General State Channel Networks

Stefan Dziembowski
Institute of Informatics
University of Warsaw, Poland
stefan.dziembowski@crypto.edu.pl

Sebastian Faust
Department of Computer Science
TU Darmstadt, Germany
sebastian.faust@cs.tu-darmstadt.de

Kristina Hostáková
Department of Computer Science
TU Darmstadt, Germany
kristina.hostakova@crisp-da.de

ABSTRACT

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

CCS CONCEPTS

• **Security and privacy** → *Cryptography*;

KEYWORDS

state channel networs; virtualization; blockchain protocols; provable secure protocols

ACM Reference Format:
Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243856

1 INTRODUCTION

In recent years we have witnessed a growing popularity of distributed cryptocurrencies such as Bitcoin [25] or Ethereum [34]. The underlying main innovation of these currencies is a consensus mechanism that allows their users to maintain the so-called *blockchain* (or *ledger*). One of the most interesting potential applications of such currencies are the *microtransactions* [23, 28, 30, 33], i.e., transactions of very small values (typically less than 1 cent) that are executed instantaneously. Once implemented, they could enable many novel business models, e.g., fair sharing of WiFi connection, or devices paying to each other in the "Internet of Things".

Unfortunately, blockchain-based systems face inherent challenges that make it very hard, if not impossible, to use them directly for microtransactions. Firstly, each transaction that is processed via the network has to be stored on the blockchain. Moreover, consensus on the blockchain requires significant time to confirm transactions, e.g., in Bitcoin confirmation takes at least around 10 minutes. This imposes a fundamental limit on how many transactions can be processed per second (for instance, the Bitcoin network is currently limited to process up to 7 transactions per second [3]). Finally, the miners that process transactions, ask for fees. Once these fees surpass the actual value assigned to a transaction, micropayments become much less attractive.

A prominent tool for addressing the above challenges are *off-chain channels* [2, 8, 19–21, 29, 31] that allow two users to rapidly exchange money between each other without sending transactions to the blockchain. Channels are implemented using so-called *smart contracts*, which allow to transfer money according to complex program rules. Below we will first briefly describe this concept, and then give a short introduction to the state of the art in off-chain channels.

Smart contracts. Informally speaking, *smart contracts* (or simply: "contracts") are programmable money, described in form of self-enforcing programs that are published on the ledger. Technically, the term "smart contract" can have two meanings: (1) a contract *code* which is a static object written is some programming language, and (2) a contract *instance* (a dynamic object that executes this code and is running on a blockchain, or inside of a state channel, see below). In the sequel we will often use this distinction (which is similar to the distinction between "programs" and "processes" in operating systems). One can think of a smart contract instance as a trusted third party to which users can send coins, and that can distribute coins between the parties, according to conditions written in its code. Probably the best known currency that supports contracts of an arbitrary complexity is *Ethereum* [34], and its most popular contract language is *Solidity*. In this system, a contract instance never acts by itself, and its actions have to be triggered by the users (who pay the so-called *fees* for every contract execution). The users communicate with the contract instances using *functions calls* (from the contract code). An instance is deployed on the ledger by a call from a special function called *constructor*. For more details on smart contracts and their formal modeling we refer to Sec. 3.

Payment channels. Payment channels are one of the most promising proposals for addressing the scalability challenges in cryptocurrencies. The main idea behind this technology is to keep the massive bulk of transactions off-chain. To this end, the parties that want to *open* a channel deploy a special "channel contract" on the blockchain and lock a certain amount of coins in it. Afterwards they can freely update the channel's balance without touching the ledger. The blockchain is contacted only when parties involved in the payment channel want to *close* the channel, or if they disagree, in which case the channel contract handles fair settlement. In the normal case, when the two parties involved in the payment channel play honestly and off-chain transactions never hit the blockchain before the channel is closed, payment channels significantly improve on the shortcomings of standard blockchain-based payments mentioned above: they limit the load put on the blockchain, allow for instantaneous payments, and reduce transaction fees.

The idea of payment channels has been extended in several directions. One of the most important extensions are the so-called *payment networks*, which enable users to route transactions via intermediary hubs. To illustrate this concept, suppose that P_1 has a payment channel with P_2, and P_2 has a payment channel with P_3. A channel network allows P_1 to route payments to P_3 via the intermediary P_2 without the need for P_1 and P_3 to open a channel between each other on the ledger. This reduces the on-chain transaction load even further. The most well known example of such a system is the *Ligthning network* that has been designed and implemented by Poon and Dryja over Bitcoin [29]. It is based on a technique called *hash-locked transactions*, in which each transaction that is sent from P_1 to P_3 is routed explicitly via P_2 – meaning that P_2 confirms that this transaction can be carried out between P_1 and P_3. For further details on hash-locked transactions, we refer the reader to, e.g., the description of the Lightning network [29] and to the full version of this paper [13].

Virtual payment channels. An alternative technique for connecting channels has recently been proposed in [12] under the name "channel virtualization". Using this technique two parties can open a virtual channel over two "extended payment channels" running on the ledger.[1] Consider the example already mentioned above, where P_1 and P_3 are not connected by a payment channel, but each of them has an extended payment channel with an intermediary called P_2. In contrast to connecting payment channels via hash-locked transactions, virtual payment channels have the advantage that the intermediary P_2 does not need to confirm each transaction routed via him. As argued in [12], virtual channels can further reduce latency and fees, while at the same time improving availability.[2] To distinguish the standard channels from the virtual ones, the former ones are also called *ledger* channels. In [12] the authors present only a construction of virtual *payment channels* over a *single* intermediary hub, leaving the general construction as an open research problem. Addressing this shortcoming is one important contribution of our work.

[1]Concretely, the contract representing the extended payment channel offers additional functionality to support connecting two payment channels.
[2]Availability is improved because payments via the virtual channel can be completed even if the intermediary is temporarily off-line.

State channels. A further generalization of payment channels are *state channels* [5], which radically enrich the functionality of payment channels. Concretely, the users of a state channel can, besides payments, execute complex smart contracts in an off-chain way. Alice and Bob who established a state channel between each other can maintain a "simulated ledger for contracts" and perform the execution of contracts on it "without registering them on the real blockchain". This happens as long as the parties do not enter into a conflict. The security of this solution comes from the fact that at any time parties can "register" the current off-chain state of the channel on the real blockchain, and let the channel contract fairly finish the execution of the contract. Examples of use cases for state channels are manifold and include contracts for digital content distribution, online gaming or fast decentralized token exchanges.

In contrast to payment channels, there has been only little work on general state channels.[3] One prominent project whose final goal is to implement general state channels over Ethereum is called *Raiden* [1], but currently it only supports simple payments, and a specification of protocols for full state channel networks has not been provided yet. The concept of an off-chain state maintained by parties was formalized in the work of Miller et al. [24], where it is used as a main building block for the payment channel construction. In contrast to [24], our general state channel construction allows two parties to have a virtual state channel whose opening does not require any interaction with the blockchain. This significantly improves the time complexity and the cost of a state channel creation. To our best knowledge, the only work considering longer general state channels is [4] recently published by Coleman et al. and developed independently from our work. The work of [4] lacks formal definitions and security proofs. On the other hand, it includes several features useful for practical implementation. We are in contact with the authors of [4] and planing collaboration to further improve our construction and move provably secure state channel networks closer to practice.

1.1 Our contribution

As described above, until now there has not been any satisfactory formal construction or security definition of general state channel networks. The main contribution of this work is to address this shortcoming by providing the *first* construction for building state channel networks of arbitrary complexity together with a formal definition and security analysis. Our construction (i) allows users to run arbitrary complex smart contracts off-chain, and (ii) permits to build channels over any number of intermediaries. Below we describe our core ideas in more detail.

Constructing state channel networks. In order to construct the general state channel networks, we follow a modular *recursive* approach where virtual state channels are built recursively on top of ledger or other – already constructed – virtual state channels. For a high-level description of our recursive approach see Sec. 2 (and Fig. 1 therein). As long as everybody is honest, the intermediaries in the virtual channel are contacted only when the channel is opened and when it is closed (and the ledger is never contacted). On the

[3]A state channel that is not application specific and allows to run arbitrarily complex contracts, is called a *general state channel*. Since we consider only general state channels in this work, we usually omit the word "general" for brevity.

other hand, let us stress that no intermediary can lose its coins even if all other parties are dishonest and every user of a virtual state channel has the guarantee that he can execute a contract created in a virtual state channel even if all other parties collude.

Modeling state channel networks and security proofs. In addition to designing the first protocols for state channel networks, we develop a UC-style model for "state channel networks" – inspired by the universal composability framework introduced in the seminal work of Canetti [9]. To this end, similarly to [12], we model money via a global ledger ideal functionality $\widehat{\mathcal{L}}$ and describe a novel ideal functionality for state channel networks that provide an ideal specification of our protocols. Using our model, we formally prove that our protocols satisfy this ideal specification. Key challenges of our analysis are (i) a careful study of timings that are imposed by the processing of the ledger, and (ii) the need to guarantee that honest parties cannot be forced to lose money by the fact that the contracts are executed off-chain even if all other parties collude and are fully malicious.

We emphasize that in the context of cryptocurrencies, a sound security analysis is of particular importance because security flaws have a direct monetary value and hence, unlike in many other settings, are guaranteed to be exploited. The later is, e.g., illustrated by the infamous attacks on the DAO [32]. Thus, we believe that before complex off-chain protocols are massively deployed and used by potentially millions of users, their specification must be analyzed using formal methods as done in our work using UC-style proofs.

Optimistic vs. pessimistic execution times. While constructing our protocols we will provide the "optimistic" and "pessimistic" execution times. The "optimistic" ones refer to the standard case when all parties behave honestly. In the optimistic case all our protocols allow for instantaneous off-chain contract execution, and a possible delay depends only on the latency of the network over which parties communicate. The "pessimistic" case corresponds to the situation when the corrupt parties try to delay the execution as much as they can by forcing contract execution on the blockchain. In our solution the pessimistic execution times grow linearly with the number of intermediaries ℓ involved. Notice that these pessimistic times can in reality happen only in the unlikely case when *all* but one party are corrupt. Since the main goal of this paper is to introduce the general framework, and not to fine-tune the parameters, we leave it as an important direction for future work to improve our construction and optimize these timings, possibly using the techniques of [24].

Further related work. One of the first proposals for building payment channels is due to Decker [11], who in particular also introduced a construction for duplex payment channels. An alternative proposal for payment channel networks has been given by Miller et al. [24]. In this work, the authors show how to reduce the pessimistic timings to constant time (i.e., independent of the length of the channel path). It is an interesting question for future work to combine the techniques from [24] with the channel virtualization. Several works focus on privacy in channel networks, path finding or money re-balancing in payment channels [19, 21, 31]. In particular, [12, 21, 24] also provide a UC-based security analysis of their

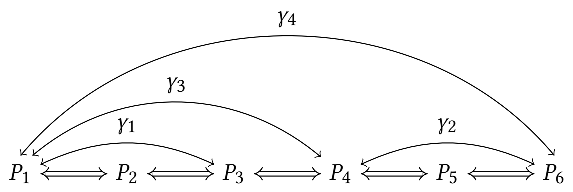

$$\begin{tikzcd}
P_1 \arrow[r, leftrightarrow, Rightarrow] \arrow[rr, bend left, leftrightarrow, "\gamma_1"] \arrow[rrr, bend left, leftrightarrow, "\gamma_3"] \arrow[rrrrr, bend left, leftrightarrow, "\gamma_4"] & P_2 \arrow[r, Leftrightarrow] & P_3 \arrow[r, Leftrightarrow] & P_4 \arrow[r, Leftrightarrow] \arrow[rr, bend left, leftrightarrow, "\gamma_2"] & P_5 \arrow[r, Leftrightarrow] & P_6
\end{tikzcd}$$

Figure 1: Example of a recursive construction of a virtual state channel γ_4 (of length 5) between P_1 and P_6.

constructions. Channel constructions based on the sequence number maturity (that we also use in this paper) have been mentioned already in [29], and recently described in more detail (as "stateful duplex off-chain micropayment channels") by Bentov et al. in [8]. Another challenge in building and maintaining complex channel networks is the fact that parties have to continuously watch what happens on the blockchain regarding the state of their channels. This problem can be addressed using so-called watchtowers [22, 27], to which users can outsource the task of watching the blockchain.

1.2 Organization of the paper

We begin with an informal description of our state channel construction in Sec. 2, where we explain how state channels are created and how they can be used. Due to the page limit, the complete protocol description is given in the the full version of this paper [13], but the specification, which, we believe is more important for future work, e.g., for protocol design, is presented in details in the main body (see Sec. 4). We introduce the necessary formalism and present security and efficiency properties required from a general state channel in Sec. 3. Our modular approach of building state channels is discussed in Sec. 5. Finally, we conclude in Sec. 6.

2 STATE CHANNEL CONSTRUCTION

Before we proceed to the more technical part of this work, let us give an intuitive explanation of our virtual state channel construction. We would like to emphasize that the description of our approach as presented in this section is very simplified and excludes many important technicalities. Formal definitions, detailed explanations of our protocols, and their full description are presented in Sections 3—5, Appx. A and in the full version of this paper [13]. As already mentioned in Sec. 1.1, we follow a recursive approach, which is shown for the case of 6 parties on Fig. 1 where we consider parties $P_1, \ldots, P_6$, with each P_i being connected with P_{i+1} via a ledger state channel $P_i \Leftrightarrow P_{i+1}$. To build a virtual state channel $\gamma_4 := P_1 \leftrightarrow P_6$, we first create a virtual state channel $\gamma_1 := P_1 \leftrightarrow P_3$ using ledger state channels $P_1 \Leftrightarrow P_2$ and $P_2 \Leftrightarrow P_3$. Then a virtual state channel $\gamma_2 := P_4 \leftrightarrow P_6$ is created using ledger state channels $P_4 \Leftrightarrow P_5$ and $P_5 \Leftrightarrow P_6$. The other virtual state channels are created recursively, as follows: first, channel $\gamma_3 := P_1 \leftrightarrow P_4$ is created using the virtual state channel γ_1 and the ledger state channel $P_3 \Leftrightarrow P_4$, and then channel γ_4 is created using the virtual state channels γ_3 and γ_2.

Ledger state channels – an overview. The terminology for ledger state channels is given in Sec. 3. and their construction is discussed in detail in Appx. A.1.

Below we explain only the main idea of the ledger state channel construction. A ledger state channel δ between Alice and Bob allows them to execute off-chain instances of some contract code $\texttt{C}$. An example could be a lottery game contract $\texttt{C}_{\texttt{lot}}$, where each user deposits 1 coin and then one user is randomly chosen to receive 2 coins. Technically, this is implemented using the standard cryptographic method based on commitment schemes (see, e.g., [6]), where the execution of the contract happens in the following steps: first the parties deposit their coins in the contract instance (call the resulting *initial state* of the game G_0)[4], then Alice sends to the contract her commitment to a random bit $r_A \in \{0, 1\}$ (which results in state G_1), afterwards Bob sends his random bit $r_B \in \{0, 1\}$ to the contract (denote the resulting state G_2). Then, Alice opens her commitment, the final state G_3 is computed, and 2 coins are given to Alice if $r_A \oplus r_B = 0$, or to Bob (otherwise). Finally, the contract instance terminates. Technically, the previous steps are implemented via function calls. For example: sending a bit r_B by Bob can be implemented as function call $\texttt{Reveal}(r_B)$ (where $\texttt{Reveal}$ is a function available in $\texttt{C}_{\texttt{lot}}$ that stores r_B in the storage of the contract).

As described in Sec. 1, two parties create a ledger state channel by deploying a *state channel contract*, (SCC), in which each party locks some amount of coins. Once the ledger state channel δ is established, parties can open instances of the contract code $\texttt{C}$ in the channel and execute them. For example the parties can open a channel in which each of them locks 10 coins and then run several instances of the lottery contract $\texttt{C}_{\texttt{lot}}$ in this channel. Every contract instance locks 1 coin of each party (from the coins that are locked in channel δ). A locked coin cannot be used for any other contract instance in δ. Once the contract instance terminates, the coins are unlocked and distributed back to the channel δ according to the rules of $\texttt{C}$. The state channel contract on the blockchain guarantees that if something goes wrong during the off-chain execution (parties disagree on a state of some contract instance, one of the parties stops communicating, etc.), they can always fairly resolve their disagreement and continue the execution via the state channel contract on the blockchain.

Off-chain contract execution in the ledger state channels. Let us now take a closer look how the off-chain contract execution is done via the ledger state channel. Let $\texttt{C}$ be a contract code, and let G denote the (dynamically changing) instance of $\texttt{C}$ that is executed in δ. To deploy G both parties agree on the initial state G_0 of G. The parties then exchanging signatures on $(G_0, 0)$. The rest of the execution is done by exchanging signatures on further states of G together with indices w that denote the *version numbers*. Assume that Alice wants to call a function f (with some parameters m) in the contract instance. Let (G_w, w) be the last state of the contract instance G on which the parties exchanged their signatures. She then (1) computes locally the new value G_{w+1} of the state, by calling $f(m)$ on G_w, and then (2) sends signed $(G_{w+1}, w + 1)$ together with f and m to Bob. Bob checks if Alice's computation was correct, and if yes then he replies with his signature on $(G_{w+1}, w + 1)$. When the instance G terminates, the coins resulting from this execution are distributed between the parties according to the outcome of the game.

For example if G is an instance of the lottery contract $\texttt{C}_{\texttt{lot}}$ described above then the states of the game are G_0, G_1, G_2 and G_3. Since the first move of the game is done by Alice, she locally computes the new state G_1 and sends it to Bob together with her commitment to r_A and her signature on $(G_1, 1)$. Then Bob replies with his signature on $(G_1, 1)$. Thereafter, Bob makes his move, i.e., he computes G_2, sends signed $(G_2, 2)$ together with his random bit r_B to Alice, and so on. Note that the interaction of the parties with the contract instance is always "local", i.e., the parties themselves compute the new states of G and then just exchange signatures.

As long as both Alice and Bob are honest, everything is done without any interaction with the blockchain. If, however, one party cheats (e.g. by refusing to communicate), the other party can always ask the SCC contract to finish the game. The version number w is used to make sure that SCC gets always the latest state of the game. More concretely: the contract is constructed in such a way that if a malicious party submits an old state, then the other party can always "overwrite" this state by providing a signed state of the contract instance with a higher version number. Once the SCC contract learns the latest state G_w, the game can be finished (starting from G_w) on-chain via SCC.

Virtual state channels – an overview. As described above, the virtual state channels are constructed recursively "on top" of the ledger state channels. Suppose that Alice and Bob want to run some contract code $\texttt{C}$ (e.g. the lottery game) in an off-chain way in γ. This time, however, they do not have an open ledger state channel between each other. Instead, both Alice and Bob have a channel with a third party, which we call Ingrid. Denote these channels α and β respectively. With the help of Ingrid but *without interacting with the blockchain*, Alice and Bob can open a virtual state channel γ that has the same functionality and provides the same guarantees as if it would be a ledger state channel between them. In particular, Alice and Bob are allowed to create a contract instance of $\texttt{C}$ in their channel γ and execute it just by communicating with each other (i.e. play their game without talking to any third party or the blockchain).

Recall that in case of the ledger state channels every dispute between Alice and Bob is resolved by the state channel contract, SCC. For the virtual state channel γ the role of such a "judge" is played by Ingrid. The main difference from the previous case is that, unlike SCC (that is executed on the ledger), *Ingrid cannot be trusted*, and in particular, she may even collude with a corrupt Alice or Bob. In order to prevent parties from cheating, we create special contracts in each of the ledger state channels α and β. Their code will be called "virtual state channel contract" (VSCC) and their instances will be denoted ν_α and ν_β, respectively. The instances ν_α provides security guarantees for Alice, and ν_β for Bob. In addition, both contract instances together provide guarantees for Ingrid. The contract code VSCC has to depend on the code $\texttt{C}$ since it needs to interpret the code $\texttt{C}$ in case the parties enter into a dispute (see below). Note that SCC depends on VSCC, and hence, indirectly, on $\texttt{C}$. This dependence is summarized in Fig. 2.

[4]A reader familiar with Ethereum may object that "simultaneous" contract instance deployment is not allowed (as Ethereum does not support "multi-input" transactions). We stress that the example above illustrates a contract that is run "inside of a channel" (not on blockchain) and is compatible with our construction.

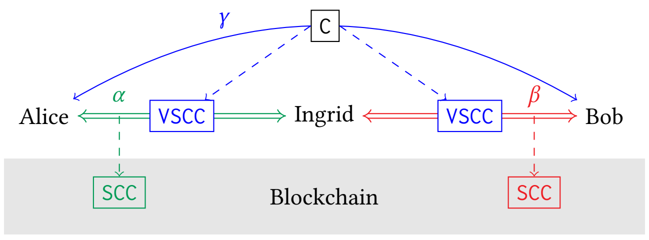


Figure 2: Construction of a virtual state channel γ_1 of length 2 in which a contract instance of C is created.

Creating the virtual state channel. Let us explain the virtual state channel creation in more detail. In the first step Alice and Bob inform Ingrid about their intention to use her as an *intermediary* for their virtual state channel γ. Alice does so by proposing to open an instance ν_α of VSCC in the channel α. This instance will contain all information about the virtual state channel γ (for example: how many coins each party wants to lock in the channel). In some sense ν_α can be viewed as a "copy" of the virtual state channel γ in which Ingrid plays the role of Bob — for example, if the initial balance in γ is 1 coin for Alice and 5 coins for Bob, then Alice would lock 1 coin and Ingrid 5 coins in ν_α. Symmetrically, Bob proposes a new instance ν_β of VSCC in the ledger state channel β that can be viewed as a "copy" of the virtual state channel γ in which Ingrid plays the role of Alice. In the example above, Ingrid would lock 1 coin and Bob 5 coins in ν_β. If Ingrid receives both proposals and she agrees to be the intermediary of the virtual state channel γ, she confirms both requests.

Contract execution in the virtual state channel γ. The *off-chain* contract execution in the virtual state channel is performed exactly in the same way as in case of the ledger state channels (see paragraph "Off-chain contract execution in the ledger state channels" above). That is, as long as both Alice and Bob are honest, they execute a contract instance G by exchanging signatures on new versions of the game states without talking to Ingrid at all, and without updating ν_α and ν_β. The case when Alice and Bob disagree needs to be handled differently, since the parties cannot contact the blockchain contract, but have to resolve this situation using the channels α and β that they have with Ingrid. Consider, for example, the situation when, in the scenario described above, Bob is malicious and stops communicating with Alice, i.e. he does not send back his signature on $(G_{w+1}, w+1)$. In this situation, Alice has to make her move "forcefully" by using the channel α she has with Ingrid. More concretely, she will execute the contract instance ν_α. It is very important to stress that the virtual state channel construction uses this instance in a *black-box way*, i.e., when describing the protocols for virtual state channel execution this protocol uses the execution of ν_α in a black-box way via the interface of the underlying channel. Internally, of course this is done by a protocol between Alice and Ingrid realizing the off-chain execution of ν_α (as long as Alice and Ingrid are honest).

First, Alice starts the "state registration procedure". The goal is to let ν_α know that she has a disagreement with Bob, and to convince ν_α that G_w is the latest state of the contract instance G. To this end, she sends to ν_α the state (G_w, w, s_B), where s_B is Bob's signature on (G_w, w). She does it by calling a function "register" (see Step 1 on Fig. 3). Of course ν_α has no reason to believe Alice that this is really the latest state of G. Therefore ν_α forwards this message to Ingrid[5], that, in turn, calls a function "register(G_w, w, s_B)" of the contract instance ν_β in channel β (see Step 2). Bob now replies (in Step 3) to ν_β with his latest version of the contract instance (i.e. he calls "register$(G_{w'}, w', s_A)$", where s_A is Alice's signature). When Ingrid learns about Bob's version from ν_β, she forwards this information to ν_α (see Step 4). Suppose that $w > w'$, i.e., Alice is honest, and Bob is cheating by submitting and old version of the instance (the other case is handled analogously). Then, both ν_α and ν_β decide that (G_w, w) is the latest version of G (i.e. they "register G_w").

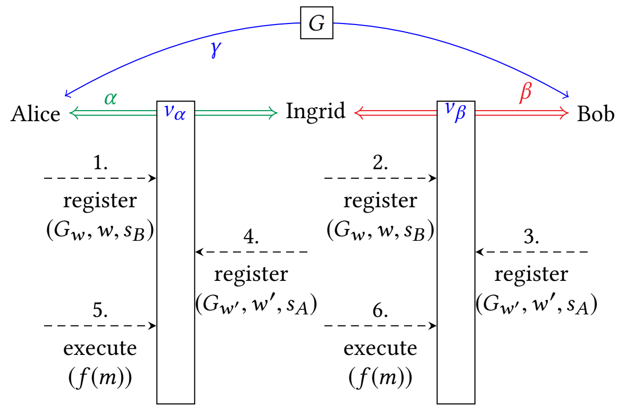
From the point of view of Ingrid, the most important security feature of this procedure is that there is a consensus among ν_α and ν_β about the latest state of G (even is Alice and Bob are *both* dishonest and playing against her). This consensus will be maintained during the entire execution of G in instances ν_α and ν_β. This is important, as otherwise she could lose coins.[6] This invariant will be maintained throughout the rest of the "forced execution procedure".

After the state registration is over, Alice calls (in Step 5, Fig. 3) a function "execute$(f(m))$" of ν_α, "asking" ν_α to execute $f(m)$ on the contract instance G starting from the registered state (G_w, w). Since we want to maintain the "consensus invariant" mentioned above, we cannot simply let ν_α perform this execution immediately after it receives this call. This is because some contracts may allow both parties to call functions at the same time[7], and Bob could simultaneously call some other function execute$(f'(m'))$ of ν_β. This situation is especially subtle because function execution is generally not commutative, i.e., executing $f(m)$ and then $f'(m')$ can produce different result than doing it in the different order. Consequently, this could result in ν_α and ν_β having different states of their local copies of γ. We solve this problem by delaying the execution of $f(m)$ until it is clear that no other function can be executed before $f(m)$. More precisely, the contract code VSCC is defined in such a way that $f(m)$ is only stored in the storage of the contract instance ν_A, resp. ν_B. The internal execution of $f(m)$ in ν_A, resp. ν_B, is performed only when the contract instance is being terminated (which happens when then virtual state channel γ is being closed).

Let us emphasize that the purpose of the description above is to explain the concepts and main ideas of our construction. The final protocol, however, works slightly differently due to several optimizations. For example, in order to decrease the pessimistic time complexity, the registration phase and the force execution phase for virtual state channels are run in parallel (i.e. Step 1 and Step 5 are happening in the same round). We refer the reader to Appx. A.2 for more details about the construction.

Applying recursion. As already highlighted earlier, longer virtual state channels are constructed recursively. The key observation that enables this recursion is that the state channels α and β that are

[5]Recall again that this execution is realized by a protocol between Alice and Ingrid.
[6]Imagine, e.g, that the final state of G in ν_α is that Alice gets all the coins locked in G, and the final state of G in ν_β is that Bob gets all the coins locked in G. Then Ingrid loses these coins in *both* channels α and β.
[7]Note that it is *not* the case of the $\mathtt{C_{lot}}$ contract, since there its always clear which party is expected to "make a move" in the game. However, in general, we do not want to have such restrictions on contracts in this paper.



G
γ
Alice
α
v_α
Ingrid
v_β
β
Bob
1. register (G_w, w, s_B)
2. register (G_w, w, s_B)
3. register $(G_{w'}, w', s_A)$
4. register $(G_{w'}, w', s_A)$
5. execute $(f(m))$
6. execute $(f(m))$

Figure 3: Illustration of the forced execution process from our example in which Alice and Bob have a virtual state channel γ in which they opened a contract instance. Only the function calls are shown (the messages sent by the contracts are omitted).

used to build γ are accessed in a "black-box" way. In other words, the only property of α and β needed in the construction of γ is that one can execute off-chain contracts in them. This "black-box" property guarantees that our virtual state channel construction works also if the channels α and β are *virtual* (not ledger), or in case one of them is virtual, and the other one is ledger.

Let us illustrate this on the situation depicted in Fig. 1. Consider first the virtual state channel γ_3 – a virtual state channel of length 3 build on top of a virtual state channel γ_1 of length 2 and the ledger state channel $P_3 \Leftrightarrow P_4$. Assume that C is the contract code whose instances can be opened in γ_3. Following the construction described earlier in this section, γ_3 can be created if both the underlying state channels γ_1 and $P_3 \Leftrightarrow P_4$ support contract instances of the virtual state channel contract VSCC which depends on C. This, in particular, implies that the ledger state channels $P_1 \Leftrightarrow P_2$ and $P_2 \Leftrightarrow P_3$, on top of which the virtual state channel γ_1 is created, must support contract instances of the virtual state channel contract VSCC′ which depends on VSCC (thus indirectly also on C).

This reasoning can be repeated for longer channels. For example, if C is a contract code whose instances can be opened in the virtual state channel γ_4, then contract instances of VSCC must be supported by both γ_2 and γ_3, contract instances of VSCC′ must be supported by γ_1, $P_3 \Leftrightarrow P_4$, $P_4 \Leftrightarrow P_5$ and $P_5 \Leftrightarrow P_6$. Finally, contract instances of the virtual state channel contract VSCC′′, which depends on VSCC′, must be supported by the ledger state channels $P_1 \Leftrightarrow P_2$, $P_2 \Leftrightarrow P_3$. More details of this recursion, including the analysis of pessimistic and optimistic timing, are provided in further sections. Let us just mention here that in order to achieve linear pessimistic time complexity (in the channel length), our construction assumes that virtual state channels are built in a balanced way as in Fig. 1 (i.e. the two state channels used to build a virtual state channel have approximately the same length).

The notion of time. In the description above we ignored the notion of time. This was done to simplify this informal description. We define this notion in the technical part of the paper (see Sec. 3.3). In our construction parties are always aware of the current time, and they pass the time information to the contract functions in the state channels. Time is modeled as a natural number, and the time unit is called a *round* (think of it as 1 second, say).

Other key features of our construction. An important property of our construction and our model is that we support full concurrency. That is, we allow several virtual state channels to be created simultaneously over the same ledger state channels, and allow parties to be involved in several concurrent executions of (possibly complex) contracts. This is possible because our ledger state channels can store and execute several contracts "independently".

Another important feature of our modular construction is that it naturally allows for building channels via multiple (possible incompatible) cryptocurrencies as long as they have a sufficiently complex scripting language (in particular, they allow to deploy a state channel contract). For illustration, consider Alice having a ledger state channel with Ingrid in cryptocurrency called "A-coin", and Bob having a ledger state channel with Ingrid in cryptocurrency called "B-coin". Now, Alice and Bob can build a virtual state channel over Ingrid, where Alice (resp. Bob) is oblivious of the details of B-coin (resp. A-coin). This makes sense as long as the exchange rate between the currencies does not change too much during the lifetime of the virtual channel. Note that, since the virtual channel opening and closing does not require interacting with the ledger, the lifetime of a virtual state channel can be made very short (minutes or hours). In addition, virtual state channels also improve on privacy. This is the case because channel updates are fully P2P and do not require involvement of intermediaries.

Finally, we point out that our concept of higher-level channel virtualization has the key feature that it adds further "layers of defense" against malicious parties before honest users need to communicate with the blockchain. Consider, for example, the situation shown in Fig. 1. Even if P_6 and the intermediary P_4 in the virtual state channel γ_4 are corrupt, then P_1 can resolve possible conflicts via the intermediary P_3 using the virtual state channel γ_1, i.e. P_1 does not need to communicate with the ledger.

3 DEFINITIONS AND SECURITY MODEL

In the sequel, following [12], we present tuples of values using the following convention. The individual values in a tuple T are identified using keywords called *attributes*: $\mathsf{attr1}, \mathsf{attr2}, \ldots$. Strictly speaking an *attribute tuple* is a function from its set of attributes to $\{0,1\}^*$. The *value of an attribute* attr in a tuple T (i.e. $T(\mathsf{attr})$) will be referred to as $T.\mathsf{attr}$. This convention will allow us to easily handle tuples that have dynamically changing sets of attributes. We assume that $(\mathsf{Gen}, \mathtt{Sign}, \mathsf{Vrfy})$ is a signature scheme that is existentially unforgeable against a chosen message attack (see, e.g., [17]). The ECDSA scheme used in Ethereum is believed to satisfy this definition.

3.1 Definitions of contracts and channels

We now present our syntax for describing contracts and channels. The notation presented in this section can be viewed as an extension of the one used in [12]. In the rest of this paper we assume that the set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of parties that use the system is fixed.

Contracts. We consider only contracts executed between two parties. A *contract storage* is an attribute tuple σ that contains at least the following attributes: (1) $\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R \in \mathcal{P}$ that denote the users involved in the contract, (2) $\sigma.\mathsf{locked} \in \mathbb{R}_{\geq 0}$ that denotes the total amount of coins that is locked in the contract and (3) $\sigma.\mathsf{cash}\colon \{\sigma.\mathsf{user}_L, \sigma.\mathsf{user}_R\} \rightarrow \mathbb{R}$ that denotes the amount of coins that the users have freely available. It must hold that $\sigma.\mathsf{locked} \geq \sigma.\mathsf{cash}(\sigma.\mathsf{user}_L) + \sigma.\mathsf{cash}(\sigma.\mathsf{user}_R)$. Let us explain the difference between locked coins and freely available coins as well as the above inequality on a concrete example. Assume that parties are playing a game where each party initially invests 5 coins. During the game, parties make a bet, where each party puts 1 coin in the "pot". Now the amount of coins *locked* in the game did not change, it is still equal to 10 coins; however, the amount of *freely available* coins decreased (each party has only 4 freely available coins). In addition to the attributes mentioned above, a contract storage may contain other application-specific data.

We will now define formally the notion of *contract code* that was already described informally in Sec. 1. Formally a contract code consists of some functions (in Ethereum they are written in Solidity) that operate on contract storage. The set of possible contract storages is usually restricted (e.g. the functions expect that it has certain attributes defined). We call the set of restricted storages the *admissible contract storages* and typically denote it Λ.

Formally, we define a *contract code* as a tuple $\mathtt{C} = (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$, where Λ are the admissible contract storages and $g_1, \ldots, g_r$ are functions called *contract constructors*, and $f_1, \ldots, f_s$ are called *contract functions.* Each contract constructor g_i is function that takes as input a tuple (P, τ, z), with $P \in \mathcal{P}, \tau \in \mathbb{N}$, and $z \in \{0,1\}^*$, and produces as output an admissible contract storage σ or a special symbol $\perp$ (in which case we say that the contract construction *failed*). The meaning of these parameters is as follows: P is the identity of the party that called the function, τ is the current round (see Sec. 3.3 for more on how we model time and rounds), and z is used to pass additional parameters to g_i. The constructors are used to create a new instance of the contract. If the contract construction did not fail, then $g_i(P, \tau, z)$ is the initial storage of a new contract instance.

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

As described in Sec. 1 a *contract instance* represents an instantiation of a contract code. Formally, a contract instance is an attribute tuple ν with a contract storage and code, where $\nu.\mathsf{code} = (\Lambda, g_1, \ldots, g_r, f_1, \ldots, f_s)$ is a contract code, and $\nu.\mathsf{storage} \in \Lambda$ is a contract storage.

Ledger state channel. We next present our terminology for ledger state channels, which is inspired by the notation for payment channels used in [12]. Formally, a ledger state channel γ is defined as an attribute tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{cspace})$. We call the attribute $\gamma.\mathsf{id} \in \{0,1\}^*$ the identifier of the ledger state channel. Attributes $\gamma.\mathsf{Alice} \in \mathcal{P}$ and $\gamma.\mathsf{Bob} \in \mathcal{P}$ are the identities of parties using the ledger state channel γ. For convenience, we also define the set $\gamma.\mathsf{end\text{-}users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ and the function $\gamma.\mathsf{other\text{-}party}$ as $\gamma.\mathsf{other\text{-}party}(\gamma.\mathsf{Alice}) := \gamma.\mathsf{Bob}$ and $\gamma.\mathsf{other\text{-}party}(\gamma.\mathsf{Bob}) := \gamma.\mathsf{Alice}$. The attribute $\gamma.\mathsf{cash}$ is a function mapping the set $\gamma.\mathsf{end\text{-}users}$ to $\mathbb{R}_{\geq 0}$ such that $\gamma.\mathsf{cash}(T)$ is the amount of coins the party $T \in \gamma.\mathsf{end\text{-}users}$ has locked in the ledger state channel γ. Finally, the attribute $\gamma.\mathsf{cspace}$ is a partial function that is used to describe the set of all contract instances that are currently open in this channel. It takes as input a *contract instance identifier cid* $\in \{0,1\}^*$ and outputs a contract instance ν such that $\{\nu.\mathsf{storage.user}_L, \nu.\mathsf{storage.user}_R\} = \gamma.\mathsf{end\text{-}users}$. We will refer to $\gamma.\mathsf{cspace}(cid)$ as the *contract instance with identifier cid in the ledger state channel* γ.

Virtual state channel. Formally, a virtual state channel γ is a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{subchan}, \gamma.\mathsf{cash}, \gamma.\mathsf{cspace}, \gamma.\mathsf{length}, \gamma.\mathsf{validity})$. The attributes $\gamma.\mathsf{id}$, $\gamma.\mathsf{Alice}$, $\gamma.\mathsf{Bob}$, $\gamma.\mathsf{cash}$ and $\gamma.\mathsf{cspace}$, are defined as in the case of a ledger state channel. The same holds for the set $\gamma.\mathsf{end\text{-}users}$ and the function $\gamma.\mathsf{other\text{-}party}$. The new attribute $\gamma.\mathsf{Ingrid} \in \mathcal{P}$ denotes the identity of the intermediary of the virtual state channel. For technical reasons (see the full version of this paper [13] for more on this) we restrict $\gamma.\mathsf{cspace}$ for virtual state channels to contain only a single contract instance. We emphasize that this is not a restrictions of the functionality since ledger state channels support an arbitrary number of contract instances, and hence we can build any number of virtual state channels.

The attribute $\gamma.\mathsf{subchan}$ is a function mapping the set $\gamma.\mathsf{end\text{-}users}$ to $\{0,1\}^*$. The value of $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ equals the identifier of the ledger/virtual state channel between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$. Analogously for the value of $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$. We often call these channels the *subchannels* of the virtual state channel γ. The attribute $\gamma.\mathsf{validity}$ denotes the round in which the virtual state channel γ will be closed (see Sec. 3.3 for more on the notion of rounds). The reason to have this parameter is to ensure that the channel γ will not remained open forever. Otherwise $\gamma.\mathsf{Ingrid}$ could have her money blocked forever, as (unlike $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$) she cannot herself request the channel closing. Finally, the attribute $\gamma.\mathsf{length} \in \mathbb{N}_{>1}$ refers to the length of the virtual state channel, i.e., the number of ledger state channels over which it is built. For example in Fig. 1 (see Page 3) we have: $\gamma_1.\mathsf{length} = 2$, $\gamma_2.\mathsf{length} = 2$, $\gamma_3.\mathsf{length} = 3$, $\gamma_4.\mathsf{length} = 5$. Sometimes it will be convenient to say that ledger state channels have length one.

3.2 Security and efficiency goals

Before presenting our formal security model in Sec. 3.3, let us start by listing some security guarantees that are desirable for a state channel network. In the following description, if it is not important whether γ is ledger state channel or a virtual state channel, and hence we will refer to γ as a *state channel.*

(1) **Consensus on creation:** A state channel γ can be successfully created only if all users of γ agree with its creation.
(2) **Consensus on updates:** A contract instance in a state channel γ can be successfully updated (this includes also creation of the contract instance) only if both end-users of γ agree with the update.
(3) **Guarantee of execution:** An honest end-user of a ledger state channel γ can execute a contract function f of a created contract instance in any round τ_0 on input value z even if the other end-user of γ is corrupt. This property holds also for virtual state channels with the restriction that $\tau_0 < \gamma.\mathsf{validity}$.
(4) **Balance security:** The intermediary of a virtual state channel γ never loses coins even if both end-users of γ are corrupt.

While property (4) provides a strong monetary security guarantee to the intermediary of a virtual state channel, the guarantees for the end-users given by properties (2) and (3) only ensure that party can not be forced to create a contract instance and that contract instances can be executed at any time. We emphasize that this is similar to what is guaranteed by the ledger to on-chain contracts. Concretely, this means that if the contract rules allow that a certain end-user may lose money (e.g., by losing the lottery as described in the example from Sec. 2), then this is not in violation with the security properties guaranteed by a state channel network.

In addition to the security properties, we identify the following two efficiency goals. Below, by constant number of rounds we mean that the required rounds for executing the procedure is independent of the channel length and the ledger delay Δ (looking ahead, the parameter Δ models the fact that changes on a blockchain come with a certain delay, see Sec. 3.3 for more details).

(1) **Constant round optimistic update/execute:** In the optimistic case when both end-users of a state channel γ are honest, they can update/execute a contract instance in γ within a constant number of rounds.
(2) **Constant round virtual state channel creation:** Successful creation of a virtual state channel γ takes a constant number of rounds.

3.3 Our model

To formally model the security of our construction, we use a UC-style model following the works of [7, 12] that consider protocols that operate with *coins.*[8] In particular, our model uses a synchronous version of the global UC framework (GUC) [10] which extends the standard UC framework [9] by allowing for a global setup.

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

Modeling communication and time. We assume a synchronous communication network, which means that the execution of the protocol happens in rounds. Let us emphasize that the notion of rounds is just an abstraction which simplifies our model (see, e.g, [15, 16, 18, 26] for a formalization of this model and its relation to the model with real time). Whenever we say that some operation (e.g. sending a message or simply staying in idle state) *takes at most* $\tau \in \mathbb{N} \cup \{\infty\}$ *rounds* we mean that it is up to the adversary to decide how long this operation takes (as long as it takes at most τ rounds). Let us now discuss the amount of time it takes for different entities to communicate with each other. The communication between two parties P_i takes exactly one round. All other communication – for example, between the adversary Adv and the environment $\mathcal{Z}$ – takes zero rounds. For simplicity we assume that any computation made by any entity takes zero *rounds* as well.

Handling coins. We follow [12] and model the money mechanics offered by crypotcurrencies such as Bitcoin or Ethereum via a global ideal functionality $\widehat{\mathcal{L}}$ using the *global UC (GUC)* model [10]. The state of the ideal functionality $\widehat{\mathcal{L}}$ is public and can be accessed by all parties of the protocol π, the adversary Adv and the environment $\mathcal{Z}$. It keeps track on how much money the parties have in their accounts by maintaining a vector of non-negative (finite precision) real numbers $(x_1, \ldots, x_n)$, where each x_i is the amount of coins that P_i owns.[9] The ledger functionality $\widehat{\mathcal{L}}$ is formally defined in the full version of this paper [13] and informally described below.

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

The GUC-security definition. Let π be a protocol working in the $\mathcal{G}$-hybrid model with access to the global ledger $\widehat{\mathcal{L}}(\Delta)$. The output of an environment $\mathcal{Z}$ interacting with a protocol π and an adversary Adv

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

[9]This is similar to the concept of a *safe* of [7].

on input 1^λ and auxiliary input z is denoted as $\text{EXEC}^{\widehat{\mathcal{L}}(\Delta),\mathcal{G}}_{\pi,\mathsf{Adv},\mathcal{Z}}(\lambda,z)$. If π is a trivial protocol in which the parties simply forward their inputs to an ideal functionality $\mathcal{F}$, then we call the parties *dummy parties*, the adversary a *simulator* Sim, and we denote the above output as $\text{IDEAL}^{\widehat{\mathcal{L}}(\Delta)}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\lambda,z)$.

To simplify the description of our protocols and the ideal functionalities, we consider a class of restricted environments which we denote $\mathcal{E}_{res}$. These restrictions typically disallow the environment to carry out certain actions, e.g., we forbid $\mathcal{Z}$ to instruct one party to start a protocol without instructing the other party to start the protocol as well.[10] We emphasize that these restrictions can easily be eliminated by integrating additional checks into the protocols and functionalities. The restrictions defining $\mathcal{E}_{res}$ are informally introduced in Sec. 4 and their complete list can be found in the full version of this paper [13]. We are now ready to state our main security definition.

Definition 3.1. Let $\mathcal{E}$ be some set of restricted environments. We say that a protocol π working in a $\mathcal{G}$-hybrid model *emulates an ideal functionality $\mathcal{F}$ with respect to a global ledger $\widehat{\mathcal{L}}(\Delta)$ against environments from class $\mathcal{E}$* if for every adversary Adv there exists a simulator Sim such that for every environment $\mathcal{Z} \in \mathcal{E}$ we have

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

(where "$\approx^c$" denotes computational indistinguishability of distribution ensembles, see, e.g., [14]).

Informally, the above definition says that any attack that can be carried out against the real-world protocol π can also be carried out against the ideal functionality $\mathcal{F}$. Since the ideal functionality is secure by design (see Sec. 4.2), also the protocol offers the same level of security. In Sec. 5 we will discuss in more detail the roles of $\mathcal{F}$ and $\mathcal{G}$.

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

4 STATE CHANNELS IDEAL FUNCTIONALITY

In this section, we describe the ideal functionality that defines how ledger state channels and virtual state channels are created, maintained and closed. Before we do so, let us establish several conventions which simplify the description of the ideal functionality.

[10]For readers familiar with UC, we notice that general UC composition of course requires arbitrary environments. In the full version of this paper [13] we prove that for our particular set of restrictions composition of our sub-protocols is preserved. An alternative approach would be to use a wrapper. However, due to the complexity of our protocol the description, of the wrapper would be highly convoluted.

4.1 Abbreviated notation

When it is clear from the context which state channel γ we are talking about, we will denote the parties of γ as $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$. We also introduce symbolic notation for sending and receiving messages. Instead of the instruction "Send the message msg to party P in round τ", we write $msg \overset{\tau}{\hookrightarrow} P$. Instead of the instruction "Send the message msg to all parties in the set γ.end−users in round τ", we write $msg \overset{\tau}{\hookrightarrow} \gamma.\mathsf{end{-}users}$. By $msg \overset{\tau}{\hookleftarrow} P$ we mean that an entity (i.e. the ideal functionality) receives a message msg from party P in round τ, and we use $msg \xleftarrow{\tau\leq\tau_1} P$ when an entity receives msg from party P latest in round τ_1. In the description of the ideal functionality we use two "timing functions": $\mathrm{TimeExeReq}(i)$ that represents the maximal number of rounds it takes to inform a party that execution of a contract instance in a state channel of length $i > 0$ was requested by the other party, and $\mathrm{TimeExe}(i)$ that represents the maximal number of rounds it takes to execute of a contract instance in a state channel of length $i > 0$. Both of these functions are of the order $O(\Delta \cdot i)$ (see the full version of this paper [13] for formal definition of these function and their relationship).

Each entity stores and maintains a set of all state channels it is aware of. Following [12] this set will be called *channel space* and denoted Γ. Sometimes we will abuse notation and interpret the channel space as a function which on input $id \in \{0,1\}^*$ returns a state channel with identifier id if such state channel exist and otherwise $\perp$. Every time a new contract instance in some of the state channels stored in Γ is successfully created (or an existing one is executed), the channels space Γ must be updated accordingly. To this end we define an auxiliary procedure `UpdateChanSpace`. The procedure takes as input a channel space Γ, a channel identifier id, a contract instance identifier cid, a new contract instance ν and two values add_A and add_B representing the required change in the cash values of the state channel with identifier id. The procedure sets $\Gamma(id).\mathsf{cspace}(cid) := \nu$, adds add_A coins to $\Gamma(id).\mathsf{cash}(A)$ and adds add_B coins to $\Gamma(id).\mathsf{cash}(B)$. Finally, it outputs the updated channel space Γ. Formal definition can be found in the full version of this paper [13].

4.2 The ideal functionality

We denote the state channel ideal functionality by $\mathcal{F}^{\widehat{\mathcal{L}}(\Delta)}_{ch}(i,C)$, where $i \in \mathbb{N}$ is the maximal length of a state channel that can be opened via the functionality, and C denotes the set of contract codes whose instances can be created in the state channels. The ideal functionality $\mathcal{F}^{\widehat{\mathcal{L}}(\Delta)}_{ch}(i,C)$ communicates with parties from the set $\mathcal{P}$, and has access to the global ideal functionality $\widehat{\mathcal{L}}$ (the ledger). $\mathcal{F}^{\widehat{\mathcal{L}}(\Delta)}_{ch}(i,C)$ maintains a channel space Γ containing all the open state channels. The set Γ is initially empty.

Since inputs of parties and the messages they send to the ideal functionality do not contain any private information, we implicitly assume that the ideal functionality forwards all messages it receives to the simulator Sim. More precisely, upon receiving the message m from party P the ideal functionality sends the message (P,m) to the simulator. The task of the simulator is to instruct the ideal functionality to make changes on the ledger and to output messages to the parties in the correct round (both depends on the choice made by the adversary Adv in the real world). In the description of the ideal functionality, we do not explicitly mention these instructions of Sim, but instead use the following abbreviation. By saying "wait for at most Δ rounds to remove/add x coins from P's account on the ledger" we mean that the ideal functionality waits until it is instructed by the simulator, which will happen within at most Δ rounds, and then request changes of P's account on the ledger. Let us emphasize this abbreviated notation does not affect the reactive nature of the ideal functionality (meaning that every action of the functionality has to be triggered by some other entity).

We present the formal definition of the $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ functionality in Fig. 4. Here we provide some intuitions behind this definition, introduce the most important restrictions on the environment (see Sec. 3.3), and argue why the ideal functionality satisfies all the security and efficiency properties stated in Sec. 3.2. Let us note that the constants appearing in the formal description of $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ follow from the technical details of our protocols which can be found, together with the formal proof of the security and efficiency properties, in the full version of this paper [13].

State channel creation. The $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ functionality consists of two "state channel creation" procedures: one for ledger and one for virtual state channels. The ledger state channel creation procedure starts with a "create" message from A (without loss of generality we assume that A always initiates the creation process). The functionality removes the coins that A wants to deposit in the ledger state channel from A's account on the ledger, and waits for B to declare that he wants to create the ledger state channel as well. If this happens within Δ rounds, then B's coins are removed from the ledger and the ledger state channel is created which is communicated to the parties with the "created" message. Otherwise A can get her money back by sending a "refund" message. Since both parties have to send the message "create", the *consensus on creation* security property is clearly satisfied for ledger state channels.

The creation procedure for a virtual state channel γ works slightly differently since its effects are visible on the subchannels of γ. The intention to create γ is expressed by $P \in \gamma.\mathsf{end\text{-}users} \cup \{I\}$ by sending a "create" message to the functionality. Once such a message is received from P, the coins that are needed to create γ are locked immediately in the corresponding subchannel of γ (if $P = I$, then coins are locked in both subchannels of γ). If the functionality receives the "create" messages from all three parties within three rounds, then the virtual state channel is created, which is communicated to $\gamma.\mathsf{end\text{-}users}$ by the "created" message.[11] Thus, the *consensus on creation* security property is satisfied also for virtual state channels and since the successful creation takes three rounds, the *constant round virtual state channel creation* holds as well.

After the virtual state channel is created, $\gamma.\mathsf{end\text{-}users}$ can use it until round $\gamma.\mathsf{validity}$. When this round comes, the parties initiate the closing procedure. The functionality then distributes the coins of γ back to its subchannels according to the balance in γ's last version. In case there exists *cid* such that $\gamma.\mathsf{cspace}(cid)$ is a contract instance with locked coins, then all of these coins go back to I in *both* subchannels of γ. This is to guarantee that I never loses coins even if end-users of γ do not terminate their contract instance in γ before $\gamma.\mathsf{validity}$.

In both cases ("ledger" and "virtual") we assume that all the honest parties involved in channel creation initiate the procedure in the same round and that they have enough funds for the new state channel. In case of a virtual state channel, we additionally assume that the length of its two subchannels differ at most by one.[12]

Contract instance update. The procedure for updating a contract instance is identical for ledger and virtual state channels (this procedure is also used for creating new contract instances). It is initiated by a party $P \in \gamma.\mathsf{end\text{-}users}$ that sends an "update" message to the ideal functionality. This message has parameters *id* and *cid* that identify a state channel γ and a contract instance in this state channel (respectively). The other parameters, $\tilde{\sigma}$ and C, denote the new storage and code of the contract instance. The party $Q := \gamma.\mathsf{other\text{-}party}(P)$ is asked to confirm the update via an "update-requested" message. If Q replies with an "update-reply" message within 1 round if both parties are honest and within T rounds otherwise (where T is a function of state channel length, see Step 2), the contract instance with identifier *cid* in γ gets replaced with a contract instance determined by the tuple $(\tilde{\sigma}, \mathsf{C})$. In the next round, both parties in $\gamma.\mathsf{end\text{-}users}$ get notified via an "updated" message. Note that Q always has to confirm the update which implies the *consensus on update* security property. The *constant round optimistic update* efficiency property holds as well since the update takes exactly 2 rounds if both parties are honest.

We assume that the environment never asks the parties to do obviously illegal things, like updating a contract instance in a state channel that does not exits, or creating a contract instance when there are not enough coins in the subchannels. Moreover, we assume that the environment never asks to update a contract instance when it is already being updated or executed.[13]

Contract instance execution. The procedure for executing a contract instance is initiated by one of the parties $P \in \gamma.\mathsf{end\text{-}users}$ that sends an "execute" message to the ideal functionality in round τ_0. This message has parameters *id* and *cid* whose meaning is as in the update procedure. Other parameters are: f denoting the contract function to be executed, and z which is an additional input parameter to the function f. The execution results in updating the contract instance with identifier *cid* according to the result of computing $f(\sigma, P, \tau, z)$, where σ is the current storage of the contract instance and $\tau := \tau_0$ in case P is honest and determined by the simulator otherwise. The other party of the state channel is notified about the execution request before round $\tau_0 + 5$ in the optimistic case and before round $\tau_0 + T_1$ otherwise. Both parties from the set $\gamma.\mathsf{end\text{-}users}$ learn the result of the execution before round $\tau_0 + 5$ in the optimistic case (which implies the *constant round optimistic execute*)

[11]Note that the intermediary I is not informed whether the virtual channel has been created. This choice is made to keep the protocol as simple as possible. Note also that I does not need this information, as she is not allowed to update this virtual channel.

[12]As discussed in Sec. 2, we make this assumption to achieve pessimistic time complexity which is linear in the state channel length.

[13]Although we forbid parallel updates of the *same* contract instance, we do not make any restrictions about parallel updates of two different contract instances even if they are in the same ledger state channel. This in particular means that we allow concurrent creation of virtual state channels.

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

This functionality accepts messages from parties in $\mathcal{P}$. We use the abbreviated notation defined in Sec. 4.1.

Ledger state channel creation

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

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

Virtual state channel creation

(1) Upon $(\text{create}, \gamma) \hookleftarrow P$, where $P \in \gamma.\text{end–users} \cup \{I\}$, record the message and proceed as follows:
   - If $P \in \gamma.\text{end–users}$ proceed as follows: If you have not yet received (create, γ) from I, then remove $\gamma.\text{cash}(P)$ coins from P's balance in $\gamma.\text{subchan}(P)$ and $\gamma.\text{cash}(\gamma.\text{other-party}(P))$ coins from I's balance in $\gamma.\text{subchan}(P)$.
   - If $P = I$, then for both $P \in \gamma.\text{end–users}$ proceed as follows: If you have not yet received (create, γ) from P then remove $\gamma.\text{cash}(P)$ coins from P's balance in $\gamma.\text{subchan}(P)$, and $\gamma.\text{cash}(\gamma.\text{other-party}(P))$ coins from I's balance in $\gamma.\text{subchan}(P)$.
(2) If within 3 rounds you record (create, γ) from all users in $\gamma.\text{end–users} \cup \{\gamma.\text{Ingrid}\}$, then define $\Gamma(\gamma.\text{id}) := \gamma$, send $(\text{created}, \gamma) \hookrightarrow \gamma.\text{end–users}$ and wait for channel closing in Step 4 (in the meanwhile accepting the update and execute messages concerning γ).
(3) Otherwise wait until round $\gamma.\text{validity}$. Then within $2 \cdot (\text{TimeExeReq}(\lceil j/2 \rceil) + \text{TimeExe}(\lceil j/2 \rceil))$ rounds, where $j := \gamma.\text{length}$, refund the coins that you removed from the subchannels in Step 1.

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

(4) Let $j := \gamma.\text{length}$. Within $2 \cdot (\text{TimeExeReq}(\lceil j/2 \rceil) + \text{TimeExe}(\lceil j/2 \rceil))$ rounds proceed as follows. Let $\hat{\gamma}$ be the current version of the virtual state channel, i.e. $\hat{\gamma} := \Gamma(\gamma.\text{id})$, and let $\hat{c}_A := \hat{\gamma}.\text{cash}(A)$ and $\hat{c}_B := \hat{\gamma}.\text{cash}(B)$.
(5) Add $\hat{c}_A$ coins to A's balance and $\hat{c}_B$ coins to I's balance in $\gamma.\text{subchan}(A)$. Add $\hat{c}_A$ coins to I's balance and $\hat{c}_B$ coins to B's balance in $\gamma.\text{subchan}(B)$. If there exists $cid \in \{0,1\}^*$ such that $\sigma_{cid} := \hat{\gamma}.\text{cspace}(cid).\text{storage} \neq \perp$ and $\hat{c} := \sigma_{cid}.\text{locked} > 0$, then add $\hat{c}$ coins to I's balance in both $\gamma.\text{subchan}(A)$ and $\gamma.\text{subchan}(B)$. Erase $\hat{\gamma}$ from Γ and $(\text{closed}, \gamma.\text{id}) \hookrightarrow \gamma.\text{end–users}$.

Contract instance update

Upon $(\text{update}, id, cid, \tilde{\sigma}, \mathsf{C}) \overset{\tau_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$, $j = \gamma.\text{length}$. If $P \notin \gamma.\text{end–users}$ then stop. Else proceed as follows:

(1) Send $(\text{update–requested}, id, cid, \tilde{\sigma}, \mathsf{C}) \xhookrightarrow{\tau_0+1} \gamma.\text{other–party}(P)$ and set $T := \tau_0 + 1$ in optimistic case when both parties in γ.end–users are honest. Else if $j = 1$, set $T := \tau_0 + 3\Delta + 1$ and if $j > 1$, set $T := \tau_0 + 4 \cdot \text{TimeExeReq}(\lceil j/2 \rceil) + 1$.
(2) If $(\text{update–reply}, ok, id, cid) \xhookleftarrow{\tau_1 \leq T} \gamma.\text{other–party}(P)$, then set $\Gamma := \mathtt{UpdateChanSpace}(\Gamma, id, cid, \tilde{\sigma}, \mathsf{C}, add_A, add_B)$, where $add_A := -\tilde{\sigma}.\text{cash}(A)$ if $\gamma.\text{cspace}(cid) = \perp$ and $add_A := \sigma.\text{cash}(A) - \tilde{\sigma}.\text{cash}(A)$ otherwise for $\sigma := \gamma.\text{cspace}(cid).\text{storage}$. The value add_B is defined analogously. Then send $(\text{updated}, id, cid) \xhookrightarrow{\tau_1+1} \gamma.\text{end–users}$ and stop.

Contract instance execution

Upon $(\text{execute}, id, cid, f, z) \overset{\tau_0}{\hookleftarrow} P$, let $\gamma := \Gamma(id)$ and $j = \gamma.\text{length}$. If $P \notin \gamma.\text{end–users}$ then stop. Else set T_1 and T_2 as:

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

(1) In round $\tau_1 \leq T_1$, send $(\text{execute–requested}, id, cid, f, z) \overset{\tau_1}{\hookrightarrow} \gamma.\text{other–party}(P)$.
(2) In round $\tau_2 \leq T_2$, let $\gamma := \Gamma(id)$, $\nu := \gamma.\text{cspace}(cid)$, $\sigma := \nu.\text{storage}$, and $\tau := \tau_0$ if P is honest and else τ is set by the simulator. Compute $(\tilde{\sigma}, add_L, add_R, m) := f(\sigma, P, \tau, z)$. If $m = \perp$, then stop. Else set $\Gamma := \mathtt{UpdateChanSpace}(\Gamma, id, cid, \tilde{\sigma}, \nu.\text{code}, add_L, add_R)$ and send $(\text{executed}, id, cid, \tilde{\sigma}, add_L, add_R, m) \overset{\tau_3}{\hookrightarrow} \gamma.\text{end–users}$.

Ledger state channel closure

Upon $(\text{close}, id) \overset{\tau_0}{\hookleftarrow} P$, let $\gamma = \Gamma(id)$. If $P \notin \gamma.\text{end–users}$ then stop. Else wait at most 7Δ rounds and distinguish the following two cases:

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

Figure 4: The state channel ideal functionality.

and before round $\tau_0 + T_2$ otherwise. The values T_1 and T_2 are functions of state channel length, see the formal description in Fig. 4. Observe that contract instance execution initiated by party P does *not* require approval of the other party of the channel (although the other party is informed about the execution request). This implies that the *guarantee of execution* security property is satisfied.

We would like to emphasize that if two different execute messages are received by the ideal functionality at the same time (or not too many rounds from each other), then it is up to the adversary to decide which function is executed first.[14] Designers of contract codes and users of the protocols should be aware of this possible asynchronicity.

Ledger state channel closure. The procedure for closing a ledger state channel γ starts when a party $P \in \gamma$.end−users sends to the ideal functionality a message (close, *id*), where *id* is the identifier of ledger state channel γ to be closed. The functionality checks (in Step 1) if there are no contract instances that are open over γ. If not, then in Step 2 the functionality distributes the coins from γ to the ledger accounts of the parties according to γ's latest balance, and notifies the parties about a successful closure.

4.3 Using the state channel ideal functionality

Let us now demonstrate how to use our ideal functionality for generalized state channel networks in practice. We do it on a concrete example of the two party lottery (already discussed in Sec. 2). The first step is to define a contract code $\mathtt{C_{lot}}(i)$ which allows two parties to play the lottery in a state channel of length at most i. A contract storage σ of $\mathtt{C_{lot}}(i)$ has, in addition to the mandatory attributes $\sigma.\mathsf{user}_L$, $\sigma.\mathsf{user}_R$, σ.cash and σ.locked (see Sec. 4.1), the attribute $\sigma.\mathsf{start} \in \mathbb{N}$, whose purpose it to store the construction round, the attribute $\sigma.\mathsf{com} \in \{0,1\}^*$, to store the commit value when submitted by $\sigma.\mathsf{user}_L$, and the attribute $\sigma.\mathsf{bit} \in \{0,1\}$ to store the secret bit when provided by $\sigma.\mathsf{user}_R$.

The contract code has one constructor $\mathtt{Init_{lot}}$ which generates the initial contract storage σ such that both $\sigma.\mathsf{cash}(\sigma.\mathsf{user}_L)$ and $\sigma.\mathsf{cash}(\sigma.\mathsf{user}_R)$ are equal to 1 (each user deposits 1 coin). The contract functions are: (i) Com which, if executed by $\sigma.\mathsf{user}_L$ on input c, stores c in σ.com, (ii) Reveal which, if executed by $\sigma.\mathsf{user}_R$ on input r_B, stores r_B in σ.bit, (iii) Open which allows $\sigma.\mathsf{user}_L$ to open the commitment stored in σ.com and pays out 2 coins to the winner, and (iv) Punish which allows a party to unlock coins from the contract instance in case the other party misbehaves. See the full version of this paper [13] for a formal definition of $\mathtt{C_{lot}}(i)$.

Assume now that parties Alice and Bob have a virtual state channel γ created via the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$, where $\mathtt{C_{lot}}(i) \in C$. If Alice wants to play the lottery using γ, she first locally executes the constructor $\mathtt{Init_{lot}}$ to obtain the initial contract storage σ. Then she sends the message (update, γ.id, *cid*, σ, $\mathtt{C_{lot}}(i)$) to $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ for some contact instance identifier *cid* never used before. The ideal functionality informs Bob about Alice's intention to play by sending the message (update−requested, γ.id, *cid*, σ, $\mathtt{C_{lot}}(i)$) to him. If Bob agrees with playing the game, he sends the reply (update−reply, *ok*, γ.id, *cid*). Alice and Bob can now start playing in a way we describe below (let τ_0 be the current round).

[14]Note that this is the case also for execution of smart contracts on the blockchain.

(1) **Commit:** In round τ_0 Alice locally chooses a random bit $r_A \in \{0,1\}$ and a random string $s \in \{0,1\}^\lambda$, where λ is the security parameter, locally computes the commit value c using the randomness s. Then she submits c by sending the message (execute, γ.id, *cid*, Com, c) to the ideal functionality.
(2) **Reveal:** If before round τ_0 + TimeExe(i) Bob receives a message from the ideal functionality that Alice committed to her secret bit, Bob locally chooses a random bit $r_B \in \{0,1\}$ which he submits by sending the message (execute, γ.id, *cid*, Reveal, r_B) to the ideal functionality. Otherwise, in round τ_0 + TimeExe(i), he sends the message (execute, γ.id, *cid*, Punish, $\perp$) to the ideal functionality to unlock all coins from the lottery contract by which he punishes Alice for her misbehavior.
(3) **Open:** If before round $\tau_0 + 2 \cdot$ TimeExe(i) Alice receives a message from the ideal functionality that Bob reveled his secret bit r_B, she opens her commitment by sending the message (execute, γ.id, *cid*, Open, (r_A, s)). Otherwise, in round $\tau_0 + 2 \cdot$ TimeExe(i), she sends the message (execute, γ.id, *cid*, Punish, $\perp$) to unlock all coins from the lottery contract by which she punishes Bob for his misbehavior.
(4) **Finalize:** If until round $\tau_0 + 3 \cdot$ TimeExe(i) Bob did not receive a message from the ideal functionality that Alice opened her commitment, Bob sends the message (execute, γ.id, *cid*, Punish, $\perp$) to the ideal functionality to unlock all coins from the lottery contract and finalize the game.

5 AN OVERVIEW OF OUR APPROACH

In this section we provide a high level idea of the modular design of our protocol realizing the state channel ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ (the main ideas behind our construction were already presented in Sec. 2).

Ledger state channels. Our first step is to define an ideal functionality $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ which models the behavior of a concrete smart contract, which we call state channel contract. This contract allows two parties to open, maintain and close a ledger state channel on the blockchain. The ideal functionality is parametrized by the set of contract codes C whose instances can be opened in the ledger state channels created via this ideal functionality. The ideal functionality $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ together with the ledger functionality $\widehat{\mathcal{L}}$ can be implemented by a cryptocurrency which supports such state channel contracts on its blockchain (a candidate cryptocurrency would be, e.g., Ethereum). We use this contract ideal functionality to design a protocol $\Pi(1, C)$ which realizes the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(1, C)$ (i.e. the protocol for ledger state channels).

The outline of the protocol is given in Appx. A.1. The formal description of the protocol $\Pi(1, C)$ and the ideal functionality $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ can be found in the full version of this paper [13], where we also prove that the protocol $\Pi(1, C)$ emulates the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(1, C)$ in the $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ hybrid world. This statement is formalized by the following theorem.

Theorem 5.1. *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. The protocol*

$\Pi(1, C)$ working in $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$-hybrid model emulates the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(1, C)$ against environments from class $\mathcal{E}_{res}$ for every set of contract codes C and every $\Delta \in \mathbb{N}$.

Virtual state channels. As already mentioned in Sec. 2, our technique allows to create virtual state channels of arbitrary length, via using the state channel functionality recursively. By this we mean that a protocol for constructing state channels of length up to i will work in a model with access to an ideal functionality for constructing state channels of length up to $i - 1$. More formally, for every $i > 1$ we construct a protocol $\Pi(i, C)$ realizing the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ in the $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i - 1, C')$-hybrid world. Here C' is a set of contract codes defined as $C' := C \cup \mathsf{VSCC}_i(C)$, where $\mathsf{VSCC}_i(C)$ is a contract code, which we call the virtual state channel contract, that allows to create a virtual state channel of length i in which contract instance with code from the set C can be opened. Thus importantly, the hybrid ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i - 1, C')$ allows to create state channels that can serve as subchannels of a virtual channel of length i.

Very briefly, the hybrid ideal functionality is used by parties of the protocol $\Pi(i, C)$ as follows. If a party receives a message regarding a state channel of length $j < i$, then it simply forwards this message to the hybrid ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i - 1, C')$. The more interesting case is when a party receives a message regarding a virtual state channel γ of length exactly i. Then it uses the hybrid ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i - 1, C')$ to make changes in the subchannels of the virtual state channels γ.

See Appx. A.2 for the outline of the protocol $\Pi(i, C)$. In the full version of this paper [13] we provide the formal description of the protocol $\Pi(i, C)$, the code of the virtual state channel contract $\mathsf{VSCC}_i(C)$, and we prove the following theorem.

THEOREM 5.2. *Suppose the underlying signature scheme is existentially unforgeable against chosen message attacks. The protocol $\Pi(i, C)$ working in $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i - 1, \mathsf{VSCC}_i(C) \cup C)$-hybrid model emulates the ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ against environments from class $\mathcal{E}_{res}$ for every set of contract codes C, every $i > 1$ and every $\Delta \in \mathbb{N}$.*

By applying the composition recursively, we get a construction of a protocol realizing $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i, C)$ in the $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(\widehat{C})$-hybrid model, where $\widehat{C}$ is a result of applying the "$C := C \cup \mathsf{VSCC}_i(C)$" equation i times recursively. See Fig. 5 for an example for $i = 3$.

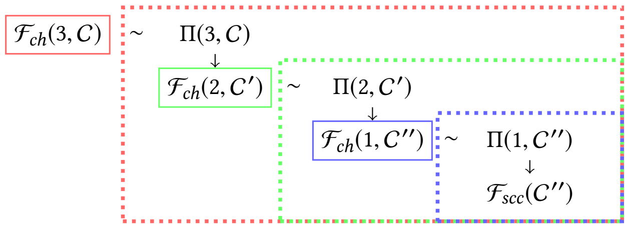


Figure 5: Our modular approach. Above $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}$, $\mathcal{F}_{scc} := \mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}$, $C' := C \cup \mathsf{VSCC}_3(C)$ and $C'' := C' \cup \mathsf{VSCC}_2(C')$.

6 CONCLUSION

We showed how to build general state channel networks, i.e., state channels of arbitrary length in which arbitrary contracts can be opened and executed off-chain. Our modular approach allows for a recursive construction of state channels (i.e. a virtual channel of length i is build on top of *two* state channels of length $\lceil i/2 \rceil$) which significantly simplifies the description of our construction. All protocols were proven to be secure in the global UC model and their optimistic time complexity is independent of the channel length. In the pessimistic case when malicious parties try to delay the protocol execution as much as possible, the time complexity of our construction is linear in channel length. We did not aim to optimize the pessimistic time complexity of our protocols since this would make their description even more complex. More fine grained timing analysis, which would reduce the constants in the pessimistic time complexity, and corresponding optimization of our state channel protocol would be highly recommended before the implementation. Another question is whether virtual state channels with time complexity independent of the channel length could be designed (for example using techniques from [24]).

Incentivizing intermediaries. An important practical question is why would a party want to become an intermediary of a virtual state channel. Although our construction does guarantee that an honest intermediary will never lose coins, the fact that an intermediary has to lock coins for the entire lifetime of the virtual channel makes this role unattractive. This problem can be solved by adding the concept of *service fees* to our construction. Let us sketch how this could be done: both Alice and Bob would lock some additional coins in the VSCC_i contract instance each of them opens in their channel with Ingrid during the virtual state channel creation. More precisely, in order to create a virtual state channel γ, Alice would lock $\gamma.\text{cash}(A)$ + serviceFee coins in the channel α she has with Ingrid and Bob would lock $\gamma.\text{cash}(B)$ + serviceFee coins in the channel β he has with Ingrid. During the closure of γ (assuming that it was successfully created), the service fee would be unlocked from the VSCC contract instances in favor of Ingrid in both channels α and β.

Suitable contract codes – a cautionary note. We would like to point out one subtle issue, that users of future real-life implementations need to be aware of. As discussed in Sec. 3.2, the security guarantees provided to the end-users of a state channel are strongly dependent on the code of the contract instance that is opened in the state channel (in other words: our system is only as secure as the contract that the user run in the channel). In principle, this is the same as in case of the standard contracts on the ledger, however there are several additional aspects that have to be taken into account when designing contract codes for state channels. Recall that all coins that are locked in a contract instance when the underlying virtual state channel is closed are assigned to the intermediary of the channel. Therefore, it is important that a contract instance is terminated by any end-user before the validity of the underlying virtual state channel expires. Another important point to keep in mind is that although our construction guarantees that end-user of a state channel can execute a contract instance in any round and on any contract function, it might take (in the pessimistic case) up to

TimeExeReq(i) rounds before the other party is notified about the execution and TimeExe(i) rounds before the execution takes place (where i denotes the length of the state channel). Thus, compared to the contract deployment directly on the blockchain, the notification and execution delay might be longer.

ACKNOWLEDGMENTS

This work has been supported by the Polish National Science Centre grant 2014/13/B/ST6/03540 and by the German Research Foundation (DFG) as part of project S7 within the CRC 1119 CROSSING and by the Emmy Noether Program FA 1320/1-1.

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

REFERENCES

[1] 2017. Update from the Raiden team on development progress, announcement of raidEX. (Feb. 2017). https://tinyurl.com/z2snp9e.
[2] 2018. Bitcoin Wiki: Payment Channels. (2018). https://en.bitcoin.it/wiki/Payment_channels.
[3] 2018. Bitcoin Wiki: Scalability. (last visited May 2018). https://en.bitcoin.it/wiki/Nanopayments.
[4] 2018. Counterfactual. (2018). https://counterfactual.com/.
[5] Ian Allison. 2016. Ethereum's Vitalik Buterin explains how state channels address privacy and scalability. (July 2016).
[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 443–458. https://doi.org/10.1109/SP.2014.35
[7] Iddo Bentov and Ranjit Kumaresan. 2014. How to Use Bitcoin to Design Fair Protocols. In *Advances in Cryptology – CRYPTO 2014, Part II (Lecture Notes in Computer Science)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8617. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 421–439. https://doi.org/10.1007/978-3-662-44381-1_24
[8] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. 2017. Instantaneous Decentralized Poker. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 410–440.
[9] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Las Vegas, NV, USA, 136–145.
[10] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2007. Universally Composable Security with Global Setup. In *TCC 2007: 4th Theory of Cryptography Conference (Lecture Notes in Computer Science)*, Salil P. Vadhan (Ed.), Vol. 4392. Springer, Heidelberg, Germany, Amsterdam, The Netherlands, 61–85.
[11] Christian Decker and Roger Wattenhofer. 2015. *A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels*. Springer International Publishing, Cham, 3–18. https://doi.org/10.1007/978-3-319-21741-3_1
[12] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2017. Perun: Virtual Payment Hubs over Cryptographic Currencies. (2017), 635 pages. http://eprint.iacr.org/2017/635 conference version accepted to the 40th IEEE Symposium on Security and Privacy (IEEE S&P) 2019.
[13] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. 2018. Foundations of State Channel Networks. Cryptology ePrint Archive, Report 2018/320. (2018). https://eprint.iacr.org/2018/320 Full version of this paper.
[14] Oded Goldreich. 2006. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA.
[15] Dennis Hofheinz and Joern Mueller-Quade. 2004. A Synchronous Model for Multi-Party Computation and the Incompleteness of Oblivious Transfer. Cryptology ePrint Archive, Report 2004/016. (2004). http://eprint.iacr.org/2004/016.
[16] Yael Tauman Kalai, Yehuda Lindell, and Manoj Prabhakaran. 2007. Concurrent Composition of Secure Protocols in the Timing Model. *Journal of Cryptology* 20, 4 (Oct. 2007), 431–492.
[17] Jonathan Katz and Yehuda Lindell. 2007. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC.
[18] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally Composable Synchronous Computation. In *TCC 2013: 10th Theory of Cryptography Conference (Lecture Notes in Computer Science)*, Amit Sahai (Ed.), Vol. 7785. Springer, Heidelberg, Germany, Tokyo, Japan, 477–498. https://doi.org/10.1007/978-3-642-36594-2_27
[19] Rami Khalil and Arthur Gervais. 2017. Revive: Rebalancing Off-Blockchain Payment Networks. In *ACM CCS 17: 24th Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 439–453.
[20] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter R. Pietzuch, and Emin Gün Sirer. 2017. Teechain: Scalable Blockchain Payments using Trusted Execution Environments. *CoRR* abs/1707.05454 (2017). arXiv:1707.05454 http://arxiv.org/abs/1707.05454
[21] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. 2017. Concurrency and Privacy with Payment-Channel Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 455–471.
[22] Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. 2018. Pisa: Arbitration Outsourcing for State Channels. *IACR Cryptology ePrint Archive* 2018 (2018), 582. https://eprint.iacr.org/2018/582
[23] Silvio Micali and Ronald L. Rivest. 2002. Micropayments Revisited. In *Topics in Cryptology – CT-RSA 2002 (Lecture Notes in Computer Science)*, Bart Preneel (Ed.), Vol. 2271. Springer, Heidelberg, Germany, San Jose, CA, USA, 149–163.
[24] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment Channels that Go Faster than Lightning. *CoRR* abs/1702.05812 (2017). http://arxiv.org/abs/1702.05812
[25] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. (2009). http://bitcoin.org/bitcoin.pdf.
[26] Jesper Buus Nielsen. 2003. On Protocol Security in the Cryptographic Model. (2003).
[27] Olaoluwa Osuntokun. 2018. Hardening Lightning. BPASE. (2018). https://cyber.stanford.edu/sites/default/files/hardening_lightning_updated.pdf
[28] Rafael Pass and Abhi Shelat. 2015. Micropayments for Decentralized Currencies. In *ACM CCS 15: 22nd Conference on Computer and Communications Security*, Indrajit Ray, Ninghui Li, and Christopher Kruegel: (Eds.). ACM Press, Denver, CO, USA, 207–218.
[29] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. (Jan. 2016). Draft version 0.5.9.2, available at https://lightning.network/lightning-network-paper.pdf.
[30] Ronald L. Rivest. 1997. Electronic Lottery Tickets as Micropayments. In *FC'97: 1st International Conference on Financial Cryptography (Lecture Notes in Computer Science)*, Rafael Hirschfeld (Ed.), Vol. 1318. Springer, Heidelberg, Germany, Anguilla, British West Indies, 307–314.
[31] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. 2017. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. *CoRR* abs/1709.05748 (2017). arXiv:1709.05748 http://arxiv.org/abs/1709.05748
[32] David Siegel. 2016. Understanding The DAO Attack. (Jun 2016). CoinDesk, http://www.coindesk.com/understanding-dao-hack-journalists/.
[33] David Wheeler. 1997. Transactions Using Bets. In *Proceedings of the International Workshop on Security Protocols*. Springer-Verlag, London, UK, UK, 89–92. http://dl.acm.org/citation.cfm?id=647214.720381
[34] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. (2014). http://gavwood.com/paper.pdf.

A OUTLINE OF THE PROTOCOL

In this section we give an outline of our ledger and virtual state channel construction. The complete formal specification appears in the full version of this paper [13].

A.1 Protocol for Ledger State Channels

We start with a high-level description of the sub-protocols for opening, maintaining and closing ledger state channels.

Create a ledger state channel. In order to create a new ledger state channel γ, the environment sends the message (create, γ) to both parties in γ.end−users. The protocol for creating a ledger state channel works at a high level as follows.

The initiating party γ.Alice requests construction of the state channel contract by sending the message (construct, γ) to the ideal functionality $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$. The ideal functionality locks the required amount of coins in her account on the ledger and sends the message (initializing, γ) to both parties. If party γ.Bob confirms the initialization by sending the message (confirm, γ), the ideal functionality

$\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ outputs (created, γ). In case γ.Bob does not confirm, the ledger state channel cannot be created and the initiating party γ.Alice has the option to refund the coins that were locked in her account on the ledger during the first step. Creation of a ledger state channels takes up to 2Δ rounds since it requires two interactions with the hybrid ideal functionality modeling a smart contract on the ledger. In case the ledger state channel is not created but γ.Alice's coins were locked in the first phase of the ledger state channel creation, she can receive them back latest after 3Δ rounds.

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

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

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

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

Update a contract instance in a ledger state channel. An update of the storage of a contract instance in a ledger state channel starts when the environment sends the message (update, id, cid, $\tilde{\sigma}$, C) to the initiating party $P \in \gamma$.end–users and works as follows. The initiating party P signs the new contract instance with increased version number (i.e. if ν is the contract instance version stored by P until now, then the new contract instance version ν' will be such that ν'.version = ν.version + 1). Party P then sends her signature on this value to the party $Q := \gamma$.other–party(P). The other party verifies the signature and informs the environment that the update was requested. If the environment confirms the update, the party Q signs the updated contract version and sends the signature to P. In this optimistic case, the update takes 2 rounds.

Let us discuss how parties behave in case the environment does not confirm the update. If Q simply aborts in this situation, P does not know if the update failed because Q is malicious or because the environment did not confirm the update. Therefore, Q has to inform P about the failure. This is, however, still not sufficient. Note that Q holds P's signature of the updated contract instance version. If Q is corrupt, he can register the updated contract instance on the ledger at any later point. Thus, party Q in order to convince P that he is not malicious, signs the *original* contract instance ν but with version number increased by 2 (i.e. the contract instance ν^* signed by Q is such that ν^*.storage = ν.storage, ν^*.code = ν.code but ν^*.version = ν.version + 2). Party Q then sends the signature to party P. Note that since ν^*.storage = ν.storage, party P does not need to send her signature on ν^* back to Q.

If P does not receive a valid signature on either the updated contract instance version or the original contract instance with increased version number from Q, it is clear that Q is malicious and therefore P initiates the registration of the contract instance on the ledger by calling the procedure Register(P, id, cid). Note that Q can still register the updated contract instance (the one that was signed by P). But importantly, after at most $2 + 3\Delta$ rounds it will be clear to both parties what the current contract instance version is.

Execute a contract instance in a ledger state channel. In order to execute a contact instance stored in a ledger state channel γ, the environment sends the message (execute, γ.id, cid, f, z) to the initiating party $P \in \gamma$.end–users. The parameter cid points to the contract instance, f is the contact function and z are additional input values for f. For $P = \gamma$.Alice the protocol works as follows. If the parties never registered the contract instance with identifier cid, then γ.Alice first tries to execute the contract instance "peacefully". This means that she locally executes f on the contract version she stores in $\Gamma^{\gamma.\mathsf{Alice}}$, signs the new contract instance and sends the signature to γ.Bob. Party γ.Bob also executes f locally on his own version of the contract instance stored in $\Gamma^{\gamma.\mathsf{Bob}}$ and thereafter verifies γ.Alice's signature. If the signature is valid, γ.Bob immediately confirms the execution by sending his signature on the new contract instance to party γ.Alice.

A technical challenge occurs when both parties want to peacefully execute the same contract instance in the same round τ since it becomes unclear what is the new contract instance. This can be resolved be having designated rounds for each party.

In case the contract instance with identifier cid has already been registered on the ledger or the peaceful execution fails, the initiating party executes the contract instance "forcefully". By this we mean that γ.Alice first initiates registration of the contract instance by calling the procedure Register(γ.Alice, id, cid) if it was not done before, and then instructs the hybrid ideal functionality $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ to execute the contract instance. The Register procedure can take up to 3Δ rounds and the contract instance execution on the ledger can take up to Δ rounds. Thus, pessimistic time complexity of the execution protocol is equal to $4\Delta + 5$ rounds.

Close a ledger state channel. In order to close a ledger state channel with identifier *id* by party $P \in \gamma$.end−users, the environment sends the message (close, *id*) to the initiating party P. Before a ledger state channel can be closed, the end-users of the ledger state channel have the chance to register all the contract instances that they have constructed off-chain. Thus, the initiating party P first (in parallel) registers all the contract instances which have been updated/peacefully executed but not registered at the ledger yet. This takes up to 3Δ rounds. Next, P asks the ideal functionality $\mathcal{F}_{scc}^{\widehat{\mathcal{L}}(\Delta)}(C)$ representing the state channel contract on the ledger to close the ledger state channel. Within Δ rounds, the ideal functionality informs both parties that the ledger state channel is being closed and gives the other end-user of the ledger state channel time 3Δ to register contract instances that were not registered by P. If after 3Δ rounds all registered contract instances are terminated, the ideal functionality adds γ.cash(γ.Alice) coins to γ.Alice's account on the ledger, and γ.cash(γ.Bob) coins to γ.Bob's account on the ledger, deletes the ledger state channel from its channel space and within Δ rounds informs both parties that the ledger state channel was successfully closed. Hence, in the pessimistic case closing can take up to 8Δ rounds.

A.2 Protocol for Virtual State Channels

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

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

The protocol $\Pi(i, C)$ has to handle messages about state channels of any length j, where $1 \leq j \leq i$. If a party P of the protocol $\Pi(i, C)$ is instructed by the environment to create, update, execute or close a state channel of length $1 \leq j < i$, the party forwards this message (possibly with some pre-processing) to the hybrid ideal functionality $\mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i - 1, \mathsf{VSCC}_i(C) \cup C)$, and hence we focus on the protocol for virtual state channels of length exactly i.

Create a virtual state channel. To create the virtual state channel γ of length i in which contract instances with code from set C can be constructed, the environment sends a message (create, γ) to all three parties γ.Alice, γ.Bob and γ.Ingrid in the same round τ_0. The creation of γ then works as follows.

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

To ensure that at the end of the protocol two honest users γ.Alice and γ.Bob can conclude whether the virtual state channel γ was successfully created, there is one additional technicality in our protocol. Notice that if γ.Ingrid is honest, once γ.Alice receives a confirmation that her update request of α was successfully competed, she can conclude that the virtual state channel is created. However, we cannot assume that γ.Ingrid is honest. Hence, to guarantee that when both γ.Alice and γ.Bob are honest they agree on whether γ was opened, they exchange confirmation messages at the end of the protocol. To conclude, if creation of a virtual state channel is successful, both end-users output (created, γ) to the environment after 3 rounds.

We emphasize that creating a virtual state channel runs in constant time – independent of the ledger processing time Δ and length of the virtual state channel. This is in contrast to the *ledger* state channels with require always 2Δ time for creation.

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

Let $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ be the contract instance with code $\mathsf{VSCC}_i(C)$ stored in the state channel γ.subchan(γ.Alice) and $cid_B := \gamma.\mathsf{Bob}||\gamma.\mathsf{id}$ the contract instance stored in γ.subchan(γ.Bob). The initiating party (assume for now that it is γ.Alice) first executes cid_A on the function $\mathtt{RegisterInstance}_i^C$ with input parameters (cid, ν^A), where ν^A is γ.Alice's current off-chain contract instance version. Notice that this execution is in a state channel of length strictly less than i and hence will be handled by the trusted hybrid ideal functionality $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathsf{VSCC}_i(C) \cup C)$. The contract function $\mathtt{RegisterInstance}_i^C$ is defined in such a way that it first verifies the validity of γ.Alice's contract instance version, and if all checks pass, it stores (cid, ν^A) together with a time-stamp in the auxiliary attribute preRegistered.

The intermediary γ.Ingrid upon receiving the information about the execution of cid_A on the function $\mathtt{RegisterInstance}_i^C$ with input parameters (cid, ν^A) can now symmetrically request execution of cid_B on $\mathtt{RegisterInstance}_i^C$ with input (cid, ν^A). We emphasize that γ.Ingrid only needs the information that cid_A is being executed and does not need to wait to start the execution until the execution of cid_A is completed.

Once γ.Bob is notified about the execution request of cid_B on $\mathtt{RegisterInstance}_i^C$ with input parameters (cid, ν^A), he immediately submits ν^B, his own off-chain contract instance version, by executing cid_B on the contract function $\mathtt{RegisterInstance}_i^C$ with input parameters (cid, ν^B). If γ.Bob's version of the contract instance with identifier cid was submitted in time and is valid, the contract function $\mathtt{RegisterInstance}_i^C$ compares the two submitted versions of cid and stores the one with higher version number in the attribute cspace(cid). Otherwise, ν^A will be considered as the registered one. Note that once honest γ.Bob learns about ν^A and submits ν^B, he knows whose contract instance version will be registered in cid_B. Thus, he can mark (id, cid) as registered in Γ_{aux}^B and update his channel space accordingly without waiting for the execution of cid_B to be completed. We emphasize that there is no particular order in which parties can register state and our protocol can handle all possible variants.

Once γ.Alice receives the information about γ.Bob's version of the contract instance, she already knows whose contract instance version will be registered in cid_A. Thus, analogously to γ.Bob, she can mark (id, cid) as registered Γ_{aux}^A and update her channel space accordingly without waiting for the execution of cid_A to be completed. If γ.Alice does not receive any information about γ.Bob's version until certain round (because γ.Bob is corrupt and did not reveal his version to γ.Ingrid or because γ.Ingrid is corrupt and did not execute cid_A with γ.Bob's version in time), she can conclude that ν^A will be the registered contract instance version in cid_A and hence mark (id, cid) as registered in Γ_{aux}^A.

To conclude, the registration procedure of a virtual state channel of length i can take up to $\mathrm{TimeReg}(i) := 4 \cdot \mathrm{TimeExeReq}(\lceil i/2 \rceil)$ rounds. This follows by inspection of the functionality $\mathcal{F}_{ch}$ and our assumption that both subchannels have length at most $\lceil i/2 \rceil$.

Update a contract instance in a virtual state channel. As long as both end-users of a virtual state channel follow the protocol, they can update a contract instance exactly the same way as if it would be a ledger state channel. The differences between updates in a ledger state channel and in a virtual state channel appears only when end-users of the state channel run into dispute, i.e., when the parties run the contract instance registration procedure, which was defined above. The pessimistic time complexity of updating a virtual state channel of length i is equal to $\mathrm{TimeReg}(i) + 2$.

Execute a contract instance in a virtual state channel. In order to execute a contract instance in a virtual state channel γ with identifier id, the environment sends a message (execute, id, cid, f, z) to one of the end-users of the virtual state channel. Let us assume for now that this party is γ.Alice and let τ_0 be the round when she received the message from the environment. The party γ.Alice first tries to execute the contract instance "peacefully", exactly as if γ would be a ledger state channel (see description above). In case the peaceful execution fails, γ.Alice needs to register the contract instance cid by calling the subproderure $\mathtt{Register}_i(\gamma.\mathsf{Alice}, id, cid)$ and execute the contract instance "forcefully" via the intermediary γ.Ingrid. Since the intermediary is not trusted, execution must be performed by executing the contract instances with code $\mathsf{VSCC}_i(C)$ stored in the underlying subchannels of γ (recall that the contract instance in the subchannel γ.subchan(γ.Alice) is stored under the identifier $cid_A := \gamma.\mathsf{Alice}||\gamma.\mathsf{id}$ and the contract instance in the state channel γ.subchan(γ.Bob) is stored under the identifier $cid_B := \gamma.\mathsf{Bob}||\gamma.\mathsf{id}$). Since both subchannels are state channels of length strictly less than i, the execution of their contract instances is handled by recursion via the trusted hybrid ideal functionality $\mathcal{F}_{ch} := \mathcal{F}_{ch}^{\widehat{\mathcal{L}}(\Delta)}(i-1, \mathsf{VSCC}_i(C) \cup C)$.

The first attempt to design the force execution protocol would be to let γ.Alice execute cid_A on the function $\mathtt{ExecuteInstance}_i^C$ with parameters $param = (cid, \gamma.\mathsf{Alice}, \tau_0, f, z, s_A)$, where s_A is γ.Alice's signature on the tuple $(cid, \gamma.\mathsf{Alice}, \tau_0, f, z)$. The contract function $\mathtt{ExecuteInstance}_i^C$ would be defined such that it verifies the execution request (for example, checks that γ.Alice's signature is valid, etc.) and then executes the contract instance with identifier cid. After successful execution of cid_A, γ.Ingrid symmetrically executes cid_B on the same contract function $\mathtt{ExecuteInstance}_i^C$ with the same input parameters $param$.

Unfortunately, this straightforward solution does not work since we allow parties to interact fully concurrently. To illustrate the problem consider an example where while the execution between γ.Alice and γ.Ingrid is running, γ.Bob also wants to forcefully execute the contract instance with identifier cid on different inputs. This means that before γ.Ingrid has time to execute cid_B on γ.Alice's request, γ.Bob executes cid_B on the function $\mathtt{ExecuteInstance}_i^C$ with his own parameters $param' = (cid, \gamma.\mathsf{Bob}, \tau_0', f', z', s_B)$. Consequently, the order of executions of the contract instance cid is different in cid_A and cid_B. Depending on the contract code of cid, this asymmetry may lead to γ.Ingrid losing money.

Therefore, the contract function $\mathtt{ExecuteInstance}_i^C$ is defined in such a way that it verifies the validity of the submitted contract instance execution request as before and if all checks pass, then it only stores the execution request in an auxiliary attribute toExecute. In other words, during the lifetime of the virtual state channel γ, the contract instances cid_A and cid_B in the subchannels of γ only collect information about the force executions of cid but they do not perform any of them. All the internal executions are postponed until the virtual state channel is being closed and the contract instance cid_A and cid_B are being terminated. If γ.Ingrid behaves honestly and always mimics requests from cid_A to cid_B and vice versa, then, after the last accepted force execution, the set toExecute stored in cid_A is equal to the set toExecute stored in cid_B. This implies that

cid_A and cid_B terminate with the same money distribution and hence γ.Ingrid can not lose money.

Since internal executions are postponed until the virtual state channel closure, end-users of the virtual state channel cannot wait until they learn the result of force execution of *cid* from the hybrid ideal functionality $\mathcal{F}_{ch}$ but they have to derive it themselves. They proceed as follows. Party γ.Alice, after initiating the force execution, waits for $2 \cdot \text{TimeExe}(\lceil i/2 \rceil) + 5$ rounds to be sure that γ.Bob did not initiate force execution of *cid* that should be performed before her own force execution request. After the waiting is over, she performs her execution of *cid* locally and outputs the result to the environment. The other party acts similarly. Once γ.Bob learns about γ.Alice's force execution, he checks if he has some pending execution requests that should take place before the one requested by γ.Alice. If this is the case then he locally executes them first. Thereafter, he locally executes the newly requested by γ.Alice and outputs the result to the environment.

Execution of a contract instance in a virtual state channel of length i as described above would take in the pessimistic case up to $5 + \text{TimeReg}(i) + 2 \cdot \text{TimeExe}(\lceil i/2 \rceil)$ rounds.

Unfortunately, it turns out that the above time complexity is polynomial in the length of the virtual state channel. In order to achieve *linear* pessimistic time complexity, we make two important observations which optimize our protocol. First note that party γ.Ingrid does not need to wait until execution of cid_A is completed in order to initiate the execution of cid_B. Similarly, γ.Bob does not need to wait until execution of cid_B is completed to locally execute the contract instance *cid* and output the result to the environment. This reduces the time complexity to $5 + \text{TimeReg}(i) + 2 \cdot \text{TimeExe Req}(\lceil i/2 \rceil)$ rounds. This time complexity can further be improved by running the registration subprocedure in parallel with the force execution phase to:

$$\text{TimeExe}(i) := 5 + 4 \cdot \text{TimeExeReq}(\lceil i/2 \rceil). \tag{1}$$

A more detailed analysis of how this improved time complexity is achieved is given in the full version of this paper.

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

For virtual state channels the situation is different. We require that the closing procedure of a virtual state channel γ always starts in round γ.validity and always results in γ being closed. In other words, both contract instances with code $\texttt{VSCC}_i(C)$ that were opened in the subchannels of γ must be terminated. This ensures that virtual state channels can never infinitely block closure of ledger state channels. Let us now explain how the protocol "Close a virtual state channel" works.

In round γ.validity both end-users of the virtual state channel start registering the contract instance if it has been created in the virtual state channel γ but has never been registered before. Thereafter, γ.Alice requests execution of the contract instance $cid_A := \gamma.\text{Alice}||\gamma.\text{id}$ stored in the subchannel γ.subchan(γ.Alice), on the contact function $\texttt{Close}_i^C$. In case γ.Alice is corrupt and does not request execution of cid_A on the function $\texttt{Close}_i^C$, γ.Ingrid can request it herself after certain time has passed. We proceed similar for γ.Bob.

The contract function $\texttt{Close}_i^C$ is defined in such a way that it first (if necessary) finalizes registration of a contract instance. This step is needed in case one party initiated registration of a contract instance with identifier *cid* but the other party did not react. As a next step the function $\texttt{Close}_i^C$ internally executes all the force execution requests stored in toExecute.

Let us now discuss what happens if there exists a registered contract instance *cid* which is however not terminated (the amount of locked coins is not equal to zero). The first idea would be to let $\texttt{Close}_i^C$ ignore the contract instance. However, this would lead to the problem that the intermediary of the virtual state channel, γ.Ingrid, loses money (because some money may still be locked in the contract) without ever having the chance to react to virtual state channel closing. Instead, the contract function $\texttt{Close}_i^C$ gives all the locked coins in the contract instance to the intermediary. This implies that end-users of a virtual channels are responsible to open a contract instance only if they are certain that they can terminate it before the channel validity expires since otherwise they will lose money.

Finally, the contract function verifies that the current value of the attribute cash is non-negative for both users and that the amount of coins that were originally invested into the virtual state channel is equal to the current amount of coins in the virtual state channel. If this is the case, $\texttt{Close}_i^C$ unlocks for each user the current amount of coins it holds in the channel contract. If one of the users have negative balance in the virtual state channel or the amount of invested coins is not equal to the current amount of coins, then any trading that happened between the end-users of γ is reverted by $\texttt{Close}_i^C$. This again guarantees that γ.Ingrid cannot lose money when γ.Alice and γ.Bob are malicious.

The time complexity of closing a virtual state channel of length i can be computed as $2 \cdot \text{TimeExeReq}(\lceil i/2 \rceil) + 2 \cdot \text{TimeExe}(\lceil i/2 \rceil)$. This follows from the simple observation that in case parties need to register a contract instance before closing the channel, both end-users should initiate the registration procedure in the same round (i.e. $\texttt{Register}_i(\gamma.\text{Alice}, id, cid)$ and $\texttt{Register}_i(\gamma.\text{Bob}, id, cid)$ are run in parallel) which reduces the time complexity of the registration phase.

Let us briefly explain one additional technicality. Recall that in case γ.Ingrid is corrupt, it can happen that the contract instances with code $\texttt{VSCC}_i(C)$ are opened in the subchannels of γ although the virtual state channel γ was not successfully created. This in particular means that the coins needed to create γ are locked in the subchannels and can be unlocked only after round γ.validity by executing the contact function $\texttt{Close}_i^C$.