Generalized Bitcoin-Compatible Channels

***Abstract*—The widespread adoption of decentralized cryptocurrencies, such as Bitcoin or Ethereum, is currently hindered by their inherently limited transaction rate. One of the most prominent proposals to tackle this scalability issue are *payment channels* which allow mutually distrusted parties to exchange an arbitrary number of payments in the form of off-chain authenticated messages while posting only a limited number of transactions onto the blockchain. Specifically, two transactions suffice, unless a dispute between these parties occurs, in which case more on-chain transactions are required to restore the correct balance. Unfortunately, popular constructions, such as the Lightning network for Bitcoin, suffer from heavy communication complexity both off-chain and on-chain in case of dispute. Concretely, the communication overhead grows exponentially and linearly, respectively, in the number of applications that run in the channel.**

In this work, we introduce and formalize the notion of *generalized channels* for Bitcoin-like cryptocurrencies. Generalized channels significantly extend the concept of payment channels so as to perform off-chain any operation supported by the underlying blockchain. Besides the gain in expressiveness, generalized channels outperform state-of-the-art payment channel constructions in efficiency, reducing the communication complexity and the on-chain footprint in case of disputes to linear and constant, respectively.

We provide a cryptographic instantiation of generalized channels that is compatible with Bitcoin, leveraging *adaptor signatures* – a cryptographic primitive already used in the cryptocurrency literature but formalized as a standalone primitive in this work for the first time. We formally prove the security of our construction in the Universal Composability framework. Furthermore, we conduct an experimental evaluation, demonstrating the expressiveness and performance of generalized channels when used as building blocks for popular off-chain applications, such as channel splitting and payment-channel networks.

I. Introduction

Blockchain technologies have spurred increasing interest over the last years, enabling secure payments, and more generally computations, among mutually distrustful parties. At the core of them lies a decentralized consensus protocol, which establishes and maintains a distributed ledger that stores all transactions. An inherent drawback of their dezentralized approach, however, is its poor transaction throughput, which, for instance, in the case of Bitcoin is around ten transactions per second, three orders of magnitude lower than credit card networks. This severely limits the widespread adoption of blockchain technologies and their potential to cater for a large user base.

Among several recent proposals to tackle this scalability problem [16, 29, 3], payment channels [4] have emerged as one of the most promising and widely deployed solutions (see, e.g., the Lightning network [27] in Bitcoin and the Raiden Network [28] in Ethereum). A payment channel enables an arbitrary number of payments between users while committing only two transactions onto the blockchain, without compromising on security. In a bit more detail, focusing on Bitcoin and its Unspent Transaction Output (UTXO) model, a payment channel between Alice and Bob is first created by a single on-chain transaction that locks bitcoins into a multi-signature address controlled by both users. They can then pay to each other (possibly many times) by exchanging authenticated off-chain messages that represent an update of their share of coins in the multi-signature address. The payment channel is finally closed when a user decides to submit the last authenticated distribution of coins to the blockchain.

Payment channels serve as a building block for a variety of off-chain services, which aim at offering better connectivity, as establishing a different channel for each possible payee would be cumbersome and financially unsustainable (e.g., one would have to lock coins in each channel). For instance, payment channel networks (PCNs) [27, 23] link payment channels to each other, forming a graph through which payments can be routed along multi-hop paths. Payment channel hubs [18] offer a different connectivity solution, with an untrusted third party acting as proxy between any pair of users and yet unable to compromise either the security or the privacy of transactions. Payment channels also serve as building block for atomic swaps [17] where two users can atomically trade their coins.

On a technical level, the key challenge in designing Bitcoin-compatible payment channels is how to revoke old states: as previously mentioned, the distribution of coins in the channel changes over time due to payments between the end-points, which poses the problem of how to prevent one of the two parties from publishing an old, financially more advantageous, state on the blockchain. The state-of-the-art approach, put forward in the Lightning Network, is based on a punishment mechanism which allows the cheated party to claim all coins from the channel. The current cryptographic realization, however, suffers from two central drawbacks, which undermine its ability to cater for the growing number of applications built on top of them:

State duplication. In order to protect parties from each other, the state of the channel is duplicated. Each copy has a built in punishment mechanism for one of the parties. State updates have to be propagated on both duplicates, leading to cumbersome and expensive protocols. This issue becomes even more pressing when users decide to use the same channel for multiple applications (e.g., [11]), which they need to update independently in parallel. For that, the channel is recursively split into *sub-channels* (with each additional application requiring a further sub-channel splitting), each of which again contains two copies of the state to faithfully point out the misbehaving user. This unfortunately makes

the number of transactions ruling the distribution of coins to grow exponentially in k where k is the number of off-chain applications built on top of each other.

Output-based revocation. Since each (sub-)channel can be used to process multiple applications, each corresponding (sub-)channel state may contain multiple outputs defining how coins can be spent. The current punishment approach follows a "punish-per-output" pattern which means that if an old state appears on the blockchain, the cheated party has to claim money from each output of the state separately, leading to a possibly large number of on-chain transactions, linear in the number of applications represented in the revoked state.

This state-of-affairs leads to the following question: *is it possible to design a simple, efficient, and expressive Bitcoin-compatible payment channel, i.e., one that reduces the channel state to be stored by each party as well as the revocation overhead on the blockchain to a minimum, while supporting a large class of off-chain applications?*

a) Our contributions: In this work, we give a positive answer to the above question, designing and formalizing a novel payment channel scheme. Concretely, our contributions are summarized below.

- We introduce the notion of *generalized channels*, which generalizes payment channels to support any application expressed in the scripting language of the underlying blockchain, thereby enhancing their expressiveness. One may view a generalized channel as a 2-party ledger for off-chain operations offering the same functionality as the underlying blockchain. Hence, our construction extends the concept offered by state channels for Ethereum [24, 10] to cryptocurrencies with limited scirpting capabilities such as Bitcoin.
- We design a novel revocation mechanism, which relies on adaptor signatures [22], to avoid state duplication thereby reducing the number of states (and thus the communication complexity) of off-chain protocols from exponential to linear in the number of applications. Additionally, our revocation mechanism based on *punish-then-split* enables revocation through a single output (and thus a single transaction), thereby reducing the overhead on the blockchain from linear to constant.
- We provide a cryptographic instantiation of generalized channels based on ECDSA-based adaptor signatures as well as Schnorr-based adaptor signatures. Our cryptographic instantiation is thus supported by virtually all cryptocurrencies, including Bitcoin.
- We formalize the security and functionality of generalized channels as ideal functionalities in the Universal Composability (UC) framework [6] and prove the security of our construction in the UC framework. While doing so, we provide the first (game-based) security definition for adaptor signatures. We beleive that the formalization of adaptor signatures is of independent interest.
- We implemented our protocols and conducted an experimental evaluation, demonstrating how to use generalized channels to implement popular off-chain applications, like payment channel network and channel splitting, and characterizing the gains in performance as compared to the construction from the Lightning Network.

b) Organization: The rest of the paper is organized as follows. In Section II, we introduce the required background and overview our solution. In Section III, we formally define generalized channels and our model in the UC framework. In Section IV, we introduce the adaptor signatures and the (game-based) security definitions. In Section V, we describe our cryptographic instantiation of generalized channels. In Section VI, we show how generalized channels can be leveraged to build different off-chain applications. In Section VII, we analyze the performance of generalized channels. We conclude in Section VIII.

II. Background and Solution Overview

A. Background and notation

Throughout this work, we use the following notation for *attribute tuples*. Let T be a tuple of values which we call attributes. Each attribute in T is identified using a unique keyword attr and referred to as $T.\mathsf{attr}$.

a) Outputs and transactions: In this work, we focus on blockchains based on the Unspent Transaction Output (UTXO) model, such as Bitcoin. In the UTXO model, coins are held in *outputs*. Formally, an output θ is an attribute tuple $(\theta.\mathsf{cash}, \theta.\varphi)$, where $\theta.\mathsf{cash}$ denotes the amount of coins associated to the output and $\theta.\varphi$ denotes the conditions that need to be satisfied to spend the output. The condition $\theta.\varphi$ can contain any script supported by the considered blockchain. We say that a user P controls or owns an output θ if $\theta.\varphi$ contains a signature verification w.r.t. the public key of P.

A *transaction* transfers coins across outputs meaning that it maps (possibly multiple) existing outputs to a list of new outputs. To avoid confusion, the existing outputs that fund the transactions are called *transaction inputs*. Formally, a transaction tx is an attribute tuple consisting of the following attributes $(\mathsf{tx.txid}, \mathsf{tx.Input}, \mathsf{tx.Output}, \mathsf{tx.Witness})$. The attribute $\mathsf{tx.txid} \in \{0,1\}^*$ is the identifier of the transaction and is calculated as $\mathsf{tx.txid} := \mathcal{H}([\mathsf{tx}])$, where $\mathcal{H}$ is a hash function modeled as a random oracle and $[\mathsf{tx}]$ is the *body of the transaction* defined as $[\mathsf{tx}] := (\mathsf{tx.Input}, \mathsf{tx.Output})$. The attribute $\mathsf{tx.Input}$ is a vector of strings identifying all inputs of tx. The attribute $\mathsf{tx.Output} = (\theta_1, \ldots, \theta_n)$ is a vector of new outputs. Finally, the attribute $\mathsf{tx.Witness} \in \{0,1\}^*$ contains the witness of the transaction allowing to spend the inputs.

To ease the readability, we illustrate the transaction flows throughout the paper in the form of charts. Let us here define and explain the symbols and notation used in the charts. A transaction is represented as a rectangle with rounded corners. Doubled edge rectangles represent transactions published on the blockchain, while single edge rectangles are transactions that could be published on the blockchain but they are not (yet). Transaction outputs are depicted as a box inside the transaction. The value of the output is written inside the output box and the output condition is written above the arrow coming from the output.

Conditions of transaction outputs might be fairly complex and hence it would be cumbersome to spell them out above the arrows. Instead, for conditions that are used frequently, we define the following abbreviated notation. If the output script contains (among other conditions) signature verification w.r.t. some public keys $pk_1, \ldots, pk_n$, we write all the public keys *below* the arrow and the remaining conditions *above* the arrow. Hence, information below the arrow denotes "who owns the output" and information above denotes "additional spending conditions". If the output script contains a check of whether a given witness hashes to a predefined hash value h, we express this by simply writing the hash value h *above* the arrow. Moreover, if the output script contains a relative time lock, i.e. a condition that is satisfied if and only if at least t rounds passed since the transaction was published, we write the string "$+t$" *above* the arrow. Finally, if the output script φ can be parsed as $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ for some $n \in \mathbb{N}$, we add a diamond shape to the corresponding transaction output. Each of the subconditions φ_i is then written above a separate arrow. An example is given in Fig. 1.

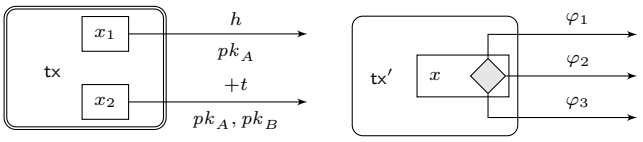

Fig. 1. (Left) Transaction $\mathtt{tx}$ is published on the blockchain. The output of value x_1 can be spent by a transaction containing a preimage of h and signed w.r.t. pk_A. The output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t rounds passed since $\mathtt{tx}$ was accepted by the blockchain. (Right) Transaction $\mathtt{tx}'$ is not published on the ledger. Its only output, which is of value x, can be spent by a transaction whose witness satisfies the output condition $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

b) Payment channels: A payment channel enables several transactions between two users without committing every single transaction to the blockchain. The cornerstone of payment channels is depositing coins into an output controlled by two users, who then authorize new deposit balances in a peer-to-peer fashion while having the guarantee that all coins are refunded at a mutually agreed time. In a bit more detail, a payment channel has three operations: *open*, *update* and *close*.

First, assume that Alice and Bob want to create a payment channel with an initial deposit of x_A and x_B coins respectively. For that, Alice and Bob agree on a *funding transaction* (that we denote by $\mathtt{TX_f}$) that sets as inputs two outputs controlled by Alice and Bob holding x_A and x_B coins respectively and transfers them to an output controlled by both Alice and Bob. When $\mathtt{TX_f}$ is added to the blockchain, the payment channel between Alice and Bob is effectively open.

Assume now that Alice wants to pay $\alpha \leq x_A$ coins to Bob. For that, they create a new *commit transaction* $\mathtt{TX_c}$ representing the commitment from both users to the new channel state. The commit transaction spends the output of $\mathtt{TX_f}$ into two new outputs: (i) one holding $x_A - \alpha$ coins controlled by Alice; and (ii) the other holding $x_B + \alpha$ coins controlled by Bob. Finally, parties exchange the signatures on the commit transaction. At this point, Alice (resp. Bob) could add $\mathtt{TX_c}$ to the blockchain. Instead, they keep it locally in their memory and overwrite it when they agree on another commitment transaction $\overline{\mathtt{TX}}_\mathtt{c}$ representing a newer channel state. This, however, leads to several commitment transactions that can possibly be added to the blockchain. Since all of them are spending the same output, only one can be accepted by the blockchain. Since it is impossible to prevent a malicious user from publishing an old commit transaction, payment channels require a mechanism that punishes such behavior.

Lightning Network [27], the state-of-the-art payment channel network for Bitcoin, implements such mechanism by introducing two commitment transactions per channel update, each of which contains a punishment mechanism for one of the users. In more detail (see also Fig. 2), the output of $\mathtt{TX}_\mathtt{c}^\mathtt{A}$ representing Alice's balance in the channel has a special condition. Namely, it can be spent by Bob if he presents a preimage of a hash value h_A or by Alice if certain number of rounds passed since the transaction was published. During a channel update, Alice chooses a value r_A, called the *revocation secret*, and presents the hash $h_A := \mathcal{H}(r_A)$ to Bob. Knowing h_A, Bob can create and sign the commit transaction $\mathtt{TX}_\mathtt{c}^\mathtt{A}$ with the built in punishment for Alice (analogously for Bob and $\mathtt{TX}_\mathtt{c}^\mathtt{B}$). During the next channel update, parties first commit to the new state by creating and signing $\overline{\mathtt{TX}}_\mathtt{c}^\mathtt{A}$ and $\overline{\mathtt{TX}}_\mathtt{c}^\mathtt{B}$, and then *revoke* the old state by sending the revocation secrets to each other thereby enabling the punishment mechanism. If a malicious Alice now publishes the old commit transaction $\mathtt{TX}_\mathtt{c}^\mathtt{A}$, Bob can spent both of its outputs and hence claim all coins locked in the channel.

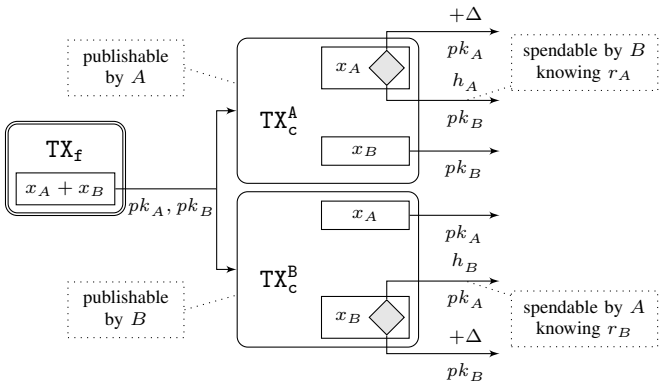

Fig. 2. A Lightning style payment channel where A has x_A coins and B has x_B coins. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. The value of Δ upper bounds the time needed to publish a transaction on a blockchain.

B. Solution Overview

The goal of our work is to extend the idea of payment channels such that parties can perform essentially *any* operation that they could do on-chain and not only pay to each other. Technically, this means that we want the commit transaction to contain arbitrary many outputs with arbitrary conditions (as long as they are supported by the underlying blockchain). The main question we need to answer when

designing such channels, which we call *generalized channels*, is how to implement the revocation mechanism in this case.

a) Revocation per update: The first idea would be to extend the revocation mechanism of payment channels explained above such that *each output* of $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ contains a punishment mechanism for Alice (analogously for Bob). This approach is taken by the Lightning network [27] whose channels support (multiple) hash-time lock payments.[1] While this solution works, it has several disadvantages: (i) If one party, say Alice, cheats and publishes an old commit transaction $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$, Bob has to spend all outputs of $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ in order to punish Alice for her misbehavior. Although Bob could group some of them within a single transaction (up to the transaction size limit), he might be forced to publish multiple transactions thereby paying high transaction fees in order to punish Alice; (ii) such revocation mechanism requires a high on-chain footprint not only for $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$, but also for Bob to get the coins from the outputs.

Our goal is to design a punishment mechanism whose on-chain footprint and potential transaction fees are independent of the channel state, i.e., the number and type of outputs in the channel. To this end, we propose the *punish-then-split* mechanism which separates the punishment mechanism from the actual outputs. In a nutshell, the commit transaction $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ has now only one output dedicated to the punishment mechanism which can be spent (i) immediately by Bob, if he proves that the commit transaction was old (i.e. he knows the revocation secret r_A of Alice); or (ii) after certain number of rounds by a *split transaction* $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ controlled by both parties and containing all the outputs of the channel (i.e. representing the channel state). Hence, if $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ is published on the blockchain, Bob has some time to punish Alice if the commit transaction was old. If Bob does not use this option, any of the parties can publish the split transaction $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ representing the channel state. Analogously for $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$.

b) One commit transactions per channel update: Another drawback of the Lightning style revocation mechanism is the need for two commit transactions for the same channel state. While this is not an issue for simple payment channels, for generalized channels it might cause redundancy in terms of communication and computational costs. This comes from the fact that generalized channels support arbitrary output conditions and hence can be used as a source of funding for, e.g., another off-chain channel as we discuss later in this work (see Section VI). Such off-chain channel would, however, have to "exist" twice. Once considering $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ being eventually published on-chain and once considering $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$. Therefore, a natural goal is to construct generalized channels that require only one commit transaction.

The naive approach to design such a single commit transaction $\mathtt{TX}_{\mathtt{c}}$ would be to simply "merge" the transactions $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ and $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$. Such $\mathtt{TX}_{\mathtt{c}}$ could be spent (i) by Alice if she knows Bob's revocation secret; (ii) by Bob if he knows Alice's revocation secret or (iii) by the split transaction $\mathtt{TX}_{\mathtt{s}}$ representing the channels state after some time. This simple proposal does not work, however, since it allows parties to misuse the punishment mechanism as follows. A malicious Alice could publish an old commit transaction $\mathtt{TX}_{\mathtt{c}}$ and since she knows Bob's revocation secret, she could immediately try to punish Bob. In order to prevent such undue punishment of honest Bob, we need to make sure that Alice can use the punishment mechanism only if it was Bob publishing $\mathtt{TX}_{\mathtt{c}}$. In other words, our punishment mechanism built in $\mathtt{TX}_{\mathtt{c}}$ requires the punishing party to prove that (i) this commit transaction was old and (ii) this commit transaction was published by the other party.

The main idea how to implement the requirement (ii), is to force the party publishing $\mathtt{TX}_{\mathtt{c}}$ to reveal some secret, which we call *publishing secret*, that could be used by the other party as a proof. We achieve this by leveraging the concept of an *adaptor signature scheme* – a signature scheme that allows a party to *pre-sign* a message w.r.t. some statement Y of a hard relation.[2] Such pre-signature can be adapted into a valid signature by anyone knowing a witness for the statement Y. Moreover, knowing both the pre-signature and the adapted full signature, it is possible to extract a witness for Y. In our context, adaptor signatures allow parties of the generalized channels to express the following: "I give you my *pre-signature* on $\mathtt{TX}_{\mathtt{c}}$ which you can turn into a valid signature and publish $\mathtt{TX}_{\mathtt{c}}$ on-chain. But this will reveal your publishing secret to me."

To conclude, our solution, depicted in Fig. 3, requires only one commit transaction $\mathtt{TX}_{\mathtt{c}}$ per update. The commit transaction has one output that can be spent (i) by Alice if she knows Bob's revocation secret r_B *and publishing secret* y_B; (ii) by Bob if he knows Alice's revocation secret r_A *and publishing secret* y_A or (iii) by the split transaction $\mathtt{TX}_{\mathtt{s}}$ representing the channels state after some time. In the depicted construction we assume that that the statement/witness pairs used for the adaptor signature scheme are public/secret keys of the blockchain signature scheme. Hence, testing if a party knows a publishing secret can be done by requiring a valid signature w.r.t. this public key. Let use emphasize that public/secret keys can also be used for the revocation mechanism instead of the hash/preimage pairs. This is actually preferable since the punishment output script will only consist of signatures and thereby we require less complex scripting language.

III. Generalized channels

In this section we formalize the notion of generalized channels and their functionality. We first introduce some basic notation and our security model which closely follows the previous works on off-chain channels [8, 9, 10].

A. Notation and security model

To formally model the security of our channel construction, we use a synchronous version of the global UC framework (GUC) [7] which extends the standard UC framework [6] by allowing for a global setup. Monetary transactions are handled by a global ledger $\mathcal{L}(\Delta, \Sigma)$, where Δ is an upper bound on

[1] Hash-time lock payment is a conditional payment that is performed conditioned on the receiver presenting a preimage of a hash function before a certain time.

[2] On a high level, a statement/witness relation is hard, if given a statement Y is it computationally hard to find a witness y.

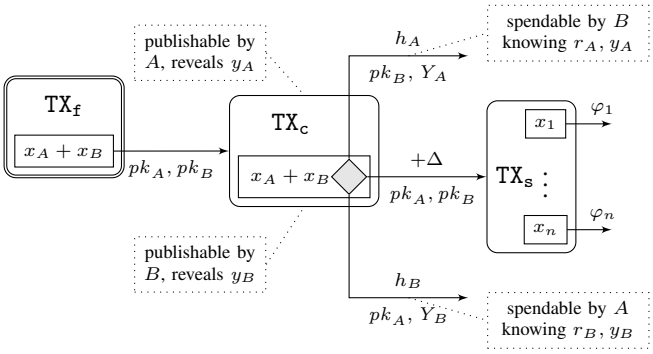

Fig. 3. A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. The value of Δ upper bounds the time needed to publish a transaction on a blockchain. If $\mathtt{TX_c}$ is published by A, publishing secret y_A corresponding to Y_A is revealed. If $\mathtt{TX_c}$ is published by B, publishing secret y_B corresponding to Y_B is revealed.

the blockchain delay (number of rounds it takes to publish a transaction) and Σ defines the signature scheme used by the blockchain. We denote by $\mathcal{P}$ the set of all parties participating in the protocols considered in this work. For more details about our model, we refer the reader to Appendix B.

We define a *generalized channel* γ as an attribute tuple $(\gamma.\mathsf{id}, \gamma.\mathsf{users}, \gamma.\mathsf{cash}, \gamma.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\gamma.\mathsf{users} \in \mathcal{P}^2$ defines the identities of the channel users, $\gamma.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ represents the total amount of coins locked in this channel and $\gamma.\mathsf{st} = (\theta_1, \ldots, \theta_n)$ is the state of the channel composed of a list of *outputs*. Each output θ_i has two attributes: the value $\theta_i.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ representing the amount of coins and the function $\theta_i.\varphi\colon \{0,1\}^* \to \{0,1\}$ representing the spending condition of the output. For convenience, we define a function $\gamma.\mathsf{otherParty}\colon \gamma.\mathsf{users} \to \gamma.\mathsf{users}$ defined as $\gamma.\mathsf{otherParty}(P) := Q$ for $\gamma.\mathsf{users} = \{P, Q\}$.

B. Ideal Functionality

We capture the desired functionality of a generalized channel protocol as an ideal functionality $\mathcal{F}$ interacting with parties from the set $\mathcal{P}$, with the adversary $\mathcal{S}$ (called the simulator) and observes the global ledger functionality $\mathcal{L}$. In a bit more detail, if a party wants to perform an action (such as open a new channel), it sends a message to $\mathcal{F}$ who executes the action and informs the party about the result. The execution might leak information to the adversary who may also influence the execution. The possible leakage and influence are modelled via the interaction with $\mathcal{S}$. Finally, $\mathcal{F}$ can observe the global ledger and hence verify that a certain transaction appeared on-chain or that a given party owns certain amount of coins. The latter is done by checking if there exists an unspent output whose condition requires signature of (only) the given party P. We denote such script $\mathtt{One\text{–}Sig}_{pk_P}$.

As a first step towards defining our functionality, we identify the security and efficiency notions of interest that a generalized channel functionality should provide. To this end, we use the following terminology. A *successful* update/create means that both parties in $\gamma.\mathsf{users}$ output the message UPDATED/CREATED. In addition a state st is called *enforced* on the ledger if a transaction with this state appears on the ledger.

Consensus on creation: A channel γ can be successfully created if and only if both parties in the set $\gamma.\mathsf{users}$ agree.

Consensus on update: Parties in $\gamma.\mathsf{users}$ reach agreement on channel update acceptance or rejection after an a-priori bounded number of rounds (the bound may depend on the blockchain delay Δ). Moreover, a channel γ can be successfully updated if and only if both parties in the set $\gamma.\mathsf{users}$ agree with the update.

Instant finality with punish: If a channel γ is successfully updated to the state $\gamma.\mathsf{st}$ and this is the latest successful update, then an honest party $P \in \gamma.\mathsf{users}$ has the guarantee that either $\gamma.\mathsf{st}$ can be enforced on the ledger or P can enforce a state where she gets all $\gamma.\mathsf{cash}$ coins.

Optimistic update: If both parties in $\gamma.\mathsf{users}$ are honest, a successful update takes a constant number of rounds (independent of the blockchain delay Δ). In other words, if both parties are honest, channel update is performed without any blockchain interaction.

Having the guarantees identified above in mind, we now design our ideal functionality $\mathcal{F}$. We assume that $\mathcal{F}$ maintains a set Γ, where it stores created channels in their latest state and the corresponding funding transaction tx. We sometimes treat Γ as a function which on input id outputs (γ, tx) s.t. $\gamma.\mathsf{id} = id$ if such channel exists and $\perp$ otherwise. To keep $\mathcal{F}$ generic, we parameterized it by two values T and k – both of which must be independent of the blockchain delay Δ. On a high level, the value T upper bounds the maximal number of consecutive off-chain communication rounds between channel users. Since different parts of the protocol might require different amount of communication rounds, the upper bound T might not be reached in all steps. For instance, channel creation might require more communication rounds than old state revocation. To this end, we give the power to the simulator to "speed-up" the process when possible. The parameter k defines the number of ways the channel state $\gamma.\mathsf{st}$ can be published on the ledger. As discussed in Section II, in this work we present a protocol realizing the functionality for $k = 1$ (see Fig. 3). Lightning style generalized channels (see Fig. 2) would be a candidate protocol for $k = 2$. Before we present $\mathcal{F}(T, k)$ formally, we discuss it on a high level and argue why it captures the aforementioned security and efficiency properties. In the text below, we abbreviate $\mathcal{F} := \mathcal{F}(T, k)$.

a) Create: If $\mathcal{F}$ receives a message of the form $(\mathtt{CREATE}, \gamma, tid_P)$ from both parties in $\gamma.\mathsf{users}$ within T rounds, it expects a channel funding transaction to appear on the ledger $\mathcal{L}$ within Δ rounds. Such transaction must spend both funding sources (defined by transaction identifiers tid_P, tid_Q) and containing one output of the value $\gamma.\mathsf{cash}$. If this is true, then $\mathcal{F}$ stores this transaction together with the channel γ in the set Γ and informs both parties about the successful channel creation via the message CREATED. Since a CREATE message is required from both parties, "consensus on creation" holds.

b) Update: The channel update is initiated by one of the parties P (called the *initiating party*) via a message $(\mathtt{UPDATE},$

$id, \vec{\theta}, t_{\mathsf{stp}}$). The parameter id identifies the channel to be updated, $\vec{\theta}$ represents the new channel state and t_{stp} denotes the number of rounds needed by the parties to setup off-chain objects (e.g. new channels or hash-time lock contracts) that are being built on top of the channel via this update request. The update is structured into two phases: (i) the prepare phase, and (ii) the revocation phase. Intuitively, the prepare phase models the fact that both parties first agree on the new channel state and get time to setup the off-chain objects on top of this new state. The revocation phase models the fact that an update is only completed once the two parties invalidate the previous channel state. We detail these two phases in the following.

The prepare phase starts when $\mathcal{F}$ receives a vector of transaction identifiers $\vec{tid} = (tid_1, \ldots, tid_k)$ from $\mathcal{S}$.[3] In the optimistic case it is completed within $3T + t_{\mathsf{stp}}$ rounds and ends when the initiating party P receives an UPDATE–OK message from $\mathcal{F}$. The setup phase can be aborted by both the initiating party P and the other party Q. In the ideal world this is achieved by P not sending the SETUP–OK and by Q not sending the UPDATE–OK message, respectively. This models two things. Firstly, the fact that Q might not agree with the proposed update and secondly, the fact that setting up off-chain objects might fail in which case parties want to abort the channel update. The abort may also result in a forceful closing of the channel via the subprocedure ForceClose (which we discuss further below). It happens when one of the parties has sufficient information to enforce the new state on-chain, while the other does not.

In order to complete the update, the revocation phase is executed. The functionality expects to receive the REVOKE message from both parties within $2T$ rounds, in which case it updates the channel state in $\Gamma(id)$ accordingly and informs both parties about the successful updated via the message UPDATED. If one of the messages does not arrive, the subprocedure ForceClose is called.

To conclude, the possibility for forceful closing guarantees the security property "consensus on update". Moreover, in case both parties are honest, the duration of an successful update is independent of the ledger delay Δ, hence the efficiency property "optimistic update" is satisfied.

c) Close: Any of the two parties can request closure of the channel via the message $(\texttt{CLOSE}, id)$, where id identifies the channel to be closed. In case both parties request closure within T rounds, *peaceful closure* is expected meaning that a transaction, spending the channel funding transaction and whose output corresponds to the latest channel state $\gamma.\mathsf{st}$, should appear on $\mathcal{L}$ within Δ rounds. In case only one of the parties requests closing, the functionality executes the ForceClose subprocedure in which case such transaction is supposed to appear on $\mathcal{L}$ within 3Δ rounds. In both cases, if the funding transaction is not spent before a certain round, an ERROR message is returned.

[3]For technical reasons, ideal functionality cannot sign transactions and thus it can also not prepare the transaction ids (which is the task of the simulator).

d) Punish: In order to guarantee "instant finality with punishments", parties continuously monitor the ledger and apply the punishment mechanism if misbehavior is detected. This is captured by the functionality in the part "Punish" which is executed at the end of each round. The functionality checks if a funding transaction of some channel was spent. If yes, then it expects one of the following to happen: (i) a punish transaction appears on $\mathcal{L}$ within Δ rounds, assigning $\gamma.\mathsf{cash}$ coins to the honest party $P \in \gamma.\mathsf{users}$; or (ii) a transaction whose output corresponds to the latest channel state $\gamma.\mathsf{st}$ appears on $\mathcal{L}$ within 2Δ rounds, meaning that the channel is peacefully or forcefully closed. If none of the above is true, ERROR is returned. Hence, under the condition that no ERROR was returned, the security property "instant finality with punish" is satisfied.

e) Simplified formal description: Since we do not aim to make any claims about privacy, we implicitly assume that every message that $\mathcal{F}$ receives/sends from/to a party is directly forwarded to $\mathcal{S}$. When $\mathcal{F}$ expects $\mathcal{S}$ to set certain values, such as the vector of tid's during the update process, and it does not do so, we implicitly assume that ERROR is returned. Moreover, we omit several natural checks that one would expect $\mathcal{F}$ to make. For example, messages with malformed or missing parameters should be ignored, channel instruction should be accepted only from channel users, etc. We formally define all those checks as a functionality wrapper $\mathcal{W}_{checks}$ in Appendix D. We use the following arrow notation in formal description below. If we write $m \overset{t}{\hookrightarrow} P$, we mean "send the message m to party P in round t." and if we write $m \overset{t}{\hookleftarrow} P$, we mean "receive a message m from party P in round t".

In summary, our functionality formally defined below satisfies the identified security and efficiency properties if no ERROR occurs. In case of an ERROR, all guarantees may be lost. Hence, we are interested only in those protocols realizing $\mathcal{F}$ that never output an ERROR.

Ideal Functionality $\mathcal{F}(T, k)$

We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Create

Upon $(\texttt{CREATE}, \gamma, tid_P) \overset{\tau_0}{\hookleftarrow} P$, let $\mathcal{S}$ define $T_1 \leq T$ and:

Both agreed: If already received $(\texttt{CREATE}, \gamma, tid_Q) \overset{\tau}{\hookleftarrow} Q$, where $\tau_0 - \tau \leq T_1$, wait if in round $\tau_1 \leq \tau + \Delta + T_1$ a transaction tx, with $\mathsf{tx}.\mathsf{Input} = (tid_P, tid_Q)$ and $\mathsf{tx}.\mathsf{Output} = (\gamma.\mathsf{cash}, \varphi)$, appears on the ledger $\mathcal{L}$. If yes, set $\Gamma(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx})$ and $(\texttt{CREATED}, \gamma.\mathsf{id}) \overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{users}$. Else stop.

Wait for Q: Else store the message and stop.

Update

Upon $(\texttt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$, let $\mathcal{S}$ define $T_1, T_2 \leq T$, parse $(\gamma, \mathsf{tx}) := \Gamma(id)$ and proceed as follows:

1) In round $\tau_1 \leq \tau_0 + T$, let $\mathcal{S}$ set $|\vec{tid}| = k$. Then $(\texttt{UPDATE–REQ}, id, \vec{\theta}, t_{\mathsf{stp}}, \vec{tid}) \overset{\tau_1}{\hookrightarrow} Q$ and $(\texttt{SETUP}, id, \vec{tid}) \overset{\tau_1}{\hookrightarrow} P$.
2) If $(\texttt{SETUP–OK}, id) \overset{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}}{\longleftarrow} P$, then $(\texttt{SETUP–OK}, id) \overset{\tau_2 + T_1}{\longrightarrow} Q$. Else stop.

3) If $(\texttt{UPDATE–OK}, id) \xleftarrow{\tau_2+T_1} Q$, then $(\texttt{UPDATE–OK}, id) \xrightarrow{\tau_2+2T_1} P$. Else distinguish:
   - If Q honest or if instructed by $\mathcal{S}$, stop (update rejected).
   - Else execute $\texttt{ForceClose}(id)$ and stop.
4) If $(\texttt{REVOKE}, id) \xleftarrow{\tau_2+2T_1} P$, $(\texttt{REVOKE–REQ}, id) \xrightarrow{\tau_2+2T_1+T_2} Q$. Else execute $\texttt{ForceClose}(id)$ and stop.
5) If $(\texttt{REVOKE}, id) \xleftarrow{\tau_2+2T_1+T_2} Q$, set $\gamma.\mathsf{st} = \vec{\theta}$ and $\Gamma(id) := (\gamma, \mathsf{tx})$. Then $(\texttt{UPDATED}, id, \vec{\theta}) \xrightarrow{\tau_2+2T_1+2T_2} \gamma.\mathsf{users}$ and stop. Else distinguish:
   - If Q honest, execute $\texttt{ForceClose}(id)$ and stop.
   - If Q corrupt, and wait for Δ rounds. If tx still unspent, then set $\vec{\theta}_{old} := \gamma.\mathsf{st}$, $\gamma.\mathsf{st} := \{\vec{\theta}_{old}, \vec{\theta}\}$ and $\Gamma(id) := (\gamma, \mathsf{tx})$. Execute $\texttt{ForceClose}(id)$ and stop.

Close

Upon $(\texttt{CLOSE}, id) \xleftarrow{\tau_0} P$, let $\mathcal{S}$ define $T_1 \leq T$ and distinguish:

Both agreed: If you received $(\texttt{CLOSE}, id) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T_1$, let $(\gamma, \mathsf{tx}) := \Gamma(id)$ and distinguish:
- If in round $\tau_1 \leq \tau + T_1 + \Delta$ a transaction tx', with $\mathsf{tx}'.\mathsf{Output} = \gamma.\mathsf{st}$ and $\mathsf{tx}'.\mathsf{Input} = \mathsf{tx}.\mathsf{txid}$, appears on $\mathcal{L}$, set $\Gamma(id) := (\bot, \mathsf{tx})$, $(\texttt{CLOSED}, id) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ and stop.
- If tx is still unspent in round $\tau + T_1 + \Delta$, output $(\texttt{ERROR}) \xrightarrow{\tau+T_1+\Delta} \gamma.\mathsf{users}$ and stop.

Wait for Q: Else wait for at most T_1 rounds to receive $(\texttt{CLOSE}, id) \xleftarrow{\tau \leq \tau_0+T_1} Q$ (in that case option "Both agreed" is executed). If such message is not received, execute $\texttt{ForceClose}(id)$ in round $\tau_0 + T_1$.

Punish (executed at the end of every round τ_0)

For each $(\gamma, \mathsf{tx}) \in \Gamma$ check if $\mathcal{L}$ contains tx' with $\mathsf{tx}'.\mathsf{Input} = \mathsf{tx}.\mathsf{txid}$. If yes, then distinguish:

Punish: For $P \in \gamma.\mathsf{users}$ honest, the following must hold: in round $\tau_1 \leq \tau_0 + \Delta$, a transaction tx'' with $\mathsf{tx}''.\mathsf{Input} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{Output} = (\gamma.\mathsf{cash}, \texttt{One–Sig}_{pk_P})$ appears on $\mathcal{L}$. Then $(\texttt{PUNISHED}, id) \xhookrightarrow{\tau_1} P$, set $\Gamma(id) := \bot$ and stop.

Close: Either $\Gamma(id) = (\bot, \mathsf{tx})$ before round $\tau_0 + \Delta$ (channels was peacefully closed) or in round $\tau_1 \leq \tau_0 + 2\Delta$ a transaction tx'', with $\mathsf{tx}''.\mathsf{Output} \in \gamma.\mathsf{st}$ and $\mathsf{tx}''.\mathsf{Input} = \mathsf{tx}'.\mathsf{txid}$, appears on $\mathcal{L}$ (channel is forcefully closed). In the latter case, set $\Gamma(id) := (\bot, \mathsf{tx})$ and $(\texttt{CLOSED}, id) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$.

Error: Otherwise $(\texttt{ERROR}) \xrightarrow{\tau_0+2\Delta} \gamma.\mathsf{users}$.

Subprocedure $\texttt{ForceClose}(id)$

Let τ_0 be the current round and $(\gamma, \mathsf{tx}) := \Gamma(id)$. If within Δ rounds tx is still an unspent transaction on $\mathcal{L}$, then $(\texttt{ERROR}) \xrightarrow{\tau_0+\Delta} \gamma.\mathsf{users}$ and stop. Else, latest in round $\tau_0 + 3\Delta$, $m \in \{\texttt{CLOSED}, \texttt{PUNISHED}, \texttt{ERROR}\}$ is output via Punish.

IV. Adaptor signatures

Before we proceed to the formalization of adaptor signatures, we recall some basic notation and definitions.

a) Digital signatures: A signature scheme consists of three algorithms $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, where: (i) $\mathsf{Gen}(1^n)$ gets as input 1^n (n is the security parameter) and outputs the secret and public keys (sk, pk); (ii) $\mathsf{Sign}_{sk}(m)$ gets as input the secret key sk and a message $m \in \{0,1\}^*$ and outputs the signature σ; and (iii) $\mathsf{Vrfy}_{pk}(m; \sigma)$ gets as input the public key pk, a message m and a signature σ, and outputs a bit b.

A signature scheme must fulfill correctness, i.e. it must hold that $\mathsf{Vrfy}_{pk}(m; \mathsf{Sign}_{sk}(m)) = 1$ for all messages m and valid key pairs (sk, pk). In this work, we use signature schemes that satisfy the notion of strong existential unforgeability under chosen message attack (or SUF–CMA). On a high level, SUF–CMA guarantees that an adversary on input the public key pk and with access to a signing oracle, cannot produce a *new* valid signature on any message m.

b) Hard relation: We next recall the definition of a hard relation R with statement/witness pairs (Y, y). Let L_R be the associated language defined as $L_R := \{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. We say that R is a *hard relation* if the following holds: (i) There exists a PPT sampling algorithm $\mathsf{GenR}(1^n)$ that on input 1^n outputs a statement/witness pair $(Y, y) \in R$; (ii) The relation is poly-time decidable; (iii) For all PPT $\mathcal{A}$ the probability of $\mathcal{A}$ on input Y outputting y is negligible.

c) Non-interactive Zero-Knowledge proof of knowledge: Finally we recall the definition of a non-interactive zero-knowledge proof of knowledge with online extractors as introduced in [12]. The online extractability property allows for extraction of a witness y for a statement Y from a proof π in the random oracle model and is useful for models where the rewinding proof technique is not allowed, such as UC. We will need this property in order to prove our ECDSA-based adaptor signature scheme secure. More formally, a pair (P, V) of PPT algorithms is called a non-interactive zero-knowledge proof of knowledge with an online extractor for a relation R, random oracle $\mathcal{H}$ and security parameter n if the following holds: (i) *Completeness*: For any $(Y, y) \in R$, it holds that $\mathsf{V}(Y, \mathsf{P}(Y, y)) = 1$ except with negligible probability; (ii) *Zero knowledge*: There exists a PPT simulator S, which on input Y can simulate the proof π for any $(Y, y) \in R$. (iii) *Online Extractor*: There exist a PPT online extractor K with access to the the sequence of queries to the random oracle and its answers, such that given (Y, π), the algorithm K can extract the witness y with $(Y, y) \in R$. It is shown in [12] how to instantiate such proof system.

A. Adaptor Signature Definition

Adaptor signatures have been introduced by the cryptocurrency community to tie together the authorization of a transaction and the leakage of a secret value. An adaptor signature scheme is essentially a two-step signing algorithm bound to a secret: first a partial signature is generated such that it can be completed only by a party knowing a certain secret, with the complete signature revealing such a secret. More precisely, we define an adaptor signature scheme with respect to a standard signature scheme Σ and a hard relation R. For any statement $Y \in L_R$, a signer holding a secret key is able to produce a *pre-signature* w.r.t. Y on any message m. Such pre-signature can be *adapted* into a valid signature on m if and only if the adaptor knows a witness for Y. Moreover, if such a valid signature is produced, it must be possible to extract a witness for Y given the pre-signature and the adapted signature.

Despite the fact that adaptor signatures have been used in previous works (e.g. [22] [13] [25]), none of these works has given a formal definition of the adaptor signature primitive and its security. As a consequence, in the following we provide the first formalization of adaptor signatures.

Definition 1 (Adaptor Signature Scheme)**.** An adaptor signature scheme wrt. a hard relation R and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ consists of four algorithms $\Xi_{R.\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ defined as:

$\mathsf{pSign}_{sk}(m, Y)$**:** is a PPT algorithm that on input a secret key sk, message $m \in \{0,1\}^*$ and statement $Y \in L_R$, outputs a pre-signature $\tilde{\sigma}$.

$\mathsf{pVrfy}_{pk}(m, Y; \tilde{\sigma})$**:** is a DPT algorithm that on input a public key pk, message $m \in \{0,1\}^*$, statement $Y \in L_R$ and pre-signature $\tilde{\sigma}$, outputs a bit b.

$\mathsf{Adapt}(\tilde{\sigma}, y)$**:** is a DPT algorithm that on input a pre-signature $\tilde{\sigma}$ and witness y, outputs a signature σ.

$\mathsf{Ext}(\sigma, \tilde{\sigma}, Y)$**:** is a DPT algorithm that on input a signature σ, pre-signature $\tilde{\sigma}$ and statement $Y \in L_R$, outputs a witness y such that $(Y, y) \in R$, or $\perp$.

In addition to the standard signature correctness, an adaptor signature scheme has to satisfy *pre-signature correctness*. Informally, it guarantees that an honestly generated pre-signature wrt. a statement $Y \in L_R$ is a valid pre-signature and can be completed into a valid signature from which a witness for Y can be extracted.

Definition 2 (Pre-signature correctness)**.** An adaptor signature scheme $\Xi_{R,\Sigma}$ satisfies pre-signature correctness if for every $n \in \mathbb{N}$, every message $m \in \{0,1\}^*$ and every statement/witness pair $(Y, y) \in R$, the following holds:

$$\Pr\left[\begin{array}{c|l} \mathsf{pVrfy}_{pk}(m, Y; \tilde{\sigma}) = 1 & (sk, pk) \leftarrow \mathsf{Gen}(1^n) \\ \wedge & \tilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y) \\ \mathsf{Vrfy}_{pk}(m; \sigma) = 1 & \sigma := \mathsf{Adapt}_{pk}(\tilde{\sigma}, y) \\ \wedge & y' := \mathsf{Ext}_{pk}(\sigma, \tilde{\sigma}, Y) \\ (Y, y') \in R & \end{array}\right] = 1.$$

We now define the security properties of an adaptor signature scheme. We begin with the notion of unforgeability which is similar to the definition of existential unforgeability under chosen message attacks but additionally requires that producing a forgery σ for some message m is hard even given a pre-signature on m w.r.t. a random statement $Y \in L_R$. Let us emphasize that allowing the adversary to learn a pre-signature on the forgery message m is crucial since for our applications unforgeability needs to hold even in case the adversary learns a pre-signature for m without knowing a corresponding witness for Y. We formally define the existential unforgeablity under chosen message attack for adaptor signature (aEUF–CMA security for short) in Definition 3.

Definition 3 (aEUF–CMA security)**.** An adaptor signature scheme $\Xi_{R,\Sigma}$ is aEUF–CMA secure if for every PPT adversary $\mathcal{A}$ there exists a negligible function ν such that: $\Pr[\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(n) = 1] \leq \nu(n)$, where the experiment $\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}$ is defined as follows:

$\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(n)$

$\mathcal{Q} := \emptyset$
$(sk, pk) \leftarrow \mathsf{Gen}(1^n)$
$m \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(pk)$
$(Y, y) \leftarrow \mathsf{GenR}(1^n)$
$\tilde{\sigma} \leftarrow \mathsf{pSign}_{pk}(m, Y)$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\tilde{\sigma}, Y)$
return $\left(m \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{pk}(m; \sigma)\right)$

$\mathcal{O}_{\mathrm{S}}(m)$

$\sigma \leftarrow \mathsf{Sign}_{sk}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return σ

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$\tilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\tilde{\sigma}$

As discussed above, adaptor signatures guarantee that a valid pre-signature w.r.t. Y can be completed to a valid signature if and only if the corresponding witness y for Y is known. An additional property that we will require is that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be completed into a valid signature using the witness y with $(Y, y) \in R$. Notice that this property is stronger than the pre-signature correctness property from Definition 2, since we require that even maliciously produced pre-signatures can always be completed into valid signatures. The next definition formalizes the above discussion.

Definition 4 (Pre-signature adaptability)**.** An adaptor signature scheme Ξ_R satisfies pre-signature adaptability if for any $n \in \mathbb{N}$, any message $m \in \{0,1\}^*$, any statement/witness pair $(Y, y) \in R$, any key pair $(sk, pk) \leftarrow \mathsf{Gen}(1^n)$ and any pre-signature $\tilde{\sigma} \leftarrow \{0,1\}^*$ with $\mathsf{pVrfy}_{pk}(m, Y; \tilde{\sigma}) = 1$, we have: $\Pr[\mathsf{Vrfy}_{pk}(m; \mathsf{Adapt}(\tilde{\sigma}, y)) = 1] = 1$.

The aEUF–CMA security together with the pre-signature adaptability ensure that a pre-signature for Y can be transferred into a valid signature if and only if the corresponding witness y is known. The last property that we are interested in is *witness extractability*. Informally, it guarantees that a valid signature/pre-signatue pair $(\sigma, \tilde{\sigma})$ for message/statement (m, Y) can be used to extract the corresponding witness y.

Definition 5 (Witness extractability)**.** An adaptor signature scheme Ξ_R is *witness extractable* if for every PPT adversary $\mathcal{A}$, there exists a negligible function ν such that the following holds: $\Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_R}(n) = 1] \leq \nu(n)$, where the experiment $\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}$ is defined as follows

$\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(n)$

$\mathcal{Q} := \emptyset$
$(sk, pk) \leftarrow \mathsf{Gen}(1^n)$
$(m, Y) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(pk)$
$\tilde{\sigma} \leftarrow \mathsf{pSign}_{pk}(m, Y)$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\tilde{\sigma})$
$y' := \mathsf{Ext}_{pk}(\sigma, \tilde{\sigma}, Y)$
return $(m \notin \mathcal{Q} \wedge (Y, y') \notin R$
$\quad \wedge \mathsf{Vrfy}_{pk}(m; \sigma))$

$\mathcal{O}_{\mathrm{S}}(m)$

$\sigma \leftarrow \mathsf{Sign}_{sk}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return σ

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$\tilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\tilde{\sigma}$

Let us stress that while the witness extractability experiment aWitExt looks fairly similar to the experiment aSigForge, there is one crucial difference; namely, the adversary is allowed to choose the forgery statement Y. Hence, we can assume that he

knows a witness for Y so he can generate a valid signature on the forgery message m. However, this is not sufficient to win the experiment. The adversary wins *only* if the valid signature does not reveal a witness for Y.

Definition 6 (Secure Adaptor Signature Scheme)**.** An adaptor signature scheme $\Xi_{R,\Sigma}$ is secure, if it is aEUF–CMA secure, pre-signature adaptable and witness extractable.

B. ECDSA-based Adaptor Signature

In this section we present an ECDSA-based adaptor signature construction that provably satisfies our security definition. The construction presented here is similar to the construction put forward by [25], however some modifications are needed for the security proof. Note that while we present here an ECDSA-based adaptor signature scheme, we additionally show a scheme based on Schnorr signatures including correctness and security proofs in the full version of this paper [14].

Recall the ECDSA signature scheme $\Sigma_{\mathsf{ECDSA}} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ for a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q. The key generation algorithm samples $x \leftarrow_{\$} \mathbb{Z}_q$ and outputs $g^x \in \mathbb{G}$ as the public key and x as the secret key. The signing algorithm on input a message $m \in \{0,1\}^*$, samples $k \leftarrow_{\$} \mathbb{Z}_q$ and computes $r := f(g^k)$ and $s := k^{-1}(\mathcal{H}(m) + rx)$, where $\mathcal{H}\colon \{0,1\}^* \to \mathbb{Z}_q$ is a hash function modelled as a random oracle and $f\colon \mathbb{G} \to \mathbb{Z}_q$.[4] The verification algorithm on input a message $m \in \{0,1\}^*$ and a signature (r,s) verifies that $f(g^{s^{-1}\mathcal{H}(m)} X^{s^{-1}r}) = r$. One of the properties of the ECDSA scheme is that if (r,s) is a valid signature for m, then so is $(r,-s)$. Consequently, Σ_{ECDSA} does not satisfy SUF–CMA security which we need in order to prove its security. In order to tackle this problem we build our adaptor signature from the *Positive ECDSA* scheme which guarantees that if (r,s) is a valid signature, then $|s| \leq (q-1)/2$. The positive ECDSA has already been used in other works such as [2, 21]. This slightly modified ECDSA scheme is not only assumed to be SUF–CMA but also prevents having two valid signatures for the same message after the signing process, which is useful in practice, e.g. for threshold signature schemes based on ECDSA. We note that the ECDSA verification accepts valid positive ECDSA signatures and hence these signatures can also be used in Bitcoin.

The adaptor signature scheme in [25] is presented with respect to a relation $R_g \subseteq \mathbb{G} \times \mathbb{Z}_q$ defined as $R_g := \{(Y,y) \mid Y = g^y\}$. The main idea of the construction is that a pre-signature (r,s) for a statement Y is computed by embedding Y into the r-component while keeping the s-component unchanged. This embedding however is rather involved in ECDSA, since the value s contains a product of k^{-1}, r and the secret key. More concretely, to compute the pre-signature for Y, the signer samples a random k and computes $K := Y^k$ and $\tilde{K} := g^k$. It then uses the first value to compute $r := f(K)$ and sets $s := k^{-1}(\mathcal{H}(m) + rx)$. To ensure that the signer uses the same value k in K and $\tilde{K}$, a zero-knowledge proof that $(\tilde{K}, K) \in L_Y := \{(\tilde{K}, K,) \mid \exists k \in \mathbb{Z}_q \text{ s.t. } g^k = \tilde{K} \wedge Y^k = K\}$ is attached to the pre-signature. We denote the prover of the NIZK as P_Y and the corresponding verifier as V_Y). The pre-signature adaptation is done by multiplying the value s with y^{-1}, where y is the corresponding witness for Y. This adjusts the randomness k used in s to ky, and hence matches with the r value.

Unfortunately, it is not clear how to prove security for the above scheme for the following reason: Ideally, we would like to reduce both the unforgeability and the witness extractability of the scheme to the strong unforgeability of positive ECDSA. More concretely, suppose there exists a PPT adversary $\mathcal{A}$ that wins the aSigForge (resp. aWitExt) experiment, then we design a PPT adversary (also called the simulator) $\mathcal{S}$ that breaks the SUF–CMA security. The main technical challenge in both reductions is that $\mathcal{S}$ has to answer queries (m, Y) to $\mathcal{O}_{\mathrm{pS}}$ by $\mathcal{A}$. This has to be done with access to the ECDSA signing oracle, but without knowledge of sk and the witness y. Thus, we need a method to "transform" full signatures into valid pre-signatures without knowing y, which seems to go against the aEUF–CMA-security (resp. witness extractability).

Due to this reason, we slightly modify this scheme. In particular, we modify the hard relation for which the adaptor signature is defined. Let R'_g consist of *pairs* (Y,π), where $Y \in L_{R_g}$ is as above, and π is a non-interactive zero-knowledge proof of knowledge that $Y \in L_{R_g}$. Formally, we define $R'_g := \{((Y,\pi), y) \mid Y = g^y \wedge \mathsf{V}_g(Y,\pi) = 1\}$ and denote by P_g the prover and by V_g the verifier of the proof system for L_{R_g}. Clearly, due to the soundness of the proof system, if R_g is a hard relation, then so is R'_g.

It might seem that we did not make it any easier for the reduction to learn a witness needed for creating pre-signatures. However, we exploit the fact that we are in the ROM and the reduction answers adversary's random oracle queries. Upon receiving a statement $I_Y := (Y,\pi)$ for which it must produce a valid pre-signature, it uses the random oracle query table to extract a witness from the proof π. Knowing the witness y and a signature (r,s), the reduction can compute $(r, s \cdot y)$ and execute the simulator of the NIZK_Y to produce a consistency proof π. This concludes the protocol description and the main proof idea. We refer the reader to the full version of this paper [14] for the detailed proof of the following theorem.

Theorem 1. *If the positive ECDSA signature scheme* Σ_{ECDSA} *is* SUF–CMA*-secure and* R'_g *is a hard relation,* $\Xi_{R'_g,\Sigma_{\mathsf{ECDSA}}}$ *from Fig. 4 is a secure adaptor signature scheme in the ROM.*

V. Protocol description

We now present a concrete protocol, which we denote Π, that realizes the channel functionality $\mathcal{F}(T,k)$ for $T = 3$ and $k = 1$. This is achieved by utilizing an adaptor signature scheme $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ for signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ used by the underlying ledger and a hard relation R. Our protocol consists of four subprotocols: Create, Update, Close and Punish. Here we explain the main ideas of the protocol. We refer the reader to Appendix C

[4] Since in ECDSA, the group $\mathbb{G}$ consists of elliptic curve points, the function f is typically defined as the projection to the x-coordinate.

$\mathsf{pSign}_{sk}(m, I_Y)$

$x := sk$
$(Y, \pi_Y) := I_Y$
$k \leftarrow_{\$} \mathbb{Z}_q$
$\tilde{K} := g^k, K := Y^k$
$r := f(K)$
$\tilde{s} := k^{-1}(\mathcal{H}(m) + rx)$
$\pi \leftarrow \mathsf{P}_Y((\tilde{K}, K), k)$
return $(r, \tilde{s}, K, \pi)$

$\mathsf{pVrfy}_{pk}(m, I_Y; \tilde{\sigma})$

$X := pk$
$(Y, \pi_Y) := I_Y$
$(r, \tilde{s}, K, \pi) := \tilde{\sigma}$
$u := \mathcal{H}(m) \cdot \tilde{s}^{-1}$
$v := r \cdot \tilde{s}^{-1}$
$K' := g^u X^v$
$b_r := (r = f(K))$
$b := \mathsf{V}_Y((K', K), \pi)$
return $(b_r \wedge b)$

$\mathsf{Ext}(\sigma, \tilde{\sigma}, I_Y)$

$(r, s) := \sigma$
$(\tilde{r}, \tilde{s}, K, \pi) := \tilde{\sigma}$
$y' := s^{-1} \cdot \tilde{s}$
if $(I_Y, y') \in R'_g$
then return y'
else return $\perp$

$\mathsf{Adapt}(\tilde{\sigma}, y)$

$(r, \tilde{s}, K, \pi) := \tilde{\sigma}$
$s := \tilde{s} \cdot y^{-1}$
return (r, s)

Fig. 4. ECDSA-based adaptor signature scheme.

for the formal descriptions. To simplify the exposition of the discussion below, we assume here that statement/witness pairs of R are valid key pairs of Σ.[5]

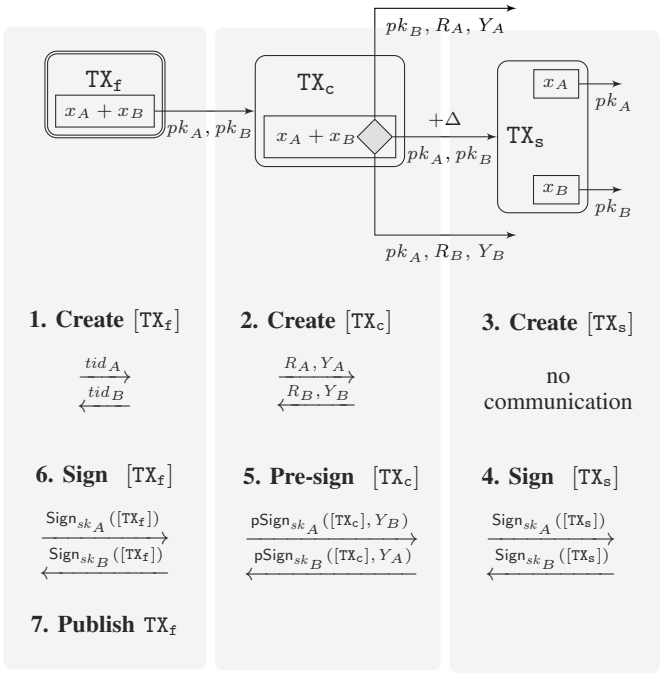

Fig. 5. Schematic description of the channel creation protocol.

a) Channel creation: In order to create a channel γ, users of the channel, let us denote them A and B, have to agree on the body of the funding transaction $[\mathtt{TX_f}]$, mutually commit to the first channel state defined by $\gamma.\mathsf{st} = ((x_A, \mathtt{One\text{–}Sig}_{pk_A}), (x_B, \mathtt{One\text{–}Sig}_{pk_B}))$, and sign and publish the funding transaction $\mathtt{TX_f}$ on the ledger. Once $\mathtt{TX_f}$ is published, the channel creation is completed. Looking at Fig. 5, one can summarize the creation process as a step-by-step creation of transaction bodies from left to right, and then step-by-step signature exchange on the transaction bodies from right to left. Let us elaborate on this in more detail.

Step 1: To prepare $[\mathtt{TX_f}]$, parties need to inform each other about their funding sources, i.e., exchange the transaction identifiers tid_A and tid_B. Each party can then locally create $[\mathtt{TX_f}]$ with $\{tid_A, tid_B\}$ as input and output requiring signature of both A and B. **Step 2:** Parties can now start committing to the initial channel state. To this end, each party $P \in \{A, B\}$ first generates a *revocation public/secret* as $(R_P, r_P) \leftarrow \mathsf{GenR}$ and *publishing public/secret* pair $(Y_P, y_P) \leftarrow \mathsf{GenR}$. The public values R_P and Y_P are sent to the other party. Each party can now locally generate the body of the commit transaction $[\mathtt{TX_c}]$ which spends $\mathtt{TX_f}$ and can be spent by a transaction satisfying one of the following conditions:

Punish A**:** It is correctly signed w.r.t. pk_B, Y_A, R_A;
Punish B**:** It is correctly signed w.r.t. pk_A, Y_B, R_B;
Channel state: It is correctly signed w.r.t. pk_A and pk_B, and at least Δ rounds have passed since $\mathtt{TX_c}$ was published.

Steps 3+4: Using the transaction identifier of $\mathtt{TX_c}$, parties can generate and exchange signatures on the body of the split transaction $\mathtt{TX_s}$ which spends $\mathtt{TX_c}$ and whose output is equal to $\gamma.\mathsf{st}$ (i.e., the coins that are owned by A and B). **Step 5:** Parties are now prepared to complete the committing phase by *pre-signing* the commit transaction to each other. This means that party A executes the pSign_{sk_A} on message $[\mathtt{TX_c}]$ and statement Y_B and sends pre-signature to B (analogously for B). **Step 6:** If valid pre-signatures are exchanged (validity is checked using the pVrfy algorithm), parties exchange signatures on the funding transaction and post it on the ledger which completes the channel creation.

The pre-signature adaptability property of Ξ guarantees that after a successful channel creation, each party P is able to adapt the pre-signature on $[\mathtt{TX_c}]$ of the other party by using the publishing secret value y_P (corresponding to Y_P), sign $[\mathtt{TX_c}]$ herself and publish this transaction on the ledger. Unless parties reveal the revocation secrets to each other, the only way to spend the posted $\mathtt{TX_c}$ is to publish $\mathtt{TX_s}$ representing the initial channel state.

b) Update: In order to update a created channel γ to a new state, represented by a vector of output scripts $\vec{\theta}$, parties have to (i) agree on the new commit and split transaction that represent the new state and (ii) invalidate the old commit transaction. Part (i) is very similar to the agreement on the initial commit and split transaction as described in detail in the creation protocol (**Steps 2-5**). There is one major difference coming from the fact that the new channel state $\vec{\theta}$ can contain outputs that fund other off-chain applications (such as sub-channels).[6] In order to setup those applications, the identifier of the new split transaction is needed. To this end, parties first prepare the commit (**Steps 2+3**) to learn the desired identifier which they output to the environment and wait for a confirmation that all applications were setup correctly. If this is the case, parties execute the second part of the committing

[5]Statements in the ECDSA-based adaptor signature can be mapped to public keys by dropping the second coordinate, i.e., the zero-knowledge proof.

[6]This is not the case during channel creation since we assume that the initial channel state consist of two accounts only.

phase (**Steps 4+5**). To realize part (ii), i.e., activate the punishment mechanism of the old commit transaction, parties simply exchange the revocation secrets corresponding to the previous commit transaction which completes the update.

The question is what happens if one party misbehaves during the update meaning that it stops communicating or sends malformed messages. As long as none of the parties pre-signed the new commit transaction, i.e. before **Step 5**, misbehavior simply implies update failure. A more problematic case is when the misbehavior occurs after at least one of the parties pre-signed the new commit transaction. There are multiple situations to be considered, for example, when one party pre-signs the new commit but the other does not; or when one party revokes the old commit and the other does not. In each of those situations, an honest party ends up in a hybrid state when the update is neither rejected nor accepted. To ensue that parties reach an agreement on the update, our protocol instructs an honest party in a hybrid situation to perform a *force close*. This means that the honest party posts the commit transaction representing the latest valid channel state on the ledger. Let us emphasize that the above high level description excludes some technical details which can be found in the formal protocol description, see Appendix C.

c) Close: The naive way to implement the closing procedure is to let parties publish the latest commit transaction parties agreed on, i.e. perform a force close. However, due to the built-in punishment mechanism, parties have to wait for a certain number of rounds after such commit transaction is accepted by the ledger to publish the split transaction representing the latest channel state.

Our protocol uses a slightly more efficient solution which eliminates the redundant waiting time for honest parties. When parties want to close a channel, they first run a "final update". On a high level, the final update preserves the latest channel state but removes the punishment layer. More precisely, parties agree on a new split transaction that has exactly the same outputs as the last split transaction but spends the funding transaction $\mathtt{TX_f}$ directly (i.e., **Steps 2+5** are skipped). Once parties jointly sign the split transaction, they can publish it on the ledger which completes the channel closure.

d) Punish: Since we are in the UTXO model, nothing can stop a corrupt party from publishing old commit transactions. However, the way we designed the commit transaction enables the honest party to punish such malicious behavior and get financially compensated. If an honest party A detects that a malicious party B posted an old commit transaction $\mathtt{TX_c}$, it can react by publishing a *punishment transaction* which spends $\mathtt{TX_c}$ and assigns all coins to A. In order to make such punishment transaction valid, A must sign it under: (i) its secret key sk_A, (ii) B's publishing secret key y_B, and (iii) B's revocation secret key r_B. The knowledge of the revocation secret r_B follows from the fact that $\mathtt{TX_c}$ was old, i.e. parties revealed their revocation secrets to each other. The knowledge of the publishing secret y_B follows from the fact that it was B who published $\mathtt{TX_c}$. Let us elaborate on this in more detail. Since $\mathtt{TX_c}$ was accepted by the ledger, it had to include a signature of A. The only signature A provided to B on $\mathtt{TX_c}$ was a *pre-signature* w.r.t. Y_B. The unforgeability and witness extractability properties of Ξ guarantee that the only way B could produce a valid signature of A on $\mathtt{TX_c}$ was by adapting the pre-signature thereby revealing the secret key y_B to A.

In the full version of this paper [14] we prove the following theorem, which essentially says that the Π protocol is a secure realization, as defined according to the UC framework, of the $\mathcal{F}(3,1)$ ideal functionality.

Theorem 2. *Let Σ be a* SUF–CMA *secure signature scheme, R a hard relation and $\Xi_{R,\Sigma}$ a secure adaptor signature scheme. Then for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}(3,1)$.*

VI. Applications

The $\mathcal{F}$ ideal functionality for generalized channels allows us to abstract from the implementation details of how a generalized channel is created, updated and closed. We now provide a generic guideline on how to interact with $\mathcal{F}$ in order to build off-chain applications on top of a generalized channel. Assume that Alice and Bob already created a channel γ via $\mathcal{F}$ and now want to use it for several applications. For that, parties have to carry out the following steps.

Initialize: Parties agree on the new state $\vec{\theta}$ of γ and the upper bound t_{stp} on the time required to setup off-chain objects. Hence, for each application parties need to agree on (i) the amount of coins they want to invest in the application and the funding condition; technically, this means that parties define $\theta_i = (\theta_i.\mathsf{cash}, \theta_i.\varphi)$, and (ii) the value t_i denoting the maximal amount of rounds that it takes to setup the corresponding application. The value t_{stp} is defined as $\max_i t_i$, thereby defining the maximal amount of rounds that it takes to setup all the applications in parallel.

Prepare: One party sends the message $(\mathtt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}})$ to $\mathcal{F}$ in order to prepare the update. Upon receiving such message, $\mathcal{F}$ responds with tid, an identifier of the transaction containing the outputs $\vec{\theta}$.

Setup: The parties exchange the application-dependent information required to fulfill the conditions $\{\theta_i.\varphi\}$ according to the rules of each application.

Complete: Parties inform $\mathcal{F}$ about setup completion by sending SETUP–OK and UPDATE–OK messages. Thereafter, $\mathcal{F}$ requests both parties to revoke the old state of γ which they do by invoking $\mathcal{F}$ on input the message REVOKE. $\mathcal{F}$ notifies the users of the completed update via the message UPDATED.

To conclude, for each application that one wants to build on top of a generalized channel, the following must be defined: (i) the amount of coins and the funding conditions of the outputs as required for the *initialize* step, (ii) the setup algorithm of the application and an upper bound on the number of rounds it takes, as needed for *setup* step. We remark that the *prepare* and *complete* steps are common to all applications.

We now demonstrate how to use this generic process on concrete examples by describing their initialize and setup steps. To avoid any repetition, for each example, we assume

there is a channel γ between parties A and B owning α_A and α_B coins and their public keys are pk_A and pk_B, respectively.

a) Channel splitting [11]: As discussed earlier in this work, a generalized channel can be split into multiple sub-channels that can be updated independently in parallel. Assume that the parties A and B want to split their channel γ into two sub-channels γ_0 and γ_1 with the coin distributions (β_A, β_B) and $(\alpha_A - \beta_A, \alpha_B - \beta_B)$ respectively. In order to do so, they follow the generic update algorithm described above with the initialize and setup steps defined as follows:

Initialize Parties create two outputs each of which funds one of the sub-channels:

- $\theta_0.\mathsf{cash} := \gamma_0.\mathsf{cash}$, $\theta_0.\varphi := \texttt{One–Sig}_{pk_A} \wedge \texttt{One–Sig}_{pk_B}$
- $\theta_1.\mathsf{cash} := \gamma_1.\mathsf{cash}$, $\theta_1.\varphi := \texttt{One–Sig}_{pk_A} \wedge \texttt{One–Sig}_{pk_B}$

and set the value $t_{\mathsf{stp}} := 2$ for the required setup steps.

Setup For each sub-channel, parties generate and sign the commit and split transactions representing the initial channel state. This procedure, explained in Section V, takes 2 rounds.

b) Payment-channel networks (PCNs) [27, 23, 22]: A payment-channel network (PCN) enables transitive payments between two users by leveraging a *path* of payment channels between the sender and the receiver. The Lightning Network implements a so-called multi-hop payment by means of a script called hash-time lock contract (HTLC). In particular, we denote by $\texttt{CheckHash}_y$ an output condition that can be spent by providing a value r such that $H(r) = y$. Using the same HTLC to update each channel in the payment path, the Lightning Network ensures that the payment is correctly carried out. In a bit more detail, the receiver sends to the sender the value y and keeps locally the value r such that $H(r) = y$. Then, the payment starts by updating each channel from the sender to the receiver so that it locks the payment amount into an output that can be spent under the condition $\texttt{CheckHash}_y$. When the channel with the receiver is updated accordingly, the receiver is sure that he can redeem the coins by revealing r. In addition, if the receiver never reveals the value r, the sender eventually gets back the locked coins with a timeout t condition, denoted by $\texttt{CheckAbsolute}_t$.

The generalized channel construction presented in this work can be used to implement PCNs. In particular, assume a payment of β coins through a payment path formed by n generalized channels. For each channel γ in the path, an update with the following initialize and setup steps is performed.

Initialize Parties exchange the hash value y, decide on the timeout value t, and create three outputs (one for the HTLC, one for the balance of A and one for the balance B):

- $\theta_0.\mathsf{cash} := \beta$, $\theta_0.\varphi := (\texttt{CheckHash}_y \wedge \texttt{One–Sig}_{pk_B}) \vee (\texttt{CheckAbsolute}_t \wedge \texttt{One–Sig}_{pk_A})$
- $\theta_1.\mathsf{cash} := \alpha_A - \beta$, $\theta_1.\varphi := \texttt{One–Sig}_{pk_A}$
- $\theta_2.\mathsf{cash} := \alpha_B$, $\theta_2.\varphi := \texttt{One–Sig}_{pk_B}$

Moreover, the parties set the value $t_{\mathsf{stp}} := 0$ as no setup is needed (more precisely, each party has enough information to prepare the transactions locally).

After the successful update of all channels on the path, the payment of α coins in the PCN is successfully set. A similar procedure can be carried out then to settle the payment when the receiver releases the value r such that $H(r) = y$.

VII. Performance Analysis

We created a proof of concept implementation for the `CREATE`, `UPDATE`, `CLOSE` and `PUNISH` operations. In a bit more detail, we utilized the `python-bitcoin-utils` library to create the required raw Bitcoin transactions encoded in the Bitcoin scripting language *Script*. Furthermore, we successfully deployed them on the Bitcoin testnet, demonstrating thereby the compatibility with the current Bitcoin network. The source code is publicly available.[7]

We evaluate the different operations for generalized channels using the following criteria: (i) the number of on- and off-chain transactions required in the protocols; (ii) the total amount of bytes that the on- and off-chain transactions sum up to; and (iii) the estimated cost (i.e. the transaction fee) for publishing the on-chain transactions required in each protocol. We remark that the transaction fee in Bitcoin is dependent on the transaction size. In our calculations, we use the price values valid at the time of writing: the average transaction fee is 14 satoshis per byte[8], or at the current exchange rate of 8869.67 USD per BTC[9], 0.00124 USD per byte.

A. Evaluation of multi-hop payments in PCNs

a) Single multi-hop payment: Let us evaluate the scenario, where we carry out one multi-hop payment, once on top of a Lightning channel and once on top of a generalized channel. To achieve this, we need three outputs, two containing the values for each of the parties and one for the HTLC.

A first difference is that in Lightning channels we need to store these outputs twice, once per commitment. If we were to update a channel to have one HTLC, we would require four off-chain transactions in the Lightning construction, two commitments with three outputs each and one transaction for the HTLC on each commitment. For the generalized channel construction, the number of transactions required for such an update is merely two, one for the commitment transaction and one for the split containing the three outputs. Note that, in the latter case, also the outputs need to be stored only once and that the HTLC does not require an additional transaction. The difference in off-chain transaction size is 1526 bytes for Lightning compared to 818 bytes for generalized channels.

A second difference is that, in the Lightning case, we need a punish mechanism per output. Hence, should an old commitment transaction get published, we would require two additional on-chain transactions with a total of 923 bytes in Lightning compared to only two transactions with 663 bytes in the generalized channel construction (including the commitment transaction in both cases). The difference for this is 1.15 USD vs 0.82 USD.

[7] https://github.com/generalized-channels/gc

[8] https://bitcoinfees.info/

[9] https://coinmarketcap.com/currencies/bitcoin/

b) Asymptotic analysis: Nodes participating in a payment-channel network typically take part in several, let us say n, multi-hop payments at once instead of just one. In this case, the Lightning solution scales even worse, as it requires $2+2\cdot n$ transactions or $706+2\cdot n\cdot 410$ bytes of off-chain transactions. With generalized channels, we only need 2 transactions with a size of $695+n\cdot 123$ bytes.

For punishment, the difference is even more pronounced, as we reduce the asymptotic complexity from linear to constant. Specifically, Lightning channels require $2+n$ on-chain transactions of $513+n\cdot 410$ bytes, which cost around $0.64+n\cdot 0.51$ USD. In generalized channels, the cost for punishment is independent of the number of HTLCs that are constructed on top of the channel. It requires 2 transactions with 663 bytes resulting in a cost of 0.82 USD. These differences can be observed in Table I for direct comparison.

TABLE I
EVALUATION OF LIGHTNING (LC) AND GENERALIZED CHANNELS (GC)

Operations	on-chain			off-chain	
	# txs	size	cost	# txs	size
update (LC)	0	0	0	$2+2\cdot n$	$706+2\cdot n\cdot 410$
update (GC)	0	0	0	2	$695+n\cdot 123$
punish (LC)	$2+n$	$513+n\cdot 410$	$0.64+n\cdot 0.51$	0	0
punish (GC)	2	663	0.82	0	0

In Table I, both constructions carry n HTLCs. *# txs* refers to the total number of transactions needed either on-chain or off-chain, *size* refers to the total number of bytes in all required on-/off-chain transactions, respectively, *cost* is in USD and denotes the estimated cost of publishing the transactions.

B. Evaluation of channel splitting

The comparison between Lightning and generalized channels in the case of channel splitting is summarized in Table II. Performing a split in a Lightning channel setting has the drawback of not only doubling off-chain objects that are potentially used on these sub-channels, but also the amount of commitment transactions, i.e., we need to create commitments for the sub-channels on both commitments of the initial channel. So if we were to split a channel, the required number of commitment transactions is four (two for every commitment) for each sub-channel with a total of 1412 bytes. In our generalized channel construction it is just one commitment and one split transaction per sub-channel, which is 695 bytes.

Once a split is performed, the sub-channels are expected to behave as a normal channel. Say that we want to split one of these sub-channels again into two: in the Lightning solution there would now be eight commitments (two for each of the four commitments) per sub-channel. Observe that after every recursive split of a channel, the amount of commitment transactions for the new sub-channel doubles for the Lightning construction. In the generalized channel construction, instead, we only need to keep track of one commitment transaction per sub-channel, therefore the amount of new commitment transactions per split is constant and not exponential. The difference between storing one and eight commitment transactions is 695 bytes for the generalized vs. 2824 bytes for the Lightning construction, not even counting any potential off-chain objects that would need to be stored eight times in the latter case. The amount of transactions needed for updates doubles for every split as well. For n splits, the difference would be 2^{n+1} additional commitment transactions in the Lightning setting against one new commitment and one new split transaction in the generalized channel setting, per sub-channel.

TABLE II
CHANNEL SPLITTING

	# txs per sub-channel	size
first split (LC)	4	1412
first split (GC)	2	695
n^{th} split (LC)	2^{n+1}	$353\cdot 2^{n+1}$
n^{th} split (GC)	2	695

VIII. CONCLUSION

Payment channels constitute one of the most promising approaches to tackle the scalability issue of decentralized blockchains. Despite the conceptually appealing design, which in principle supports different types of off-chain applications, existing constructions for Bitcoin-like cryptocurrencies suffer from a heavy communication complexity as well as on-chain footprint. This fundamentally undermines the potential of payment channels to serve as building block for a variegated multi-application off-chain ecosystem.

In this work, we formalize for the first time the notion of *generalized channels* for Bitcoin-like cryptocurrencies, a generalization of the concept of payment channels that provides off-chain support for any operation supported by the underlying blockchain. Besides the gain in expressiveness and the streamlined design of off-chain applications, generalized channels lead to a significant performance improvement, reducing the communication complexity and the on-chain footprint in case of disputes to linear and constant, respectively, in the number of applications leveraging the channel. Additionally, we provide a cryptographic instantiation of a generalized channel compatible with Bitcoin with provable security guarantees in the Universal Composability framework. To this end, we also introduce the first formalization of *adaptor signatures*, which we believe is of independent interest.

Generalized channels can be integrated today in the Lightning Network and other Bitcoin-compatible off-chain applications, thereby improving their performance. Most importantly, we believe generalized channels pave the way for the design of novel off-chain applications, such as Bitcoin-compatible virtual channels and more efficient and expressive payment channel hub constructions, a research direction we intend to explore in the near future.

REFERENCES

[1] C. Badertscher et al. “Bitcoin as a Transaction Ledger: A Composable Treatment”. In: *CRYPTO 2017, Part I*. Ed. by J. Katz and H. Shacham. Vol. 10401. LNCS. Springer, Heidelberg, Aug. 2017, pp. 324–356.

[2] W. Banasik et al. “Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts”. In: *ESORICS 2016, Part II*. Ed. by I. G. Askoxylakis et al. Vol. 9879. LNCS. Springer, Heidelberg, Sept. 2016, pp. 261–280. DOI: 10.1007/978-3-319-45741-3_14.

[3] S. Bano et al. “Consensus in the Age of Blockchains”. In: *CoRR* abs/1711.03936 (2017). arXiv: 1711.03936. URL: http://arxiv.org/abs/1711.03936.

[4] *Bitcoin Wiki: Payment Channels*. https://en.bitcoin.it/wiki/Payment_channels. 2018.

[5] D. Boneh et al. “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps”. In: *EUROCRYPT 2003*. Ed. by E. Biham. Vol. 2656. LNCS. Springer, Heidelberg, May 2003, pp. 416–432.

[6] R. Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145.

[7] R. Canetti et al. “Universally Composable Security with Global Setup”. In: *TCC 2007*. Ed. by S. P. Vadhan. Vol. 4392. LNCS. Springer, Heidelberg, Feb. 2007, pp. 61–85.

[8] S. Dziembowski et al. “General State Channel Networks”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 949–966.

[9] S. Dziembowski et al. “Multi-party Virtual State Channels”. In: *EUROCRYPT 2019, Part I*. Ed. by V. Rijmen and Y. Ishai. LNCS. Springer, Heidelberg, May 2019, pp. 625–656. DOI: 10.1007/978-3-030-17653-2_21.

[10] S. Dziembowski et al. “Perun: Virtual Payment Hubs over Cryptocurrencies”. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019, pp. 106–123.

[11] C. Egger et al. “Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. ACM, 2019, pp. 801–815.

[12] M. Fischlin. “Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors”. In: *CRYPTO 2005*. Ed. by V. Shoup. Vol. 3621. LNCS. Springer, Heidelberg, Aug. 2005, pp. 152–168.

[13] L. Fournier. *One-Time Verifiably Encrypted Signatures A.K.A. Adaptor Signatures*. https://github.com/LLFourn/one-time-VES/blob/master/main.pdf. 2019.

[14] *Generalized Bitcoin-Compatible Channels (Full Version)*. https://generalized-channels.github.io/.

[15] O. Goldreich. *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press, 2006. ISBN: 0521035368.

[16] L. Gudgeon et al. *SoK: Off The Chain Transactions*. Cryptology ePrint Archive, Report 2019/360. https://eprint.iacr.org/2019/360. 2019.

[17] E. Heilman et al. *The Arwen Trading Protocols (Full Version)*. Cryptology ePrint Archive, Report 2020/024. https://eprint.iacr.org/2020/024. 2020.

[18] E. Heilman et al. *TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub*. Cryptology ePrint Archive, Report 2016/575. http://eprint.iacr.org/2016/575, accepted to the Network and Distributed System Security Symposium (NDSS) 2017. 2016.

[19] J. Katz et al. “Universally Composable Synchronous Computation”. In: *TCC 2013*. Ed. by A. Sahai. Vol. 7785. LNCS. Springer, Heidelberg, Mar. 2013, pp. 477–498. DOI: 10.1007/978-3-642-36594-2_27.

[20] A. Kiayias and O. S. T. Litos. *A Composable Security Treatment of the Lightning Network*. Cryptology ePrint Archive, Report 2019/778. https://eprint.iacr.org/2019/778. 2019.

[21] Y. Lindell. “Fast Secure Two-Party ECDSA Signing”. In: *CRYPTO 2017, Part II*. Ed. by J. Katz and H. Shacham. Vol. 10402. LNCS. Springer, Heidelberg, Aug. 2017, pp. 613–644.

[22] G. Malavolta et al. “Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability”. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/.

[23] G. Malavolta et al. “Concurrency and Privacy with Payment-Channel Networks”. In: *ACM CCS 17*. Ed. by B. M. Thuraisingham et al. ACM Press, 2017, pp. 455–471.

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

[25] P. Moreno-Sanchez and A. Kate. *Scriptless Scripts with ECDSA*. lightning-dev mailing list. https://lists.linuxfoundation.org/pipermail/lightning-dev/attachments/20180426/fe978423/attachment-0001.pdf.

[26] A. Poelstra. *Scriptless scripts*. https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-05-milan-meetup/slides.pdf. 2017.

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

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

[29] A. Zamyatin et al. *SoK: Communication Across Distributed Ledgers*. Cryptology ePrint Archive, Report 2019/1128. https://eprint.iacr.org/2019/1128. 2019.

APPENDIX

A. Related work

a) Adaptor signatures: Poelstra introduced the notion of adaptor signatures. In a nutshell, an adaptor signature

(AS) is a modified version of a digital signature so that a valid signature can be created only given a witness for a cryptographic hardness assumption (e.g., discrete logarithm problem) [26]. Adaptor signatures have been proven useful in off-chain applications such as PCNs [22].

Given the utility of AS, there have been some attempts to formally use them. For instance, Malavolta et al. [22] use AS as building block to define and realize multi-hop payments in PCNs. However, they do not define AS as a stand alone primitive that can be then used in other works. Concurrent to our work, Fournier [13] attempts to formalize AS as an instance of one-time verifiable encrypted signatures. Yet, in their definition the adversary is not given a pre-signature on the challenge message in the unforgeability and extractability games. However, in applications including our generalized channels, the adversary learns a pre-signature on the message for which it wishes to forge a signature.

Boneh et al. [5] define the notion of verifiably encrypted signatures (VES). In this setting the signer wishes to show the verifier that she has signed a message correctly without revealing the signature. In another similar work, Banasik et al. [2] introduce a method that allows two parties (buyer and seller) to exchange a digital asset using cryptocurrencies that do not support Turing complete programs (smart contracts). Neither of the those works provide a construction that can realize the properties expected from an AS.

b) Generalized channels: The authors in [20] provide a formalization of the Lightning Network (LN) in the UC framework. This formalization is however tailored to the details of the current LN and cannot be leveraged to formalize generalized channels as we propose in this work. State channels enable to execute arbitrary computations off-chain [8, 9, 24]. Moreover, the authors have also provided a formal model in the UC framework. These constructions, however, require a highly expressive scripting functionality (e.g., as in Ethereum) that is not available in many cryptocurrency, including Bitcoin.

B. On the usage of the UC-Framework

To formally model the security of our construction, we use a synchronous version of the global UC framework (GUC) [7] which extends the standard UC framework [6] by allowing for a global setup. Since our model is essentially the same as in [8, 9], parts of this section are taken verbatim from there.

a) Protocols and adversarial model: We consider a protocol π that runs between parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ that takes as input a security parameter 1^n (with $n \in \mathbb{N}$) 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 $\mathcal{A}$ takes full control over P_i and learns its internal state. Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{E}$ – which represents anything "external" to the current protocol execution. The environment also observes all outputs returned by the parties of the protocol.

b) Modeling time and communication: 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 and allows us to argue about the time complexity of our protocols in a natural way. We follow [9], which in turn follows [19], and formalize the notion of rounds via an ideal functionality $\mathcal{F}_{clock}$ representing "the clock". On a high level, the ideal functionality requires all honest parties to indicate that they are prepared to proceed to the next round before the clock is "ticked". We treat the clock functionality as a *global* ideal functionality using the GUC model. This means that all entities are always aware of the given round.

We assume that parties of a protocol are connected via authenticated communication channels with guaranteed delivery of exactly one round. This means that if a party P sends a message m to party Q in round t, party Q receives this message in beginning of round $t+1$. In addition, Q is sure that the message was sent by party P. The adversary can see the content of the message and can reorder messages that were sent in the same round. However, it can not modify, delay or drop messages sent between parties, or insert new messages. The assumptions on the communication channels are formalized as an ideal functionality $\mathcal{F}_{GDC}$. We refer the reader to [9] its formal description.

While the communication between two parties of a protocol takes exactly one round, all other communication – for example, between the adversary $\mathcal{A}$ and the environment $\mathcal{E}$ – takes zero rounds. For simplicity, we assume that any computation made by any entity takes zero rounds as well.

c) Handling coins: We model the money mechanics offered by UTXO crypotcurrencies, such as Bitcoin, via a *global* ideal functionality $\mathcal{L}$ using the GUC model. Our functionality is parameterized by a *delay parameter* Δ which upper bounded in the maximal number of rounds it takes to publish a valid transaction, and a signature scheme Σ. The functionality accepts messages from a fixed set of parties $\mathcal{P}$.

The ledger functionality $\mathcal{L}$ is initiated by the environment $\mathcal{E}$ via the following steps: (1) $\mathcal{E}$ instructs the ledger functionality to generate public parameter of the signature scheme pp; (2) $\mathcal{E}$ instructs every party $P \in \mathcal{P}$ to generate a key pair (sk_P, pk_P) and submit the public key pk_P to the ledger via the message $(\text{register}, pk_P)$; (3) sets the initial state of the ledger meaning that it initialize a set `TX` defining all published transactions.

Once initialized, the state of $\mathcal{L}$ is public and can be accessed by all parties of the protocol, the adversary $\mathcal{A}$ and the environment $\mathcal{E}$. Any party $P \in \mathcal{P}$ can at any time post a transaction on the ledger via the message $(\text{post}, \mathsf{tx})$. The ledger functionality waits for at most Δ rounds (the exact number of rounds is determined by the adversary). Thereafter, the ledger verifies the validity of the transaction and adds it to the transaction set `TX`. The formal description of the ledger functionality follows.

Ideal Functionality $\mathcal{L}(\Delta, \Sigma)$

The functionality accepts messages from all parties that are in the set $\mathcal{P}$ and maintains a PKI for those parties. The functionality maintains the set of all accepted transactions TX and all unspent transaction outputs UTXO. The set $\mathcal{V}$ defines valid output conditions.
Initialize public keys: Upon $(\text{register}, pk_P) \overset{\tau_0}{\hookleftarrow} P$ and it is the first time P sends a registration message, add (pk_P, P) to PKI.
Post transaction: Upon $(\text{post}, \mathsf{tx}) \overset{\tau_0}{\hookleftarrow} P$, check that $|\mathtt{PKI}| = |\mathcal{P}|$. If not, drop the message, else wait until round $\tau_1 \leq \tau_0 + \Delta$ (the exact value of τ_1 is determined by the adversary). Then check if:
1) The id is unique, i.e. for all $(t, \mathsf{tx}') \in \mathtt{TX}$, $\mathsf{tx}'.\mathsf{txid} \neq \mathsf{tx}.\mathsf{txid}$.
2) All the inputs are unspent and the witness satisfies all the output conditions, i.e. for each $(tid, i) \in \mathsf{tx}.\mathsf{Input}$, there exists $(t, tid, i, \theta) \in \mathtt{UTXO}$ and $\theta.\varphi(\mathsf{tx}, t, \tau_1) = 1$.
3) All outputs are valid, i.e. for each $\theta \in \mathsf{tx}.\mathsf{Output}$ it holds that $\theta.\mathsf{cash} > 0$ and $\theta.\varphi \in \mathcal{V}$.
4) The value of the outputs is not larger than the value of the inputs. More formally, let $I := \{\mathtt{utxo} := (t, tid, i, \theta) \mid \mathtt{utxo} \in \mathtt{UTXO} \wedge (tid, i) \in \mathsf{tx}.\mathsf{Input}\}$, then $\sum_{\theta' \in \mathsf{tx}.\mathsf{Output}} \theta'.\mathsf{cash} \leq \sum_{\mathtt{utxo} \in I} \mathtt{utxo}.\theta.\mathsf{cash}$
5) The absolute time-lock of the transaction has expired, i.e. $\mathsf{tx}.\mathsf{TimeLock} \leq \mathsf{now}$.

If all the above checks return true, add (τ_1, tx) to TX, remove the spent outputs from UTXO, i.e., $\mathtt{UTXO} := \mathtt{UTXO} \setminus I$ and add the outputs of tx to UTXO, i.e., $\mathtt{UTXO} := \mathtt{UTXO} \cup \{(\tau_1, \mathsf{tx}.\mathsf{txid}, i, \theta_i)\}_{i \in [n]}$ for $(\theta_1, \ldots, \theta_n) := \mathsf{tx}.\mathsf{Output}$. Else, ignore the message.

Let us emphasize that our ledger functionality is fairly simplified. In reality, parties can join and leave the blockchain system dynamically. Moreover, we completely abstract from the fact that transactions are published in blocks which are proposed by parties and the adversary. Those and other features are captured by prior works, such as [1], that provide a more accurate formalization of the Bitcoin ledger in the UC framework [6]. However, interaction with such ledger functionality is fairly complex. To increase the readability of our channel protocols and ideal functionality, which is the main focus on our work, we decided for this simpler ledger.

d) The GUC-security definition: Let π be a protocol with access to the global ledger $\mathcal{L}(\Delta, \Sigma)$ and the global clock $\mathcal{F}_{clock}$. The output of an environment $\mathcal{E}$ interacting with a protocol π and an adversary $\mathcal{A}$ on input 1^n and auxiliary input z is denoted as $\text{EXE}^{\mathcal{L}(\Delta,\Sigma),\mathcal{F}_{clock}}_{\pi,\mathcal{A},\mathcal{E}}(n, z)$. Let $\phi_{\mathcal{F}}$ be the ideal protocol for an ideal functionality $\mathcal{F}$ with access to the global ledger $\mathcal{L}(\Delta, \Sigma)$ and the global clock $\mathcal{F}_{clock}$. This means that $\phi_{\mathcal{F}}$ is a trivial protocol in which the parties simply forward their inputs to the ideal functionality $\mathcal{F}$. The output of an environment $\mathcal{E}$ interacting with a protocol $\phi_{\mathcal{F}}$ and a adversary $\mathcal{S}$ (sometimes also call *simulator*) on input 1^n and auxiliary input z is denoted as $\text{EXE}^{\mathcal{L}(\Delta,\Sigma),\mathcal{F}_{clock}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{E}}(n, z)$.

We are now ready to state our main security definition which, informally, says that if a protocol π UC-realizes an ideal functionality $\mathcal{F}$, then any attack that can be carried out against the real-world protocol π can also be carried out against the ideal protocol $\phi_{\mathcal{F}}$.

Definition 7. We say that a protocol π *UC-realizes an ideal functionality $\mathcal{F}$ with respect to a global ledger $\mathcal{L} := \mathcal{L}(\Delta, \Sigma)$* and a global clock $\mathcal{F}_{clock}$ if for every adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that we have

$$\left\{\text{EXE}^{\mathcal{L},\mathcal{F}_{clock}}_{\pi,\mathcal{A},\mathcal{E}}(n, z)\right\}_{\substack{n \in \mathbb{N}, \\ z \in \{0,1\}^*}} \overset{c}{\approx} \left\{\text{EXE}^{\mathcal{L},\mathcal{F}_{clock}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{E}}(n, z)\right\}_{\substack{n \in \mathbb{N}, \\ z \in \{0,1\}^*}}$$

(where "$\overset{c}{\approx}$" denotes computational indistinguishability of distribution ensembles, see, e.g., [15]).

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.

C. Additional material to channel protocol

We now formally describe the protocol for generalized channels Π described on high level in Section V. The protocol internally uses a secure adaptor signature scheme $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ for the ledger signature scheme Σ and a relation R. We assume that statement/witness pairs of R are public/secret key of Σ. More precisely, we assume there exists a function ToKey that takes as input a statement $Y \in L_R$ and outputs a public key pk. The function is s.t. the distribution of $(\mathsf{ToKey}(Y), y)$, for $(Y, y) \leftarrow \mathsf{GenR}$, is equal to the distributions of $(pk, sk) \leftarrow \mathsf{Gen}$. We emphasize that both ECDSA and Schnorr based adaptor signatures satisfy this condition (ECDSA, the ToKey simply drops the NIZK, for Schnorr ToKey is the identity function). We discuss how to modify our protocol if this assumption does not hold in [14]. We first introduce some conventions.

We assume that each party $P \in \mathcal{P}$ maintains a set Γ^P of all open channels together with auxiliary information about the channel (such as the funding transaction, latest commit transaction and corresponding revocation secret etc.). In addition to the channel set, we assume that each party maintains a set Θ^P containing all revoked commit transactions and corresponding revocation secretes. Similarly to the the formal description of the ideal functionality, we make use of a arrow notation for sending and receiving messages. Moreover, our formal description excludes some natural check an honest party should make. Those checks are define as a protocol wrapper in Appendix E. In the protocol description, we abbreviate $\mathtt{One\text{–}Sig}_{pk_1} \wedge \cdots \wedge \mathtt{One\text{–}Sig}_{pk_n}$ as $\mathtt{Multi\text{–}Sig}_{pk_1,\ldots,pk_n}$.

Generalized channel protocol

Below, we abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Create

Party P upon $(\mathtt{CREATE}, \gamma, tid_P) \overset{t_0}{\hookleftarrow} \mathcal{E}$:
1) Set $id := \gamma.\mathsf{id}$, generate $(R_P, r_P) \leftarrow \mathsf{GenR}$, $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\text{createInfo}, id, tid_P, R_P, Y_P) \overset{t_0}{\hookrightarrow} Q$.
2) If $(\text{createInfo}, id, tid_Q, R_Q, Y_Q) \overset{t_0+1}{\hookleftarrow} Q$, create:

$$[\mathtt{TX_f}] := \mathtt{GenFund}((tid_P, tid_Q), \gamma)$$
$$[\mathtt{TX_c}] := \mathtt{GenCommit}([\mathtt{TX_f}], I_P, I_Q, 0)$$
$$[\mathtt{TX_s}] := \mathtt{GenSplit}([\mathtt{TX_c}].\mathsf{txid} \| 1, \gamma.\mathsf{st})$$

for $I_P := (pk_P, R_P, Y_P)$, $I_Q := (pk_Q, R_Q, Y_Q)$. Else stop.
3) Compute $s^P_{\mathsf{c}} \leftarrow \mathsf{pSign}_{sk_P}([\mathtt{TX_c}], Y_Q)$, $s^P_{\mathsf{s}} \leftarrow \mathsf{Sign}_{sk_P}([\mathtt{TX_s}])$

and send $(\text{createCom}, id, s_{\mathtt{c}}^P, s_{\mathtt{s}}^P) \xhookrightarrow{t_0+1} Q$.

4) If $(\text{createCom}, id, s_{\mathtt{c}}^Q, s_{\mathtt{s}}^Q) \xhookleftarrow{t_0+2} Q$, s.t. $\mathsf{pVrfy}_{pk_Q}([\mathtt{TX}_{\mathtt{c}}], Y_P; s_{\mathtt{c}}^Q) = 1$ and $\mathsf{Vrfy}_{pk_Q}([\mathtt{TX}_{\mathtt{s}}]; s_{\mathtt{s}}^Q) = 1$, $s_{\mathtt{f}}^P \leftarrow \mathsf{Sign}_{sk_P}([\mathtt{TX}_{\mathtt{f}}])$ and send $(\text{createFund}, id, s_{\mathtt{f}}^P) \xhookrightarrow{t_0+2} Q$. Else stop.

5) If $(\text{createFund}, id, s_{\mathtt{f}}^Q) \xhookleftarrow{t_0+3} Q$, s.t. $\mathsf{Vrfy}_{pk_Q}([\mathtt{TX}_{\mathtt{f}}]; s_{\mathtt{f}}^Q) = 1$, $\mathtt{TX}_{\mathtt{f}} := ([\mathtt{TX}_{\mathtt{f}}], \{s_{\mathtt{f}}^P, s_{\mathtt{f}}^Q\})$ and $(\text{post}, \mathtt{TX}_{\mathtt{f}}) \xhookrightarrow{t_0+3} \mathcal{L}$. Else stop.

6) If $\mathtt{TX}_{\mathtt{f}}$ is accepted by $\mathcal{L}$ in round $t_1 \leq t_0 + 3 + \Delta$, set $\mathtt{TX}_{\mathtt{c}} := ([\mathtt{TX}_{\mathtt{c}}], \{\mathsf{Sign}_{sk_P}([\mathtt{TX}_{\mathtt{c}}]), \mathsf{Adapt}(s_{\mathtt{c}}^Q, y_P)\})$, $\mathtt{TX}_{\mathtt{s}} := ([\mathtt{TX}_{\mathtt{s}}], \{s_{\mathtt{s}}^P, s_{\mathtt{s}}^Q\})$, store $\Gamma^P(\gamma.\mathsf{id}) := (\gamma, \mathtt{TX}_{\mathtt{f}}, (\mathtt{TX}_{\mathtt{c}}, r_P, R_Q, Y_Q, s_{\mathtt{c}}^P), \mathtt{TX}_{\mathtt{s}})$ and $(\mathtt{CREATED}, id) \xhookrightarrow{t_1} \mathcal{E}$.

Update

Party P upon $(\mathtt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \xhookleftarrow{t_0} \mathcal{E}$

1) Generate $(R_P, r_P) \leftarrow \mathsf{GenR}$, $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\text{updateReq}, id, \vec{\theta}, t_{\mathsf{stp}}, R_P, Y_P) \xhookrightarrow{t_0} Q$.

Party Q upon $(\text{updateReq}, id, \vec{\theta}, t_{\mathsf{stp}}, R_P, Y_P) \xhookleftarrow{\tau_0} P$

2) Generate $(R_Q, r_Q) \leftarrow \mathsf{GenR}$ and $(Y_Q, y_Q) \leftarrow \mathsf{GenR}$.

3) Set $t_{\mathsf{lock}} := \tau_0 + t_{\mathsf{stp}} + 4 + \Delta$, extract $\mathtt{TX}_{\mathtt{f}}$ from $\Gamma^P(id)$ and

$$[\mathtt{TX}_{\mathtt{c}}] := \mathtt{GenCommit}([\mathtt{TX}_{\mathtt{f}}], I_P, I_Q, t_{\mathsf{lock}})$$
$$[\mathtt{TX}_{\mathtt{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathtt{c}}].\mathsf{txid} \| 1, \vec{\theta})$$

where $I_P := (pk_P, R_P, Y_P)$, $I_Q := (pk_Q, R_Q, Y_Q)$.

4) Sign $s_{\mathtt{s}}^Q \leftarrow \mathsf{Sign}_{sk_Q}([\mathtt{TX}_{\mathtt{s}}])$, send $(\text{updateInfo}, id, R_Q, Y_Q, s_{\mathtt{s}}^Q) \xhookrightarrow{\tau_0} P$, $(\mathtt{UPDATE\text{–}REQ}, id, \vec{\theta}, t_{\mathsf{stp}}, \mathtt{TX}_{\mathtt{s}}.\mathsf{txid}) \xhookrightarrow{\tau_0+1} \mathcal{E}$.

Party P upon $(\text{updateInfo}, id, h_Q, Y_Q, s_{\mathtt{s}}^Q) \xhookleftarrow{t_0+2} Q$

5) Set $t_{\mathsf{lock}} := t_0 + t_{\mathsf{stp}} + 5 + \Delta$, extract $\mathtt{TX}_{\mathtt{f}}$ from $\Gamma^Q(id)$ and

$$[\mathtt{TX}_{\mathtt{c}}] := \mathtt{GenCommit}([\mathtt{TX}_{\mathtt{f}}], I_P, I_Q, t_{\mathsf{lock}})$$
$$[\mathtt{TX}_{\mathtt{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathtt{c}}].\mathsf{txid} \| 1, \vec{\theta}),$$

for $I_P := (pk_P, R_P, Y_P)$ and $I_Q := (pk_Q, R_Q, Y_Q)$. If $\mathsf{Vrfy}_{pk_Q}([\mathtt{TX}_{\mathtt{s}}]; s_{\mathtt{s}}^Q) = 1$, $(\mathtt{SETUP}, id, \mathtt{TX}_{\mathtt{s}}.\mathsf{txid}) \xhookrightarrow{t_0+2} \mathcal{E}$. Else stop.

6) If $(\mathtt{SETUP\text{–}OK}, id) \xhookleftarrow{t_1 \leq t_0+2+t_{\mathsf{stp}}} \mathcal{E}$, compute $s_{\mathtt{c}}^P \leftarrow \mathsf{pSign}_{sk_P}([\mathtt{TX}_{\mathtt{c}}], Y_Q) s_{\mathtt{s}}^P \leftarrow \mathsf{Sign}_{sk_P}([\mathtt{TX}_{\mathtt{s}}])$ and send $(\text{updateComP}, id, s_{\mathtt{c}}^P, s_{\mathtt{s}}^P) \xhookrightarrow{t_1} Q$. Else stop.

Party Q

7) If $(\text{updateComP}, id, s_{\mathtt{c}}^P, s_{\mathtt{s}}^P) \xhookleftarrow{\tau_1 \leq \tau_0+2+t_{\mathsf{stp}}} P$, s.t. $\mathsf{pVrfy}_{pk_P}([\mathtt{TX}_{\mathtt{c}}], Y_Q; s_{\mathtt{c}}^P) = 1$ and $\mathsf{Vrfy}_{pk_P}([\mathtt{TX}_{\mathtt{s}}]; s_{\mathtt{s}}^P) = 1$, output $(\mathtt{SETUP\text{–}OK}, id) \xhookrightarrow{\tau_1} \mathcal{E}$. Else stop.

8) If $(\mathtt{UPDATE\text{–}OK}, id) \xhookleftarrow{\tau_1} \mathcal{E}$, pre-sign $s_{\mathtt{c}}^Q \leftarrow \mathsf{pSign}([\mathtt{TX}_{\mathtt{c}}], Y_P)$ and send $(\text{updateComQ}, id, s_{\mathtt{c}}^Q) \xhookrightarrow{\tau_1} P$. Else send $(\text{updateNotOk}, id, r_Q) \xhookrightarrow{\tau_1} P$ and stop.

Party P

9) In round $t_1 + 2$ distinguish the following cases:
- If $(\text{updateComQ}, id, s_{\mathtt{c}}^Q) \xhookleftarrow{t_1+2} Q$, s.t. $\mathsf{pVrfy}_{pk_Q}([\mathtt{TX}_{\mathtt{c}}], Y_P; s_{\mathtt{c}}^Q) = 1$, output $(\mathtt{UPDATE\text{–}OK}, id) \xhookrightarrow{t_1+2} \mathcal{E}$.
- If $(\text{updateNotOk}, id, r_Q) \xhookleftarrow{t_1+2} Q$, s.t. $(R_Q, r_Q) \in R$, add $\Theta^P(id) := \Theta^P(id) \cup ([\mathtt{TX}_{\mathtt{c}}], r_Q, Y_Q, s_{\mathtt{c}}^P)$ and stop.
- Else, execute the procedure $\mathtt{ForceClose}^P(id)$ and stop.

10) If $(\mathtt{REVOKE}, id) \xhookleftarrow{t_1+2} \mathcal{E}$, parse $\Gamma^P(id)$ as $(\gamma, \mathtt{TX}_{\mathtt{f}}, (\overline{\mathtt{TX}}_{\mathtt{c}}, \bar{r}_P, \bar{R}_Q, \bar{Y}_Q, \bar{s}_{\mathtt{Com}}^P), \overline{\mathtt{TX}}_{\mathtt{s}})$ and update the channel space as $\Gamma^P(id) := (\gamma, \mathtt{TX}_{\mathtt{f}}, (\mathtt{TX}_{\mathtt{c}}, r_P, R_Q, Y_Q, s_{\mathtt{c}}^P), \mathtt{TX}_{\mathtt{s}})$, for $\mathtt{TX}_{\mathtt{s}} := ([\mathtt{TX}_{\mathtt{s}}], \{s_{\mathtt{s}}^P, s_{\mathtt{s}}^Q\})$ and $\mathtt{TX}_{\mathtt{c}} := ([\mathtt{TX}_{\mathtt{c}}], \{\mathsf{Sign}_{sk_P}([\mathtt{TX}_{\mathtt{c}}]), \mathsf{Adapt}(s_{\mathtt{c}}^Q, y_P)\})$., and send $(\text{revokeP}, id, \bar{r}_P) \xhookrightarrow{t_1+2} Q$. Else, execute $\mathtt{ForceClose}^P(id)$ and stop.

Party Q

11) Parse $\Gamma^Q(id)$ as $(\gamma, \mathtt{TX}_{\mathtt{f}}, (\overline{\mathtt{TX}}_{\mathtt{c}}, \bar{r}_Q, \bar{R}_P, \bar{Y}_P, \bar{s}_{\mathtt{Com}}^Q), \overline{\mathtt{TX}}_{\mathtt{s}})$. If $(\text{revokeP}, id, \bar{r}_P) \xhookleftarrow{\tau_1+2} P$, s.t. $(\bar{R}_P, \bar{r}_P) \in R$, $(\mathtt{REVOKE\text{–}REQ}, id) \xhookrightarrow{\tau_1+2} \mathcal{E}$. Else execute $\mathtt{ForceClose}^Q(id)$ and stop.

12) If $(\mathtt{REVOKE}, id) \xhookleftarrow{\tau_1+2} \mathcal{E}$ as a reply, set

$$\Theta^Q(id) := \Theta^Q(id) \cup ([\overline{\mathtt{TX}}_{\mathtt{c}}], \bar{r}_P, \bar{Y}_P, \bar{s}_{\mathtt{Com}}^Q)$$
$$\Gamma^Q(id) := (\gamma, \mathtt{TX}_{\mathtt{f}}, (\mathtt{TX}_{\mathtt{c}}, r_Q, R_P, Y_P, s_{\mathtt{c}}^Q), \mathtt{TX}_{\mathtt{s}}),$$

for $\mathtt{TX}_{\mathtt{s}} := ([\mathtt{TX}_{\mathtt{s}}], \{s_{\mathtt{s}}^P, s_{\mathtt{s}}^Q\})$, $\mathtt{TX}_{\mathtt{c}} := ([\mathtt{TX}_{\mathtt{c}}], \{\mathsf{Sign}_{sk_Q}([\mathtt{TX}_{\mathtt{c}}]), \mathsf{Adapt}(s_{\mathtt{c}}^P, y_Q)\})$, and send $(\text{revokeQ}, id, \bar{r}_Q) \xhookrightarrow{\tau_1+2} P$. In the next round $(\mathtt{UPDATED}, id) \xhookrightarrow{\tau_1+3} \mathcal{E}$ and stop. Else, in round $\tau_1 + 2$, execute $\mathtt{ForceClose}^Q(id)$ and stop.

Party P

13) If $(\text{revokeQ}, id, \bar{r}_Q) \xhookleftarrow{t_1+4} Q$ s.t. $(\bar{R}_Q, \bar{r}_Q) \in R$, then set $\Theta^P(id) := \Theta^P(id) \cup ([\overline{\mathtt{TX}}_{\mathtt{c}}], \bar{r}_Q, \bar{Y}_Q, \bar{s}_{\mathtt{Com}}^P)$ and $(\mathtt{UPDATED}, id) \xhookrightarrow{t_1+4} \mathcal{E}$. Else execute $\mathtt{ForceClose}^P(id)$ and stop.

Close

Party P upon $(\mathtt{CLOSE}, id) \xhookleftarrow{t_0} \mathcal{E}$

1) Extract $\mathtt{TX}_{\mathtt{f}}$ and $\mathtt{TX}_{\mathtt{s}}$ from $\Gamma^P(id)$ and set:

$$[\overline{\mathtt{TX}}_{\mathtt{s}}] := \mathtt{GenSplit}(\mathtt{TX}_{\mathtt{f}}.\mathsf{txid} \| 1, \mathtt{TX}_{\mathtt{s}}.\mathsf{Output})$$

2) Compute $s_{\mathtt{s}}^P \leftarrow \mathsf{Sign}_{sk_P}([\overline{\mathtt{TX}}_{\mathtt{s}}])$ and send $s_{\mathtt{s}}^P \xhookrightarrow{t_0} Q$.

3) If $s_{\mathtt{s}}^Q \xhookleftarrow{t_0+1} Q$ s.t. $\mathsf{Vrfy}_{pk_Q}([\overline{\mathtt{TX}}_{\mathtt{s}}]; s_{\mathtt{s}}^Q) = 1$, set $\overline{\mathtt{TX}}_{\mathtt{s}} := ([\overline{\mathtt{TX}}_{\mathtt{s}}], \{s_{\mathtt{s}}^P, s_{\mathtt{s}}^Q\})$ and send $(\text{post}, \overline{\mathtt{TX}}_{\mathtt{s}}) \xhookrightarrow{t_0+1} \mathcal{L}$. Else, execute $\mathtt{ForceClose}^P(id)$ and stop.

4) Let $t_2 \leq t_1 + \Delta$ be the round in which $\overline{\mathtt{TX}}_{\mathtt{s}}$ is accepted by $\mathcal{L}$. Set $\Gamma^P(id) = \bot$, $\Theta^P(id) = \bot$ and send $(\mathtt{CLOSED}, id) \xhookrightarrow{t_2} \mathcal{E}$.

Punish

Party P upon $\mathtt{PUNISH} \xhookleftarrow{t_0} \mathcal{E}$:

For each $id \in \{0,1\}^*$ s.t. $\Theta^P(id) \neq \bot$:

1) Parse $\Theta^P(id) := \{([\mathtt{TX}_{\mathtt{c}}^{(i)}], r_Q^{(i)}, Y_Q^{(i)}, s^{(i)})\}_{i \in m}$ and extract γ from $\Gamma^P(id)$. If for some $i \in [m]$, there exist a transaction tx on $\mathcal{L}$ such that $\mathsf{tx}.\mathsf{txid} = \mathtt{TX}_{\mathtt{c}}^{(i)}.\mathsf{txid}$, then parse the witness as $(s_P, s_Q) := \mathsf{tx}.\mathsf{Witness})$, where $\mathsf{Vrfy}_{pk_P}([\mathsf{tx}]; s_P) = 1$, and set $y_Q^{(i)} := \mathsf{Ext}(s_P, s^{(i)}, Y_Q^{(i)})$.

2) Define the body of the punishment transaction $[\mathtt{TX}_{\mathtt{pun}}]$ as:

$$\mathtt{TX}_{\mathtt{pun}}.\mathsf{Input} := \mathsf{tx}.\mathsf{txid} \| 1,$$
$$\mathtt{TX}_{\mathtt{pun}}.\mathsf{Output} := \{(\gamma.\mathsf{cash}, \mathtt{One\text{–}Sig}_{pk_P})\}$$

3) Sign $s_y \leftarrow \mathsf{Sign}_{y_Q^{(i)}}([\mathtt{TX}_{\mathtt{pun}}]), s_r \leftarrow \mathsf{Sign}_{r_Q^{(i)}}([\mathtt{TX}_{\mathtt{pun}}]), s_P \leftarrow \mathsf{Sign}_{pk_P}([\mathtt{TX}_{\mathtt{pun}}])$, and set $\mathtt{TX}_{\mathtt{pun}} := ([\mathtt{TX}_{\mathtt{pun}}], s_y, s_r, s_P)$. Then $(\text{post}, \mathtt{TX}_{\mathtt{pun}}) \xhookrightarrow{t_0} \mathcal{L}$.

4) Let $\mathtt{TX}_{\text{pun}}$ be accepted by $\mathcal{L}$ in round $t_1 \le t_0+\Delta$. Set $\Theta^P(id) = \bot$, $\Gamma^P(id) = \bot$ and output $(\mathtt{PUNISHED}, id) \overset{t_1}{\hookrightarrow} \mathcal{E}$.

Subprocedures

$\mathtt{GenFund}(\vec{tid}, \gamma)$:
Return [tx], where tx.Input := $\vec{tid}$ and tx.Output := $\{(\gamma.\mathsf{cash}, \mathtt{Multi\text{–}Sig}_{\gamma.\mathsf{users}})\}$.

$\mathtt{GenCommit}([\mathtt{TX_f}], (pk_P, R_P, Y_P), (pk_Q, R_Q, Y_Q), t)$:
Let $(c, \mathtt{Multi\text{–}Sig}_{pk_P, pk_Q}) := \mathtt{TX_f}.\mathsf{Output}[1]$ and denote

$$\varphi_1 := \mathtt{Multi\text{–}Sig}_{\mathsf{ToKey}(R_Q), \mathsf{ToKey}(Y_Q), pk_P},$$
$$\varphi_2 := \mathtt{Multi\text{–}Sig}_{\mathsf{ToKey}(R_P), \mathsf{ToKey}(Y_P), pk_Q},$$
$$\varphi_3 := \mathtt{CheckRelative}_{\Delta} \wedge \mathtt{Multi\text{–}Sig}_{pk_P, pk_Q}.$$

Return [tx], where tx.Input = $\mathtt{TX_f}.\mathsf{txid}\|1$, tx.Output := $(c, \varphi_1 \vee \varphi_2 \vee \varphi_3)$ and set tx.TimeLock to t if $t >$ now and to 0 otherwise.

$\mathtt{GenSplit}(tid, \vec{\theta})$:
Return [tx], where tx.Input := tid and tx.Output := $\vec{\theta}$.

$\mathtt{ForceClose}^P(id)$:
Let t_0 be the current round.
1) Extract $\mathtt{TX_c}$ and $\mathtt{TX_s}$ from $\Gamma(id)$.
2) Wait until round $t_1 := \max\{t_0, \mathtt{TX_c}.\mathsf{TimeLock}\}$ and send $(\text{post}, \mathtt{TX_c}) \overset{t_1}{\hookrightarrow} \mathcal{L}$.
3) Let $t_2 \le t_1 + \Delta$ be the round in which $\mathtt{TX_c}$ is accepted by the blockchain. Wait for Δ rounds to $(\text{post}, \mathtt{TX_s}) \overset{t_2+\Delta}{\hookrightarrow} \mathcal{L}$.
4) Once $\mathtt{TX_s}$ is accepted by $\mathcal{L}$ in round $t_3 \le t_2+2\Delta$, set $\Theta^P(id) = \bot$ and $\Gamma^P(id) = \bot$ and output $(\mathtt{CLOSED}, id) \overset{t_3}{\hookrightarrow} \mathcal{E}$.

D. Simplifying functionality description

The formal description of the functionality $\mathcal{F}(T, k)$ as presented in Section III is simplified. Namely, several natural checks that one would expect an ideal functionality to make when receiving a message are excluded from its description. For example a functionality should ignore a message that is malformed (e.g. missing or additional parameters), requests an update of a channel that was never created, etc. We define all those check using a wrapper $\mathcal{W}_{\text{checks}}(T, k)$.

Functionality wrapper: $\mathcal{W}_{\text{checks}}(T, k)$

Below, we abbreviate $\mathcal{F} := \mathcal{F}(T, k)$.
Create: Upon $(\mathtt{CREATE}, \gamma, tid) \overset{\tau_0}{\hookleftarrow} P$, where $P \in \gamma.\mathsf{users}$, check if: $\Gamma(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created; γ is valid according to the definition given in Section III-A; $\gamma.\mathsf{st} = \{(c_P, \mathtt{One\text{–}Sig}_{pk_P}), (c_Q, \mathtt{One\text{–}Sig}_{pk_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\ge 0}$; and there exists $(t, id, i, \theta) \in \mathcal{L}.\mathtt{UTXO}$ such that $\theta = (c_P, \mathtt{One\text{–}Sig}_P)$ for $(id, i) := tid$;[a] If one of the above checks fails, drop the message. Else proceed as $\mathcal{F}$.
Update: Upon $(\mathtt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$ check if: $\gamma := \Gamma(id) \neq \bot$; $P \in \gamma.\mathsf{users}$; there is no other update being preformed; let $\vec{\theta} = (\theta_1, \dots \theta_\ell) = ((c_1, \varphi_1), \dots, (c_\ell, \varphi_\ell))$, then $\sum_{j\in[\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If not, drop the message. Else proceed as $\mathcal{F}$.
Upon $(\mathtt{SETUP\text{–}OK}, id) \overset{\tau_2}{\hookleftarrow} P$ check if: you accepted a message $(\mathtt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$, where $t_2 - t_0 \le t_{\mathsf{stp}} + T$ and the message is a reply to the message $(\mathtt{SETUP}, id, \vec{tid})$ sent to P in round τ_1 such that $\tau_2 - \tau_1 \le t_{\mathsf{stp}}$[b]. If not, drop the message. Else proceed as $\mathcal{F}$.
Upon $(\mathtt{UPDATE\text{–}OK}, id) \overset{\tau_0}{\hookleftarrow} P$, check if the message is a reply to the message $(\mathtt{SETUP\text{–}OK}, id)$ sent to P in round τ_0. If not, drop the message. Else proceed as $\mathcal{F}$.
Upon $(\mathtt{REVOKE}, id) \overset{\tau_0}{\hookleftarrow} P$, check if the message is a reply to either the message $(\mathtt{UPDATE\text{–}OK}, id)$ sent to P in round τ_0 or the message $(\mathtt{REVOKE\text{–}REQ}, id)$ sent to P in round τ_0. If not, drop the message. Else proceed as $\mathcal{F}$.
Close: Upon $(\mathtt{CLOSE}, id) \overset{\tau_0}{\hookleftarrow} P$, check if $\gamma := \Gamma(id) \neq \bot$ and $P \in \gamma.\mathsf{users}$. If not, drop the message. Else proceed as $\mathcal{F}$.
All other messages are dropped.

[a] In case more channels are being created at the same time, then none of the other creation requests can use of the tid.
[b] See Appendix B what we formally meant by "reply".

E. Simplifying the protocol descriptions

Similarly as the descriptions of our ideal functionality, the description of the protocol Π presented in Appendix C excludes many natural checks that we would want an honest party to make. Let us give a few examples of requests which an honest party drops if received from the environment: (i) The environment sends a malformed message to a party P (e.g. missing or additional parameters); (ii) A party P receives an instruction to create a channel γ but $P \notin \gamma.\mathsf{users}$; (iii) A party P receives an instruction to create a channel using the UTXO defined by tid but this UTXO is not spendable by P etc. We define all those check as a wrapper $\mathcal{W}_{\text{checksP}}$.

Protocol wrapper: $\mathcal{W}_{\text{checksP}}$

Party $P \in \mathcal{P}$ proceeds as follows:
Create: Upon $(\mathtt{CREATE}, \gamma, tid) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$ check if: $P \in \gamma.\mathsf{users}$; $\Gamma^P(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created; γ is valid according to the definition given in Section III-A; $\gamma.\mathsf{st} = \{(c_P, \mathtt{One\text{–}Sig}_{pk_P}), (c_Q, \mathtt{One\text{–}Sig}_{pk_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\ge 0}$; there exists $(t, id, i, \theta) \in \mathcal{L}.\mathtt{UTXO}$ such that $\theta = (c_P, \mathtt{One\text{–}Sig}_P)$ for $(id, i) := tid$. If one of the above checks fails, drop the message. Else proceed as in Π.
Update: Upon $(\mathtt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$ check if: $\gamma := \Gamma^P(id) \neq \bot$; there is no other update being preformed; let $\vec{\theta} = (\theta_1, \dots \theta_\ell) = ((c_1, \varphi_1), \dots, (c_\ell, \varphi_\ell))$, then $\sum_{j\in[\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If on of the checks fails, drop the message. Else proceed as in Π. Upon $(\mathtt{SETUP\text{–}OK}, id) \overset{\tau_2}{\hookleftarrow} \mathcal{E}$ check if: you accepted a message $(\mathtt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, where $t_2 - t_0 \le t_{\mathsf{stp}} + T$ and the message is a reply to the message $(\mathtt{SETUP}, id, \vec{tid})$ you sent in round τ_1 such that $\tau_2 - \tau_1 \le t_{\mathsf{stp}}$[a]. If not, drop the message. Else proceed as in Π.
Upon $(\mathtt{UPDATE\text{–}OK}, id) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if the message is a reply to the message $(\mathtt{SETUP\text{–}OK}, id)$ you sent in round τ_0. If not, drop the message. Else proceed as in Π.
Upon $(\mathtt{REVOKE}, id) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if the message is a reply to either $(\mathtt{UPDATE\text{–}OK}, id)$ or $(\mathtt{REVOKE\text{–}REQ}, id)$ you sent in round τ_0. If not, drop the message. Else proceed as in Π.
Create: Upon $(\mathtt{CLOSE}, id) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if $\gamma := \Gamma^P(id) \neq \bot$. If not, drop the message. Else proceed as in Π.
All other messages are dropped.

[a] See Appendix B what we formally meant by "reply".