Lightning Factories

Alejandro Ranchal-Pedrosa[a,b,c,d], Maria Potop-Butucaru[b], Sara Tucci-Piergiovanni[a]

[a] *CEA LIST, PC 174, Gif-sur-Yvette, France*
[b] *Sorbonne Université, CNRS, Laboratoire dInformatique de Paris 6, LIP6, Paris, France*
[c] *Department of Networks and Telecommunication Services (RST), Télécom SudParis, Paris-Saclay University, 91011 Evry, France*
[d] *Department of Information Engineering and Communications, Universitat Autónoma de Barcelona, Barcelona, Spain*

Abstract

Bitcoin, the most popular blockchain system, does not scale even under very optimistic assumptions. The Lightning Network, a layer on top of Bitcoin, composed of one-to-one lightning channels make it scale to up to 105 Million users. Recently, Duplex Micropayment Channel factories have been proposed based on opening multiple one-to-one payment channels at once. Duplex Micropayment Channel factories rely on time-locks to update and close their channels. This mechanism yields to a situation where users' funds time-locking for long periods increases with the lifetime of the factory and the number of users. This makes DMC factories not applicable in real-life scenarios.

In this paper, we propose the first channel factory construction, the Lightning Factory, that offers a constant collateral cost, independent of the lifetime of the channel and members of the factory. We compare our proposed design with Duplex Micropayment Channel factories, obtaining better performance results by a factor of more than 3000 times in terms of the worst-case constant collateral cost incurred when malicious users use the factory. The message complexity of our factory is n while Duplex Micropayment Channel factories need n^2 messages, where n is the number of users. Moreover, our factory copes with an infinite number of updates while in Duplex Micropayment

Email addresses: alejandro.ranchal_pedrosa@etu.upmc.fr (Alejandro Ranchal-Pedrosa), maria.potop-butucaru@lip6.fr (Maria Potop-Butucaru), sara.tucci@cea.fr (Sara Tucci-Piergiovanni)

Channel factories the number of updates is bounded by the initial time-lock.

Finally, we discuss the necessity for our Lightning Factories of BNN, a non-interactive aggregate signature cryptographic scheme, and compare it with Schnorr and ECDSA schemes used in Bitcoin and Duplex Micropayment Channels.

Keywords: Bitcoin, Blockchain, Scalability, Lightning Network.

---

1. Introduction

The Bitcoin blockchain aims at becoming the main system for e-commerce. However, it has a big problem: it does not scale. The way Bitcoin works at the time of writing, all (full) nodes need to know all bitcoin transactions ever made. Following this approach, the Bitcoin Network will need to generate more than 1TB of transactions per day to reach VISA's peak transaction rate [14]. Even if the network achieved such numbers, becoming a Bitcoin node would be a very resource-consuming task. This hinders the use of standard computational resources which, in turn, leads to a centralized network of a few powerful nodes, thus threatening its trustless nature.

It is, therefore, reasonable to consider ways of creating blockchain-enforceable information, without actually bloating the network. This approach is similar to that of the judicial system: citizens (members of the network) sign contracts constantly (court-enforceable information), but they do not enforce these contracts unless there is a dispute in which the counter-party does not cooperate. This is actually the idea of *payment channels*, i.e. blockchain-enforceable contracts, whose content is the balance of involved parties. Opening and closing the contract takes place in the blockchain, but from the moment parties open the channel till the moment they close it, they can perform transactions with each other without publishing them in the blockchain, unless there is a dispute, to enforce the correct transaction. Let us note that this approach, called often *Layer*2 of the blockchain, does not only improve scalability, but offers a number of advantages for end-users. First, members of a payment channel can perform payments without paying any fees, if they have an open channel, or with some fees determined by relay nodes in the path, instead of a blockchain fee. Second, the payments performed within a payment channel, provided all participants are online and responsive, take place at the speed of their communication protocol. Third, the possibility to perform fast, free of charge payments opens Bitcoin's way

into a new set of applications based on micropayments.

Recently the idea of payment channels has been further improved by the use of intermediate nodes that can also route payments, creating a network of payment channels, such as *Lightning Network* [14]. However, as pointed out by Poon et al. [14], the Lightning Network does not scale well enough. Even under the very generous assumption that each user only publishes 3 transactions per year (to open and/or close channels), the network scales to only 35 million users, far from covering the world's population. For this reason, Burchert et al. [5] propose *Channel Factories*. Channel factories allow for various users to simultaneously open independent channels in one single transaction, reducing drastically the number of blockchain hits required. Their solution bases on Duplex Micropayment Channels (DMCs) [7], in which closing transactions rely on timelocks relative to the funding transaction entering the blockchain. The timelock makes the transaction invalid until an amount of time in the blockchain elapses (either actual time in seconds, or block-depth). This mechanism makes DMCs simple to setup and track, but it shows an important trade-off between the lifetime of the channel and the worst-case temporary lock-in of funds. On the one hand, a higher locktime will reduce the number of blockchain hits. On the other hand, if one party goes unresponsive, the counterparty will have to wait for the entire locktime before retrieving their funds. In contrast to DMCs, Lightning Channels tackle this trade-off efficiently, leading to a constant worst-case locktime independent of the lifetime of the channel – for this reason we propose in this paper a factory solution based on Lightning Channels instead of DMCs.

The contributions of our work is as follows. To the best of our knowledge, we propose the first *Lightning Factory*, solving the trade-off between the lifetime of the factory and the risk of funds lock-in. We compare our Lightning Factories with DMC Factories (the current state of the art). We show that Lightning Factory offers a constant collateral cost, independent of the lifetime of the channel and members of the factory, enabling actual applicability of factories for scalability, besides disincentivizing frauds by penalization. We obtain better performance results by a factor of more than 3000 times with respect to DMC Factories. From a cryptographic point of view, our solution requires for multi-signatures a non-interactive aggregate signature scheme. Maxwell et al. [10] recently proposed a scheme for Schnorr Multi-signatures with applications to Bitcoin. This scheme is however, interactive, not non-interactive, as we require. For this reason, we apply the BNN [1] non-interactuve scheme for our Lightning Factory, instead of Schnorr

and ECDSA, used in Bitcoin and DMCs, and we compare them.

The remaining of this document is structured as follows: in Section 2 we discuss related work, while in Section 3 we introduce the necessary background and basic notions on payment channels; Section 4 shows our Lightning Factory construction; in Section 5 we compare Lightning Factories with state of the art, and finally we conclude in Section 6.

2. Related Work

Decker et al. [7] firstly introduced Duplex Micropayment Channels (DMCs), with the usage of decreasing timelocks to update the channel. Poon and Dryja's Lightning Network and channel construction [14] followed, gaining popularity as the most promising proposal for a payment channel network. Decker et al. [6] recently proposed eltoo, a proposal for removing incentives to updates for the updating phase of Lightning channels. Prihodko et al. [13] proposed FLARE, a routing algorithm for the Lightning Network.

An important aspect of the Lightning Network not yet extensively studied is its overall usage and impact, i.e. how the fees will be, how scalable the routing will really be, the impact it can generate on the blockchain, etc. Zohar et al. [4] studied this in two rather simple, static Lightning Network topologies.

Other proposals focused on more versatile blockchains. Poon and Vitalik released Plasma [12], a specification of off-chain childchains for Ethereum, as an intermediate layer between Lightning and the rootchain. Miller et al. [11] considered improvements in terms of collateral cost of HTLC-based routing for Ethereum, wile Khalil et al. [9] proposed a rebalancing protocol for exhausted channels.

Because payment channels do not scale well enough by themselves [14], Burchert et al. [5] firstly suggested setting up multiple channels at once in what they referred to as a DMC factory. Decker et al. [6] shortly mentioned that their eltoo approach can be extended to factories. However, they did not provide a protocol. While eltoo-based approach speaks of Lightning penalizations as toxic, the absence of penalizations for fraud in an eltoo-based factory can lead to all users committing to each valid state that maximizes their benefit, bloating the network and causing tension and distrust in the network, which can be more toxic when scaling to multiple parties than penalizing fraudsters. Additionally, the diversity of options for a second

layer in Bitcoin required a common notation of them, which has not been yet performed for Bitcoin, though it has for state channel networks [8].

3. Payment Channels and Factories

In this section, we sketch the functioning of payment channels and factories.

3.1. Channels

A payment channel between n parties, also called n-party channel, consists of a funding transaction that locks up funds, and a sequence of update transactions that deterministically specify how the locked funds are split among participants/users. The structure of a generic transaction is introduced below.

Transaction. Each transaction T_S is a data structure specifying an agreement among a set of subscribers S to move funds among accounts. T_S has the following fields: $T_S.out$: the set of outputs of the transaction, i.e., a set of elements of type o_j, where o_j indicates the fund o to transfer to the account a_i; $T_S.in$: the set of inputs of the transaction. Each element of this set is an output of another transaction $T'_{S'}.out$, i.e. an amount spendable in the transaction T_S; $T_S.conds$: the set of conditions for the transaction T_S to be valid. A valid transaction makes the outputs of the transactions T_S spendable (needed signatures, locktimes, etc.). Figure 1 shows the chaining of two transactions through their inputs and outputs. From the bottom to the top, the transaction T^1_B moves an amount of 20 from the B's account to the C's one, i.e. $T^1_B.out = \{(20, C)\}$. The relationship between the T^1_B's input and T^0_A's output (represented by an arrow in the figure) creates a dependency between transactions, we say in this case that T^1_B spends (fully or partially[1]) the output of the transaction T^0_A. Note that each transaction is executed when registered in the blockchain and that by construction T^1_B cannot be registered before T^0_A since T^1_B refers to T^0_A outputs.

Two-party Channels. Channels have two types of transactions: a *funding* transaction, that opens the channel, and subsequent *refund* transactions. Note that specific protocols can instantiate these transactions in a specific

[1]Usually if the total referred amount is not used, as in this case, an additional transfer from B to himself is added to fully spend the referred output. For sake of conciseness, in the paper this additional transfer is not explicitly represented.

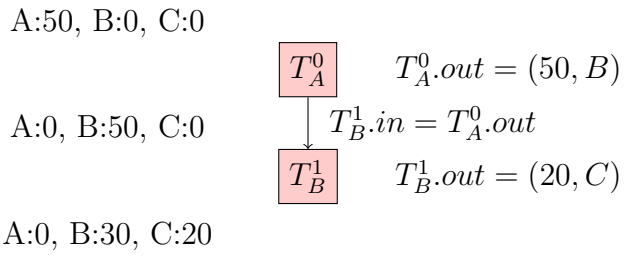

Figure 1: Example of a chain of transactions moving funds from A to B and from B to C. On the left-hand side of the picture, the state of balances before and after the execution of each transaction.

way and/or add other types of transactions. We give an intuition about general principles of channels before deepening into the details of the Lightning channels.

Funding Transaction. Any payment channel is initialized with a *funding* transaction $T_{i,j}^0$ that creates a common account for the participants i and j. The the set of inputs refers to input transactions spendable separately by u_i and u_j; the output specifies instead an output moving funds to an account shared by u_i and u_j; conditions refers to the fact that to spend the common output the spending transaction must by signed by both u_i and u_j.

Refunding transaction. After locking up funds with the funding transaction $T_{i,j}^0$, each subsequent transaction will represent a two-party agreement on a new redistribution of funds, i.e. a refunding transaction. This means that any refunding transaction $T_{i,j}^k$ with $k \geq 1$ has $T_{i,j}^k.in = T_{i,j}^0.out$, and outputs move funds back to i and j on independent accounts a_i and a_j. This means that spending transactions can spends $T_{i,j}^k$ outputs only be signed by u_i or u_j, depending on the output referred, either o_i or o_j.

Let us note that transactions are created by participants by following a message-passing protocol to open a channel (creating the fund and the first refund transaction) and to update the channel (creating the subsequent refund transactions). Figure 2 shows a protocol to open a channel among Alice and Bob. Transactions are exchanged through messages that must be signed, i.e., $T_{\cdot\cdot}^0$ indicates nobody signed yet. Once the transaction is created and fully signed, it can be sent to the blockchain. Let us note that each refund transaction spends the same locked funds, this means that only one of them can really hit the blockchain, otherwise a *double spending* would occur. In this respect, we say that a transaction is *on-chain* when the transaction is

registered in the blockchain in a confirmed block. We then refer to an *off-chain* transaction as a transaction ready to be published on-chain, i.e. it is blockchain-enforceable, but not yet sent to the blockchain.

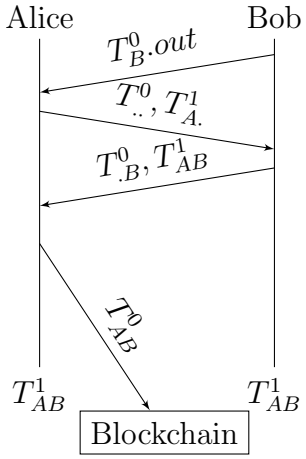

Figure 2: Example of opening a channel by exchaning messages. The funding transaction and the first refund transactions are created and the funding transaction is sent to the blockchain.

Figure 3 illustrates a sequence of k update transactions, being the last transaction T^k, where each transaction spends the outputs of the funding transaction T^0, which is on chain. In green, transactions T^1, T^2, $\cdots$, T^k that spend from the same outputs of T^0 (conditition represented by the incoming arrow) and are off-chain. These transactions are mutually exclusive, only one of them can be included in the blockchain, condition represented by the $\bigotimes$. In purple, we show how much money each participant receives (that is, the balance of each participant at each state).

In the context of n-party channels, a malicious party may want to publish on-chain an old balance, if this balance favors them. Consider an older prefix $\mathcal{T}_{i,j}$ such that the balance $b_{i,k}, k < n$, is greater than $b_{i,n}$, then u_i may want the blockchain to confirm $T^k_{i,j}$ instead of $T^n_{i,j}$. Note that, if $T^k_{i,j}$ hits the blockchain, $T^n_{i,j}$ will be discarded (albeit perhaps after some locktime), because both transactions spend from the same outputs. From now on, we will refer to a party that successfully makes the blockchain confirm an old transaction as a party that *commits a fraud.* A party can go however unresponsive either maliciously or involuntarily.

Once the funding transaction is created as well as the first refunding

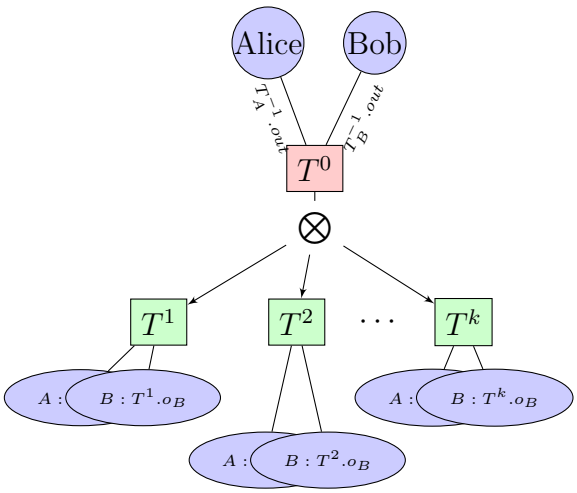

Figure 3: Example of a channel between Alice and Bob. In red, the transaction T^0 that is on-chain. In purple, The initial inputs and the final outputs, split between the participants Alice and Bob. In green, transactions T^1, T^2, $\cdots$, T^k that spend from the same outputs of T^0 and are off-chain. These transactions are mutually exclusive, only one of them can be included in the blockchain, represented by the $\bigotimes$.

transaction, other refund transactions T^k can be created off-chain. In the reminder of the paper we will interchangeably use the term refund transaction T^k and channel at state k, where at each state k the users of the channel have balances determined by the execution of the transaction T^k.

As already mentioned, a payment channel is implemented by a message-passing protocol among participants. Any protocol to correctly implement a payment channel must be fraud-resistant and cope with unresponsive behavior. Honest parties should always own enough transactions to be able to get back at least an amount of funds equivalent to the last agreed-upon balance (*no-steal*). Moreover, if a new update cannot be fully singed due to an unresponsive behavior of one of the party, then the other party must get back the initial fund published with the $funding$ transaction (*no-lock*). For instance, in Figure 2 the first update transaction is signed before the fund transaction, to guarantee *no-lock*.

Underlying mechanisms of current proposals.. Depending on the update mechanism, we list here three different channels. *Duplex Micropayment Channels*[7] (DMCs) update by creating new transactions with decreasing time-

locks for each update, achieving the determinism of the updates. New updates are locked for less time, thus replacing the older ones. Note that in this protocol frauds cannot be commited under the assumption that the blockchain well-behaves. *Eltoo Channels*[6] update by creating a set of transactions that invalidate previous refund transactions when creating the new update. New updates invalidate old ones, but frauds can be committed. In this case the protocol can recover to the correct state under the assumption that the fraud is detected. *Lightning Channels*[14] follow eltoo channels approach, but with the additional feature of penalizing parties that commits frauds.

3.2. Factories

A channel factory is an n-party channel which creates a funding transaction among n nodes, i.e. all of them sign the funding transaction. Further, instead of having an update consisting of a refunding transaction signed by all the parties, a special *Allocation* transaction create funding transactions for 2-party channels. The update of the factory consists in updating the allocation transaction. The channel factory concept has been introduced in [5] in which the funding transaction to open the factory is called *Hook* transaction and the first allocation transaction has an associated timelock. Updating the factory means opening/closing channels, by creating a new allocation transaction with lower locktime. Finally, closing the factory means publishing the lastly signed allocation transaction (with the lowest locktime), or else cooperating to sign a last agreed-upon transaction with no locktime.

4. From Lightning Channels to Lightning Factories

In this section we present the Lightning Factory construction. We first detail Lightning and Duplex Micropayment Channels, then introduce the cryptographic scheme needed to cope with the challenges of extending lightning channels to n parties. Furthermore, we explain the protocol for opening, updating and closing a Lightning Factory.

4.1. Lightning Channels

A Lightning channel is opened by creating a funding transaction and a first refund transaction, as shown in Figure 2 . To align to Lightning we will denote the funding transaction as F_{AB}. For the channel update in Lightning, outdated states are invalidated by creating specific transactions that we detail in the following. These specific transactions, if a malicious party commits a

fraud, allow the honest one to remedy by publishing a specific transaction that gets back *all* the funds to the honest party – *a proof of fraud*. The set of created transactions and their dependency are shown in Figure 4. Let us note that all the transactions have now a subscript indicating the party that, once the transaction is created, stores the transaction locally. The figure shows two so-called Commitments transactions: $C^{k,A}_{AB}$ that only Alice can send to the blockchain, and $C^{k,B}_{AB}$ that only Bob can send to the blockchain. In case of unresponsive party or because one party wants to unilaterally close a channel, funds can be retrieved by A thanks to a so-called Revocable Delivery transactions $RD^{k,A}_{AB}$ after a timelock (by B through $RD^{k,B}_{AB}$, respectively), illustrated in the dotted paths in Figure 4. During the creation of $RD^{k,A}_{AB}$ the protocol makes sure to create as well $D^{k,A}_{AB}$ which refunds B immediately (no time-locks). Proofs-of-frauds can be achieved through the Breach Remedy transactions $BR^{k,A}_{AB}$, $BR^{k,B}_{AB}$, respectively. These transactions spend the same outputs as $RD^{k,A}_{A,B}$ and $RD^{k,B}_{A,B}$, but without a timelock and they give all the balance to the counterparty, illustrated in the dashed paths in Figure 4.

In the proposed scheme, if B (the same applies to A) gets unresponsive, funds can be retrieved unilaterally by A thanks to $RD^{k,A}_{AB}$, but only after a timelock, this way *no-lock* is preserved. Moreover, if one of the two party sends to the blockchain a stale state, the timelock allows for B to react and send a breach remedy.

4.2. Duplex Micropayment Channel Factories[5].

Before defining our proposed Lightning Factory, and for the purpose of comparing performance, let us outline the state-of-the-art factory construction: DMC Factories [5].

Suppose a set of users $u_0, ..., u_{n-1}$ want to open an amount of channels within them (e.g. $\{\{u_0, u_1\}, \{u_0, u_2\}, \{u_2, u_n\}, ...\}$.). If they want to open m channels, $n \leq m \leq \binom{n}{2}$, they would need to publish m transactions. With a DMC Factory, they can instead join together into a n-of-n multisig output, called the Hook transaction, $H_{\{u_j\}_{j=0}^{n-1}}$ (i.e. T^0 in figure 3). The output of this transaction is the input of another transaction, called the Allocation transaction $A_{\{u_j\}_{j=0}^{n-1}}$ (i.e. T^1 in figure 3). The allocation transaction allocates the funds of each channel, and has a timelock that is decreased in each update. That is, having the first allocation transaction $A^1_{\{u_j\}_{j=0}^{n-1}}(tlock : t_1)$ then, when updating, the new allocation is as follows: $A^2_{\{u_j\}_{j=0}^{n-1}}(tlock : t_2 \leq t_1 - \delta_t)$, being δ_t the minimum time required to guarantee inclusion of A^2 before A^1

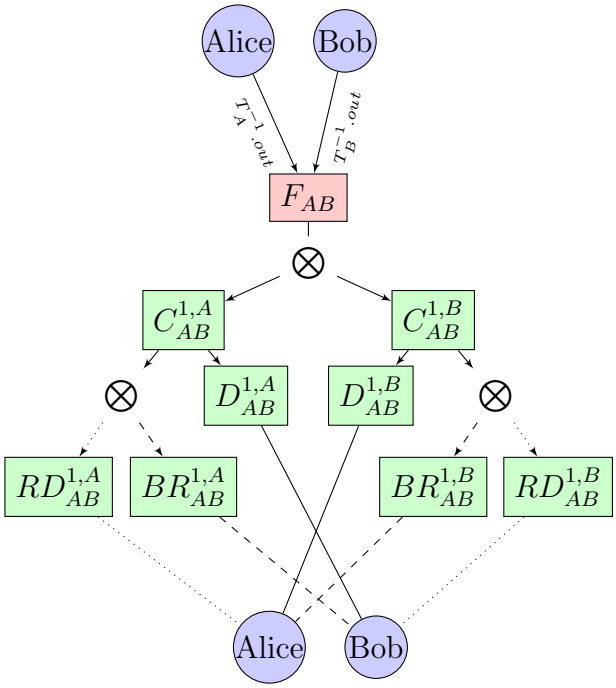

Figure 4: Example of a channel between Alice and Bob. Alice can only broadcast the left-hand side by publishing $C_{AB}^{1,A}$. If Bob has $BR_{AB}^{1,A}$ then Alice loses all her funds (dashed path). Otherwise, she can publish $RD^{1,A}$ after waiting some locktime (dotted path), retrieving her funds. Bob receives his funds immediately in this case through $D^{1,A}$. $C^{1,A}$ and $C^{1,B}$ are mutually exclusive, only one can be published.

in the blockchain, with a high probability. A more in-depth discussion about δ_t, and Δ_t, can be found in section 5.2. Therefore, the hook and allocation transactions extend the DMC concepts of funding and refund transactions to DMC Factories, respectively. Figure 5 shows an example of a DMC factory.

The creation of each channel (i.e. each funding an refunding transaction), the first allocation transaction $A^1_{\{u_j\}_{j=0}^{n-1}}$ and the hook transaction $H_{\{u_j\}_{j=0}^{n-1}}$ represent the opening of the factory. Updating the factory means opening/closing channels, by creating a new allocation transaction $A^2_{\{u_j\}_{j=0}^{n-1}}$, with lower locktime relative to the hook. Analogously, users involved in a particular channel can update it by signing a new refund transaction with a lower locktime. Finally, closing the factory means publishing the lastly signed allocation transaction (with the lowest locktime), or else cooperating to sign a

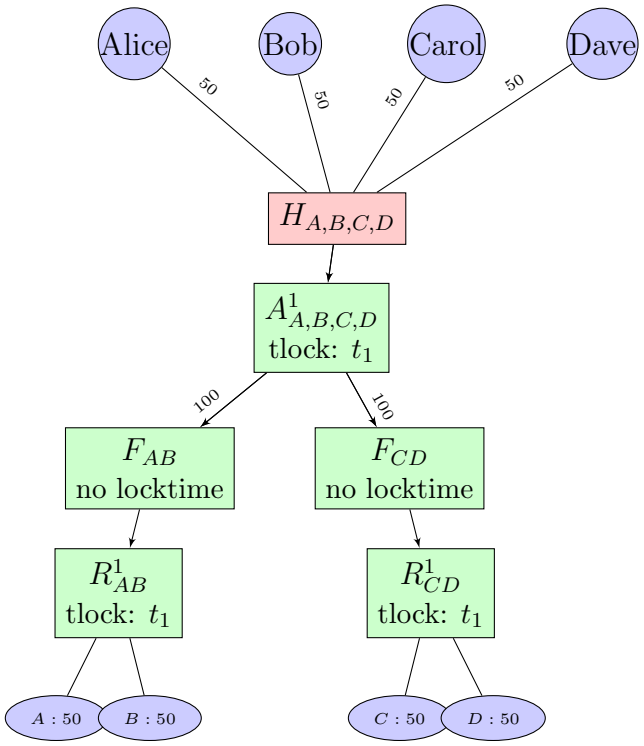

Figure 5: DMC factory between Alice, Bob, Carol and Dave with two payment channels inside: F_{AB} and F_{CD}. In red, the on-chain transaction. In green, transactions that do not ever hit the blockchain if A,B,C and D cooperate.

last agreed-upon transaction with no locktime. We deepen into the protocol specification of DMCs, analyzing their performance, in the appendix.

4.3. Cryptographic Scheme

Lightning Factories do not extend as straightforwardly from Lightning Channels as DMC Factories do from DMCs. Let us note that in a two-party Lightning Channel both participants sign everything because every change in the Lightning Channel involves them, and an ejection of one of them implies closing the channel. For this reason, a two-party Lightning Channel works perfectly with a 2-of-2 multisignature, in which both participants sign the same message m, which represents the last balance. Lightning Factories to be effective need to take into account ejection of participants in the factory. Moreover, participants in a Lightning Factory sign and share a part of a transaction, so that each user can later reconstruct transactions as needed.

This requires for a cryptographic scheme based on aggregate signatures. As detailed by Boneh et al. [2], an Aggregate Signature (AS) scheme is a digital signature scheme with the additional property that a sequence of signatures $\sigma_1, ..., \sigma_n$ of some message m_i under some public key pk_i can be condensed into a single, compact aggregate signature σ that simultaneously validates the fact that m_i has been signed under pk_i for all $i = 1, ..., n$. The verification process takes input $(pk_1, m_1), \ldots, (pk_n, m_n)$, and accepts or rejects. Boneh et al. [2] propose an aggregate signature scheme based on BLS [3], called BGLS. Bellare et al. [1] improve this scheme by removing the per-signer distinct messages restriction to BGLS in a new scheme, BNN.

Using a non-interactive aggregated signature scheme, such as BNN, Alice, Bob and Carol sign a part of a transaction each, instead of the full transaction. A part of a transaction can be considered similar to a partially signed transaction with the sighash-single flag, or a partially signed n-of-n multisig. Typically, this is referred to as an *aggregate signature*. We will also refer to the signed message that needs to be aggregated with others to form a full transaction as a *transaction fragment*. As such, an n-of-n aggregate signature needs n transaction fragments signed by n different users (each user signs one), in order to get a fully signed transaction.

4.4. Lightning Factory Protocol

Actions of users. In order to depict the Lightning Factory protocol, we define a set of actions that a user can perform. The protocol will decide the rules for the actions to be taken and the kind of transactions to build.

- $create_i(T)$: the transaction T is created by u_i and stored locally at u_i;
- $sign_i(T)$: the transaction T is signed by the user u_i;
- $broadcast_i(T)$: the transaction is sent to all n-channel participants;,
- $deliver_i^j(T)$: the transaction T is delivered by u_i from u_j;
- $publish_i(T)$: the transaction T is published on-chain by u_i and stored in the blockchain, if and only if the transaction is valid (correct signatures and timelocks expired).

Note that the party u_i can store the transaction T if and only if they have created it or another party u_j shared it with i, i.e. u_i delivered from u_j.

In the following for each protocol phase we detail the types of transaction built and we detail the corresponding protocol.

4.4.1. Opening a Lightning Factory

Lightning Factory sets up funds by locking them up into a n-of-n aggregated output, by means of a hook transaction. A Lightning Factory extends the concept of a Lightning Channel to Factories trough the equivalent of an Allocation transaction $A^k_{\{u_i\}}$, i.e. a set of Allocation Commitment transactions $\{C^{k,j}_{\{m_i\}}\}^{n-1}_{j=0}$, one per user per state to ascribe blame, same way we defined commitment transactions $C^{k,A}_{AB}, C^{k,B}_{AB}$ as the equivalent of a refund transactions. The Allocation Commitment under the aggregate signature scheme is $C^{k,j}_{\{m_i\}^{n-1}_{i=0}} = \sum_{i=0}^{n-1} C^{k,j}_{m_i}$, where k is the state number, j is the user that owns this commitment transaction, and $\{m_i\}$ indicates all required messages are aggregated. The input of this transaction is the output of the hook. The output of this transaction points at a revocable allocation transaction $RA^{k,j}_{\{m_i\}^{n-1}_{i=0}} = \sum_{i=0}^{n-1} RA^{k,j}_{m_i}$, with a locktime relative to the inclusion of $C^{k,j}_{\{m_i\}^{n-1}_{i=0}}$ in the blockchain.

Specifically, a transaction fragment is a tuple $< \mathcal{P}, \mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{O}, \mathcal{S}t >$.

Following, we explain each of the parts of the tuple, with an example of the fragment Bob signs for the Alice's commitment transaction at the initial state, $C^{1,A}_{m_B}$: $\mathcal{P}$ is the issuer of the message (e.g. Alice in the case of $C^{1,A}_{m_B}$);$\mathcal{S}$ is signer of the message (e.g. Bob in $C^{1,A}_{m_B}$); $\mathcal{T}$ is the type of the message: either timelocked or not. In the case of $C^{1,A}_{m_B}$ $\mathcal{T}$ has no locktime; $\mathcal{I}$ is the input for this fragment's transaction. The $H_{\{u_i\}^{n-1}_{i=0}}$ channel hook (funding) output for the fragment the transaction belongs to; $\mathcal{O}$ is the output for this fragment's transaction. For $C^{1,A}_{m_B}$ this output is the input of the Revocable Allocation; $\mathcal{S}t$ is state identifier for which this message is valid. In the case of $C^{1,A}_{m_B}$, state is 1.

Notice that only $\mathcal{P}$ and $\mathcal{S}t$ are newly proposed fields for Bitcoin. $\mathcal{P}$, the issuer, can be simply a one bit flag indicating that the signer of this fragment is not the issuer. $\mathcal{S}t$ can be defined in some of the remaining bits still unspecified in the sequence_no field[2].

All users need to agree and sign for the state, so that they cannot reuse a fragment for a future state. Therefore, the aggregated Allocation Commitment transaction $C^{1,A}_{\{m_i\}^{n-1}_{i=0}} = \sum_{i=0}^{n-1} C^{1,A}_{m_i}$ would contain the following extra fields: $\mathcal{S}$ is σ_i, $\forall C^{1,A}_{m_i}$; $\mathcal{T}$ is No locktime in the aggregated message in $C^{1,A}_{\{m_i\}}$;

[2] `https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki`

$\mathcal{O}$ is one aggregate signature output, $C^{1,A}_{\{m_i\}}.o$. Throughout this paper, we use T_{m_i} for the fragment of transaction T created and signed by user i, $T_{\{m_i\}^k_{i=0}}$ for an aggregation of fragments of transaction T created and signed by all users $\{u_i\}^k_{i=0}$, and $T_{\{m_i\}}$ as the same as $T_{\{m_i\}^{n-1}_{i=0}}$.

Let us note that, for this application, we require only one output for the commitment transaction (as detailed above). The output represents the balance that this message commits to. It can only be relative to the signer, i.e. Bob can only sign the amount Bob receives from the factory.

Analogously, one can also extend the concept of a revocable allocation transaction into a set of transaction fragments $\{RA^{1,A}_{m_i}\}$ that, aggregated, create a valid transaction $RA^{1,A}_{\{m_i\}} = \sum_{i=0}^{n-1} RA^{1,A}_{m_i}$ spending the outputs of $C^{1,A}_{\{m_i\}}$. As such, provided that Alice already committed to this state by broadcasting $C^{1,A}_{\{m_i\}}$, an aggregated revocable allocation message for her, $RA^{1,A}_{\{m_i\}} = \sum_{i=0}^{n-1} RA^{1,A}_{m_i}$, would result in each part of the aggregated tuple $RA^{1,A}_{\{m_i\}} =< \mathcal{P}, \mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{O}, \mathcal{S}t >$: $\mathcal{P}$ is any, $\forall RA^{1,A}_{A,m_i}$; $\mathcal{S}$ is σ_i, $\forall RA^{1,A}_{m_i}$; $\mathcal{T}$ is relative locktime for all, $\forall RA^{1,A}_{A,m_i}$, dependent on when the corresponding commitment transaction was published; $\mathcal{I}$ is $C^{1,j}_{\{u_j\}}$, regardless of the j (similar to SIGHASH_NOINPUT); $\mathcal{O}$ is one aggregate signature output, $\mathcal{O}$=$o^{RA^1}_i$, $\forall RA^{1,A}_{m_i}$; $\mathcal{S}t$ is 1, $\forall\, RA^{1,A}_{m_i}$.

Notice that $\sum_{i=0}^{n} o^{RA^1}_i = \mathcal{B}$, being $\mathcal{B}$ the total amount locked at setup (the total balance). $o^{RA^1}_i$ act as the output of a funding transaction, used as input for each refunding transaction of two-party channels. In order to allow each signer to specify with which output to aggregate, we use output indices. Also, each transaction fragment signs in its fragment the output indices with a list of signatures that can aggregate to this output, in order to prevent an outsider to lock funds of a channel by including their signature in the output. [3].

Figure 6 shows an example of a Lightning Factory for u_0's case (i.e. only u_0 can publish the commitment transaction shown). Transaction fragments are shown within each transaction they form, vertically aligned with the user that signed for it. The lowest transactions represent each individual commitment transaction of each two-party lightning channel. Here we illustrate them as commitment transactions, considering them as lightning channels,

[3]This requirement is no different from how other works require different ways of representing outputs [10] for further scalability)

but they can be refunding transactions from DMCs. Also, another factory within the factory can be created at this level. Notice how, the same way

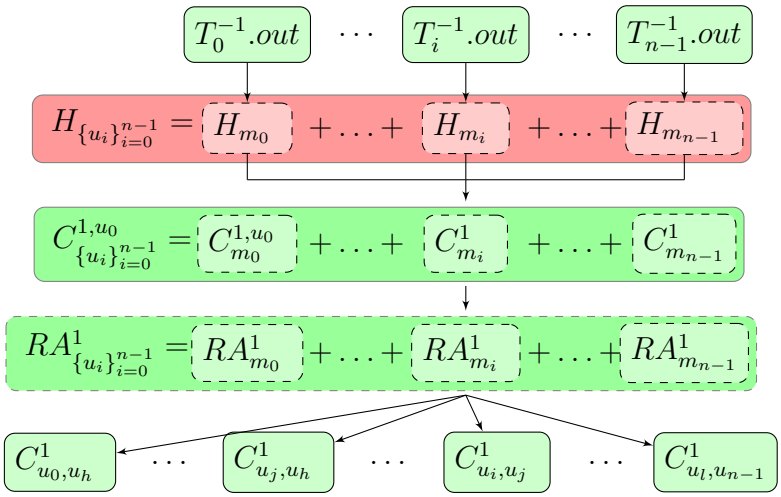

Figure 6: Lightning Factory's transactions and fragment for u_0's case (i.e. only u_0 can publish the commitment transaction shown). The transaction fragments are shown within each transaction they form, vertically aligned with the user that signed for it. Notice how the lowest Commitment transactions (e.g. $C^1_{u_0,u_h}$) are those of each individual channel.

inputs are aggregated as needed (initially all need to aggregate their key), outputs are as well. Each user needs to sign also for which output indexes they want to sign, in order to add its key to the output (and, thus, require its signature in order to spend it). In the case of the hook, each transaction fragment H_{m_i} signs only the input $T_i^{-1}.o$, which only requires user u_i's key, and the output n-of-n aggregated output $H.o$, which requires all other users. As for the revocable allocation fragments RA_{m_i}, the outputs user u_i signs are only those that are used as inputs in channels that involve u_i. We provide the protocol to open the factory LFsetup() in Figure 7), and we prove its correctness in the Appendix.

4.4.2. Updating a Lightning Factory

Updating to state $k+1$ requires a two-step process:

1. sign and share transaction fragments for the new commitment and revocable allocation transactions for state k+1, $C^{k+1,A}_{\{m_i\}} = \sum_{i=0}^{n-1} C^{k+1,A}_{m_i}$ and $RA^{k+1,A}_{\{m_i\}} = \sum_{i=0}^{n-1} RA^{k+1,A}_{m_i}$
2. invalidate the previous state k, creating Proofs-of-Fraud.

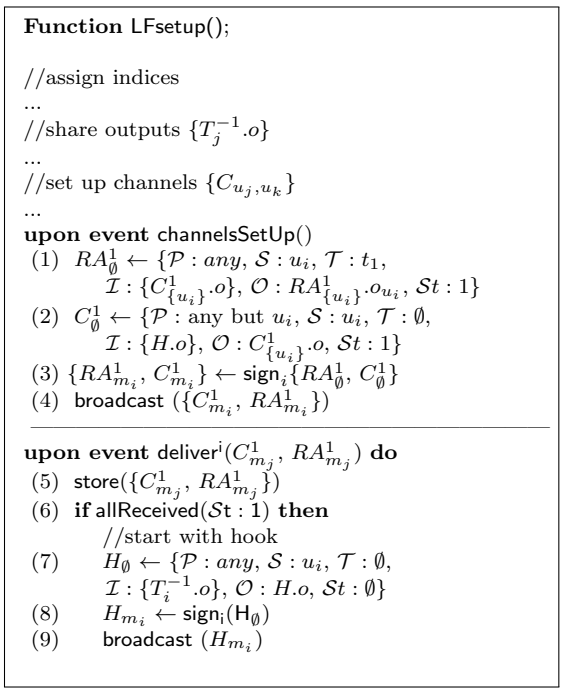

Figure 7: Opening a Lightning Factory. The initial comments refer selecting an ordering, sharing outputs to be spent, and setting up 2-party channels inside the factory.

Invalidation of Alice's transaction for state k means for Alice to create and share a Breach Remedy transaction fragment, $BR^{k,A}_{m_A}$, that spends from $C^{k,A}_{\{u_j\}}$ without a timelock, same as the fragment $C^{k,A}_{m_i}$ does with a timelock Δ_t. More in detail, $BR^{1,A}_{m_A}$'s transaction fragment fields are as follows: $BR^{1,A}_{m_A} =< \mathcal{P}=$any, $\mathcal{S}=$Alice, $T=$no locktime, $\mathcal{I}=C^{1,A}_{\{m_i\}}$, commitment transaction of Alice, $\mathcal{S}$t$=1$, $\mathcal{O}=\emptyset >$.

This way, if Alice publishes the previous state, Bob or Carol can prove fraud, and restore the channel without requiring Alice's signature anymore. Notice this requires for the transaction fragments $\{C^{k,A}_{m_j}\}_{j\neq A}$ to be only valid for the particular state k. This is why $\mathcal{S}t$ is required as part of the transaction fragment.

There are two possible options when proving fraud with Breach Remedy transaction fragments, we will refer to them as Breach Remedy Restoration (BRR) and Breach Remedy Closing (BRC) transactions. Both can be signed and transferred during the update protocol, but only one of them is required to guarantee the invalidation of previous states and, therefore, correctness of

the factory.

BRRs: Proof-of-Fraud to expel fraudster. BRR is used to expel a fraudster upon committing to a fraud, but leave the rest of the factory intact. The output of a PoF in a BRR is simply a new (n-1)-(n-1) aggregated signature, that removes the fraudster. This way, since the fraudster's fragments are not required anymore (nor are they accepted), then the Commitment and Revocable Allocation transactions will not take them into account. This means the key of the fraudster will not figure in the outputs that the fraudster signed for in its Revocable Allocation fragment. Hence, every 2-of-2 multisig output that funded a channel for this fraudster with someone else becomes a single-sig output for the counter-party, effectively giving all funds in the channel to the counter-party. In this sense, The BR acts as a new hook for a new Lightning Factory without the fraudster.

In order for BRRs to be reproducible, we also introduce *idle transaction fragments* (Is). That is, each participant signs one and gives it to all the rest, once for the entire lifetime of the factory. This fragment simply adds the key of the signer for the input and the output of the Proof-of-Fraud, making sure a non-fraudster is still part of the factory, whereas the BR only adds the key to the input. Therefore, when a fraudster tries to commit fraud by publishing an invalid Commitment Transaction, any participant must create a BRR transaction by aggregating the BR of the fraudster with the idle transaction fragments of the rest of the factory members.

We illustrate an example for a 4-party factory in figure 8, where Alice tried to commit fraud. In this channel factory, Alice aggregates $C^{1,A}_{A,B,C,D} = C^{1,A}_{m_A} + C^{1,A}_{m_B} + C^{1,A}_{m_C} + C^{1,A}_{m_D}$ and their public keys, i.e. $pk_{A,B,C,D} = pk_A + pk_B + pk_C + pk_D$, thus it is certainly possible that a BRR, $BRR = BR^{1,A}_{m_A} + I_{m_B} + I_{m_C} + I_{m_D} = BR^{1,A}_{m_A} + I_{m_B,m_C,m_D}$, with $BR^{1,A}_{m_A}$ signed by Alice, could point its output to $pk_{B,C,D}$, being $C^{2,B}_{B,C,D}$ a valid transaction that spends this output. After A cheated, her messages are not necessary, and her buildable transactions not valid.

I_{m_i} serves as confirmation that user i agrees with unlocking the current output o of the commitment transaction $C^{j,A}_{\{u_i\}}$, as long as the new output o' is such that pk_i is still part of it, and no new members have been added (only removal). On the other hand, $BR^{j,A}_{m_A}$ agrees with unlocking this output. This approach is similar to that of a member of a multisig signing a new state in which this member is not part of the multisig anymore. In this case, if o was the output of a hook transaction that depended on the four, $H_{A,B,C,D}$, o' is

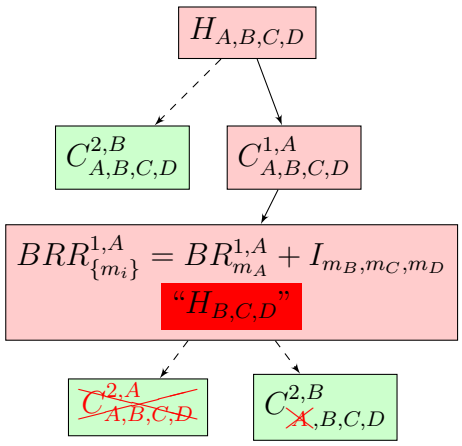

Figure 8: Expelling fraudster using a Breach Remedy Restoration transaction (BRR). After A cheated, her messages are not necessary, and her buildable transactions not valid.

the output of a BRR that depends on Bob, Carol and Dave only, similar to a hook $H_{B,C,D}$. Hence, a new $BRR_{B,C}$ can be created, should Bob try cheating with $C^{2,B}_{\{m_i\}}$, by aggregating $BRR^{2,B}_{B,C} = BR^{2,B}_{m_B} + I_{m_C,m_D}$, which would result in the funds depending only on Carol and Dave's keys. This reproducibility is necessary to guarantee correctness.

BRCs: Proof-of-Fraud to close factory. To close the factory while proving fraud, one can create a BRC transaction, made out of Breach Remedy and Revocable Allocation transaction fragments. The challenge here, as for BRRs, is to point at the proper outputs. Note that a BRC might be at the same time a fraud in itself by a second fraudster, and a third honest party must be able to proof two nested frauds (i.e. it must be reproducible). This is why this Proof-of-Fraud is revocable, as opposed to previous cases. BRRs already tackle this problem, since the factory is restored, not closed.

Figure 9 shows this scenario in a channel factory between Alice, Bob, Carol and Dave, for which the set of state updates are as follows:

User	State 1	State 2	State 3
A	50BTC	20BTC	20BTC
B	50BTC	80BTC	20BTC
C	50BTC	50BTC	110BTC
D	50BTC	50BTC	50BTC

Notice how it is Alice's interest to publish state 1, even if it is deprecated.

Therefore, Alice tries cheating by publishing $C^{1,A}_{\{m_i\}} = \sum_{i=0}^{n-1} C^{1,A}_{m_i}$. After that, Bob, who is also dishonest, prepares a transaction $BRC^{\{1,A\},\{2,B\}}_{\{m_i\}} = BR^{1,A}_{m_A} + C^{2,B}_{m_B} + C^{2,B}_{m_C} + C^{2,B}_{m_D}$, indicating that this transaction invalidates the committed state 1 by Alice, and commits to state 2 by Bob. Finally, Carol, who is honest and whose best interest is also to publish the last state, prepares the last $BRC^{\{1,A\},\{2,B\},\{3,C\}}_{\{m_i\}} = BR^{2,B}_{m_B} + C^{3,C}_{m_C} + C^{3,C}_{m_D}$.

In this case, if there was a two-party channel between Alice and Bob in the factory, then A's and B's balances in the last valid state would go as fees. The remaining parties can, when cooperative, create new sets of Commitment transactions that split among them the balances of the malicious parties.

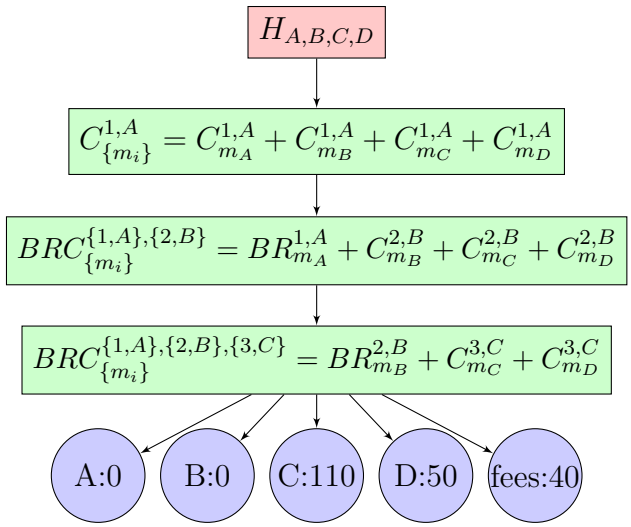

Figure 9: Nested BRCs. Valid state is state 3, not 2. Bob was trying to cheat with a BRC.

Update Protocol. Figure 10 shows the update protocol, regardless of how BRs are used (as part of a BRR or a BRC). Notice that this protocol does not generate a new state if one user is offline, which can be exploited to retrieve all required signatures for a new state, without sharing them, affecting the correctness. For this reason, any update that does not fully succeed paralyzes the money-flow in the factory, leaving it stale, until a further update/close event finishes. We refer to this as a stale factory, and a stale factory attack. However, a stale attack does not have a big performance impact for Lightning Factories, as detailed in section 5. It is nevertheless possible to select an ordering of the users and require users to share keys one by one only when

receiving one key. A protocol like this would require $\lceil \frac{n}{2} \rceil$ users to collude in order to successfully achieve a stale situation.

Nonetheless, a stale attack can be suspected any time an update is not fully finished, and, given the trustless-oriented nature of blockchains, one should always assume that the rest of users of the factory might be colluding to steal one's funds. Furthermore, this protocol performs significantly faster than ordering and sharing the fragments one by one with one particular user at a time, since all fragments are delivered to all.

Function LFupdate($C^k_{\{u_j\}}, RA^k_{\{u_j\}}$);

//set up, update channels $\{F_{u_j,u_k}\}$ ...
upon event channelsUpdated()
(1) $C^{k+1,i}_{\emptyset} \leftarrow \{\mathcal{P} : i, \mathcal{S} : u_i, \mathcal{T} : \emptyset, \mathcal{I} : \{H_{\{u_j\}}.o\},$
$\mathcal{O} : C^{k+1}_{\{u_j\}}.o, \mathcal{S}t : 1\}$
(2) $RA^{k+1}_{\emptyset} \leftarrow \{\mathcal{P} : any, \mathcal{S} : u_i, \mathcal{T} : t_1, \mathcal{I} : \{C^{k+1}_{\{u_j\}}.o\},$
$\mathcal{O} : RA^{k+1}_{\{u_j\}}.o_{u_i}, \mathcal{S}t : k+1\}$
(3) $C^{k+1}_{\emptyset} \leftarrow \{\mathcal{P} :$ any but $u_i, \mathcal{S} : u_i, \mathcal{T} : \emptyset, \mathcal{I} : \{H_{\{u_j\}}.o\},$
$\mathcal{O} : C^{k+1}_{\{u_j\}}.o, \mathcal{S}t : k+1\}$
(4) $\{C^{k+1,i}_{m_i}, RA^{k+1}_{m_i}, C^{k+1}_{m_i}\} \leftarrow \mathsf{sign}_i(\{C^{k+1,i}_{\emptyset}, RA^{k+1}_{\emptyset}, C^{k+1}_{\emptyset}\})$
(5) $\mathsf{broadcast}^{k+1}_0$ $(\{C^{k+1}_{m_i}, RA^{k+1}_{m_i}\})$

---

upon event $\mathsf{deliver}^i(\{C^{k+1}_{m_j}, RA^{k+1}_{m_j}\})$ **do**
(6) $\mathsf{store}(\{C^{k+1}_{m_j}, RA^{k+1}_{m_j}\})$
(7) **if** $\mathsf{allReceived}(\mathcal{S}\mathsf{t} : \mathsf{k}+1)$ **then** //start with breach remedy
(8) $BR^{k,i} \leftarrow \{\mathcal{P} : any, \mathcal{S} : u_i, \mathcal{T} : \emptyset, \mathcal{I} : \{C^{1,i}.o\}, \mathcal{O} : \emptyset, \mathcal{S}t : k\}$
(9) $BR^{k,i}_{\emptyset} \leftarrow create_i(BR)$
(10) $BR^{k,i}_{m_i} \leftarrow \mathsf{sign}_i(\mathsf{BR}^{k,i}_{\emptyset})$
(11) $\mathsf{broadcast}_i(BR^{k,i}_{m_i})$

---

upon event timeout_protocol **do**
(12) **if** $\mathsf{notAllReceived}(\mathcal{S}\mathsf{t} : \mathsf{k}+1)$ **then**
//publish lastly valid one (no breach remedy issued)
(13) $\mathsf{publish}_i(\sum_{j=0}^{j-1} C^{k,i}_{m_j})$

Figure 10: Updating a Lightning Factory.

4.4.3. Closing a Lightning Factory

In order for user u_j to properly close a Lightning Factory, being $l_f - 1$ the last state of the factory, they add the proper last Allocation Commitment fragments into an Allocation Commitment transaction $C^{k,j}_{\{m_i\}_{i=0}^{n-1}} = \sum_{i=0}^{n-1} C^{k,j}_{m_i}$. Then, after waiting for the timelock Δ_t, any user $u_{j\prime}$, including u_j, can publish

$RA^{k,j\prime}_{\{m_i\}_{i=0}^{n-1}}$ in order to close the factory. Notice that, should all users agree, they can create a last state l_f, not revocable, that directly outputs into the accounts they agree upon, instead of setting up the channels of the factory when closing the factory.

5. Complexity, Resilience and Performance Analysis

In this section, we analyze the correctness and performance of our Lightning Factory construction, and compare it with others. We show better performance by a factor of more than 3000 by decreasing the worst-case collateral cost of Lightning Factories, when compared to Duplex Micropayment Factories, among others. For this purpose, we first define here correctness, Bitcoin as a Clock, the adversarial model and components of payment channels.

5.1. Correctness specification

Any protocol to correctly implement a payment channel must be fraud-resistant and cope with unresponsive behavior. Honest parties should always own enough signed transactions to be able to get back at least an amount of funds equivalent to the last agreed-upon balance (*no-steal*). If an update transaction is not fully signed and retrieved by all parties before the protocol times out, then the party must get back the initial fund published with the $funding$ transaction (*no-lock*).

More formally, given a sequence of transactions $\mathcal{T}_{i,j} = \{T_i^{-1}, T_j^{-1}, T_{i,j}^0, \ldots, T_{i,j}^k, \ldots T_{i,j}^n\} : n > 1$, then we define the following properties:

No-steal. The two-party protocol guarantees no-steal if and only if all honest parties own enough off-chain and on-chain transactions to enforce their last balance unilaterally.

No-lock. the transaction $T_{i,j}^0$ is never on-chain, nor has it any participant fully signed (off-chain), without having all participants at least one, fully-signed off-chain $T_{i,j}^k$ that spends the outputs of such funding $T_{i,j}^0$. This guarantees no participant can publish $T_{i,j}^0$ and then not cooperate, not signing, to lock funds.

We prove in the appendix the correctness of Lightning Factories.

5.2. Bitcoin as a Clock

n-party channels meet the no-steal property assuming Bitcoin as a Clock is somewhat reliable. That is, we require that the blockchain has some degree

of reliability to guarantee that after some time t any transaction that has met an average fee will hit the blockchain. If this can not be guaranteed, timelocks can be overdue and valid transactions can be invalidated. In general, one should take great care in deciding the timelock for transactions T^i. As a matter of fact, full correctness can not be guaranteed for certain, without assuming an ideal blockchain resilient to attacks that delay the inclusion of transactions in the blockchain, only with high-probability. A study of the proper time to select is a work in itself and is out of the scope of this paper. We will simply define two different times:

δ_t, considering it as the minimum time that two subsequent transactions that spend from the same output must differ in order to guarantee inclusion of the newest with a desired probability. Therefore, an update from $T^k_{AB}(tlock : t_k)$ would require signing a new $T^{k+1}_{AB}(tlock : t_k - \delta_t)$. DMCs and DMC factories require new update transactions to have a timelock at least δ_t smaller relative to the previous update.

Δ_t, considering it as the minimum time a transaction must wait before spending an output to guarantee enough time for a counterparty to realize that the output is spendable (i.e. the transaction that generated the output hit the blockchain), prepare a transaction with no locktime, and publish it first. Lightning Channels and Lightning Factories require Δ_t to give enough time to counterparties to prove fraud. Notice that $\Delta_t > \delta_t$.

Additionally, other factors have a significant impact in the decision for Δ_t and δ_t, or even in publishing the current state to prevent risk, such as forced expiration spam attacks, colluding miner attacks, or even signed fees becoming obsolete (i.e. too low to be included by miners) [14]. These attacks and their workarounds are out of the scope of this document.

Having such basic notation, it is possible to deepen into correctness properties and performance measures of channels and factories.

5.3. Components of n-party Payment Channels.

We will assume that a group U of n users want to create an n-party channel. Any n-party payment channel has four components:

1. A cryptographic scheme, with specified algorithms and protocols for key generation, signing and verification. For a Lightning Factory, we require a non-interactive Aggregate Signature (AS) scheme, such as Bellare et al.'s [1] BNN.
2. A channel creation protocol, in which members of U sign and share signed messages m_i in order to set up the channel.

3. A channel update protocol, in which members update the outputs of the channel, that precedes and/or invalidates all previous updates' outputs, including the first one of the setup.
4. A channel close protocol, with which members enforce the last update.

These components must be correct. That is, as long as the adversary does not get explicit access to honest parties' keys, these components must guarantee the no-lock and no-steal properties of honest parties' funds, described in section 5.

5.4. The Adversarial Model

We consider an adversary F with the following capabilities:

- F can control the network to read, drop or redirect all messages. While this is all the correctness a channel requires, its performance improves if F is allowed to drop messages, but not to read or redirect them to a new recipient.
- F can take full control and corrupt any subset of players $S \subsetneq U$ at any time, learning its entire state (stored messages, signatures, etc.). if $S = U$ then there is no honest party (i.e. there is no victim of the adversary).
- For any uncorrupted user u_i, F can decide to conduct and adaptive *chosen-message-and-subroup* attack: at any time, it can request u_i to execute the protocol on some specified message with some specified co-signers (who might be, in turn, corrupted by F).

5.5. Attacks in Factories

In this section, we present two attacks common to factories in general, before we compare the impact of these attacks in the different factories. To the best of our knowledge, we are the first to bring up these attacks.

5.5.1. Broken factory attack

Burchert et al. [5] propose a workaround in a channel factory $\mathcal{F}_{\{u_i\}_{i=0}^{n-1}}$ in which one of the members, u_j, goes unresponsive, either during updating (Stale Factory attack) or not. When this happens, the factory cannot be updated, meaning that opened channels must remain open for the period of time that they lastly signed when u_j was still responsive, $A^{l_f}_{\{u_i\}_{i=0}^{n-1}}(tlock :$

$t_{min} < t_1 - (l_f - 1)\delta_t)$, being $A^{l_f}_{\{u_i\}_{i=0}^{n-1}}$ the lastly signed allocation transaction of a DMC factory. However, inner-channel states that do not depend on u_j can still be updated within the factory. According to Burchert et al. [5], there is a workaround to create a new factory with these channels in which u_j has no stake, in a way that does not require waiting for t_{min}, by splicing out u_j from the new factory. We show this procedure in figure 11, proving that the workaround does not work, since it can be exploited.

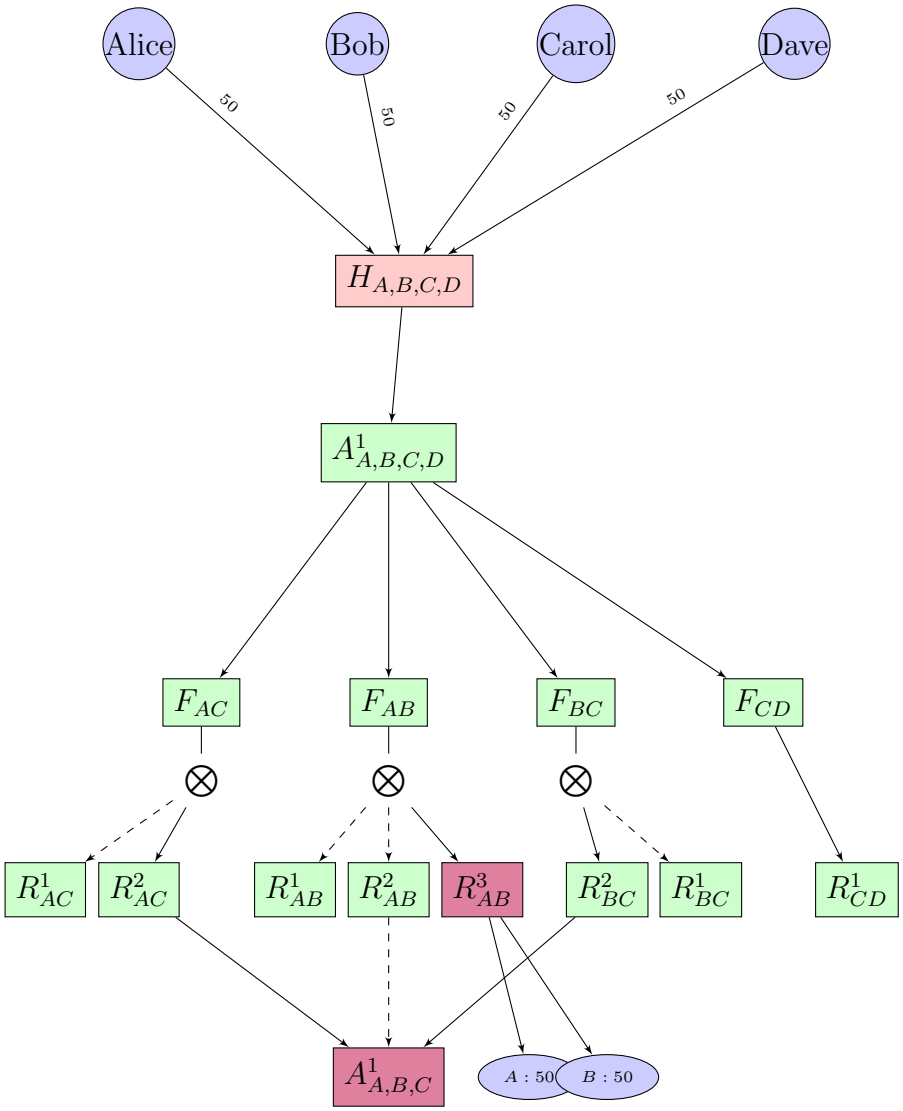

Figure 11: Redirection of inner-channel outputs to a new factory, in the event Dave goes unresponsive, as claimed by Burchert et al. [5]. Notice how the new refund transaction R^3_{AB} invalidates the Allocation $A^1_{A,B,C}$, both in purple. Alice and Bob are the only ones required for such update. Carol might think Alice and Bob's last state is R^2_{AB}, and might accept payments from them in channels of the channel factory of $A_{A,B,C}$. Afterwards, Alice and Bob can publish R^3_{AB}, which will hit the blockchain first and thus Alice and Bob will steal from Carol.

The two signed offline transactions in purple, $A^1_{A,B,C}$ and R^3_{AB}, are at

conflict. If R^3_{AB}, the third (and last) refunding transaction of the DMC between Alice and Bob, is published, then $A^1_{A,B,C}$, the allocation transaction of the new factory, is not valid. Also, Carol does not know that R^3_{AB} exists, thus Alice and Bob are stealing from Carol.

5.5.2. Stale factory attack

A stale factory attack does not affect the correctness of the factory (contrary to a broken factory attack), but rather its performance. Before explaining a stale factory, let us consider a stale channel.

Stale Channel. Suppose a channel between Alice and Bob, with current valid state R^1_{AB}, in which they wish to update to a new state R^2_{AB}. Alice prepares the transaction $R^2_{A.}$, signs it, and gives it to Bob. In such a case, if Bob is malicious, he can decide to stop responding, and Alice will not know if R^1_{AB} will be valid, or if Bob will publish R^2_{AB} instead. We show a stale channel in figure 12.

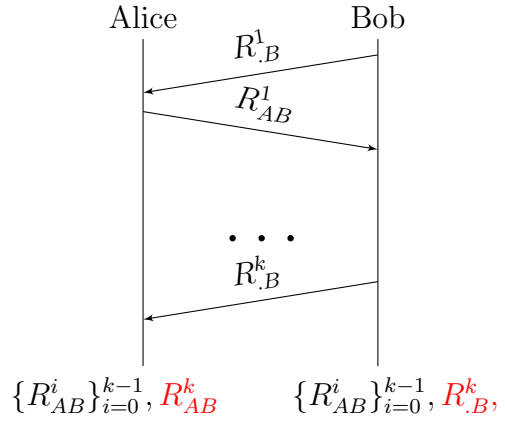

Figure 12: Stale channel. Alice becomes unresponsive after receiving $R^k_{.B}$. Bob cannot know whether R^{k-1}_{AB} is still valid, or if Alice will sign and publish R^k_{AB} instead.

This situation creates an ambiguity that damages Bob as much as Alice, by timelocking his own funds. Also, it does not directly affect the no-steal and no-lock property, but creates ambiguity for an amount of time $O(t_1)$.

Stale Factory attack. A Stale Factory attack does not affect the correctness of the factory, but rather its performance. It can be, however, significantly damaging for the performance of a DMC Factory. This attack takes place at

the moment of updating the factory. In a DMC Factory, or channel, there is no moment in which two states are equally valid, since one of them will always have a lower locktime. For this reason, an update can be a critical moment, if the last member to sign is malicious.

In a DMC Factory $\mathcal{F}_{\{u_i\}_{i=0}^{n-1}}$, regardless of the communication protocol to update [5], there will always be a set of users $\{u_i\}_{i=0}^{j-1}$ that receive the signatures of the rest $\{u_i\}_{i=j}^{n-1}$ first. As such, not all users will get all signed transactions simultaneously, but some of them will have to sign and provide before ($\{u_i\}_{i=0}^{j-1}$ in this case). Without loss of generality, we treat the set of users who receive the signatures first as a single malicious user, u_j, who receives the new state $A^2_{\{u_i\}_{i\neq j}}(tlock : t_2)$ signed by everybody else but them. If u_j is malicious, they can decide not to communicate the signed $A^2_{\{u_i\}_{i=0}^{n-1}}$, and to go offline instead. Should this happen, any other channel that was modified by this update cannot be used anymore, since the members of the channel have no idea if they have to update the old channel or the new one, resulting in all these channels completely paralyzed for the amount of time t_2. The rest of channels can continue normal operation, and perform their inner-channel updates.

Notice that, contrary as previously thought [5], updating both states for as long as there exists ambiguity is sometimes not possible. If previous state $A^1_{\{u_i\}_{i=0}^{n-1}}$ has two channels $\{F_{AB}, F_{CD}\}$, and the new state $A^2_{\{u_i\}_{i=0}^{n-1}}$ has two different channels $\{F_{AC}, F_{BD}\}$, then a payment Alice to Bob can not happen in the new state, whereas it can in the previous one. Routing within the factory in the second state is also not possible. Same happens for a payment Alice to Carol. As such, Alice's funds are timelocked.

The stale factory attack is a significant drawback of DMC factories. Even if the communication protocol to update, in the best of cases, requires for a stale factory attack that $n-1$ of the members collude, only with one user going unresponsive is sufficient for the rest to suspect that the other $n-1$ might be colluding for a stale factory attack. That is, any update request that was initiated and not finished is potentially a stale factory attack.

5.6. Complexity Analysis

In this section we compare the complexity and security performances of our Lightning Factory versus DMC Factories.

Worst-case lock-in time DMC Factories' worst-case lock-in time is t_1, where t_1 being $A^1_{\{u_i\}}(tlock : t_1)$. Lightning Factories have a constant worst-case lock-in time of Δ_t. Notice that $\Delta_t = c\delta_t,\ c \in \mathbb{R}$, and $\Delta_t << t_1$.

Blockchain check time DMC Factories only have to check the blockchain at $t_1 - (l_f - 1)\delta_t$, where $(l_f - 1)$ being $A^{l_f-1}_{\{u_i\}}$ the lastly signed state. Lightning Factories have to periodically check the status of the blockchain at least every Δ_t, in order to have enough time to prove fraud.

Memory footprint DMC Factories simply need to store the lastly signed $A^{l_f-1}_{\{u_i\}}$. Lightning Factories need to store, for Alice's case, $\{C^{l_f-1,A}_{\{u_i\}}, RA^{l_f-1,A}_{\{u_i\}}\}$ along with all $\{I_{m_i}\}_{i\neq A}$ and the last $\{BR^{l_f-1,i}_{m_i}\}_{i\neq A}$, since the last Breach Remedy fragments can make use of
SIGHASH_NOINPUT to match previous old states. However, if memory is a constraint, Breach Remedy fragments can also be aggregated, and Idle transaction fragments are not necessary for the correctness of the factory, requiring ultimately the size of 3 transactions, compared to that of 1 for DMC Factories. Recall that, at the moment of writing, DMC Factories have been proposed with Schnorr signatures, whose size is twice as much as our proposed BLS signatures, resulting in a final 3/2 ratio of size required for Lightning Factories compared to DMC Factories.

Number of updates DMC Factories are upper-bounded in the number of updates by $\lfloor \frac{t_1}{\delta_t} \rfloor$, where t_1 being $A^1_{\{u_i\}}(tlock : t_1)$. Lightning Factories have an unlimited amount of updates.

Message complexity The protocol proposed by Burchert et al. [5] requires exchanging n^2 messages for each update. Lighting Factory requires $2n$ messages, being ordered in two sets. A first set of n allocation commitment and revocable allocation fragments that are broadcast in an indistinct order, and a second set of n breach remedy fragments that are broadcast afterwards. Note that fragments and their signatures are smaller in size to the transactions and signatures reported in DMC Factories. That is, BLS signatures are half as big as Schnorr.

5.7. Resilience Analysis

Previous work [5] suggested a mechanism for splicing out unresponsive/malicious parties which attempts against the correctness of the factory. The authors suggest the redirection of inner-chanel outputs to a new factory. We have the evidence that this splicing out mechanism is vulnerable to broken factory attack, which cracks the no-steal correctness property.

Using Lightning Factories, there is no risk for a counter party going unresponsive, since the factory can be closed uncooperatively with a small locktime Δ_t to allow for disputes, instead of a period of time representative of

the lifetime of the factory t_1, where t_1 the timelock for the first state, as with DMC. This means that the trade-off between locktime and lifetime of the factory is addressed, being possible to have unlimited lifetime with constant locktime. Also, malicious parties are disincentivized to try fraud, since any other member of the factory can publish a Proof-of-Fraud, and make them lose all funds.

Furthermore, the attacks possible in a channel factory make it difficult for DMC factories to tackle their trade-off, since the properties of long-lasting channels/factories (high number of updates) and low worst-case lock-in time of funds are in direct conflict. However, in a Lightning Factory, given the smaller locktime, not dependent on the number of updates, and the fact that invalidation of states are only signed once everybody has a validation of the new state, the stale factory attack is significantly less bothersome. We compare the impact of such an attack in section 5.8.

Broken factory attack in Lightning Factories. The broken factory attack is common to all factories constructions, including Lightning Factories. The purpose of the broken factory attack is to illustrate that splicing out an unresponsive member from the factory will require waiting the entire timelock, which means that splicing out a member takes as much time and transactions as closing and reopening the factory. For a DMC Factory $\mathcal{F}_{\{u_i\}_{i=0}^{n-1}}$ with initial Allocation transaction $A^1_{\{u_i\}}(tlock : t_1)$, the number of updates possible in the factory is $\lfloor \frac{t_1}{\delta_t} \rfloor$. Considering the factory breaks (i.e. one member becomes unresponsive) at state $A^{t_1}_{\{u_i\}}$, then the amount of time that it takes to wait before being able of 'splicing out' this member is $t_1 >> \delta_t$. We can see how, in order to make the factory useful, t_1 should be high enough. However, the greater i, the greater the worst-case lock-in time t_1.

For a Lightning Factory, this issue is much less relevant. Since all states are invalidated and have a constant locktime Δ_t, not only is the number of updates not upper-bounded, but also the worst-case lock-in time is constant Δ_t, regardless of the number of possible updates.

Stale factory attack in Lightning Factories. The stale factory attack is not damaging in Lightning Factories. In a DMC Factory, a stale factory attack uses the fact that all states are actually valid, but with a different locktime. New states have a lower locktime, which means that the existence and the publication of a new state invalidates previous ones, since these have a higher locktime. As such, if a set of users with current state defined by $A^{k,u_j}_{\{u_i\}}(tlock :$

$t_k = t_1 - (k-1)\delta_t$) initiate but do not finalize sharing with each other the fully signed new Allocation transaction $A^{k+1,u_j}_{\{u_i\}}(tlock : t_{k+1} = t_1 - k * \delta_t)$, users cannot be sure of which of the two transactions will be published, since they do not know if any other user actually obtained a fully signed $A^{k+1,u_j}_{\{u_i\}}$, or if all users have only a partially signed version (i.e. $A^{k,u_j}_{\{u_i\}}$ would be the lastly enforceable state). As such, the factory cannot be further used, at least for a subset of the channels (see section Appendix B.4). In this case, again, users will have to wait $t_k >> \delta_t$ time, unless a new $A^{k+2}_{\{u_i\}}(tlock : t_{k+2} = t_1 - (k+1)\delta_t)$ becomes fully signed.

A Lightning Factory, however, can have two equally valid states, at the same time. Since users do not invalidate the previous state until they receive a new, fully signed one, they are sure that they always have an enforceable state of the factory. As such, if a state update does not complete, and they suspect there is ambiguity as to which state is the valid one in the factory, they can simply publish their owned state, wait for Δ_t, and close the factory with their enforceable version of the state.

5.8. Performance Analysis

In this section, we compare the impact of different timelocks, that of a Lightning Factory and of a DMC Factory. If the factory faces a stale factory situation, some funds from some channels or, in the worst case, all funds from all channels may be locked for the locktime that was set by the timelock. If the factory simply can not be updated because of one or several users being offline, the funds can be moved within the already opened channels, but not outside, for as long as the timelock has not finished.

In both cases, we consider the cost of holding unusable liquidity during each locktime. We call this the interest rate. Similar to the value chosen by Zohar et al's [4], we choose an interest rate of $r = 0.0001096$ per iteration step, when fixed. For our simulation, we consider the factory wishes to update at each iteration step, and each iteration steps reduces the timelock by one. The two above-mentioned cases will change the impact of the locktime, that is, the value of r, but the locktime is not dependent on it. Hence, we use a generic model that works for both cases.

Let p be the probability of a user going unresponsive and/or malicious during an update (or in between updates) in a binomial distribution, and let p be the same for all users, then the expected number of possible updates is $E(n) = \frac{1}{1-(1-p)^n}$. If the factory was opened defining l_f updates, then the

remaining updates are $l_f - E(n)$ otherwise. Finally, to consider the cost, we consider the remaining updates and multiply them by the interest rate r, being the cost $(l_f - E(n))r$ if $E(n) < l_f$.

Figure 13 shows the simulation results[4] of the remaining percentage of updates as a function on the total lifetime of the factory l_f and on the number of users n, expected number of updates j as a function on the probability of a malicious/offline party p, and simulation results showing resulting cost when increasing the lifetime, from left to right. Both in Figure 13 and Figure 14, when fixed, the chosen values of each parameter are: $p = 10^{-7}$ (1 user in 10 million goes unresponsive, either during update or not), $l_f = 10000$ (after 10000 updates the DMC Factory closes), $n = 1000$ (1000 users in the factory). One can see how increasing the number of users immediately affects the lifetime of the channel, due to the increasing chance of a stale attack. Increasing the lifetime of the channel also increases such chance, given the more tries. This is strongly dependent on the value on p chosen, as shown in Figure 13, which we consider to be generous for the DMC construction in our results.

Figure 14 shows the cost as a function on the number of users n, the probability of a malicious/offline party p and the interest rate r, from left to right. The right-most plot of Figure 13, along with the tables in Figure 14, illustrate how the cost, dependent on the interest ϕ, is much lower in our construction compared to a DMC factory, by a factor of more than 3000 in almost all results, and increasing for DMC while remaining constant for our construction LF. These results were obtained by a simulation on 1000 factories per result.

It is, therefore, clear that Lightning Factories scale well better when considering the locktime of fund than DMC factories, regardless of the actual values for n, l_f, p and r. Additionally, we prevent selfish users from continuously publishing outdated states that maximize their rewards, by penalizing them.

5.9. BNN vs Schnorr

Schnorr-based signatures schemes have recently been suggested for Bitcoin [10]. Here, we compare our proposal of a non-interactive AS scheme

[4]the code to obtain such results is available at `https://github.com/ranchalp/LightningFactories-simulations`

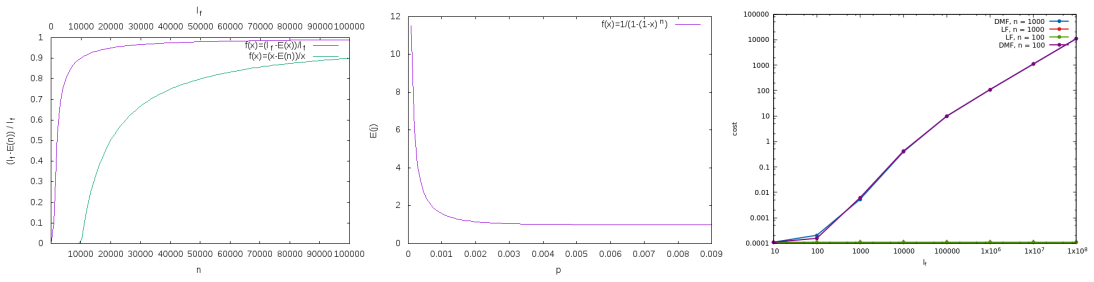

Figure 13: Remaining percentage of updates as a function on the total lifetime of the factory.

n	LF	DMF
10	ϕ	3437ϕ
100	ϕ	3576ϕ
3000	ϕ	6839ϕ
10^4	ϕ	8977ϕ

Interest	LF	DMF
ϕ	ϕ	3785ϕ
5ϕ	5ϕ	3729ϕ
9ϕ	9ϕ	3551ϕ
10ϕ	10ϕ	3686ϕ

p	LF	DMF
10^{-7}	ϕ	3645ϕ
$2 \cdot 10^{-7}$	ϕ	5661ϕ
$5 \cdot 10^{-7}$	ϕ	7963ϕ
10^{-6}	ϕ	8957ϕ

Figure 14: Cost as a function on the number of users n, the probability of a malicious/offline party and the interest rate.

based on BLS [3], such as BNN, with Schnorr-based interactive AS schemes proposed by Maxwell et al. [10].

Signatures size is an important matter in the blockchain, since every bit that is stored in it has a cost, and it will be replicated by all nodes. BLS signatures consist of one group element (i.e. 256 bits), whereas Schnorr signatures are longer, with one group element plus one integer of the size of the group order, (i.e. 512 bits). Therefore, the improvements in terms of size, derived by Maxwell et al. [10], from switching from ECDSA to Schnorr are not only preserved with our proposed BLS-based system, but improved by a factor of 2.

While this design is not exclusive to Bitcoin, adapting it to Bitcoin would require additional modifications from the perspective of validation semantics. Again, this is not a new requirement, and in fact Maxwell et al. [10] already require a new Opcode for cross-input multi-signatures.The reason is that it is necessary to enforce all and only the required transaction fragments at each step. This can be implemented with a new Opcode, in a backward compatible way, such that miners that do not want to upgrade will simply believe transactions involving this Opcode in other miners' blocks.

Furthermore, key aggregation, as an improvement in terms of size for multi-signatures, is not always possible in Bitcoin-like blockchains without a

non-interactive AS. Flags such as sighash-single[5], which allows modification of other outputs in the same transaction, make it impossible to aggregate this other outputs non-interactively with Schnorr-based schemes. Moreover, cross-input multi-signatures, i.e. aggregating all transactions and inputs in a block into one signature, would not be possible with Schnorr-based schemes, since they require the interaction of each input signer. Therefore, there are at least two of Bitcoin's state-of-the-art proposals that are not possible without a non-interactive AS.

6. Conclusions and Discussion

In this paper, we proposed the first extension of Lightning Channels to Lightning Factories, solving the trade-off between the lifetime of the factory and the risk of temporary funds lock-in existing in DMC Factories. Our design scales well better than DMC Factories, offering a constant collateral cost, independent of the lifetime of the channel and the members of the factory. Moreover, our Lightning Factories are resilient to attacks. Driven by the necessity to implement non-interactive aggregate signatures, we proposed BNN as signature scheme. Note that advantages of BNNs lie in a reduced signature size with respect to the Schnorr-based interactive AS schemes proposed by Maxwell et al. [10] by a factor of 2. Moreover, it would be possible to implement it in Bitcoin, requiring additional modifications from the perspective of validation semantics. The reason is that it is necessary to enforce all and only the required transaction fragments at each step. This can be implemented with a new Opcode, in a backward compatible way, such that miners that do not want to upgrade will simply believe transactions involving this Opcode in other miners' blocks. Other than Bitcoin, we believe that this design would be beneficial to other existing and upcoming blockchains.

Appendix A. Payment Channels

Poon et al. [14] proposed initially Lightning Channels for the Lightning Network. Yet, there are currently several on-going projects to implement the concept, most of which are open to other types of payment channels, not only Lightning Channels. In this section, we take a look not only at Lightning Channels, but rather at a variety of possible constructions that can

[5] `https://bitcoin.org/en/glossary/sighash-single`

empower an off-chain payment network. As such, Payment Channels are the cornerstone of the Lightning Network. In this section, we show how to open, update and close a payment channel. We explain Duplex Micropayment Channels in section Appendix A.1, while we mention Lightning Channels in section Appendix A.2.

Appendix A.1. Duplex Micropayment Channels

As already mentioned, all channels have three main phases: open, update and close (sometimes referred to as setup, negotiation and settlement [6]. Duplex micropayment channels rely on timelocks to update and close their channels. Potentially, every update of the channel is a new close of the channel, with smaller timelock. We deepen into the details here below.

Appendix A.1.1. Opening a DMC

Figure A.15 shows the negotiation to create a payment channel in a Bitcoin-like blockchain, with a particular example in figure A.16. To this end, Alice and Bob create a funding and a first refund transaction. First, Bob tells Alice which outputs o he wants to include in this channel (i.e. how much money). Then, Alice creates a funding transaction $F_{\cdot\cdot}$ with hers and Bob's outputs as inputs (2-of-2 multisig), which she does not sign. Notice the two dots in $F_{\cdot\cdot}$ indicate that none of them signed. Following the generic example of figure 3, here is both transactions involving DMC channels:

$F_{\cdot\cdot} = T^0_{\cdot\cdot}$

$T^0_{AB}.kind =$*funding*,

$T^0_{AB}.in = \{T^{-1}_A.out, T^{-1}_B.out\}$,

$T^0_{AB}.out = o_{AB}$,

$T^0_{AB}.conds = (sk_A,\, T^{-1}_A.out), (sk_B,\, T^{-1}_B.out)$ where sk_A and sk_B are the secret keys owned by A and B to digitally sign transactions.

$R^1_{AB} = T^1_{AB}$

$T^1_{AB}.in = F_{AB}.o_{AB}$,

$T^1_{AB}.out = (o^1_A, o^1_B) : o^1_A = o_A \wedge o^1_B = o_B$,

$T^1_{AB}.conds = (sk_A \wedge sk_B, F_{AB}.o_{AB})$ where sk_A with sk_B allow for spending $F_{AB}.o_{AB}$ as long as $blockheight > block(F_{AB}) + tlock$, where tlock is the timelock (e.g. 14400 blocks $\simeq$ 100 days).

The outputs of $F_{\cdot\cdot}$ are the inputs of $R^1_{A\cdot}$. Bob then sings both of them, R^1_{AB}, $F_{\cdot B}$, and shares them with Alice. At this point, since R_{AB} has been signed by both of them, R^1_{AB} is ready to be published. However, the inputs of R^1_{AB} point to some outputs that are not in the blockchain, because $F_{\cdot B}$,

is not fully signed and published yet. Now Alice signs and publishes F_{AB}. Note that at this point Alice and Bob both stored locally R^1_{AB}, ensuring they both can redeem the funds if they want to.

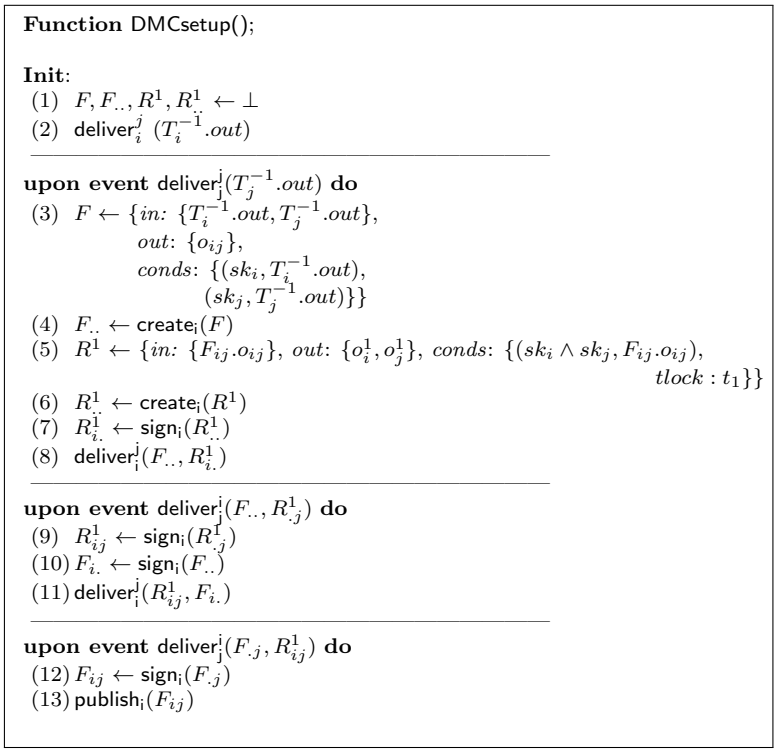

Figure A.15: Opening a DMC. the dots in $F_{..}$ represent two missing signatures, while in $R^1_{i.}$ only j's signature is missing.

Appendix A.1.2. Updating a DMC

Updating a channel balance is equivalent to replacing R^k_{AB} by R^{k+1}_{AB} and so on, that is, being $R^k_{AB} = T^k_{AB}$, then $R^{k+1}_{AB} = T^{k+1}_{AB}$. A payment channel protocol must do this in a way that R^{k+1}_{AB} invalidates R^k_{AB}, to prevent any party from stealing funds by publishing a former state.

For DMCs, update transactions are actually timelocked for an amount of time i, e.g. $R^1_{AB}(tlock : t_1)$ where t_1 is the locktime. This locktime can be either absolute (i.e. absolute blockheight) or relative to the blocknumber in which the funding transaction got included, or instead relative to a subse-

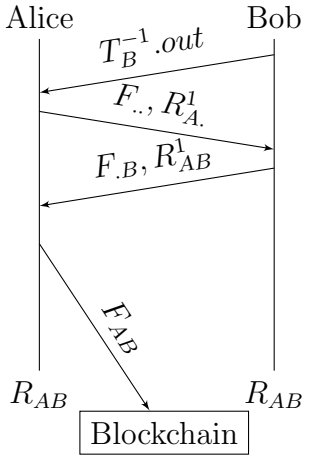

Figure A.16: Example of opening a DMC following protocol described in figure A.15 [7].

quent kick-off transaction [7, 5][6]. Therefore, if the current state corresponds to the refund transaction $R^1_{AB}(tlock : t_1)$, then updating the state is simply signing a new refund transaction with lower locktime $R^2_{AB}(tlock : t_1 - \delta_t)$. This way, assuming no forced expiration spam nor colluding miner attacks, the transaction R^2_{AB} would enter the blockchain before R^1_{AB}, since it has a lower locktime, i.e. if $\delta_t = 1\,day$ then it has one more day as advantage.

Function DMCupdate(R^k_{ij});

Init:
(1) $R^{k+1}, R^{k+1}_{..} \leftarrow \perp$
(2) $R^{k+1} \leftarrow \{in\text{: } \{F_{ij}.o_{ij}\},\ out\text{: } \{o_i^{k+1}, o_j^{k+1}\},\ conds\text{: } \{(sk_i \wedge sk_j, F_{ij}.o_{ij}),\ tlock : t_{k+1} = t_1 - k\delta_t\}\}$
(3) $R^{k+1}_{..} \leftarrow \mathsf{create_i}(R^{k+1})$
(4) $R^{k+1}_{i.} \leftarrow \mathsf{sign_i}(R^{k+1}_{..})$
(5) $\mathsf{deliver}^{\mathsf{j}}_{\mathsf{i}}(R^{k+1}_{i.})$

---

upon event $\mathsf{deliver}^{\mathsf{i}}_{\mathsf{j}}(R^{k+1}_{.j})$ **do**
(6) $R^{k+1}_{ij} \leftarrow \mathsf{sign_i}(R^{k+1}_{.j})$
(7) $\mathsf{deliver}^{\mathsf{j}}_{\mathsf{i}}(R^{k+1}_{ij})$

Figure A.17: Updating a DMC.

[6] `https://en.bitcoin.it/wiki/Timelock`

Appendix A.1.3. Closing a DMC

Given the DMC construction, unilaterally closing the channel requires simply to publish the last refund transaction R_{AB}^{max} before the second last refund transaction becomes valid. Failing to do so may end up in an old state R_{AB}^{j}, $j < max$ being published before, and one of the parties stealing money. Furthermore, if both parties cooperate and are online, they can sign a last state that does not require any locktime, to redeem funds immediately.

Appendix A.1.4. Disadvantages of DMCs

One major disadvantage of DMCs is its trade-off between its lifetime and its funds lock-in. If the first refund transaction $R_{AB}^{1}(tlock : t_1)$ sets too high a locktime t_1 and one of the parties goes unresponsive, then the counterparty has to wait the amount of time t_1 before it can use the locked amount of money. Instead, if t_1 is too low, then the usability of the channel decreases, since participants in the channel will have to close or reopen it more frequently, as the amount of updates is $\left\lfloor \frac{t_1}{\delta_t} \right\rfloor$. A tree structure of nested refund transactions that point to a new $R_{AB}'^{2}$ with a reset timelock can alleviate the trade-off [7], but still have a collateral cost in terms of worst-case blockchain hits. Also, an off-chain kick-off transaction in between the F_{AB} and R_{AB}^{k} can delay the the clock triggering [7], but the number of updates is still upper-bounded by the initial timelock.

Appendix A.2. Lightning Channels

In a Lightning channel, the refund transaction and the subsequent updates do not have any timelock. State transaction invalidation is indeed handled by creating additional transactions. If a state transaction is published by a malicious party, then the honest one has the possibility to remedy to the fraud publishing a Proof-of-Fraud, i.e. a specific transaction that gets back all the funds to the honest party. The different types of transactions and protocol details are presented in the following paragraphs.

In Lightning, as in DMCs, the channel is opened by creating a funding transaction and a first update transaction for refunding. Nevertheless, in Lightning, each update transaction T_{AB}^{k} is split into two transactions, called commitment transactions: $C_{AB}^{k,A}$, which only Alice can publish, and $C_{AB}^{k,B}$, which only Bob can publish. Moreover, for each commitment transaction $C_{AB}^{k,A}$, a Delivery transaction $D^{k,A}$, a Revocable Delivery transaction $RD^{k,A}$ and a Breach Remedy transaction $BR^{k,A}$ are created.

Appendix A.2.1. Opening a Lightning Channel

We show the protocol to create the set of transactions corresponding to T^k_{AB}, state $k = 1$ in figure A.19. Note that we show this procedure for Alice's transactions, being Bob's analogously created, replacing the apex A by B. This first commitment transaction $C^{1,A}_{AB}$ specifies two outputs, which two other transactions spend as follows:

- the output for Bob $C^{1,A}_{AB}.o_B$ is spent by a delivery transaction $D^{1,A}$ that only Bob can spend. This is equivalent to giving to Bob its balance at state 1, as committed by $C^{1,A}_{AB}$.
- the output for Alice $C^{1,A}_{AB}.o_A$ is spent by a Revocable Delivery transaction RD^A_{AB}, that Alice can spend after a timelock $t \geq \Delta_t$, relative to the blockheight (i.e. OP_CSV) in which the commitment transaction $C^{1,A}_{AB}$ is included in the blockchain.

Δ_t should be a time value high enough to give time to the counterparty to see an invalid $C^{k,A}_{AB}$ entering the blockchain, search for the corresponding Proof-of-Fraud (see section Appendix A.2.2), and publish it on the blockchain. One can notice that $\Delta_t \geq \delta_t$. The particular value of Δ_t depends on the connectivity of the nodes to the blockchain. let p_A, p_B the maximum period of time between successive blockchain checks for Alice and Bob (i.e. Alice checks the blockchain every p_A time units), respectively, then $\Delta_t \geq max(p_A, p_B) + \delta_t$. We will simply refer to this value as Δ_t.

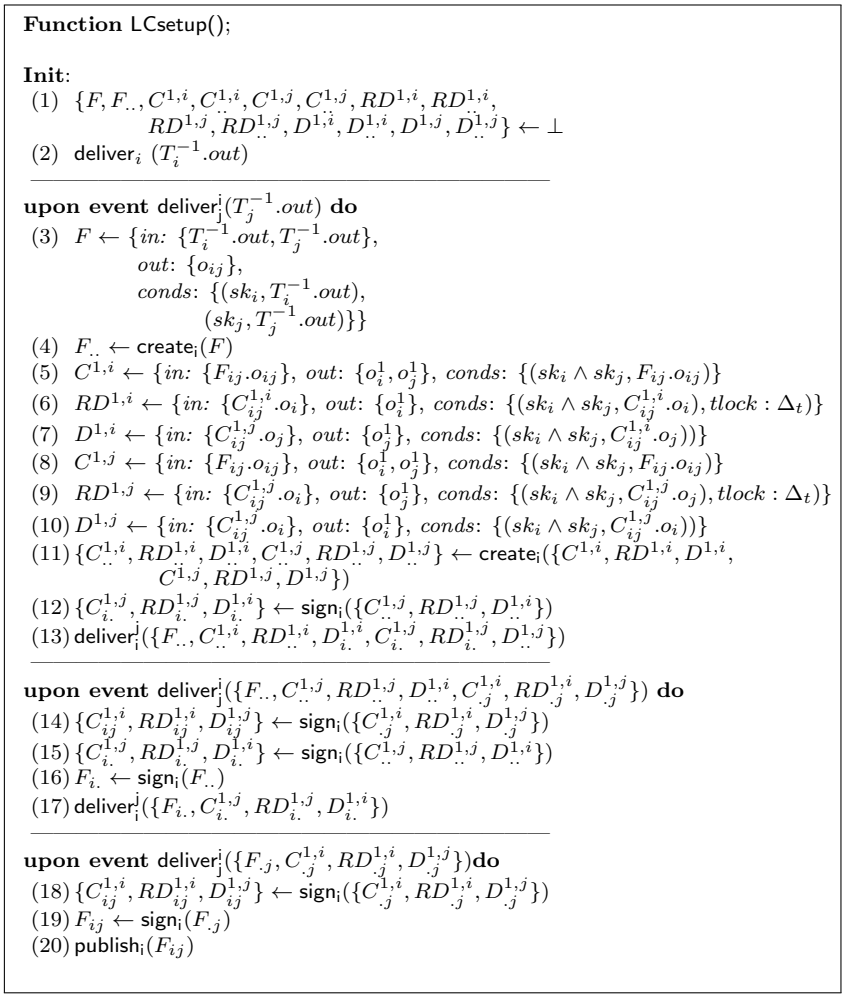

Figure A.18: Opening a Lightning Channel. the dots in $F_{..}$ represent two missing signatures, while in $C_{i.}^{1,i}$ only j's signature is missing.

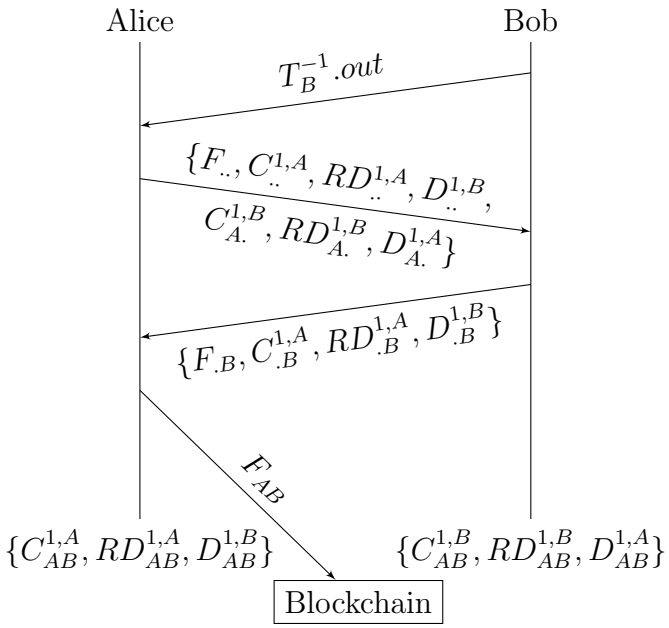

Figure A.19: Example of opening a Lightning channel following protocol described in figure A.18.

Appendix A.2.2. Updating a Lightning Channel

Function LCupdate($C_{ij}^{k,i}, C_{ij}^{k,j}, RD_{ij}^{k,i}, RD_{ij}^{k,j}, D_{ij}^{k,i}, D_{ij}^{k,j}$);

Init:
(1) $\{C^{k+1,i}, C_{..}^{k+1,i}, C^{k+1,j}, C_{..}^{k+1,j}\} \leftarrow \perp$
(2) $\{RD^{k+1,i}, RD_{..}^{k+1,i}, RD^{k+1,j}, RD_{..}^{k+1,j}\} \leftarrow \perp$
(3) $\{D^{k+1,i}, D_{..}^{k+1,i}, D^{k+1,j}, D_{..}^{k+1,j}\} \leftarrow \perp$
(4) $\{BR^{k,i}, BR_{..}^{k,i}, BR^{k,j}, BR_{..}^{k,j}\} \leftarrow \perp$
(5) $finished \leftarrow 0$
(6) $C^{k+1,i} \leftarrow \{in: \{F_{ij}.o_{ij}\}, out: \{o_i^{k+1}, o_j^{k+1}\}, conds: \{(sk_i \wedge sk_j, F_{ij}.o_{ij})\}$
(7) $RD^{k+1,i} \leftarrow \{in: \{C_{ij}^{k+1,i}.o_i\}, out: \{o_i^{k+1}\}, conds: \{(sk_i \wedge sk_j, C_{ij}^{k+1,i}.o_i), tlock: \Delta_t)\}$
(8) $D^{k+1,i} \leftarrow \{in: \{C_{ij}^{k+1,j}.o_j\}, out: \{o_j^{k+1}\}, conds: \{(sk_i \wedge sk_j, C_{ij}^{k+1,i}.o_j))\}$
(9) $C^{k+1,j} \leftarrow \{in: \{F_{ij}.o_{ij}\}, out: \{o_i^{k+1}, o_j^{k+1}\}, conds: \{(sk_i \wedge sk_j, F_{ij}.o_{ij})\}$
(10) $RD^{k+1,j} \leftarrow \{in: \{C_{ij}^{k+1,j}.o_i\}, out: \{o_j^{k+1}\}, conds: \{(sk_i \wedge sk_j, C_{ij}^{k+1,j}.o_j), tlock: \Delta_t)\}$
(11) $D^{k+1,j} \leftarrow \{in: \{C_{ij}^{k+1,j}.o_i\}, out: \{o_j^{k+1}\}, conds: \{(sk_i \wedge sk_j, C_{ij}^{k+1,j}.o_i))\}$
(12) $\{C_{..}^{k+1,i}, RD_{..}^{k+1,i}, D^{k+1,i}, C^{k+1,j}, RD^{k+1,j}, D^{k+1,j}\} \leftarrow$
$\mathsf{create_i}(\{C^{k+1,i}, RD^{k+1,i}, D^{k+1,i}, C^{k+1,j}, RD^{k+1,j}, D^{k+1,j}\})$
(13) $\{C_{i.}^{k+1,j}, RD_{i.}^{k+1,j}, D_{i.}^{k+1,i}\} \leftarrow \mathsf{sign_i}(\{C_{..}^{k+1,j}, RD_{..}^{k+1,j}, D_{..}^{k+1,i}\})$
(14) $\mathsf{deliver_i^j}(\{C_{..}^{k+1,i}, RD_{..}^{k+1,i}, D_{i.}^{k+1,i}, C_{i.}^{k+1,j}, RD_{i.}^{k+1,j}, D_{..}^{k+1,j}\})$

---

upon event $\mathsf{deliver_j^i}(\{C_{..}^{k+1,j}, RD_{..}^{k+1,j}, D_{..}^{k+1,i}, C_{.j}^{k+1,i}, RD_{.j}^{k+1,i}, D_{.j}^{k+1,j}\})$ **do**
(15) $\{C_{ij}^{k+1,i}, RD_{ij}^{k+1,i}, D_{ij}^{k+1,j}\} \leftarrow \mathsf{sign_i}(\{C_{.j}^{k+1,i}, RD_{.j}^{k+1,i}, D_{.j}^{k+1,j}\})$
(16) $\{C_{i.}^{k+1,j}, RD_{i.}^{k+1,j}, D_{i.}^{k+1,i}\} \leftarrow \mathsf{sign_i}(\{C_{..}^{k+1,j}, RD_{..}^{k+1,j}, D_{..}^{k+1,i}\})$
(17) $BR^{k,i} \leftarrow \{in: \{C_{ij}^{k,j}.o_i\}, out: \{o_j^{k}\}, conds: \{(sk_i \wedge sk_j, C_{ij}^{k,j}.o_i))\}$
(18) $BR_{..}^{k,i} \leftarrow \mathsf{create_i}(BR^{k,i})$
(19) $BR_{i.}^{k,i} \leftarrow \mathsf{sign_i}(BR_{..}^{k,i})$
(20) $finished = -1$
(21) $\mathsf{deliver_i^j}(\{C_{i.}^{k+1,j}, RD_{i.}^{k+1,j}, D_{i.}^{k+1,i}, BR_{i.}^{k,i}\})$

---

upon event $\mathsf{deliver_j^i}(\{C_{.j}^{k+1,i}, RD_{.j}^{k+1,i}, D_{.j}^{k+1,j}, BR_{.j}^{k,j}\})$**do**
(22) $\{C_{ij}^{k+1,i}, RD_{ij}^{k+1,i}, D_{ij}^{k+1,j}\} \leftarrow \mathsf{sign_i}(\{C_{.j}^{k+1,i}, RD_{.j}^{k+1,i}, D_{.j}^{k+1,j}\})$
(23) $BR_{ij}^{k,j} \leftarrow \mathsf{sign_i}(BR_{.j}^{k,j})$
(24) $BR^{k,i} \leftarrow \{in: \{C_{ij}^{k,j}.o_i\}, out: \{o_j^{k}\}, conds: \{(sk_i \wedge sk_j, C_{ij}^{k,j}.o_i))\}$
(25) $BR_{..}^{k,i} \leftarrow \mathsf{create_i}(BR^{k,i})$

The timelock for the Revocable Delivery is to allow for Bob to prove Alice is committing fraud, by publishing a Breach Remedy transaction BR^{A}_{AB}. The output $C^{1,A}.o_A$ will end up belonging either only to Bob, if Bob proves fraud before the timelock times out, or only to Alice, after the timelock. If $BR^{1,A}_{AB}$ exists, and Bob publishes it, he proved that Alice committed fraud, receiving all the funds, as $C^{1,A}.o_B$ depends only of Bob's signature. The transactions are analogous for $C^{1,B}_{AB}$.

The result is that, if Alice publishes $C^{1,A}_{AB}$, being $C^{1,A}_{AB}$ an old state (i.e. $BR^{1,A}_{AB}$ exists), then Bob can redeem all funds immediately. If instead state 1 is the last state (i.e. $BR^{1,A}_{AB}$ does not exist), Alice will receive her funds after waiting an amount of time $t \geq \Delta_t$. Proofs-of-Fraud (PoF) are created by counterparties to invalidate the previous state, as we detail in section Appendix A.2.2. In this case, a PoF simply consists of publishing $BR^{1,A}_{AB}$. If $BR^{1,A}_{AB}$ is not published within time t from $C^{1,A}_{AB}$, then the protocol assumes Alice was not fraudulent and allows her to receive her funds using $RD^{1,A}_{AB}$.

Updating a lightning channel goes as following:

1. create new $C^{2,A}_{AB}$, $C^{2,B}_{AB}$, along with the new Delivery and Revocable Delivery transactions. At this moment, the four $C^{1,A}_{AB}$, $C^{1,B}_{AB}$, $C^{2,A}_{AB}$ $C^{2,B}_{AB}$ are simultaneously valid.
2. create both Breach Remedy transactions. $BR^{1,A}_{A.}$ (created and signed by A and given to B), and $BR^{1,B}_{.B}$ (created and signed by B and given to A), that spend the same outputs as $RD^{1,A}_{AB}$ and $RD^{1,B}_{AB}$, respectively, but without a timelock. $BR^{1,A}_{A.}$ and $BR^{1,B}_{.B}$ give all balance to the counterparty, B and A, respectively. Breach Remedy transactions represent thus a sign of honesty from the signer.

Appendix A.2.3. Closing a Lightning Channel

Being $\mathcal{C}^{\mathcal{A}}_{\mathcal{AB}} = \{C^{i,A}_{AB}\}^{l-1}_{i=0}$ the set of commitment transactions that ascribe blame to Alice in the channel between Alice and Bob (i.e. they updated the channel l times), she can unilaterally close a lightning channel publishing the last commitment transaction, $C^{l-1,A}_{AB}$. Bob will immediately receive its funds, while Alice will have to wait for the previously negotiated and signed for amount of time Δ_t, to allow for Bob to prove fraud, after which Alice can redeem her funds. Again, if both parties cooperate and are online, they can simply sign a last state C^{l}_{AB} that directly refunds without any timelock.

Appendix A.3. eltoo Channels

Decker et al. recently proposed eltoo channels [6]. These channels have two types of transactions for updating and closing, called update and settlement transactions, apart from the funding transaction (common to all channels). The update transactions are similar to the commitment transactions in Lightning Channels, in that they commit to a particular state. The settlement transactions are similar to the revocable delivery and delivery transactions, in that they return the balances. One major difference with Lightning Channels is that eltoo channels do not penalize the fraudster. We detail in this section the protocol to open, update and close eltoo channels.

Decker et al. firstly suggested the concept of *floating transactions* for eltoo. A floating transaction is a transaction with the SIGHASH_NOINPUT flag, which instructs the signature creation and the signature verification code to blank the previous output field of the input that is being signed. This way, a the transaction can be rewritten to reference a different transaction output, as long as the input scripts match the output scripts. The process of rewriting the transaction to reference different outputs is called *binding*.

Appendix A.3.1. Opening an eltoo channel

The protocol to setup a channel is no different from that of DMCs, detailed in section Appendix B.1, but changing R^1_{AB} for S^1_{AB}. S^1_{AB} is the settlement transaction. The settlement transaction has a constant timelock $S^1_{AB}(tlock : t)$ such that $t \geq \Delta_t$, similar to the commitment transactions of Lightning Channels.

Appendix A.3.2. Updating an eltoo channel

Updating an eltoo channel, as for Lightning, consists of two steps:

1. Creating and signing a new settlement transaction S^2_{AB} that spends the outputs of a not yet signed update transaction $U^2_{..}$, after some locktime.
2. Signing the update transaction $U^2_{..}$, with no locktime.

Decker et al. propose the inclusion of a new concept, that of state numbers, in order to enforce an ordering of different update transactions U^i with the usage of SIGHASH_NOINPUT. That is, each U^i is flagged with SIGHASH_NOINPUT, in order for each U^i to match the outputs of each other U^j, and of the funding transaction F. However, it is necessary to enforce that U^i can spend the outputs of U^j only if $i > j$. Therefore, state numbers

verify this. This way, the last state can be enforced, since there is no U^l that can be attached to U^{l-1}.

Since the two steps process is similar to that of Lightning Channels, the protocol shown in figure A.20 can be applied here too, replacing all BR by U and the rest of the transactions C, RD, D by S. Also, the same conclusions and properties are applicable here. The only difference is that there is no penalization for fraudsters to prevent them from committing fraud.

Appendix A.3.3. Closing an eltoo channel

Closing a channel consists of publishing the last U_{AB}^{l-1}, waiting for $t \geq \Delta_t$, and later publishing the last S_{AB}^{l-1}. Selfish nodes, however, will always try to commit fraud by publishing the state U^i that gives them the most amount of funds. After, that, the honest node can publish U^{l-1}, invalidating U^i, but not penalizing the selfish node. As above-mentioned, if both parties cooperate they can sign a last U^l that has no locktime, and even share the transaction fee.

Appendix A.4. Comparison

Recall that eltoo channels would get the same results in this comparison as Lightning, other than the message complexity, and the fact that they do not penalize fraudsters.

Worst-case lock-in time DMCs' worst-case lock-in time is t_1, being the amount of updates required $\lfloor \frac{t_1}{\delta_t} \rfloor$, with $R_{AB}^1(tlock : t_1)$. Lightning channels have a constant worst-case lock-in time of Δ_t. Notice that $\Delta_t << t_1$.

Blockchain check time DMCs only have to care about the blockchain at time $t_1 - (l-1)\delta_t$, being R_{AB}^{l-1} the lastly signed state. Lightning channels have to periodically check the status of the blockchain at least every Δ_t time, in order to have enough time to prove fraud.

Memory footprint DMCs simply need to store the lastly signed state R_{AB}^{l-1}. Lightning channels need to store, for Alice's case, $C_{AB}^{l-1,A}$, $RD_{AB}^{l-1,A}$ and $D_{AB}^{l-1,A}$ along with all $\{BR_{AB}^{i,B}\}_{i=1}^{l-2}$, to be able to prove fraud. However, with the help of SIGHASH_NOINPUT, it would be possible to only need to store the last $BR_{AB}^{l-2,B}$, similar to how Decker et al. propose to use it in eltoo [6]. Since eltoo channels do not penalize, they require two transaction less than Lightning to be stored (no need for BR_{AB} nor D_{AB}).

Number of updates DMCs are upper-bounded in the number of updates by $\lfloor \frac{t_1}{\delta_t} \rfloor$, being $R_{AB}^1(tlock : t_1)$. Lightning channels have an unlimited amount of updates.

Message complexity The DMC protocol requires exchanging 3 messages for opening the channel,and 2 per update. In contrast, Lightning Channels require exchanging 3 messages for opening a channel (although of bigger size since the messages contain more transactions), and 3 per update, analogous to eltoo channels.

Stale channel This is not critical in a DMC channel, since Bob can actually only timelock funds to Alice, while he might still timelock his own funds. Nonetheless, in a Lightning Channel, a situation like this would not take place, since Alice will never provide to bob a Breach Remedy of the previous state $BR_A^{1,A}$ if she does not receive the new update $C_{AB}^{2,A}$ signed by Bob. Therefore, if Alice suspects Bob is trying to lock the funds, she can simply publish $C_{AB}^{1,A}$ and close the channel.

Appendix B. DMC Factories

Suppose a set of users $u_0, ..., u_{n-1}$ want to open an amount of channels with each other (e.g. $\{\{u_0, u_1\}, \{u_0, u_2\}, \{u_2, u_n\}, ...\}$.). If they want to open m channels, $n \leq m \leq \binom{n}{2}$, they would need to publish m transactions. However, they can instead join together into a n-of-n multisig output, that of the Hook transaction, $H_{\{u_j\}_{j=0}^{n-1}}$ (notice the subindex still refers to which users signed this transaction). The output of this transaction is the input of another transaction, called the Allocation transaction $A_{\{u_j\}_{j=0}^{n-1}}$, similar to how the outputs of F_{AB} are the inputs of R_{AB} in section Appendix B.1.

Finally, the outputs of $A_{\{u_j\}_{j=0}^{n-1}}$ are the inputs of the respective F_{u_a,u_b} that setup each respective channel, whose outputs point to R_{u_a,u_b} as in section Appendix B.1. Figure 5 shows an instance of a DMC Factory.

Appendix B.1. Opening a DMC Factory

In Figure A.16, we outlined the communication protocol for opening a two-party DMC. Since the DMC Factory involves n parties, the communication protocol to set up and update the channel, i.e. to properly sign $H_{\{u_j\}_{j=0}^{n-1}}$ is not trivially derived from such figure. Figure B.21 shows the protocol to setup the DMC Factory. First, each user is assigned one unique number, creating an ordering $\{u_0, \ldots, u_i, \ldots, u_{n-1}\}$.

After creating each individual channel (as explained in section Appendix A, and having shared the outputs that will serve as input for the Hook, each user u_i prepares and shares a signed Allocation transaction $A_{u_i}^1$. From

this moment on, each user will only sign and share a transaction with r signatures once it receives a new transaction with r signatures, of which at least 1 signature it did not have before. Only the user 0 will sign and release a transaction with r signatures when receiving a transaction with $r-1$ signatures.

This protocol ensures that $\lceil \frac{n}{2} \rceil$ of the users should be cooperatively malicious in order to receive the required n signatures before the rest of the non-malicious users. if the number of malicious users is less than $\lceil \frac{n}{2} \rceil$, the non-malicious users can cooperate to retrieve fully signed transactions $A^1_{\{u_i\}}, H_{\{u_i\}}$, as well as the malicious ones. Notice that $H_{\{u_i\}}$ is never signed before receiving a fully a signed $A_{\{u_i\}}$, thus guaranteeing the no-lock property. The no-steal property is also always met. The malicious users can, however, generate a stale situation, if they are more at least half of the users.

Function DMCFsetup();

//assign indices
...
//share outputs $\{T_j^{-1}.o\}$
...
//set up channels $\{F_{u_j,u_k}\}$
...
upon event channelsSetUp()
(1) $\{H, H_\emptyset, A^1, A^1_\emptyset\} \leftarrow \perp$
(2) $r \leftarrow 1$
(3) $signedAllocation \leftarrow 0$
(4) $A^1 \leftarrow \{in\text{: } \{H_{\{u_j\}}.o\},\ out\text{: } \{F_{u_j,u_k}.in\},\ conds\text{: } \{(\{sk_j\}_{j=0}^{n-1}, H_{\{u_j\}}.o),\ tlock : t_1\}\}$
(5) $A^1_\emptyset \leftarrow create_i(A^1)$
(6) $A^1_{u_i} \leftarrow \mathsf{sign_i}(\mathsf{A}^1_\emptyset)$
(7) $\mathsf{broadcast}_i\ (A^1_i)$

---

upon event $\mathsf{deliver^i}(A^1_{\{u_j\}_{j=i-r}^{i-1}})$ **do**
(8) **if** $i == 0$ **then**
(9) $A^1_{\{u_j\}_{j=n-(r-1)}^{n}} \leftarrow \mathsf{sign_i}(A^1_{\{u_j\}_{j=n-(r-1)}^{n-1}})$
(10) $\mathsf{deliver}^1_0(A^1_{\{u_j\}_{j=n-(r-1)}^{n}})$
(11) **else then**
(12) $A^1_{\{u_j\}_{j=i-(r-1)}^{i}} \leftarrow \mathsf{sign_i}(A^1_{\{u_j\}_{j=i-(r-1)}^{i-1}})$
(13) $\mathsf{deliver}^{i+1}_i(A^1_{\{u_j\}_{j=i-(r-1)}^{i}})$
(14) $r \leftarrow r + 1$

---

upon event $\mathsf{deliver^i}(A^1_{\{u_j\}_{j=i-(n-1)}^{i-1}})$ **do**
(15) $A^1_{\{u_j\}_{j=0}^{n-1}} \leftarrow \mathsf{sign_i}(A^1_{\{u_j\}_{j=i-(n-1)}^{i-1}})$
(16) $\mathsf{broadcast}_i\ (A^1_{\{u_j\}_{j=0}^{n-1}})$
(17) $H \leftarrow \{in\text{: } \{T_j^{-1}.o\},\ out\text{: } \{H_{u_j}.o\},\ conds\text{: } \{(sk_j, T_j^{-1}.o)\}_{j=0}^{n-1}\}$
(18) $H_\emptyset \leftarrow create_i(H)$
(19) $H_{u_i} \leftarrow \mathsf{sign_i}(\mathsf{H}_\emptyset)$
(20) $r \leftarrow 1$
(21) $signedAllocation \leftarrow 1$
(22) **if** $i == 0$ **then**
(23) $\mathsf{deliver}^1_0\ (H_0)$

---

upon event $\mathsf{deliver^i}(A^1_{\{u_j\}_{j=0}^{n-1}})$ **do**
(24) $r \leftarrow 1$
(25) $signedAllocation \leftarrow 1$
(26) **if** $i == 0$ **then**
(27) $\mathsf{deliver}^1_0\ (H_0)$

---

upon event $\mathsf{deliver}^{\mathsf{i}}_{\mathsf{i-1}}(H_{\{u_j\}_{j=i-r}^{i-1}})$ **do**
(28) **if** $signedAllocation == 1$ **then**
(29) **if** $i == 0$ **then**
(30) $H_{\{u_j\}_{j=n-(r-1)}^{n}} \leftarrow \mathsf{sign_i}(H_{\{u_j\}_{j=n-(r-1}^{n-1}})$
(31) $\mathsf{deliver}^1_0(H_{\{u_j\}_{j=n-(r-1)}^{n}})$
(32) **else then**
(33) $H_{\{u_j\}_{j=i-(r-1)}^{i}} \leftarrow \mathsf{sign_i}(H_{\{u_j\}_{j=i-(r-1)}^{i-1}})$
(34) $\mathsf{deliver}^{i+1}_i(H_{\{u_j\}_{j=i-(r-1)}^{i}})$
(35) $r \leftarrow r + 1$

---

upon event $\mathsf{deliver}^{\mathsf{i}}_{\mathsf{i-1}}(H_{\{u_j\}_{j=i-(n-1)}^{i-1}})$ **do**
(36) $H_{\{u_j\}_{j=0}^{n-1}} \leftarrow \mathsf{sign_i}(H_{\{u_j\}_{j=i-(n-1)}^{i-1}})$
(37) **if** $\mathsf{read_i}(H_{\{u_j\}}) == \perp$ **then** // not published
(38) $\mathsf{publish}_i(H_{\{u_j\}})$

Appendix B.2. Updating a DMC Factory

Updating the state of each independent channel within the channel factory is no different from what we detail in section Appendix A.1.2. In fact, one can instead have a Lightning Channel construction at the two-party level, instead of a DMC construction. Only the two parties of this channel in this channel factory, u_a, u_b, need to sign to update the balance of the channel.

However, the channel factory is at the level of the Hook and Allocation transactions. $A^k_{\{u_i\}_{i=0}^{n-1}}(tlock : t_k)$ has a timelock t_k, similar to that of $R^k_{AB}(tlock : t_k)$ in DMCs. To update the channel factory (i.e. closing all channels and leave factory, or closing and/or opening some channels within the factory), all users $u_0, ..., u_{n-1}$ need to agree and sign a new allocation transaction $A^{k+1}_{\{u_i\}_{i=0}^{n-1}}(tlock : t_{k+1})$ such that $t^{k+1} \leq t - \delta_t$. As before, this new transaction will hit the blockchain before the old one, given the lower timelock. Figure B.22 shows the protocol for updating. It follows the same approach illustrated when opening the DMC Factory, but without creating a hook H and with an allocation A^{k+1} instead of A^1.

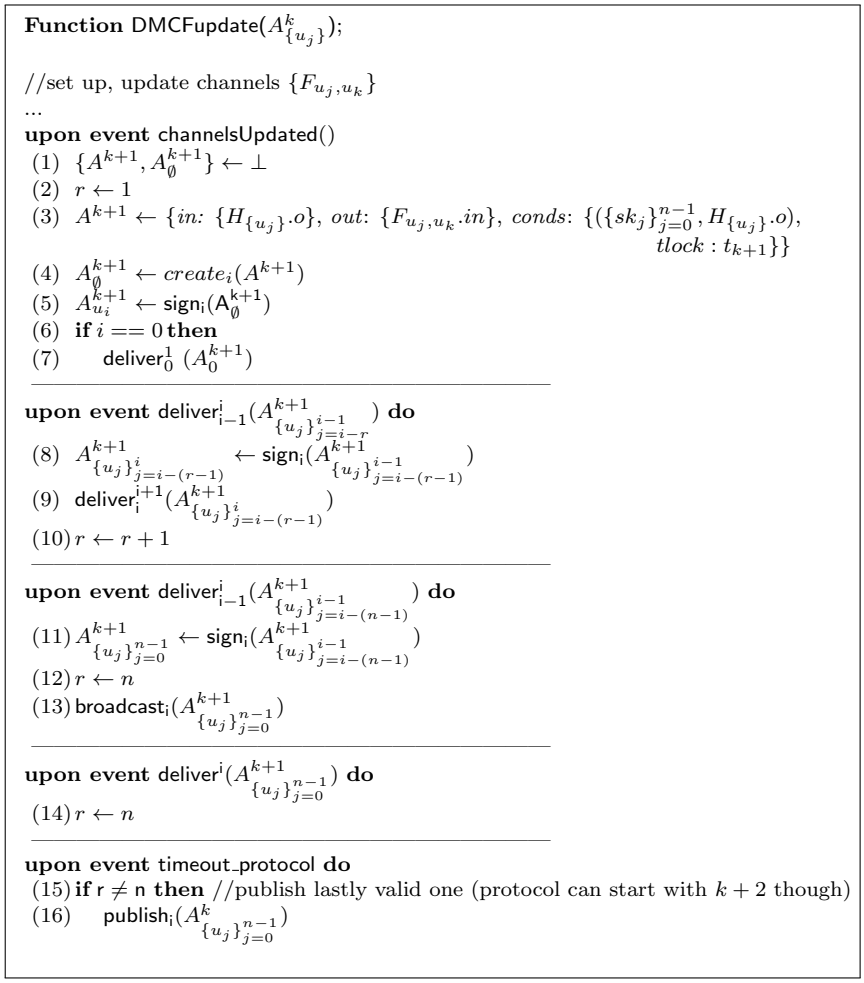

Figure B.22: Updating a DMC Factory, after selecting an ordering, having set up/updated the new/existing channels of the factory (as detailed in section Appendix B.1). Notice $i \in \mathbb{Z}_n$.

Appendix B.3. Closing a DMC Factory

Again, closing a two-party independent channel in a channel factory is equivalent to the procedure explained in section Appendix A.1.3. For the DMC Factory, the procedure is also analogous. If all parties cooperate, they sign a last state that has no relative timelock, referred to as Settlement transaction, and publish it immediately after. Otherwise, they will publish the last Allocation transaction, with the smallest relative timelock, and after such timelock, they can redeem their funds.

Appendix B.4. On DMC Factories

The downsides of using DMCs increase when using a DMC Factory. The trade-off between usability of the channel and the risk of temporary funds lock-in increases drastically in this case. In a DMC Factory of n users, it requires only one of them to completely timelock the funds inside the factory. Getting funds out of channels within the factory before the last signed timelock is impossible for as long as one user remains unresponsive.

Again, while this trade-off can be improved by using a kick-off transaction and nested refund transactions, also the worst-case blockchain footprint increases drastically, as only one user can enforce all the nested transactions into the blockchain, increasing the blockchain fees.

Appendix C. Correctness of Lightning Factories

Appendix C.1. Lightning Factories: proofs

Lemma 1. *Under the assumption that Bitcoin as a Clock is reliable, and that the channels of the factory have been created following a correct protocol, the protocol* LFsetup() *satisfies the no-lock property.*

Proof. We can prove this by contradiction. Suppose the no-lock property does not hold. This means that, given the group of n users $\{u_j\}_{i=0}^{n-1}$ that want to setup the factory, a subgroup of them, of size $a < n$, $\{u_j\}_{i=0}^{a-1}$, has succeeded in locking up the funds of the rest $\{u_j\}_{i=0}^{n-a-1}$. This is only possible if the group of attackers owned and published a fully signed hook transaction, before at least one of the malicious users signed and shared the allocation commitment and the revocable allocation transaction with the honest users. However, this means that all honest users should have executed line 8 in figure 7 before at least one of the malicious users executed line 4. This is a contradiction, because no user will execute line 8 unless line 6 is true, which is only true if all users executed line 4. It follows that the protocol satisfies the no-lock property. □

Lemma 2. *Under the assumption that Bitcoin as a Clock is reliable, and that the channels of the factory have been created following a correct protocol, the protocol* LFsetup() *satisfies the no-steal property.*

Proof. This proof is analogous to that of no-lock property, since line 6 of figure 7 also verifies that all users shared and signed fragments matching the agreed-upon balance, otherwise it returns false. □

Theorem 1. *Under the assumption that Bitcoin as a Clock is reliable, and that the channels of the factory have been created following a correct protocol, the protocol* LFsetup() *is correct,*

Proof. The protocol is correct if it satisfies the no-lock and no-steal properties. As proved in lemmas 1 and 2, it satisfies both. □

Lemma 3. *Under the assumption that Bitcoin as a Clock is reliable, and that the channels of the factory have been created following a correct protocol, the protocol* LFupdate() *satisfies the no-lock property.*

Proof. Again, we can prove this by contradiction. Suppose the no-lock property does not hold. This means that, given the group of n users $\{u_j\}_{i=0}^{n-1}$ in the factory, a subgroup of them, of size $a < n$, $\{u_j\}_{i=0}^{a-1}$, has succeeded in locking up the funds of the rest $\{u_j\}_{i=0}^{n-a-1}$. This is only possible if the group of attackers owned the breach remedy fragment of state k of each honest user, before at least one of the malicious users signed and shared the allocation commitment and the revocable allocation transaction of the new state k with the honest users. However, this means that all honest users should have executed line 11 in figure 10 before at least one of the malicious users executed line 5. This is a contradiction, because no user will execute line 11 unless line 7 is true, which is only true if all users executed line 5. It follows that the protocol satisfies the no-lock property. □

Lemma 4. *Under the assumption that Bitcoin as a Clock is reliable, and that the channels of the factory have been created following a correct protocol, the protocol* LFupdate() *satisfies the no-steal property.*

Proof. This proof is analogous to that of lemma 3, since line 7 of figure 10 also verifies that all users shared and signed fragments matching the agreed-upon balance in state $k + 1$, otherwise it returns false. □

Theorem 2. *Under the assumption that Bitcoin as a Clock is reliable, and that the channels of the factory have been created following a correct protocol, the protocol* LFupdate() *is correct,*

Proof. The protocol is correct if it satisfies the no-lock and no-steal properties. As proved in lemmas 3 and 4, it satisfies both. □

[1] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted Aggregate Signatures. *International Colloquium on Automata, Languages and Programming - ICALP*, (June), 2007.

[2] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. pages 416–432, 2003. URL: https://doi.org/10.1007/3-540-39200-9_26, http://dx.doi.org/10.1007/3-540-39200-9_26 doi:10.1007/3-540-39200-9_26.

[3] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. volume 17, pages 297–319, 2004. URL: https://doi.org/10.1007/s00145-004-0314-9, http://dx.doi.org/10.1007/s00145-004-0314-9 doi:10.1007/s00145-004-0314-9.

[4] Simina Brânzei, Erel Segal-Halevi, and Aviv Zohar. How to charge lightning. *CoRR*, abs/1712.10222, 2017. URL: http://arxiv.org/abs/1712.10222, http://arxiv.org/abs/1712.10222 arXiv:1712.10222.

[5] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 361–377. Springer, 2017.

[6] Christian Decker, Rusty Russell, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. *White paper: https://blockstream.com/eltoo.pdf*.

[7] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. pages 3–18, 2015. URL: https://doi.org/10.1007/978-3-319-21741-3_1, http://dx.doi.org/10.1007/978-3-319-21741-3_1 doi:10.1007/978-3-319-21741-3_1.

[8] Stefan Dziembowski, Sebastian Faust, and Kristina Hostakova. Foundations of state channel networks. *IACR Cryptology ePrint Archive*, 2018:320, 2018. URL: https://eprint.iacr.org/2018/320.

[9] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM*

SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pages 439–453, 2017. URL: http://doi.acm.org/10.1145/3133956.3134033, http://dx.doi.org/10.1145/3133956.3134033 doi:10.1145/3133956.3134033.

[10] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *IACR Cryptology ePrint Archive*, 2018.

[11] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Christopher Cordi, and Patrick McCorry. Sprites and State Channels: Payment Networks that Go Faster than Lightning. *CoRR*, 2017. URL: http://arxiv.org/abs/1702.05812, http://arxiv.org/abs/1702.05812 arXiv:1702.05812.

[12] Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. *White paper*, pages 1–47, 2017. URL: http://plasma.io/plasma.pdf.

[13] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, and Aleksey Ostrovskiy. Flare: An Approach to Routing in Lightning Network. *White Paper (bitfury. com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_n etwork_7_7_2016. pdf)*, page 40, 2016.

[14] Draft Version, Joseph Poon, and Thaddeus Dryja. The Bitcoin Lightning Network. *draft version 0.5*, i:1–22, 2016.