Lightning Factories

Alejandro Ranchal Pedrosa
EIT Digital Master School
Sorbonne Université, Paris, France
CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
alejandro.ranchal_pedrosa@etu.upmc.fr

Maria Potop-Butucaru
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, Paris, France
maria.potop-butucaru@lip6.fr

Sara Tucci-Piergiovanni
CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
sara.tucci@cea.fr

ABSTRACT

Bitcoin, the most popular blockchain system, does not scale even under very optimistic assumptions. Lightning networks, 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 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 where 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 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.

CCS CONCEPTS

• **Computer systems organization** → *Peer-to-peer architectures*; *Dependable and fault-tolerant systems and networks*;

KEYWORDS

Bitcoin, Blockchain, Scalability, Lightning Network.

ACM Reference Format:
Alejandro Ranchal Pedrosa, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. 2019. Lightning Factories. In *Proceedings of ACM SAC Conference (SAC'19).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 *Layer2* 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 *Lihgtning Network* [14]. However, as pointed out by Poon et al. [14], Lightning Networks do 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_B^1 moves an amount of 20 from the B's account to the C's one, i.e. $T_B^1.out = \{(20, C)\}$. The relationship between the T_B^1's input and T_A^0's output (represented by an arrow in the figure) creates a dependency between transactions, we say in this case that T_B^1 spends (fully or partially[1]) the output of the transaction T_A^0. Note that each transaction is executed when registered in the blockchain and that by construction T_B^1 cannot be registered before T_A^0 since T_B^1 refers to T_A^0 outputs.

[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.

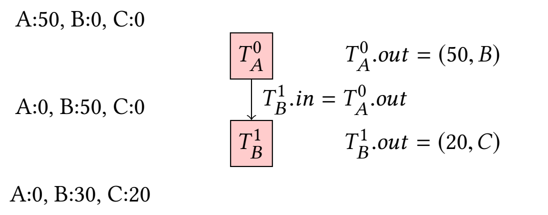

Figure 1: Example of a chain of transactions moving funds from A to B and from B to C. On the left-hand side of the picture, the state of balances before and after the execution of each transaction.

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 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., $T_{..}^0$ 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.

Once the fund transaction is created as well as the first refunding 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. Let us note that in this context, a malicious

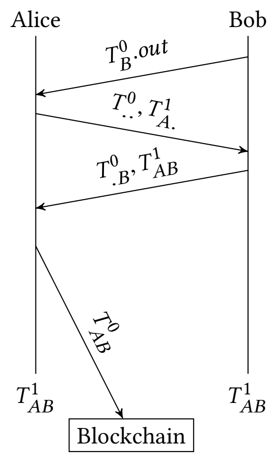

Figure 2: Example of opening a channel by exchaning messages. The funding transaction and the first refund transactions are created and the funding transaction is sent to the blockchain.

party may want to publish on-chain an old balance (if this balance favors him), in this case we say that the malicious party *commits a fraud*. Moreover, a party can go unresponsive either maliciously or involuntarily.

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 timelocks 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 the Lightning 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 3. 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: $C_{AB}^{k,A}$ that only Alice can send to the blockchain, and $C_{AB}^{k,B}$ 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 $RD_{AB}^{k,A}$ after a timelock (by B through $RD_{AB}^{k,B}$, respectively). During the creation of $RD_{AB}^{k,A}$ the protocol makes sure to create as well $D_{AB}^{k,A}$ which refunds B immediately (no time-locks). Proofs-of-frauds can be achieved through the Breach Remedy transactions $BR_{AB}^{k,A}$, $BR_{AB}^{k,B}$, respectively. These transactions spend the same outputs as $RD_{A,B}^{k,A}$ and $RD_{A,B}^{k,B}$, but without a timelock and they give all the balance to the counterparty.

In the proposed scheme, if B (the same applies to A) gets unresponsive, funds can be retrieved unilaterally by A thanks to $RD_{AB}^{k,A}$, 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 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,

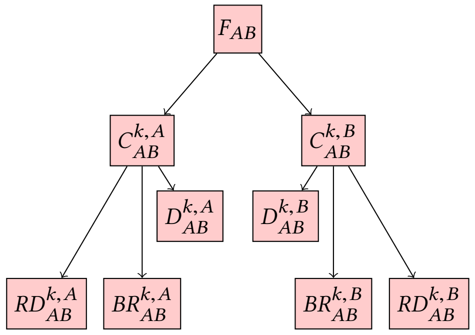

Figure 3: Channel state k between Alice and Bob.

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 $(pk_1, m_1), \dots, (pk_n, m_n)$, 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.

4.3 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.

4.3.1 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 $C^{k,j}_{\{m_i\}^{n-1}_{i=0}} = \sum^{n-1}_{i=0} C^{k,j}_{m_i}$, 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 $RA^{k,j}_{\{m_i\}^{n-1}_{i=0}} = \sum^{n-1}_{i=0} RA^{k,j}_{m_i}$, with a lock-time relative from the inclusion of $C^{k,j}_{\{m_i\}}$ 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 $C^{1,A}_{m_B}$ $\mathcal{T}$ is no locktimed; $\mathcal{I}$ 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 $C^{1,A}_{\{m_i\}} = \sum^{n-1}_{i=0} C^{1,A}_{m_i}$ 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 $C^{1,A}_{\{m_i\}}$; $\mathcal{O}$ is one aggregate signature output, $C^{1,A}_{\{m_i\}}.o$.

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.

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

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 $RA^{1,A}_{\{m_i\}} = \sum^{n-1}_{i=0} RA^{1,A}_{m_i}$ 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, $RA^{1,A}_{\{m_i\}} = \sum^{n-1}_{i=0} RA^{1,A}_{m_i}$, 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 $\sum^n_{i=0} o^{RA^1}_i = \mathcal{B}$, 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].

Notice how, the same way 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 $T^{-1}_i.o$, 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 4a).

4.3.2 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, $C^{k+1,A}_{\{m_i\}} = \sum^{n-1}_{i=0} C^{k+1,A}_{m_i}$ and $RA^{k+1,A}_{\{m_i\}} = \sum^{n-1}_{i=0} RA^{k+1,A}_{m_i}$
(2) invalidate the previous state k, creating Proofs-of-Fraud.

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

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

$\{C^{k,A}_{m_j}\}_{j \neq A}$ 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.

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.

Update Protocol. Figure 4b 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.

4.3.3 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 $u_{j'}$, including u_j, can publish $RA^{k,j'}_{\{m_i\}_{i=0}^{n-1}}$ 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

5.1 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

```
Function LFsetup();

//assign indices
...
//share outputs $\{T_j^{-1}.o\}$
...
//set up channels $\{C_{u_j,u_k}\}$
...
upon event channelsSetUp()
    $RA_\emptyset^1 \leftarrow <\mathcal{P} = any,\ \mathcal{S} = u_i,\ \mathcal{T} = t_1,$
        $\mathcal{I} = \{C_{\{u_i\}}^1.o\},\ \mathcal{O} = RA_{\{u_i\}}^1.o_{u_i},\ \mathcal{S}t = 1>$
    $C_\emptyset^1 \leftarrow <\mathcal{P} = \text{any but } u_i,\ \mathcal{S} = u_i,\ \mathcal{T} = \emptyset,$
        $\mathcal{I} = \{H.o\},\ \mathcal{O} = C_{\{u_i\}}^1.o,\ \mathcal{S}t = 1>$
$\{RA_{m_i}^1,\ C_{m_i}^1\} \leftarrow \text{sign}_i\{RA_\emptyset^1,\ C_\emptyset^1\}$
    broadcast $(\{C_{m_i}^1,\ RA_{m_i}^1\})$
——————————————————————————
upon event deliver$^i(C_{m_j}^1,\ RA_{m_j}^1)$ do
    store$(\{C_{m_j}^1,\ RA_{m_j}^1\})$
    if allReceived($\mathcal{S}$t = 1) then
        //start with hook
        $H_\emptyset \leftarrow <\mathcal{P} = any,\ \mathcal{S} = u_i,\ \mathcal{T} = \emptyset,$
        $\mathcal{I} = \{T_i^{-1}.o\},\ \mathcal{O} = H.o,\ \mathcal{S}t = \emptyset>$
        $H_{m_i} \leftarrow \text{sign}_i(H_\emptyset)$
        broadcast $(H_{m_i})$
```

(a) 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.

```
Function LFupdate($C_{\{u_j\}}^k,\ RA_{\{u_j\}}^k$);

//set up, update channels $\{F_{u_j,u_k}\}$ ...
upon event channelsUpdated()
    $C_\emptyset^{k+1,i} \leftarrow <\mathcal{P} = i,\ \mathcal{S} = u_i,\ \mathcal{T} = \emptyset,\ \mathcal{I} = \{H_{\{u_j\}}.o\},$
        $\mathcal{O} = C_{\{u_j\}}^{k+1}.o,\ \mathcal{S}t = 1>$
    $RA_\emptyset^{k+1} \leftarrow <\mathcal{P} = any,\ \mathcal{S} = u_i,\ \mathcal{T} = t_1,\ \mathcal{I} = \{C_{\{u_j\}}^{k+1}.o\},$
        $\mathcal{O} = RA_{\{u_j\}}^{k+1}.o_{u_i},\ \mathcal{S}t = k + 1>$
    $C_\emptyset^{k+1} \leftarrow <\mathcal{P} = \text{any but } u_i,\ \mathcal{S} = u_i,\ \mathcal{T} = \emptyset,\ \mathcal{I} = \{H_{\{u_j\}}.o\},$
        $\mathcal{O} = C_{\{u_j\}}^{k+1}.o,\ \mathcal{S}t = k + 1>$
$\{C_{m_i}^{k+1,i},\ RA_{m_i}^{k+1},\ C_{m_i}^{k+1}\} \leftarrow \text{sign}_i(\{C_\emptyset^{k+1,i},\ RA_\emptyset^{k+1},\ C_\emptyset^{k+1}\})$
    broadcast$_0^{k+1}$ $(\{C_{m_i}^{k+1},\ RA_{m_i}^{k+1}\})$
——————————————————————————
upon event deliver$^i(\{C_{m_j}^{k+1},\ RA_{m_j}^{k+1}\})$ do
    store$(\{C_{m_j}^{k+1},\ RA_{m_j}^{k+1}\})$
    if allReceived($\mathcal{S}$t = k + 1) then //start with breach remedy
        $BR^{k,i} \leftarrow <\mathcal{P} = any,\ \mathcal{S} = u_i,\ \mathcal{T} = \emptyset,\ \mathcal{I} = \{C^{1,i}.o\},\ \mathcal{O} = \emptyset,\ \mathcal{S}t = k\}$
        $BR_\emptyset^{k,i} \leftarrow create_i(BR)$
        $BR_{m_i}^{k,i} \leftarrow \text{sign}_i(\text{BR}_\emptyset^{k,i})$
        broadcast$_i(BR_{m_i}^{k,i})$
——————————————————————————
upon event timeout_protocol do
    if notAllReceived($\mathcal{S}$t = k + 1) then
        //publish lastly valid one (no breach remedy issued)
        publish$_i(\sum_{j=0}^{j-1} C_{m_j}^{k,i})$
```

(b) Updating a Lightning Factory.

Figure 4: Left: Lightning Factory setup protocol. Right: Lightning Factory update protocol.

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.

5.2 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.

5.3 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 $E(n) < l_f$.

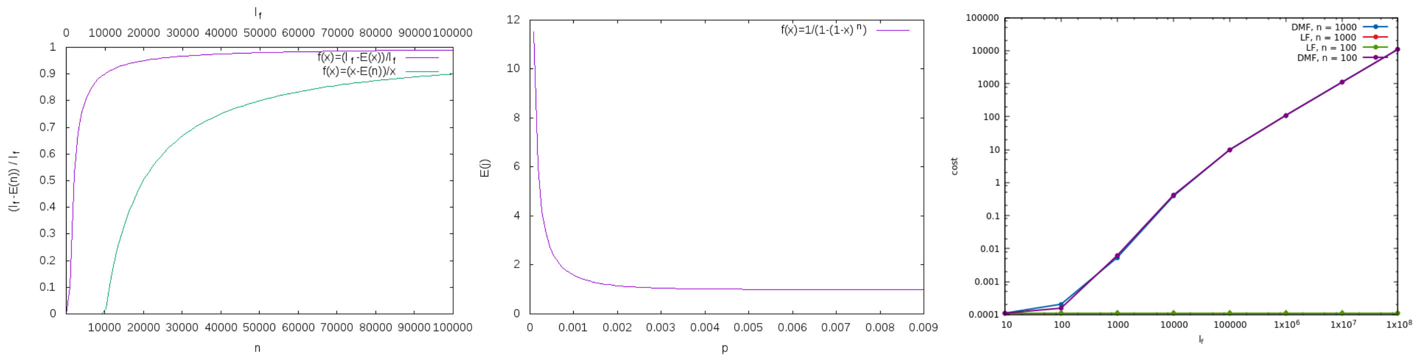

Figure 5: Remaining percentage of updates as a function on the total lifetime of the factory.

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

Figure 5 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 5 and Figure 6, 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 5, which we consider to be generous for the DMC construction in our results.

Figure 6 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 5, along with the tables in Figure 6, 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.

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.

ACKNOWLEDGMENTS

The authors would like to thank Yannick Seurin for our many insightful discussions related to the cryptographic schemes mentioned in this paper.

REFERENCES

[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, 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, 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, arXiv:1712.10222.

[5] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks. In *International Symposium on*

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

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, 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, 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, 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 et-work_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.