Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures

Lukas Aumayr[1(✉)], Oguzhan Ersoy[2(✉)], Andreas Erwig[3(✉)], Sebastian Faust[3(✉)], Kristina Hostáková[4(✉)], Matteo Maffei[1(✉)], Pedro Moreno-Sanchez[5(✉)], and Siavash Riahi[3(✉)]

[1] Technische Universität Wien, Vienna, Austria
{Lukas.Aumayr,Matteo.Maffei}@tuwien.ac.at
[2] Delft University of Technology, Delft, Netherlands
o.ersoy@tudelft.nl
[3] Technische Universität Darmstadt, Darmstadt, Germany
{Andreas.Erwig,Sebastian.Faust,Siavash.Riahi}@tu-darmstadt.de
[4] ETH Zürich, Zürich, Switzerland
kristina.hostakova@inf.ethz.ch
[5] IMDEA Software Institute, Madrid, Spain
pedro.moreno@imdea.org

Abstract. Decentralized and permissionless ledgers offer an inherently low transaction rate, as a result of their consensus protocol demanding the storage of each transaction on-chain. A prominent proposal to tackle this scalability issue is to utilize off-chain protocols, where parties only need to post a limited number of transactions on-chain. Existing solutions can roughly be categorized into: (i) application-specific channels (e.g., payment channels), offering strictly weaker functionality than the underlying blockchain; and (ii) state channels, supporting arbitrary smart contracts at the cost of being compatible only with the few blockchains having Turing-complete scripting languages (e.g., Ethereum).

In this work, we introduce and formalize the notion of *generalized channels* allowing users to perform any operation supported by the underlying blockchain in an off-chain manner. Generalized channels thus extend the functionality of payment channels and relax the definition of state channels. We present a concrete construction compatible with any blockchain supporting transaction authorization, time-locks and constant number of Boolean $\wedge$ and $\vee$ operations – requirements fulfilled by many (non-Turing-complete) blockchains including the popular Bitcoin. To this end, we leverage *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 generalized channel construction in the Universal Composability framework.

As an important practical contribution, our generalized channel construction outperforms the state-of-the-art payment channel construction, the Lightning Network, in efficiency. Concretely, it halves the off-chain communication complexity and reduces the on-chain footprint in case of disputes from linear to constant in the number of off-chain applications funded by the channel. Finally, we evaluate the practicality of our

M. Tibouchi and H. Wang (Eds.): ASIACRYPT 2021, LNCS 13091, pp. 635–664, 2021.
https://doi.org/10.1007/978-3-030-92075-3_22

construction via a prototype implementation and discuss various applications including financially secured fair two-party computation.

Keywords: Blockchain · Adaptor signatures · Off-chain protocols and channels

1 Introduction

One of the most fundamental technical challenges of decentralized and permissionless blockchains is scalability. Since transactions are processed via a costly distributed consensus protocol run among a set of parties (so-called miners), transaction throughput is limited and transaction confirmation is slow. There has been a plethora of work on improving scalability of blockchains, with off-chain protocols being one of the most promising solutions.

Intuitively, off-chain protocols build a second layer over the blockchain (often referred to as the *layer-1*) by allowing the vast majority of transactions to be processed directly between the involved participants, with the blockchain being used only in the initial setup and in case of disputes, thereby drastically improving transaction throughput and confirmation time.

While there exists a large variety of different off-chain (or layer-2) solutions (see, e.g., [6,30,32,53] and many more), *payment channels* [10,19,47] are by far the most prominent one. Intuitively, a payment channel works in three phases. First, the two users *open* a channel by locking a certain amount of coins on-chain into an account controlled by both users. Then they perform an arbitrary amount of payments by exchanging authenticated messages *off-chain*. Finally, they *close* the channel by announcing the outcome of their trades to the ledger.

Off-chain computations in Ethereum. Ethereum supports on-chain transactions specified in a *Turing-complete scripting language*, which enables the execution of arbitrarily complex programs, also called smart contracts, thereby going beyond simple payments. The underlying blockchain is organized accordingly in the account-based model, in which the balance associated to an account is explicitly stored in its memory and programmatically updated via smart contracts. By leveraging the expressiveness of Turing-complete scripting languages, payment channels can be generalized into so-called *state channels* [22,23,43], whose functionality goes far beyond simple payments. Namely, state channels enable users to execute arbitrarily complex smart contracts in an off-chain manner, thereby making their execution faster and cheaper.

Turing-complete vs restricted scripting. The majority of current blockchains (e.g., Bitcoin, Zcash, Monero, and Cardano's ADA) only support a restricted scripting language and are based on the Unspent Transaction Output (UTXO) model: intuitively, they enable a restricted class of transactions, possibly conditioned to some events, that transfer money from an unspent transaction to a new unspent transaction. There are several reasons behind the choice of a limited scripting language. First, the simplicity of design and usage, which is believed to be beneficial for security: countless examples of smart contract vulnerabilities on Ethereum show that complex contract logic and increased expressiveness pave the

way for critical bugs, which may have severe consequences for the stability of the underlying currency as shown by the infamous DAO hack [48]. Second, blockchains with simple transaction logic are less costly to maintain: this is important as transaction execution is done by many parties, and even normal users. Finally, restricted scripting languages are expressive enough to encode many interesting computations (e.g., lotteries [2], auctions [21], and more [7,8,37]).

Unfortunately, current state channel constructions are not applicable without a Turing-complete scripting language, thereby excluding the majority of blockchains. In this work, we investigate the following question: *Can we generically lift any transaction logic offered by layer-1 to layer-2 even for blockchains with restricted transaction logic?* Besides its practical importance, we believe that this question is theoretically interesting. It may constitute a first step towards a more general research agenda exploring the feasibility (or impossibility) of generic off-chain computation from blockchains with limited expressiveness.

1.1 Our Contribution

Our main contribution is to put forward the notion of *generalized channels* – a generic extension of payment channels to support off-chain execution of *arbitrary transaction logic* supported by the underlying blockchain. State channels can hence be seen as a special case of generalized channels for blockchains with Turing-complete scripting languages. We briefly outline our main contributions below. A technical overview of our construction is given in Sect. 2.

Generalized Channels. We show that if the underlying UTXO-based blockchain supports transaction authorization, time-locks and basic Boolean logic (constant number of $\wedge$, $\vee$ operations), then *any* transaction logic available on layer-1 can be lifted to layer-2 securely and generically.

As most cryptocurrencies, including the by far most prominent Bitcoin, satisfy the assumptions of our construction, they can benefit from generalized channels as a scalability solution. This, in particular, implies that our construction directly enables to execute *any* Bitcoin transaction off-chain. Moreover, we stress that our construction can also be deployed over any blockchain that can simulate a UTXO-based system, which, in particular, includes blockchains with support for Turing-complete smart contracts, e.g., Ethereum or Hyperledger Fabric [1].

A novel revocation mechanism for generalized channels. The main technical challenge in our generalized channel design is to propose an efficient mechanism for old channel state revocation while putting minimal assumptions on the scripting language of the underlying blockchain. The state-of-the-art approach, put forward by the Lightning Network [47], uses a punishment mechanism which allows the cheated party to claim all coins from the channel. As we argue, a straightforward generalization of the Ligthning-style revocation is unsuitable for generalized channels. Firstly, the blockchain communication complexity in case of misbehavior depends on the number of parallel conditional payments funded by the channel. This significantly increases the blockchain overhead when processing a punishment (if triggered). Secondly, the security of the revocation mechanism relies on state duplication, hence each off-chain transaction funded by the channel has to be performed twice (once on each duplicate). This is particularly problematic when

channels are built on top of channels [26] as the off-chain communication complexity grows exponentially with the number of channel layers.

To overcome these drawbacks, we design a novel revocation mechanism reducing the on-chain complexity in case of a dispute from linear to constant, and the off-chain communication complexity from exponential to linear.

Formalization of adaptor signatures. A key idea of our novel revocation mechanism is to utilize an *adaptor signature scheme* [46] – a cryptographic primitive introduced by the cryptocurrency community to tie together the authorization of a transaction and the leakage of a secret value. Although adaptor signatures have been used in previous works (e.g. [29,41,45]), a formal definition has never been presented. We fill this gap by providing the first formalization of adaptor signatures and their security (in terms of cryptographic games), and proving that ECDSA and Schnorr-based schemes satisfy our notions. We believe that our formalization and security analysis of adaptor signatures is of independent interest (see details on the impact of our work below).

Formalization of generalized channels. In order to formally define the security guarantees of a generalized channel protocol, we utilize the extended Universal Composability model allowing for global setup (the GUC model for short) put forward by Canetti et al. [15]. More precisely, we model money mechanics of an UTXO-based blockchain via a global ledger ideal functionality and provide an ideal specification of a generalized channel protocol via a novel ideal functionality. Thereafter, we prove that our generalized channel construction satisfies this ideal specification. The key challenges of our security analysis are to ensure the consistency of timings imposed by the blockchain processing delay, and to ensure that no honest party can ever lose coins by participating in a channel.

Evaluation and applications. We implemented our protocols and conducted an experimental evaluation, demonstrating how to use generalized channels as a building block for popular off-chain applications, like payment routing through a payment channel network (PCN) [41,42,47] and channel splitting [26]. Concretely, our evaluation demonstrates that, already when routing *one* payment through a channel, the amount of blockchain fees in case of a dispute is reduced by 28% compared to the state-of-the-art Lightning network solution. In practice, there have been cases of disputes in channels with 50 concurrent payments [40], which currently costs 553.66 USD in fees to resolve in Lightning and only 17.47 USD with generalized channels. For channel splitting, we reduce the transactions to be exchanged off-chain per sub-channel from exponential to constant.

Moreover, we discuss how to use generalized channels to realize the Claim-or-Refund functionality of Bentov and Kumaresan [8]. This functionality, can be used to build a fair two-party computation protocol over Bitcoin, where fairness is achieved by financially penalizing malicious parties. Realizing the Claim-or-Refund functionality, in particular, implies that generalized channels allow parties to execute any two-party computation off-chain.

1.2 Other Related Work

We briefly discuss related work on off-chain protocols and adaptor signatures, where the latter is an important building block in our construction.

Off-chain protocols. As already mentioned before, there has been an extensive line of work on various types of payment channels [10,19,47] and payment channel networks (PCNs) [41,42,47]. However, these constructions only support simple payments and do not extend to support more complex transaction logic. The authors in [34] 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 here. Most related to our work is the research on state channels [22,23,43], as these constructions allow to lift any transaction logic supported by the underlying blockchain off-chain. However, state channels crucially rely on the underlying blockchain to support smart contracts and hence do not work for blockchains with restricted scripting language. Finally, eltoo [20] is a payment channel construction which does not rely on a punishment mechanism, yet requires Bitcoin to adapt a new scripting command (op-code). This op-code, however, has not been included to Bitcoin's scripting language in the past due to security concerns. In the case of address reuse or lazy wallet designs, funds can be stolen by replaying transactions [52]. Moreover, the security of the eltoo protocol has not been formally proven and it only supports simple payments.

Apart from payment and state channels, numerous other solutions have been proposed in order to perform heavy on-chain computation off-chain. For instance, various previous works (e.g., [17,18,35]) focus on realizing on-chain functionality off-chain by using Trusted Execution Environments which, however, inherently add an additional trust assumptions that may not hold in practice (e.g., [12,13,16]). A proposal to remove these assumptions is to use MPC protocols [8,37], which however require collateral linear in the number of conditional payments. In contrast, generalized channels only require constant collateral for the execution of an arbitrary number of such payments. There have been proposals to remedy the collateral requirement in MPC protocols [9,36,38] but they are incompatible with many existing UTXO blockchains, including Bitcoin.[1]

Adaptor signatures. Poelstra [46] introduced the notion of adaptor signatures (AS), which intuitively allows to create partial signatures whose completion is conditioned on solving a cryptographic hard problem – a feature that has been proven useful in off-chain applications such as PCNs [41] and payment-channel hubs [49]. For instance, Malavolta et al. [41] use AS as building block to define and realize multi-hop payments in PCNs. Moreover, AS have been used as an off-the-shelf cryptographic building block for multi-path payments [25] and Monero-compatible PCNs [51]. Banasik et al. [5] construct a scheme satisfying a similar notion to AS in order to allow two parties to exchange a digital asset using cryptocurrencies that do not support Turing-complete programs. None of these works, however, define AS as a stand-alone primitive. Concurrently to our work, Fournier [29] attempts to formalize AS as an instance of one-time verifiable encrypted signatures [11]. Yet, the definition of [29] is weaker than the one we give in this work and does not suffice for the channel applications. Also concurrent to this work, Thyagarajan and Malavolta [50] define *lockable signatures*. While

[1] These solutions require the underlying blockchain to either support verification of signatures on arbitrary messages or Turing-complete smart contracts.

similar to AS in spirit, lockable signatures are a weaker primitive as the partial signature must be created honestly (e.g., through MPC) and the solution to the cryptographic hardness problem must be known beforehand. On the other hand, lockable signatures can be built from any signature scheme while AS cannot be constructed from unique signatures [27].

2 Background and Solution Overview

Blockchain transactions. 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 a tuple (cash, φ), where cash denotes the amount of coins associated to the output and φ defines the conditions (also known as scripts) that need to be satisfied to spend the output.

A *transaction* transfers coins across outputs meaning that it maps (possibly multiple) existing outputs to a list of new outputs. The existing outputs that fund the transactions are called *transaction inputs*. In other words, transaction inputs are those tied with previously unspent outputs of older transactions. Formally, a transaction tx is a tuple of the form $(\mathsf{txid}, \mathsf{In}, \mathsf{Out}, \mathsf{Witness})$, where $\mathsf{txid} \in \{0,1\}^*$ is the unique identifier of tx and is calculated as $\mathsf{txid} := \mathcal{H}([\mathsf{tx}])$, where $\mathcal{H}$ is a hash function modeled as a random oracle and $[\mathsf{tx}]$ is the *body of the transaction* defined as $[\mathsf{tx}] := (\mathsf{In}, \mathsf{Out})$; In is a vector of strings identifying all transaction inputs; $\mathsf{Out} = (\theta_1, \ldots, \theta_n)$ is a vector of new outputs; and $\mathsf{Witness} \in \{0,1\}^*$ contains the witness allowing to spend the transaction inputs.

To ease the readability, we illustrate the transaction flows using charts (see Fig. 1 for examples). We depict transactions as rectangles 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 frequently used conditions, we define the following abbreviated notation. If the output script contains (among other conditions) signature verification w.r.t. some public keys $pk_1, \ldots, pk_n$ on the body of the spending transaction, 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 h, we express this by 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 "$+t$" *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 sub-conditions φ_i is then written above a separate arrow.

Payment channels. A payment channel [47] enables several payments between two users without submitting every single transaction to the blockchain. The

Fig. 1. (Left) 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 tx was accepted by the blockchain. (Right) tx' is not published yet. Its only output can be spent by a transaction whose witness satisfies $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

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.

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 (i.e., its spending condition mandates both Alice's and Bob's signature). 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 owned by Alice; and (ii) the other holding $x_B + \alpha$ coins owned by Bob. Finally, parties exchange the signatures on the commit transaction, thereby complete the channel *update*. Alice (resp. Bob) could now add $\mathtt{TX_c}$ to the blockchain. Instead, they keep it locally in their memory and overwrite it when they agree on another commit transaction, let us denote it $\overline{\mathtt{TX_c}}$, representing a newer channel state. This, however, leads to several commit transactions that can possibly be added to the blockchain. Since all of them are spending the same output, only one can be accepted. As it is impossible to prevent a malicious user from publishing an old commit transaction, payment channels require a mechanism punishing such behavior.

Lightning Network [47], the state-of-the-art payment channel for Bitcoin, implements such mechanism by introducing *two* commit transactions, denoted $\mathtt{TX_c^A}$ and $\mathtt{TX_c^B}$, 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 $\mathtt{TX_c^A}$ 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 $\mathtt{TX_c^A}$ with the built-in punishment for Alice (analogously for Bob and $\mathtt{TX_c^B}$). During the next channel update, parties first commit to the new state by creating and signing $\overline{\mathtt{TX_c^A}}$ and $\overline{\mathtt{TX_c^B}}$, and then *revoke* the old state by

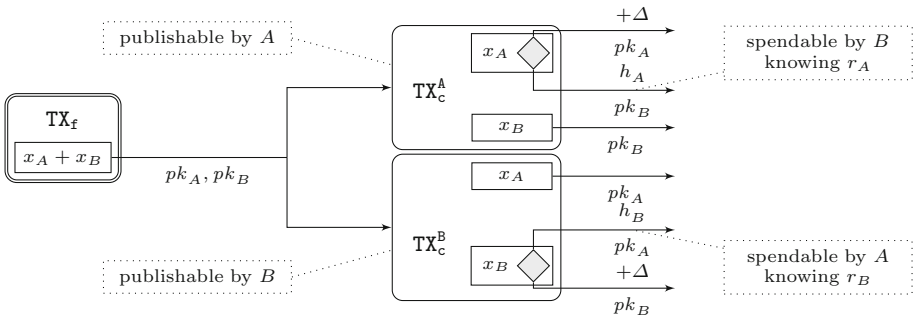

Fig. 2. A Lightning style payment channel where A has x_A coins and B has x_B coins. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. Δ upper bounds the time needed to publish a transaction on a blockchain.

sending the revocation secrets to each other thereby enabling the punishment mechanism. If a malicious Alice now publishes the old commit transaction $\mathtt{TX_c^A}$, Bob can spend both of its outputs and claim all coins locked in the channel.

2.1 Solution Overview

The goal of our work is to extend the idea of payment channels such that parties can agree on *any* conditional payment that they could do on-chain and not only direct payments. 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.

Revocation per update. The first idea would be to extend the revocation mechanism explained above such that *each output* of $\mathtt{TX_c^A}$ contains a punishment mechanism for Alice (analogously for Bob). While this solution works, it has several disadvantages. If one party, say Alice, cheats and publishes an old commit transaction $\mathtt{TX_c^A}$, Bob has to spend all outputs of $\mathtt{TX_c^A}$ to punish Alice. 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. Moreover, such revocation mechanism requires a high on-chain footprint not only for $\mathtt{TX_c^A}$, but also for Bob getting 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 $\mathtt{TX_c^A}$ 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*

$\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ owned 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 $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$.

One commit transaction 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 undesirable 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 other off-chain applications, e.g., a fair two-party computation or another off-chain channel as we discuss later in this work (see Sect. 7). Such off-chain application would, however, have to "exist" twice. Once considering $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ being eventually published on-chain and once considering $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$. Especially when considering channels built on top of channels, the overhead grows exponentially. Our goal is to construct generalized channels that require only one commit transaction and hence avoid any redundancy.

A naive approach to design such a single commit transaction $\mathtt{TX}_{\mathtt{c}}$ would be to "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. Unfortunately, this simple proposal 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. To prevent such undue punishment of honest Bob, we need to make sure that Alice can use the punishment mechanism only if Bob published $\mathtt{TX}_{\mathtt{c}}$.

The main idea of how to implement this additional requirement is to force the party publishing $\mathtt{TX}_{\mathtt{c}}$ to reveal some secret, which we call *publishing secret*, that the other party could use as 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 (at a high level, a statement/witness relation is hard, if given a statement Y is it computationally hard to find a witness y). Such pre-signature can be adapted into a valid signature by anyone knowing a witness for the statement Y. Also, it is possible to extract a witness y for Y by knowing both the pre-signature and the adapted full signature. In our context, adaptor signatures allow users of a generalized channel to express the following: "I give you my *pre-signature* on $\mathtt{TX}_{\mathtt{c}}$ that you can turn into a full signature and publish $\mathtt{TX}_{\mathtt{c}}$, which 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 statement/witness pairs used for

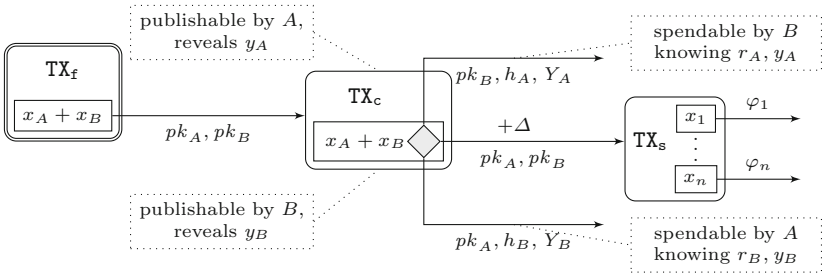

Fig. 3. A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. In the figure, pk_A denotes Alice's public key, (h_A, r_A) her revocation public/secret values, and (Y_A, y_A) her publishing public/secret values (analogously for Bob). The value of Δ upper bounds the time needed to publish a transaction on a blockchain.

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 (not only in our construction but also in the Lightning-style channels) since the punishment output script will only consist of signature verification, thereby requiring less complex scripting language. As a result, our solution does not only work over Bitcoin, but over any UTXO based blockchain that supports transaction authorization (if there exists an adaptor signature scheme w.r.t. the considered digital signature), relative time-locks and constant number of $\wedge$ and $\vee$ in output scripts.

3 Preliminaries

We denote by $x \leftarrow_{\$} \mathcal{X}$ the uniform sampling of the variable x from the set $\mathcal{X}$. Throughout this paper, n denotes the security parameter and all our algorithms run in polynomial time in n. By writing $x \leftarrow \mathsf{A}(y)$ we mean that a *probabilistic polynomial time* algorithm A (or PPT for short) on input y, outputs x. If A is a *deterministic polynomial time* algorithm (DPT for short), we use the notation $x := \mathsf{A}(y)$. A function $\nu\colon \mathbb{N} \to \mathbb{R}$ is *negligible in* n if for every $k \in \mathbb{N}$, there exists $n_0 \in \mathbb{N}$ s.t. for every $n \geq n_0$ it holds that $|\nu(n)| \leq 1/n^k$. 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 $T.\mathsf{attr}$. Let us now briefly recall the cryptographic primitives used in this paper to establish the used notation.

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 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 $\mathsf{SUF\text{–}CMA}$). At a high level, $\mathsf{SUF\text{–}CMA}$ guarantees that a PPT adversary on input the public key pk and with access to a signing oracle, cannot produce a *new* valid signature on any message m.

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 $\{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. We say that R is a *hard relation* if the following holds: (i) There exists a PPT sampling algorithm GenR 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 a valid witness y is negligible.

Finally, we recall the definition of a non-interactive zero-knowledge proof of knowledge (NIZK) with online extractors as introduced in [28]. 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 need this property to prove our ECDSA-based adaptor signature scheme secure. More formally, a pair (P, V) of PPT algorithms is called a NIZK 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, 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 sequence of queries to the random oracle and its answers, such that given (Y, π), the algorithm K can extract the witness y with $(Y, y) \in R$. An instance of such proof system is in [28].

4 Generalized Channels

4.1 Notation and Security Model

To formally model the security of generalized channels, we use the global UC framework (GUC) [15] which extends the standard UC framework [14] by allowing for a global setup. Here we discuss our security model (which follows the previous works on off-chain channels [22–24]), only briefly and refer the reader to the full version of this paper [4] for more details.

We consider a protocol π that runs between parties from a fixed set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ who can *corrupt* any party P_i at the beginning of the protocol execution (so-called static corruption). Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{Z}$ – which represents anything "external" to the current protocol execution. We assume a synchronous communication network meaning that protocol execution happens in rounds, formalized via a global ideal functionality $\mathcal{F}_{clock}$ representing "the clock" [33]. Parties in the protocol are connected with authenticated communication channels with guaranteed delivery of exactly one round, formalized via an ideal functionality $\mathcal{F}_{GDC}$. For simplicity, we assume that all other communication (e.g., messages sent between the adversary

and the environment) as well as local computation take zero rounds. Monetary transactions are handled by a global ideal ledger functionality $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$, where Δ is an upper bound on the blockchain delay (number of rounds it takes to publish a transaction), Σ defines the signature scheme and $\mathcal{V}$ defines valid output conditions. Furthermore, the global ledger maintains a PKI.

Generalized channel syntax. A *generalized channel* γ is an attribute tuple $(\gamma.\mathsf{id}, \gamma.\mathsf{users}, \gamma.\mathsf{cash}, \gamma.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the channel identifier, $\gamma.\mathsf{users} \in \mathcal{P} \times \mathcal{P}$ defines the identities of the channel users, $\gamma.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ represents the total amount of coins locked in γ, and $\gamma.\mathsf{st} = (\theta_1, \ldots, \theta_n)$ is the state of γ 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 $\theta_i.\varphi\colon \{0,1\}^* \to \{0,1\}$ defining the spending condition. For convenience, we use $\gamma.\mathsf{otherParty}\colon \gamma.\mathsf{users} \to \gamma.\mathsf{users}$ defined as $\gamma.\mathsf{otherParty}(P) := Q$ for $\gamma.\mathsf{users} = \{P, Q\}$.

4.2 Ideal Functionality

We capture the desired functionality of a generalized channel protocol as an ideal functionality $\mathcal{F}$. As a first step towards defining our functionality, we informally identify the most important security and efficiency notions of interest that a generalized channel protocol should provide.

Consensus on creation: A generalized channel γ is successfully created only if all parties in $\gamma.\mathsf{users}$ agree with the creation. Moreover, parties in $\gamma.\mathsf{users}$ reach agreement whether the channel is created or not after an a-priori bounded number of rounds.

Consensus on update: A generalized channel γ is successfully updated only if both parties in $\gamma.\mathsf{users}$ agree with the update. Moreover, parties in $\gamma.\mathsf{users}$ reach agreement whether the update is successful or not after an a-priori bounded number of rounds.

Instant finality with punish: An honest party $P \in \gamma.\mathsf{users}$ has the guarantee that either the current state of the channel can be enforced on the ledger, or P can enforce a state where she gets all $\gamma.\mathsf{cash}$ coins. A state st is called *enforced* on the ledger if a transaction with this state appears on the ledger.

Optimistic update: If both parties in $\gamma.\mathsf{users}$ are honest, the update procedure takes a constant number of rounds (independent of the blockchain delay Δ).

Having the guarantees identified above in mind, we now design our ideal functionality $\mathcal{F}$. It interacts with parties from the set $\mathcal{P}$, with the adversary $\mathcal{S}$ (called the simulator) and the ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$. 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 which is modeled via the interaction with $\mathcal{S}$. Finally, $\mathcal{F}$ observes the ledger and can verify that a certain transaction appeared on-chain or the ownership of coins.

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. At 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 Sect. 2, in this work we present a protocol realizing the functionality for $k = 1$ (see Fig. 3). A generalized channel construction using Lightning style revocation mechanism (see Fig. 2) would be a candidate protocol for $k = 2$.

We assume that the functionality maintains a set Γ of created channels in their latest state and the corresponding funding transaction tx. We present $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$ formally in Fig. 4. Here we discuss each part of the functionality at a high level and argue why it captures the aforementioned security and efficiency properties identified above. We abbreviate $\mathcal{F} := \mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$.

Create. If $\mathcal{F}$ receives a message of the form $(\texttt{CREATE}, \gamma, tid_P)$ 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 a transaction must spend both funding sources (defined by transaction identifiers tid_P, tid_Q) and contain one output of the value $\gamma.\mathsf{cash}$. If this is true, $\mathcal{F}$ stores this transaction together with the channel γ in Γ and informs both parties about the successful channel creation via the message `CREATED` (how this can be done within the UC model is discussed in the full version of this paper [4]). Since a `CREATE` message is required from both parties and both parties receive `CREATED`, "consensus on creation" holds.

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. This means 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. If only one of the parties requests closing, $\mathcal{F}$ executes the `ForceClose` subprocedure in which case such transaction is supposed to appear on $\mathcal{L}$ within 3Δ rounds modelling possible dispute resolution. In both cases, if the funding transaction is not spent before a certain round, an `ERROR` is returned to both users.

Update. The channel update is initiated by one of the parties P (called the *initiating party*) via a message $(\texttt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}})$. 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 set up off-chain applications (e.g., new channels or fair two-party computation) 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 set up the off-chain applications 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 the 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 $\mathcal{S}$.[2] 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. 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, 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`. 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 $\mathcal{F}$ updates the channel state in Γ accordingly and informs both parties about the successful update 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" as it ensures termination of the update process and allows both parties see the state in which the channel was closed. Moreover, in case both parties are honest, the update duration is independent of the ledger delay Δ, hence the efficiency property "optimistic update" is satisfied.

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.

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

Notation used in the formal description in Fig. 4. Messages sent between parties and $\mathcal{F}$ have the following format: (`MESSAGE_TYPE`, *parameters*). To shorten the description, we use following arrow notation: by $m \overset{t}{\hookrightarrow} P$, we mean "send the message m to party P in round t." and by $m \overset{t}{\hookleftarrow} P$, we mean "receive a message m from party P in round t". To indicate that a message should be sent/received before/after a certain round, we use inequality symbols above the arrows. When $\mathcal{F}$ expects $\mathcal{S}$ to set certain values (such as the vector of *tid*'s during the update

[2] 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).

process or the exact round in which a message should be sent to parties) and it does not do so, we implicitly assume that ERROR is returned. 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 $\mathcal{S}$. In the formal description, we treat the channel set Γ as a function which on input id outputs (X, tx), where X is a set of channels s.t. for every $\gamma \in X$ $\gamma.\mathsf{id} = id$, if such channel exists and $\perp$ otherwise. We denote the script requiring signature of (only) P as $\texttt{One–Sig}_{pk_P}$. Moreover, we omit several natural checks that one would expect $\mathcal{F}$ to make. For example, messages with missing parameters should be ignored, channel instruction should be accepted only from channel users, etc. We formally define all checks as a functionality wrapper in the full version of this paper [4]. Finally, we omit the read queries that $\mathcal{F}$ sends to $\mathcal{L}$ in order to learn its state.

5 Adaptor Signatures

Our goal is to realize the ideal functionality for generalized channel for $k = 1$, meaning that there is only one way to publish the channel state on-chain. As explained at a high level in Sect. 2.1, we achieve our goal by utilizing an adaptor signature scheme – a cryptographic primitive that we discuss in this section.

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 digital 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, 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. [29,41,45]), none of these works has given a formal definition of the adaptor signature primitive and its security. In the following, we fill this gap and provide the first game-based formalization of adaptor signatures. As already mentioned, Erwig et al. [27] recently extended our definition to a two-party case.

Definition 1 (Adaptor signature scheme). *An adaptor signature scheme w.r.t. a hard relation R and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ consists of four algorithms $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ with the following syntax: $\mathsf{pSign}_{sk}(m, Y)$ 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; and $\mathsf{Ext}(\sigma, \tilde{\sigma}, Y)$ 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 $\perp$.*

Upon $(\texttt{CREATE}, \gamma, tid_P) \xleftarrow{\tau_0} P$, distinguish:

Both agreed: If already received $(\texttt{CREATE}, \gamma, tid_Q) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T$: If tx s.t. $\mathsf{tx.In} = (tid_P, tid_Q)$ and $\mathsf{tx.Out} = (\gamma.\mathsf{cash}, \varphi)$, for some φ, appears on $\mathcal{L}$ in round $\tau_1 \leq \tau + \Delta + T$, set $\Gamma(\gamma.\mathsf{id}) := (\{\gamma\}, \mathsf{tx})$ and $(\texttt{CREATED}, \gamma.\mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$. Else stop.

Wait for Q**:** Else wait if $(\texttt{CREATE}, id) \xleftarrow{\tau \leq \tau_0 + T} Q$ (in that case "Both agreed" option is executed). If such message is not received, stop.

Upon $(\texttt{UPDATE}, id, \boldsymbol{\theta}, t_{\mathsf{stp}}) \xleftarrow{\tau_0} P$, parse $(\{\gamma\}, \mathsf{tx}) := \Gamma(id)$, set $\gamma' := \gamma$, $\gamma'.\mathsf{st} := \boldsymbol{\theta}$:

1. In round $\tau_1 \leq \tau_0 + T$, let $\mathcal{S}$ define $\boldsymbol{tid}$ s.t. $|\boldsymbol{tid}| = k$. Then $(\texttt{UPDATE–REQ}, id, \boldsymbol{\theta}, t_{\mathsf{stp}}, \boldsymbol{tid}) \xhookrightarrow{\tau_1} Q$ and $(\texttt{SETUP}, id, \boldsymbol{tid}) \xhookrightarrow{\tau_1} P$.
2. If $(\texttt{SETUP–OK}, id) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} P$, then $(\texttt{SETUP–OK}, id) \xhookrightarrow{\tau_3 \leq \tau_2 + T} Q$. Else stop.
3. If $(\texttt{UPDATE–OK}, id) \xleftarrow{\tau_3} Q$, then $(\texttt{UPDATE–OK}, id) \xhookrightarrow{\tau_4 \leq \tau_3 + T} P$. Else distinguish:
   - If Q honest or if instructed by $\mathcal{S}$, stop *(reject)*.
   - Else set $\Gamma(id) := (\{\gamma, \gamma'\}, \mathsf{tx})$, run $\texttt{ForceClose}(id)$ and stop.
4. If $(\texttt{REVOKE}, id) \xleftarrow{\tau_4} P$, send $(\texttt{REVOKE–REQ}, id) \xhookrightarrow{\tau_5 \leq \tau_4 + T} Q$.
   Else set $\Gamma(id) := (\{\gamma, \gamma'\}, \mathsf{tx})$, run $\texttt{ForceClose}(id)$ and stop.
5. If $(\texttt{REVOKE}, id) \xleftarrow{\tau_5} Q$, $\Gamma(id) := (\{\gamma'\}, \mathsf{tx})$, send $(\texttt{UPDATED}, id, \boldsymbol{\theta}) \xhookrightarrow{\tau_6 \leq \tau_5 + T} \gamma.\mathsf{users}$ and stop *(accept)*. Else set $\Gamma(id) := (\{\gamma, \gamma'\}, \mathsf{tx})$, run $\texttt{ForceClose}(id)$ and stop.

Upon $(\texttt{CLOSE}, id) \xleftarrow{\tau_0} P$, distinguish: **Both agreed:** If already received $(\texttt{CLOSE}, id) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T$, run $\texttt{ForceClose}(id)$ unless both parties are honest. In this case let $(\{\gamma\}, \mathsf{tx}) := \Gamma(id)$ and distinguish:

- If tx', with $\mathsf{tx}'.\mathsf{In} = \mathsf{tx.txid}$ and $\mathsf{tx}'.\mathsf{Out} = \gamma.\mathsf{st}$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + \Delta$, set $\Gamma(id) := \bot$, send $(\texttt{CLOSED}, id) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ and stop.
- Else output $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop.

Wait for Q**:** Else wait if $(\texttt{CLOSE}, id) \xleftarrow{\tau \leq \tau_0 + T} Q$ (in that case "Both agreed" option is executed). If such message is not received, run $\texttt{ForceClose}(id)$ in round $\tau_0 + T$.

At the end of every round τ_0: For each $id \in \{0,1\}^*$ s.t. $(X, \mathsf{tx}) := \Gamma(id) \neq \bot$, check if $\mathcal{L}$ contains tx' with $\mathsf{tx}'.\mathsf{In} = \mathsf{tx.txid}$. If yes, then define $S := \{\gamma.\mathsf{st} \mid \gamma \in X\}$, $\tau := \tau_0 + 2\Delta$ and distinguish: **Close:** If tx'' s.t. $\mathsf{tx}''.\mathsf{In} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{Out} \in S$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, set $\Gamma(id) := \bot$ and $(\texttt{CLOSED}, id) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ if not sent yet.

Punish: If tx'' s.t. $\mathsf{tx}''.\mathsf{In} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{Out} = (\gamma.\mathsf{cash}, \texttt{One–Sig}_{pk_P})$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, for P honest, set $\Gamma(id) := \bot$, $(\texttt{PUNISHED}, id) \xhookrightarrow{\tau_1} P$ and stop.

Error: Else $(\texttt{ERROR}) \xhookrightarrow{\tau} \gamma.\mathsf{users}$.

$\texttt{ForceClose}(id)$: Let τ_0 be the current round and $(X, \mathsf{tx}) := \Gamma(id)$. If within Δ rounds tx is still unspent on $\mathcal{L}$, then $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop. *Note that otherwise, message* $m \in \{\texttt{CLOSED}, \texttt{PUNISHED}, \texttt{ERROR}\}$ *is output latest in round* $\tau_0 + 3 \cdot \Delta$.

Fig. 4. The ideal functionality $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T, k)$. We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$.

An adaptor signature scheme $\Xi_{R,\Sigma}$ *must satisfy* pre-signature correctness *stating that for every* $m \in \{0,1\}^*$ *and every* $(Y,y) \in R$*, the following holds:*

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

The first security property, *existential unforgeability under chosen message attack for adaptor signature* (aEUF–CMA security for short), protects the signer. It is similar to EUF–CMA for digital signatures 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 stress that allowing the adversary to learn a pre-signature on the forgery message m is crucial since, for our applications, signature unforgeability needs to hold even in case the adversary learns a pre-signature for m without knowing a witness for Y.

Definition 2 (Existential unforgeability). *An adaptor signature scheme* $\Xi_{R,\Sigma}$ *is* aEUF–CMA *secure if for every PPT adversary* $\mathcal{A}=(\mathcal{A}_1,\mathcal{A}_2)$ *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:*

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

1 : $\mathcal{Q} := \emptyset, (sk,pk) \leftarrow \mathsf{Gen}(1^n)$
2 : $(Y,y) \leftarrow \mathsf{GenR}(1^n)$
3 : $(m,\mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}(\cdot),\mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(pk,Y)$
4 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m,Y)$
5 : $\sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}(\cdot),\mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\tilde{\sigma},\mathsf{st})$
6 : **return** $\left(m \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{pk}(m;\sigma)\right)$

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

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

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

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

The reason why the game computes $\tilde{\sigma}$ in step 4 (although $\mathcal{A}$ could obtain it by querying $\mathcal{O}_{\mathrm{pS}}$) is that it allows $\mathcal{A}$ to learn $\tilde{\sigma}$ without m being added to $\mathcal{Q}$.

The second property, called *pre-signature adaptability*, protects the verifier. It guarantees that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be completed into a valid signature using a witness y with $(Y,y) \in R$. Notice that this property is stronger than the pre-signature correctness property from Definition 1, since we require that even pre-signatures that were not produced by pSign but are valid, can be completed into valid signatures.

Definition 3 (Pre-signature adaptability). *An adaptor signature scheme* $\Xi_{R,\Sigma}$ *satisfies pre-signature adaptability if for any message* $m \in \{0,1\}^*$*, any statement/witness pair* $(Y,y) \in R$*, any public key* pk *and any pre-signature* $\tilde{\sigma} \in \{0,1\}^*$ *with* $\mathsf{pVrfy}_{pk}(m,Y;\tilde{\sigma})=1$*, we have* $\mathsf{Vrfy}_{pk}(m;\mathsf{Adapt}(\tilde{\sigma},y))=1$*.*

The last property that we are interested in is *witness extractability* which protects the signer. Informally, it guarantees that a valid signature/pre-signatue pair $(\sigma,\tilde{\sigma})$ for message/statement (m,Y) can be used to extract a witness y for Y. Hence a malicious verifier cannot use a pre-signature $\tilde{\sigma}$ to produce a valid signature σ without revealing a witness for Y.

Definition 4 (Witness extractability). *An adaptor signature scheme* $\Xi_{R,\Sigma}$ *is* witness extractable *if for every PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, there exists a negligible function* ν *such that the following holds:* $\Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(n) = 1] \leq \nu(n)$*, where the experiment* $\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}$ *is defined as follows*

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

1 : $\mathcal{Q} := \emptyset, (sk, pk) \leftarrow \mathsf{Gen}(1^n)$
2 : $(m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}(\cdot),\mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(pk)$
3 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y)$
4 : $\sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}(\cdot),\mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\tilde{\sigma}, \mathsf{st})$
5 : **return** $((Y, \mathsf{Ext}_{pk}(\sigma, \tilde{\sigma}, Y)) \notin R \wedge m \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{pk}(m; \sigma))$

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

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

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

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

Let us stress that while the 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 they know a witness for Y so they 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 5. *An adaptor signature scheme* $\Xi_{R,\Sigma}$ *is secure, if it is* aEUF–CMA *secure, pre-signature adaptable and witness extractable.*

Note that none of the security definitions explicitly states that pre-signatures are unforgeable. However, it is implied by the definitions as we discuss in the full version of this paper [4].

5.1 ECDSA-based Adaptor Signature

We now construct a provably secure adaptor signature scheme based on ECDSA digital signatures that are commonly used by blockchains. The construction presented here is similar to the construction put forward by [45], however some modifications are needed for the security proof. In addition to the ECDSA-based adaptor signature scheme presented here, we show a scheme based on Schnorr digital signatures, including correctness and security proofs, in the full version of this paper [4].

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 modeled as a random oracle and $f\colon \mathbb{G} \to \mathbb{Z}_q$ (i.e., f is typically defined as the projection to the x-coordinate since in ECDSA the group $\mathbb{G}$ consists of elliptic curve points). 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

$\mathsf{pSign}_{sk}(m, I_Y)$	$\mathsf{pVrfy}_{pk}(m, I_Y; \tilde{\sigma})$	$\mathsf{Adapt}(\tilde{\sigma}, y)$	$\mathsf{Ext}(\sigma, \tilde{\sigma}, I_Y)$
$x := sk, (Y, \pi_Y) := I_Y$	$X := pk, (Y, \pi_Y) := I_Y$	$(r, \tilde{s}, K, \pi) := \tilde{\sigma}$	$(r, s) := \sigma$
$k \leftarrow_{\$} \mathbb{Z}_q, \tilde{K} := g^k$	$(r, \tilde{s}, K, \pi) := \tilde{\sigma}$	$s := \tilde{s} \cdot y^{-1}$	$(\tilde{r}, \tilde{s}, K, \pi) := \tilde{\sigma}$
$K := Y^k, r := f(K)$	$u := \mathcal{H}(m) \cdot \tilde{s}^{-1}$	**return** (r, s)	$y' := s^{-1} \cdot \tilde{s}$
$\tilde{s} := k^{-1}(\mathcal{H}(m) + rx)$	$v := r \cdot \tilde{s}^{-1}$		**if** $(I_Y, y') \in R'_g$
$\pi \leftarrow \mathsf{P}_Y((\tilde{K}, K), k)$	$K' := g^u X^v$		**then return** y'
return $(r, \tilde{s}, K, \pi)$	**return** $((r = f(K)) \wedge \mathsf{V}_Y((K', K), \pi))$		**else return** $\perp$

Fig. 5. ECDSA-based adaptor signature scheme.

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 $\mathsf{SUF\text{–}CMA}$ 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 [5,39]. This slightly modified ECDSA scheme is not only assumed to be $\mathsf{SUF\text{–}CMA}$ 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. As the ECDSA verification accepts valid positive ECDSA signatures, these signatures can be used by any blockchain that uses ECDSA, e.g., Bitcoin.

The adaptor signature scheme in [45] is presented w.r.t. 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 is rather involved, 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. 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 $\mathsf{aSigForge}$ (resp. $\mathsf{aWitExt}$) experiment. Having $\mathcal{A}$, we want to design a PPT adversary (also called the simulator) $\mathcal{S}$ that breaks the $\mathsf{SUF\text{–}CMA}$ security. The main technical challenge in both reductions is that $\mathcal{S}$ has to answer queries (m, Y) to the pre-signing oracle $\mathcal{O}_{\mathrm{pS}}$ 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 be a relation whose statements are *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 the paper [4] 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. 5 is a secure adaptor signature scheme in the ROM.*

6 Generalized Channel Construction

We now present a concrete protocol, denoted Π, that requires only one commit transaction, i.e., implements the punish-then-split mechanism. 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. Throughout this section, 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 distribution of $(pk, sk) \leftarrow \mathsf{Gen}$. We emphasize that both ECDSA and Schnorr based adaptor signatures satisfy this condition as discussed in the full version of the paper [4], where we also explain how to modify our protocol when this condition does not hold. Our protocol consists of four subprotocols: Create, Update, Close and Punish. We discuss each subprotocol separately at a high level here and refer the reader to the full version of the paper [4] for the pseudo-code description.

Channel creation. In order to create a channel γ, the users of the channel, say 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

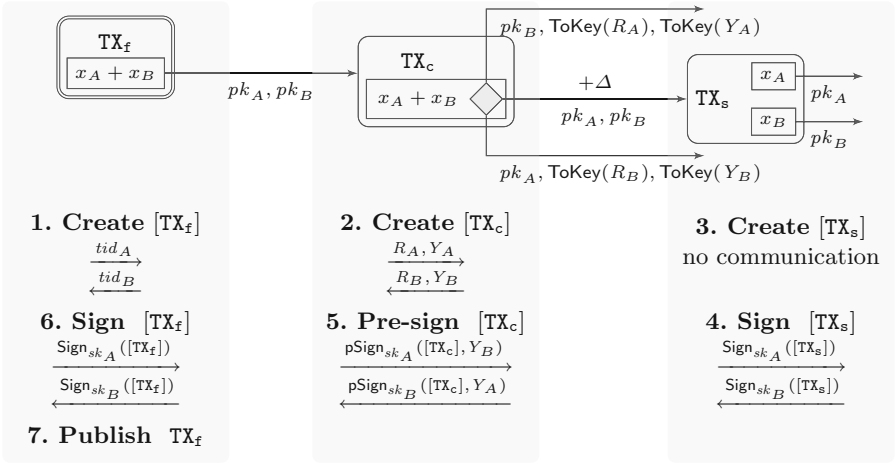

Fig. 6. Schematic description of the generalized channel creation protocol.

ledger. Recall that $\texttt{One–Sig}_{pk}$ represents the script that verifies that the transaction is correctly signed w.r.t. the public key pk. Once $\texttt{TX}_\texttt{f}$ is published, the channel creation is completed. Looking at Fig. 6, one can summarize the creation process as a step-by-step creation of transaction bodies from left to right, and then a 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 $[\texttt{TX}_\texttt{f}]$, 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 the body of the funding transaction $[\texttt{TX}_\texttt{f}]$ with $\{tid_A, tid_B\}$ as input and output requiring the signature of both A and B. **Step 2:** Parties can now start committing to the initial channel state. To this end, each party $P \in \{A, B\}$ generates a *revocation public/secret* pair $(R_P, r_P) \leftarrow \mathsf{GenR}$ and *publishing public/secret* pair $(Y_P, y_P) \leftarrow \mathsf{GenR}$, and sends the public values R_P, Y_P to the other party. Parties can now locally generate $[\texttt{TX}_\texttt{c}]$ which spends $\texttt{TX}_\texttt{f}$ and can be spent by a transaction satisfying one of these conditions:

Punish A: It is correctly signed w.r.t. $pk_B, \mathsf{ToKey}(Y_A), \mathsf{ToKey}(R_A)$;
Punish B: It is correctly signed w.r.t. $pk_A, \mathsf{ToKey}(Y_B), \mathsf{ToKey}(R_B)$;
Channel state: It is correctly signed w.r.t. pk_A and pk_B, and at least Δ rounds have passed since $\texttt{TX}_\texttt{c}$ was published.

Steps 3+4: Using the transaction identifier of $\texttt{TX}_\texttt{c}$, parties can generate and exchange signatures on the body of the split transaction $\texttt{TX}_\texttt{s}$ which spends $\texttt{TX}_\texttt{c}$ and whose output is equal to initial state of the channel $\gamma.\mathsf{st}$. **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 $[\texttt{TX}_\texttt{c}]$ and statement Y_B and sends the 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 in **Step 7**. If the funding transaction is accepted by the ledger, channel creation is successfully completed.

The question is what happens if one of the parties misbehaves during the creation process by aborting or sending a malformed message (w.l.o.g. let B be the malicious party). If the misbehavior happens before A sends her signature on $\texttt{TX}_\texttt{f}$ (i.e., before step 6), party A can safely conclude that the creation failed and does not need to take any action. If the misbehavior happens during step 6, A is in a hybrid situation. She cannot post $\texttt{TX}_\texttt{f}$ on-chain as she does not have B's signature needed to spend tid_B. However, since she already sent her signature on $\texttt{TX}_\texttt{f}$ to B, she has no guarantee that B will not post $\texttt{TX}_\texttt{f}$ later. To resolve this issue, our protocol instructs A to spend her output tid_A. Now within Δ rounds, tid_A is spent – either by the transaction posted by A (in which case creation failed) or by $\texttt{TX}_\texttt{f}$ posted by B (in which case creation succeeded).

To conclude, channel creation as described above takes 5 off-chain communication rounds and up to Δ rounds are needed to publish the funding transaction. Our formal protocol description contains two optimizations that reduce the number of off-chain communication rounds to 3. The optimizations are based on the observations that messages sent during steps 1 and 2 can be grouped into one as well as the messages sent during steps 4 and 5.

Channel closure. The purpose of the closing procedure is to collaboratively publish the latest channel state on the blockchain. The naive implementation is to let parties publish the latest agreed upon commit transaction and thereafter the corresponding split transaction representing the latest channel state. However, due to the punishment mechanism built-in the commit transaction, parties have to wait for Δ rounds after such a transaction is accepted by the ledger to publish the split transaction. To realize our ideal functionality, we need to design a more efficient solution eliminating the redundant waiting for honest parties.

When parties want to close a channel, they first run a "final update". In short, 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 $\texttt{TX}_\texttt{f}$ directly (i.e., **Steps 2+5** from Fig. 6 are skipped). Once parties jointly sign the split transaction, they can publish it on the ledger which completes the channel closure. If the final update fails, parties close the channel forcefully. Namely, they first publish the latest commit transaction, wait until the time for punishments expires. Then they post the split transaction representing the final channel state. It takes at most Δ rounds to publish the commit transaction and at most 2Δ rounds to publish the split transaction once the commit transaction is accepted which corresponds to the upper bound dictated by our ideal functionality. Since forceful closing might also be triggered during a channel update (as we discuss next), we define forceful closure as a separate subprocedure `ForceClose`.

Channel Update. To update a channel γ to a new state, given by a vector of output scripts $\vec{\theta}$, parties have to (i) agree on the new commit and split transaction capturing 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 the creation protocol (**Steps 2–5** in Fig. 6). 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).[3] In order to set up these 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 and set up all applications off-chain. Once this is done, which is signaled by "SETUP–OK" and takes at most t_{stp} rounds, parties execute the second part of the committing phase (**Steps 4+5**).

To realize part (ii), in which the punishment mechanism of the old commit transaction is activated, parties simply exchange the revocation secrets corresponding to the previous commit transaction which completes the update. Note that in this optimistic case when both parties are honest, the update is performed entirely off-chain and takes at most $5 + t_{\mathsf{stp}}$ rounds.

We now discuss what happens if one party misbehaves during the update. 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. This happens, e.g., 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 these situations, an honest party ends up in a hybrid state when the update is neither rejected nor accepted. In order to realize our ideal functionality requiring consensus on update in bounded number of rounds, our protocol instructs an honest party to `ForceClose` the channel. This means that the honest party posts the latest commit transaction that both parties agreed on to the ledger guaranteeing that $\mathtt{TX}_{\mathtt{f}}$ is spent within Δ rounds. If the transaction spending $\mathtt{TX}_{\mathtt{f}}$ is the new commit transaction, the channel is closed in the updated state. Otherwise, the update fails and either the channel is closed in the state before the update, or the punishment mechanism is activated and the honest party gets financially compensated (as discussed in the next paragraph).

Punish. Since we are in the UTXO model, nothing can stop a corrupted party from publishing an old commit transaction, thereby closing the channel in an old state. 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 $\overline{\mathtt{TX}}_{\mathtt{c}}$, it can react by publishing a *punishment transaction* which spends $\overline{\mathtt{TX}}_{\mathtt{c}}$ and assigns all coins to A. In order to make such punishment transaction valid, A must sign it under: (i) her secret key sk_A, (ii) B's publishing secret key $\bar{y}_B$, and (iii) B's revocation secret key $\bar{r}_B$. The knowledge of the revocation secret $\bar{r}_B$ follows from the fact that $\overline{\mathtt{TX}}_{\mathtt{c}}$ was old, i.e., parties revealed their revocation secrets to each other. The knowledge of the publishing secret $\bar{y}_B$ follows from the fact that it was B who published $\overline{\mathtt{TX}}_{\mathtt{c}}$. Let us elaborate on this in more detail. Since $\overline{\mathtt{TX}}_{\mathtt{c}}$ was accepted by the ledger, it had to include a signature of A.

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

The only signature A provided to B on $\overline{\mathtt{TX}}_\mathtt{c}$ was a *pre-signature* w.r.t. $\bar{Y}_B$. The unforgeability and witness extractability properties of $\Xi_{R,\Sigma}$ guarantee that the only way B could produce a valid signature of A on $\overline{\mathtt{TX}}_\mathtt{c}$ was by adapting the pre-signature and hence revealing the secret key $\bar{y}_B$ to A.

Security analysis. We now formally state our main 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. Let $\mathcal{L}(\Delta,\Sigma,\mathcal{V})$ be a ledger, where $\mathcal{V}$ allows for transaction authorization w.r.t. Σ, relative time-locks and constant number of Boolean operations $\wedge$ and $\vee$. Then the protocol Π UC-realizes the ideal functionality $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(3,1)$.*

The formal UC proof of the Theorem 2 can be found in the full version of this paper [4]. Let us here just argue at a high level, why our protocol satisfies the most complex property defined by the ideal functionality, i.e., instant finality with punishment.

We first argue that instant finality holds after the channel creation, meaning that each of the two parties (alone) is able to unlock her coins from a created channel if it was never updated. The pre-signature adaptability property of $\Xi_{R,\Sigma}$ guarantees that after a successful channel creation, each party P is able to adapt the pre-signature of the other party Q on $[\mathtt{TX}_\mathtt{c}]$ by using the publishing secret value y_P (corresponding to Y_P). Party P can now sign $[\mathtt{TX}_\mathtt{c}]$ herself and post $\mathtt{TX}_\mathtt{c}$ on the ledger. Since parties never signed any other transaction spending $\mathtt{TX}_\mathtt{f}$, the posted $\mathtt{TX}_\mathtt{c}$ will be accepted by the ledger within Δ rounds. Note that here we rely on the unforgeability of the signature scheme and the unforgeability of the adaptor signature scheme. Let us stress that parties have not revealed their revocation secrets, i.e., the values r_P and r_Q, to each other yet. Hardness of the relation R implies that none of the two parties is able to use the punishment mechanism of the published commit transaction. Thus, after Δ rounds, P can post the split transaction $\mathtt{TX}_\mathtt{s}$ on the ledger by which she unlocks her x_P coins.

After a successful update, each party P possesses a pre-signature of the other party Q on the new commit transaction $\mathtt{TX}_\mathtt{c}$ and the revocation secret of the other party on the previous commit transaction. The former implies that P is able to complete Q's pre-signature, sign $[\mathtt{TX}_\mathtt{c}]$ herself and post $\mathtt{TX}_\mathtt{c}$ on-chain. Assume first that the funding transaction of the channel $\mathtt{TX}_\mathtt{f}$ is not spent yet, hence $\mathtt{TX}_\mathtt{c}$ is accepted by the ledger within Δ rounds. Since party Q does not know the revocation secret of party P corresponding to $\mathtt{TX}_\mathtt{c}$, by hardness of the relation R, the only way how $\mathtt{TX}_\mathtt{c}$ can be spent is by publishing $\mathtt{TX}_\mathtt{s}$ representing the latest channel state. Hence, instant finality holds in this case.

Assume now that $\mathtt{TX}_\mathtt{f}$ is already spent and hence $\mathtt{TX}_\mathtt{c}$ is rejected by the ledger. The only transaction that could have spent $\mathtt{TX}_\mathtt{f}$ is one of the old commit transactions. This is because P never signed or pre-signed any other transaction spending $\mathtt{TX}_\mathtt{f}$. Let us denote the transaction spending $\mathtt{TX}_\mathtt{f}$ as $\overline{\mathtt{TX}}_\mathtt{c}$. Since $\overline{\mathtt{TX}}_\mathtt{c}$ is an old transaction P knows Q's revocation secret r_Q. Moreover, the extractability property of the adaptor signature scheme implies that P can extract Q's

publishing secret y_Q from the pre-signature that she gave to Q on this transaction and the completed signature contained in $\overline{\mathtt{TX}}_\mathtt{c}$. Hence, P can create a valid punishment transaction spending $\overline{\mathtt{TX}}_\mathtt{c}$. As our protocol instructs an honest party P to constantly monitor the blockchain and publish the punishment transaction immediately if $\overline{\mathtt{TX}}_\mathtt{c}$ appears on-chain, the punishment transaction will be accepted by the blockchain before the relative time-lock of $\overline{\mathtt{TX}}_\mathtt{c}$ expires. Hence, P receives all the coins locked in the channel which is what we needed to show.

7 Applications

Our generalized channels support a variety of applications such as PCNs [41,42,47], payment channel hubs [31,49], multi-path payments in PCNs [25], financially fair two-party computation [8], channel splitting [26], virtual payment channels [3] or watchtowers [44]. Furthermore, generalized channels prove to be highly versatile in interoperable applications, i.e., applications that run across multiple blockchains. As generalized channels rely only on on-chain signature verification, time-locked transactions and basic Boolean logic, they can be implemented on a multitude of different blockchains, easing thus the design and execution of cross-chain applications. Here, we first generally discuss which applications can be built on top of generalized channels and then focus on several concrete examples.

Suitable applications. We are interested in applications that are executed among two parties (i.e., two-party applications) and whose goal is to redistribute coins between them. We call the initial transaction outputs holding coins of the two parties the *funding source* of the application. If all outputs of the funding source are contained in already published transactions, we say that the application is *funded directly by the ledger*. If the outputs are part of a generalized channel state, we say that the application is *funded by a generalized channel*.

In principle, any two-party application that can be funded directly by the underlying ledger can also be funded by a generalized channel. There are, however, two subtleties one should keep in mind. Firstly, generalized channels provide "only" instant finality with punishment. This implies that generalized channels are suitable for two-party applications in which parties are willing to accept financial compensation in exchange for an off-chain state loss. Secondly, it takes up to 3Δ rounds to publish the funding source of the application. Hence, the protocol implementing the application needs to adjust the dispute timings accordingly (if applicable). We summarize this statement in the full version of this paper [4], where we also explain how to add applications to a generalized channel. Here we now discuss several concrete applications that benefit from generalized channels.

Fair two-party computation. One important example of an application that can be built on top of generalized channels is the *claim-or-refund* functionality introduced by Bentov and Kumaresan [8], and used in a series of work to realize multiple applications over Bitcoin [37]. At a high level, claim-or-refund allows one party, say A, to lock β coins that can be claimed by party B if she presents

a witness satisfying a condition f. After a predefined number of rounds, say t, the payment of β coins is refunded back to A if the witness is not revealed.

In their work, Bentov and Kumaresan demonstrated how to utilize this simple functionality to realize secure two-party protocol with penalties over a blockchain. Hence, the fact that claim-or-refund can be built on top of generalized channels naturally implies that two parties can execute any such protocol *off-chain*. Off-chain execution offers several advantages if both parties collaborate: (i) they do not have to pay fees or wait for the on-chain delay when deploying and funding the claim-or-refund as well as when one of the parties rightfully claims (resp. refunds) coins; (ii) they can run several simultaneous instances of claim-or-refund fully off-chain, thus improving efficiency; and (iii) a blockchain observer is oblivious to the fact that the claim-or-refund functionality has been executed off-chain. In case of misbehavior during the execution of a claim-or-refund instance, the channel punishment procedure ensures that the honest party is financially compensated with all funds locked in the channel.

Channel splitting. A generalized channel can be split into multiple sub-channels that can be updated independently in parallel. This idea appears already in [26] where two users A and B want to split a channel γ with coin distribution (α_A, α_B) 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.

Executing multiple applications without prior channel splitting requires all applications to share a single funding source (i.e., that provided by the channel) and thus to be adjusted with every single channel update (i.e., even if the update is required for a single application), which might significantly increase the off-chain communication complexity. However, first splitting the channel into sub-channels effectively makes the execution of applications in each sub-channel independent of each other. For instance, two applications that benefit from channel splitting are *payment channels with watchtower* [44] and *virtual channels* [3] – both of which rely on generalized channels, and which we discuss next.

We elaborate on further applications in the full version of this paper [4].

8 Performance Analysis

We implemented a proof of concept for our generalized channels construction, creating the necessary Bitcoin transactions. We successfully deployed these transactions on the Bitcoin testnet, demonstrating thereby the compatibility

Table 1. Costs of lightning (LC) and generalized channels (GC) funding m HTLCs.

	on-chain (dispute)			off-chain (update)	
	# txs	size (bytes)	cost (USD)	# txs	size (bytes)
LC	$2 + m$	$513 + m \cdot 410$	$13.52 + m \cdot 10.80$	$2 + 2 \cdot m$	$706 + 2 \cdot m \cdot 410$
GC	2	663	17.47	2	$695 + m \cdot 123$

with the current Bitcoin network. The source code is available at https://github.com/generalized-channels/gc. For the different operations, we measure the (i) number and (ii) byte size for off- and on-chain transactions required for the protocol. On-chain, we additionally measure the current estimated fee cost (May 2021). Note that the transaction fee in Bitcoin is dependent on the transaction size. We compare these numbers to Lightning-based channels.

Evaluation of multiple HTLCs. Users in a PCN typically take part in several multi-hop payments at once inside one channel. We evaluate the costs of performing m parallel payments, over both Lightning channels (LC) and generalized channels (GC). To realize multiple payments in a channel, there needs to be $2 + m$ outputs: Two of which account for the balances of each user, and m representing one payment each in a "Claim-or-Refund" contract (HTLC).

To update to a channel with m parallel payments, parties need to exchange $2+2{\cdot}m$ transactions in LC and only 2 transactions in GC. The advantage of GC is two-fold: The state is not duplicated and the HTLCs do not require an additional transaction. The difference in off-chain transaction size is $706 + 2 \cdot m \cdot 410$ bytes for LC compared to $695 + m \cdot 123$ bytes for GC.

In case of a dispute, the difference in on-chain cost is even more pronounced. To punish in LC, the honest party needs to spend $m + 1$ outputs: the one representing the balance of the malicious party and one per HTLC. This is in contrast to GC, where the honest party publishes the punishment transaction only. As a result, the total size of on-chain transactions in the LC is $513 + m \cdot 410$ bytes, which cost around $13.52 + m \cdot 10.80$ USD. In GC, the on-chain transaction size is 663 bytes resulting in a cost of 17.47 USD. There have already been disputes for channels with 50 active HTLCs [40]. To settle such a dispute in LC, transactions with 21013 bytes or a cost of 553.66 USD have to be deployed. In GC, again we only need 663 bytes or 17.47 USD. GC thus reduce the on-chain cost from linear on m to constant in the case of a dispute as shown in Table 1.

Evaluation of channel splitting. The state duplication impacts other applications as well, e.g., channel splitting (see Sect. 7). For a LC, two commit transactions need to be exchanged per update. Hence, if we split a LC into two sub-channels, parties need to create these sub-channels for both commit transactions. Moreover, for each sub-channel two commit transactions are required. This is a total of 4 commit transactions per sub-channel. GC needs only one commitment and one split transactions per sub-channel.

After a channel split, sub-channels are expected to behave as normal channels. If we want to split a LC sub-channel further, we would need eight commit transactions (two for each of the four commitments) per sub-channel. Observe, that for every recursive split of a channel, the amount of LC commit transactions for the new subchannel doubles. For the m^{th} split, we need 2^{m+1} additional commit transactions in the LC setting. In the GC setting, there is no state duplication, therefore the amount of transactions per sub-channel is always one commit and one split transaction. We reduce the complexity for additional transactions on the m^{th} split from exponential to constant.

Acknowledgment. This work was partly supported by the German Research Foundation (DFG) Emmy Noether Program *FA 1320/1-1*, by the *DFG CRC 1119 CROSSING* (project S7), by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 771527-BROWSEC), by the Austrian Science Fund (FWF) through PROFET (grant agreement P31621) and the Meitner program (grant agreement M 2608-G27), by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 projects SBA and ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by CoBloX Labs and by the ERC Project PREP-CRYPTO 724307.

References

1. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: EuroSys, pp. 30:1–30:15 (2018). https://doi.org/10.1145/3190508.3190538
2. Andrychowicz, M., et al.: Secure multiparty computations on bitcoin. Commun. ACM **59**(4), 76–84 (2016)
3. Aumayr, L., et al.: Bitcoin-compatible virtual channels. In: IEEE S&P, Matteo Maffei (2021)
4. Aumayr, L., et al.: Generalized channels from limited blockchain scripts and adaptor signatures. Cryptology ePrint Archive, Report 2020/476 (2020). https://ia.cr/2020/476
5. Banasik, W., et al.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: ESORICS, pp. 261–280 (2016)
6. Bano, S., et al.: SoK: Consensus in the age of blockchains. In: ACM AFT, pp. 183–198. ACM (2019)
7. Bartoletti, M., Zunino, R.: Bitml: A calculus for bitcoin smart contracts. In: David, L., Mohammad, M., Michael, B., XiaoFeng, W. (eds.) CCS, pp. 83–100 (2018)
8. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
9. Bentov, I., Kumaresan, R., Miller, A.: Instantaneous decentralized poker. In: ASIACRYPT, pp. 410–440 (2017)
10. Bitcoin wiki: Payment channels. https://tinyurl.com/y6msnk7u
11. Boneh, D., et al.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 416–432. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_26
12. Brasser, F., et al.: Software grand exposure: SGX cache attacks are practical. In: 11th USENIX Workshop on Offensive Technologies (2017)
13. Bulck, J.V., et al.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: USENIX (2018)
14. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press, October 2001

15. Canetti, R., et al.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
16. Chen, G., et al.: Pectre attacks: Leaking enclave secrets via speculative execution. In: IEEE Euro S&P, pp. 142–157 (2018)
17. Cheng, R., et al.: Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In: IEEE EuroS&P, pp. 185–200 (2019)
18. Das, P., et al.: Fastkitten: Practical smart contracts on bitcoin. In: USENIX 2019, pp. 801–818 (2019)
19. Decker, C., Wattenhofer, R.: A fast and scalable payment network with bitcoin duplex micropayment channels. In: Stabilization, Safety, and Security of Distributed Systems 2015, pp. 3–18 (2015)
20. Decker, C., et al.: eltoo: A simple layer2 protocol for bitcoin. https://blockstream.com/eltoo.pdf
21. Deuber, D., et al.: Minting mechanisms for blockchain - or - moving from cryptoassets to cryptocurrencies. Cryptology ePrint Archive, Report 2018/1110 (2018). https://eprint.iacr.org/2018/1110
22. Dziembowski, S., et al.: General state channel networks. In: ACM CCS 18, pp. 949–966 (2018)
23. Dziembowski, S., et al.: Multi-party virtual state channels. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 625–656. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_21
24. Dziembowski, S., et al.: Perun: Virtual payment hubs over cryptocurrencies. In: IEEE S&P 2019, pp. 106–123 (2019)
25. Eckey, L., et al.: Splitting payments locally while routing interdimensionally. ePrint Archive (2020). https://eprint.iacr.org/2020/555
26. Egger, C., et al.: Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks. In: ACM CCS 19, pp. 801–815. ACM (2019)
27. Erwig, A., et al.: Two-party adaptor signatures from identification schemes. In: PKC (2021)
28. Fischlin, M.: Communication-efficient non-interactive proofs of knowledge with online extractors. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 152–168. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_10
29. Fournier, L.: One-time verifiably encrypted signatures a.k.a. adaptor signatures, October 2019. https://tinyurl.com/y4qxopxp
30. Gudgeon, L., et al.: Off the chain transactions. In: FC, Sok (2020)
31. Heilman, E., et al.: Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In: NDSS, 01 2017. 10.14722/ndss.2017.23086
32. Jourenko, M., et al.: Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies. Cryptology ePrint Archive, Report 2019/352 (2019). https://eprint.iacr.org/2019/352
33. Katz, J., et al.: Universally composable synchronous computation. In: Amit, S., (ed.) TCC 2013, volume 7785 of LNCS, pp. 477–498. Springer, Heidelberg, March 2013. https://doi.org/10.1007/978-3-642-36594-2_27
34. Kiayias, A., Litos, O.S.T.: A composable security treatment of the lightning network. In: IEEE CSF 2020, pp. 334–349 (2020)
35. Kosba, A., et al.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE S&P, pp. 839–858 (2016)
36. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: ACM CCS 2016, pp. 418–429 (2016)

37. Kumaresan, R., Bentov, I.: How to use bitcoin to incentivize correct computations. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 14, pp. 30–41. ACM Press, November 2014
38. Kumaresan, R., et al.: How to use bitcoin to play decentralized poker. In: ACM CCS, pp. 195–206 (2015)
39. Lindell, Y.: Fast secure two-party ECDSA signing. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10402, pp. 613–644. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63715-0_21
40. lnchannels. https://ln.bigsun.xyz/ (2020)
41. Malavolta, G., et al.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS 2019. https://www.ndss-symposium.org/ndss-paper/anonymous-multi-hop-locks-for-blockchain-scalability-and-interoperability/
42. Malavolta, G., et al.: Concurrency and privacy with payment-channel networks. In: Bhavani, M., Thuraisingham, D.E., Tal, M., Dongyan, X., (eds.) ACM CCS 17, pp. 455–471. ACM Press, October/November 2017
43. Miller, A., et al.: Sprites and state channels: Payment networks that go faster than lightning. In: Ian, G., Tyler, M., (eds.) FC 2019, volume 11598 of Lecture Notes in Computer Science, pp. 508–526 (2019)
44. Mirzaei, A., et al.: A fair and privacy preserving watchtower for bitcoin. In: FC, Fppw (2021)
45. Moreno-Sanchez, P., Kate, A.: Scriptless scripts with ecdsa. https://tinyurl.com/yxtjo47l
46. Poelstra, A.: Scriptless scripts. https://tinyurl.com/ludcxyz, May 2017
47. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. https://tinyurl.com/q54gnb4 (2016)
48. Siegel, A.: Understanding the dao attack. https://tinyurl.com/2bzxkn7a (2016)
49. Tairi, E., et al.: A^2l: Anonymous atomic locks for scalability in payment channel hubs. In: IEEE S&P (2021)
50. Thyagarajan, S.A.K., Malavolta, G.: Lockable signatures for blockchains: Scriptless scripts for all signatures. In: IEEE S&P (2021)
51. Thyagarajan, S.A.K., et al.: Paymo: Payment channels for monero. Cryptology ePrint Archive (2020). https://eprint.iacr.org/2020/1441
52. Transcripts from coredev.tech amsterdam 2019 meeting on sighash noinput. https://tinyurl.com/49ryfutr
53. Wang, G., et al.: Sharding on blockchain. In: ACM AFT, Sok, pp. 41–61 (2019)