Fulgurite

Trey Del Bonis - 2018-12-05 - https://tr3y.io/

Fulgurite is a new way to think about how to “do” Lightning on Bitcoin that I developed at the DCI in October/November 2018. I’ve been at the DCI since June and I work on Lit, our Lightning implementation. Lit: https://github.com/mit-dci/lit

I’m designing how roughly how Fulgurite should “*actually work*” in Rust. It’s not complete yet but the core logic is laid out and explains some things I might accidentally gloss over here. I didn’t do this in LaTeX because this *is not* trying to be a formal research paper or anything of the sort. See on GitLab, under “xchan” (for “eXtensible”): https://gitlab.com/delbonis/roz

(master at time of writing is `d356886852686da61aefc5367b073e2e5abf5476`)

There’s also a presentation I gave at one of our research meetings before setting out to work on the code. Some minor things have changed, and I’ve had to implement some more things that the presentation doesn’t cover. I also stole some images from those slides, which is why the color schemes might not match. You don’t have to read that presentation to understand this.

Link: https://docs.google.com/presentation/d/1oeho3JRTGskx0K0gEBFJbHJeTt0ELeoQmlGQGgrFfBI/edit?usp=sharing

Abstract
Lightning has made strides towards making day-to-day use of Bitcoin more fluid, but it’s not as flexible as it can be. Fulgurite is a design for Lightning to make it general enough to support many of the hypothetical features that have been discussed in the community, like splicing funds between channels and on-chain addresses, multiparty channels, recursive channels, and others with relatively little effort by tracking channel state in a semi-novel way by borrowing ideas from other projects like Plasma. On-chain, it looks exactly the same as Lightning does today, but this proposes a new way to manage and communicate the state/signature information for channels. This would require some significant changes to the BOLT peer protocol but would make the network more extensible and may allow for future upgrades with less coordination.

1. Lightning as a Blockchain

There's a rough correspondence between the utxoset and the state of a channel that's been discussed before in other places. Blocks describe a transition between two sets of utxos. Transactions are the basic units in these state transitions, working on small parts of the utxoset.

In Plasma, state channels are blockchains that are linked to their host chain with a parent contract on the host chain. Funds are moved into the contract and users can interact with them on the plasma chain, and these changes can be reflected on the host chain in some implementation-specific way. Ethereum gives you a lot of flexibility in this regard. But since Lightning channels are fully synchronous and require cooperation from all parties to update, we don't actually need the added flexibility. In fact, our channels aren't *technically* blockchains in that they don't rely on them for determining the current state (like you would do when resyncing a chain), but instead it's used to track state history and unambiguously refer to states using the hash of the transition that produced it. We can actually throw out very old historical state transitions without any loss of security if we really want to.

Really we just care about the "hash-linked state transitions" part. We can define a state transition as an aggregate of multiple individual operations.

ops : state transitions :: txs : blocks

If we keep a base case of the initial state during channel construction, and then exclusively make changes to the state by applying particular operations to it, then we can be sure we can't construct "invalid" states. These operations operate on "channel partitions", and I'll talk about those more later. Borrowing syntax from Haskell, the function signature looks like this:

```
applyStateTsn :: [Operation] -> ChanState -> ChanState
```

So it should be expected that we have something that looks like a mempool except for ops instead of txs. As far as deciding when to include a set of ops as a state transition, it's a lot looser than for a cryptocurrency blockchain. Under light load it might make sense to *immediately* have a node produce a new transition when it decides it needs to. Under heavier load it might make more sense to have an agreed-upon batching window and produce them every 1 second, with all of the ops initiated during that period included in the transition.

The operations in Lightning as originally specified, would be something like this:

- Push (not done in BOLT, although commonly used for direct payments in Lit)
- AddHtlc
- ClaimHtlc
- ExpireHtlc

2. Generalize Funds as Partitions

Really, what does a channel participant's balance look like on-chain? It's just an output to a pubkey hash. And HTLCs are *also* just outputs, but to a script hash (with some extra logic on our end attached to them for on-chain claiming). But in the channel, we think of them as a portion of the total balance, that goes out to some particular contract/address/etc., instead of a numerical quantities for "alice's balance" and "bob's balance" and "this HTLC in this direction" and "that HTLC in that direction" and things.

Instead, we have "user balance" partitions and "HTLC partitions" and other partitions for anything else we care about. Partitions know their balance, and keep with them the other parameters involved in the operation validation rules. They also have a name that makes it easier to refer to them when constructing operations, think of it like an Ethereum account address. We need something to define them more generally than just an "owner", since things like an HTLC or a subchannel don't really *have* single owners. A subchannel partition is just be a partition that pays to the funding address of another channel. There's some special semantics doing state updates with subchannels that I'll cover later on.

When rendering a tx, each partition produces some kind of Contract structure (or potentially more than one) that's used when generating the break tx blobs. These contracts are collected from all of the partitions and it's up to the tx engine to decide what to do with them, in the case of LN-penalty it's setting up maturity clauses around them, in the case of Eltoo it can just put them all on the timelocked draining tx. And of course it's simpler for cooperative close txs since we just put the contract outputs without any extra logic spending from the fund tx and that's it.

Operations are signed by one or more channel participants (like you'd sign a tx, but with a different set of keys than chain keys), and have a TTL where we can enforce timeouts to applying operations (and get some replay protection). So for a Push operation, you'd include the amount you're pushing and who you're sending it to.

Any funds coming from inputs that partitions don't claim can be implicitly included in fees calculation, see later.

2.1 Why not just UTXOs?

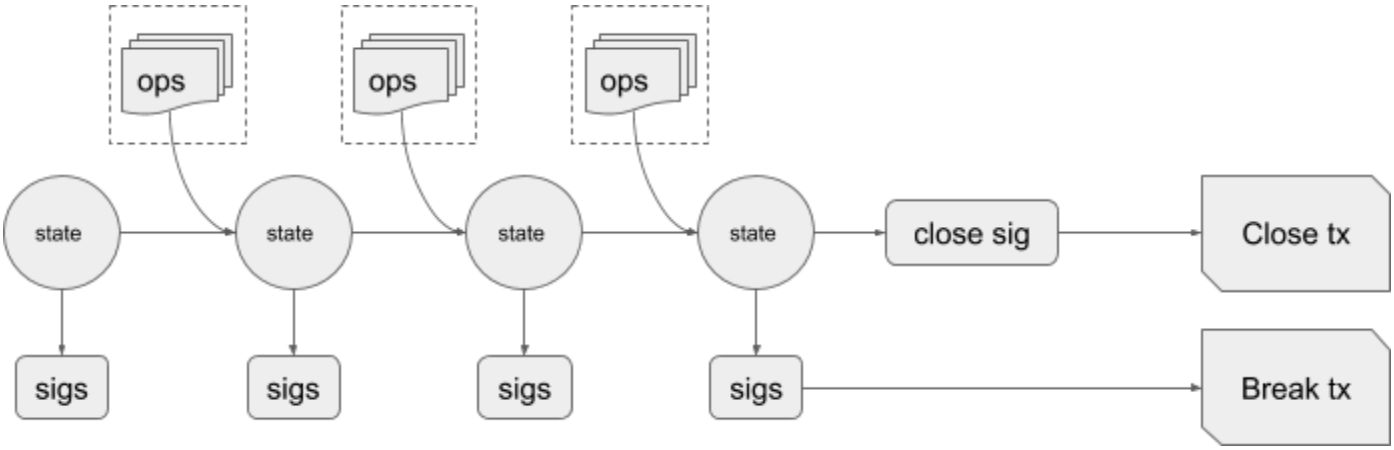
We could just go all the way with these partitions and just have a little tiny Bitcoin and do *normal Bitcoin transactions* that have standard Bitcoin outputs in the channel, a lot more like a sidechain. Each state would commit to a set of outpoints to spend to on-chain. This ends up simplifying state validation since we can reuse a bunch of Bitcoin code for manipulating txs and chain data. But this would make justice transactions more complicated (how do we know who to punish and how?) and it would also make it a lot harder to build a Fulgurite-like system on an

account-based currency like Ethereum. And while a lot of people don't really care about that last point, *I* do, since I think it'd be really cool to be able to do seamless swaps between Bitcoin and Ethereum with a broadly-multicoin Lightning Network.

Though it would be *very* fun to make a "sub-Bitcoin" style channel system that supports arbitrary kinds of transactions and then design a way to be able to forward HTLC multihop payments across it with BOLT. It'd probably need Eltoo to work well though.

2.2 Rendering Transactions

Since we can think of the inputs and outputs as a pure function of the state of the channel and general context about the participants, our information flow looks something like this:



(This leaves out things like justice txs, etc.)

Your tx builder would need to be aware of the different kinds of behavior that channel partitions can specify, and would need to know how to deal with different combinations of that without caring about what the users are trying to do with it.

Fees Discussion

There's been a lot of discussion both on the mailing list and on other venues about *what to do about fees*. For the uninformed, there's some problems with the game theory of channel fees, where different things your counterparty can do to make you stuck with having to pay break tx fees, but the details of that discussion aren't the focus of this work. All that actually matters to us is that both parties agree to use the same parameters to the tx builder.

One proposal was to include an OP_TRUE script that can be used to child-pays-for-parent the channel close tx (which is a little easier with Eltoo). In the Fulgurite tx construction scheme we could say "create an OP_TRUE with unallocated fees, skimming off the top of user balances in such-and-such way for whatever the unallocated funds don't cover".

Maybe instead we can have a special “Fees” partition type and set up some operation validation rules regarding those. Perhaps penalize people who perform more operations by having each signer of an operation implicitly add some funds to the Fees partition, giving back some of the balance to the people who *aren’t* signers of an operation.

The point is, it doesn’t matter really what we do. Dealing with fees is beyond the scope of this, but there’s plenty of space in the design for an arbitrarily complicated fee scheme if we decide we want/need it.

3. Complex Channel Manipulation

So we represent channels as hash-linked chains of state operations that manipulate a set of partitions. Except it really ends up looking more like just a graph when you start doing some of the more advanced things with it since nodes reference nodes in other chains by their hashes. So instead of thinking of these state transitions as block, I used the term “Node” in the code since we *are* building a large graph with different types of nodes and different types of edges between them.

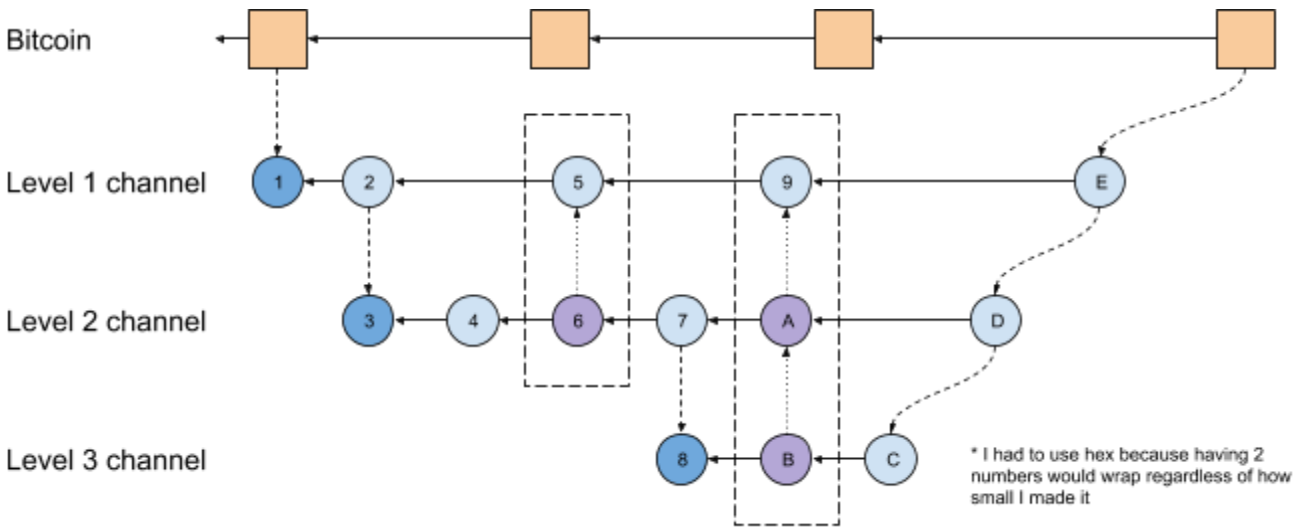
If we can have a general enough mechanism for advancing what we consider the current state, then all of the common use cases we need are actually just particular ways of manipulating the channel state graph. So in our implementation we should want to expose a rich way of describing low-level state manipulation so performing high-level operations is just working within that interface. I’m a little vague in this section because there’s often multiple designs for some of these with tradeoffs at the transaction level that should be bikeshedded elsewhere.

3.1 Subchannels

I’m talking about subchannels before I talk about splicing and stuff because we need to talk about how we manage the channel state graph before we talk about that.

So above we talked about how Lightning channels are kinda like blockchains. Let’s draw everything out so we can understand what goes on under the hood when higher level stuff occurs.

(See diagram next page.)



Oh wow that's a lot isn't it. First let's say what all the circles and colors mean:

- Orange: Bitcoin blocks
- Dark blue: Funding tx node
- Light blue: Simple state transition node
- Lavender: Reanchor node, with indirect link to reanchor target
- Solid line: Direct reference via a hash of some sort
- Dashed line: Indirect reference, like a txid or partition layout information
- Dotted line: Indirect reference to state, with state node hash
- Dashed box: Linked states with coordinated/interleaved update procedure

Now let's describe what happens in order:

1. A channel funding tx is included in a block, for State 1.
2. We open a subchannel (L2, for State 3) in the first transition (State 2) of the L1 channel.
    a. This would allocate a "subchannel" partition in the L1 channel with the initial layout of the subchannel.
3. We make some contained state transition (State 4) in the L2 channel. It doesn't matter for the purposes of this discussion, pretend that Alice pushed 100 sats to Bob or something.
4. Some state transition (State 5) happens in the L1 channel. Oh no! That means the commitment break tx has a different txid, and messing up the input for the L2 channel. Let's go through the steps of doing that state update:
    a. Sign and broadcast sig for L1 state. (State 5)
    b. Wait for all sigs for L1 state.
    c. Sign and broadcast sigs for L2 state. (State 6)
    d. Wait for all sigs for L2 state.
    e. Internally accept new L1 and L2 states as committed.
    f. Broadcast revs for prev L1 state. (State 4)

    g. Wait for revs for prev L1 state.
    h. Internally mark prev L1 state as revoked and mark new states as finalized.
    i. Broadcast revs for prev L2 state. (Note: this isn't strictly necessary since its funding is already revoked and this can't be valid anymore)
    j. Optional: Wait for revs for prev L2 state.
    k. Internally mark prev L2 state as revoked.
5. Create another subchannel (L3, for State 8) in the next state transition (State 7) in L2.
6. Some new state transition happens in the L1 channel. This is even more work! I'm not going to write this out, but in this case we have a procedure a lot like the previous one, except instead of just L1 waiting on L2 to finish signing, we also have L2 wait on L3 to finish finalizing before letting L2 continue to broadcasting revocations. It's like a stack.
7. Have some state transition (State C) in the L3 channel (again, like a Push).
8. Close the L3 channel into the L2 channel (in State D), providing final channel layout.
9. Close the L2 channel into the L1 channel (in State E), providing final channel layout.
10. Cooperatively close the L1 channel on-chain (using State E).

The important thing here is how information flows. Parents don't really care what their children do until they're done. But children do care about their parents, so if a parent updates the child needs to too. But with SIGHASH_NOINPUT this is a bit simpler! Children don't have to care about their parents, and we can use the child commitment tx on top of any output that the parent produces, so we just have to keep around the subchannel partition in the parent and leave it. There's some downsides in that txs that should be invalidated stay valid when they shouldn't be, which can cause complications when adjusting subchannels into parents.

It might be worth investigating having channels hierarchies with heterogeneous commitment schemes, such as LN-penalty in the parent but Eltoo in children or vice-versa. This is especially important since if we need to invoke justice transactions for a parent channel then we end up having a different funding txid than what we would have for an honest break, so we need extra signatures to do subchannel breaks in those cases. Really it just ends up being a mess and we should just use Eltoo where it makes sense to.

So in more detail, there's 3 types of channel state nodes we need to start off with:

- Root
    - Has initial channel layout, agreed upon before creation
- Deriv (as in "derived from previous state")
    - Simple state transitions, manipulating the channel layout
    - Has state hash of parent node
- Reanchor
    - Used when a parent update and the child needs to update
    - Has state hashes of parent node and reanchor target
    - Also probably will be used with splicing

And of course to create/destroy these subchannels we need new operation types. Note that there we don't need any information about "what goes on in the subchannel" in these operations, just information about how to construct the initial state and how to reflect the final state.

- FundSubchannel
    - Provide list of where funds are coming from, automatically deducted from parent participant funds
    - Signed by all subchannel participants
    - (In my implementation it has a merkle commitment to the list of participants, although I don't think it's strictly necessary.)
- CloseSubchannel
    - Provide final layout of partitions, then merged with parent channel partition set, user fund partitions collect together

Another advantage of having states be derived from discrete nodes like this is it allows you to refer to a state based on the hash of the node that produced it (or "state hash" as I've been calling it here), which can simplify gossip code when passing signatures around between participants and you end up needing less context when processing incoming signatures.

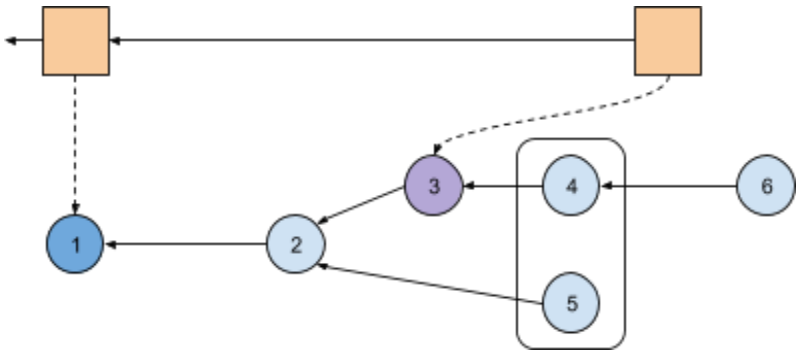
With this we also sort of get two parts of the overall channel state. The channel inputs, and then the partition layout. The latter of which I've been referring to as "*the* channel state", but it's actually important to consider all of it since it all goes into generating txs, it's just that only the latter kind is expected to change regularly in Deriv nodes.

3.2 Splicing

Once you've gotten past all the noise with reanchor nodes, it's pretty simple to understand what needs to happen to make splicing work. There's a few different schemes to make it work, but this is a rough procedure so we don't care *too* much about the details and we can also bikeshed those somewhere else.

Splicing in and splicing out end up looking similar to each other, and you could theoretically do a splice in and out simultaneously, if you find a way to negotiate it. Since partition balances are different, a valid operation on one branch might not be valid on another, so when we're doing the parallel transitions like below we can only attempt a state transition if it's valid on *both* tips, until the branch is finalized. So the state graph ends up looking kinda like this:

(See diagram next page.)



And the rough procedure:

1. Create a splice plan: decide which UTXOs should be spent into this channel and/or which addresses should be spent to.
2. Tell counterparty splice details, create new tx with proper inputs/outputs to new channel tx, exchanging signatures.
3. Create new channel state (State 3) with reanchor node, reanchoring onto the new output, don't invalidate previous state yet, publish tx.
4. Until new tx confirms, apply new "linked" states in parallel to both reanchor state and previous state, revoking states as necessary. (States 4 and 5)
5. When reanchor state confirms, forget about (or revoke for good measure) the now-invalid branch without the reanchor state, adding new states (State 6) on valid tip.
6. Globally gossip new channel txid.

So the reanchor state would need to be a little smarter to be able to differentiate between "reanchor to this parent channel state hash" (what it is basically how I've written it right now) and "reanchor to this funding output (or these justice outputs)". Although I still need to iron out some of the details with this since we should only actually need the latter of those two, and then all types of reanchoring are kinda the same.

3.3 Parent Closures

This is a little tricky and is more poorly defined. When you close a parent channel, each of the subchannels become free-standing channels of their own. We don't actually have to do anything with the channel state when channels close unilaterally in an honest way. In non-honest closes we might want to mark the channel as "temporarily unstable" or something, but *theoretically* we could keep using it as long as we made sure the correct state ended up on-chain and our signatures were based on what we expected. For cooperative closures child states need to resign from the new funding tx before publishing signatures for the parent close tx, of course.

3.4 Multiparty

I've been talking about "participants" as if there's more than 2 of them for much of this so far. That's because Fulgurite makes things a little more straightforward to do that. Multiparty channels do make updating state a bit more complicated though, beyond just the fact you need more signatures from more people...

In LN-penalty, the amount of justice transactions you have to make and sign ends up being something like O(n^2) where n is the number of participants, so there ends up being O(n^3) or more signatures and revocations flying around. It also increases the number of steps to create all these justice txs since you need to know the txid for higher level justice txs before you get to lower-level ones. (Unless you want to assume there isn't going to be more than m evil parties in an n party channel, where m < n.) So that's not only a lot of data, it's also a lot of waiting for other peers. But it's a lot simpler with Eltoo, so let's assume we're doing that. That isn't to say doing multiparty in LN-penalty is impossible, but it makes things more confusing and breaks the incentive model a little bit since you might not necessarily always lose *all* your money if you publish old state. So because they don't scale too well (even with Eltoo) I don't expect Fulgurite multiparty channels to be done with more than a dozen or so peers, unless you're doing channel factories where you only rarely do updates in the parent. This is an area for future discussion.

As far as channel operations, having multiparty channels doesn't change a whole lot. The way I've implemented it so far includes "destination participants" in operations and participant fund partitions know the address of their owner. Where most of the complications lie is in the state update mechanics. You need to make sure that you know to get sigs/revs from all of the peers. If there's subchannels you need to update then participants that aren't in one of those subchannels don't actually care about them, and publish revs as soon as they get sigs, so participants *with* subchannels should know how to handle getting revs before they expect. This isn't actually that hard and the state machine I designed for handling the update procedure already handles it fairly well, although it would need a bit of refactoring for trying to support multiparty channels on LN-penalty due to the recursive blowup of the number of revocation signatures you need to exchange. Instead of being a finite automaton it ends up having to be a generalized pushdown automaton, I think.

3.5 Extracting Subchannels (Is this useful?)

Let's say you have an L1 channel with a L2 subchannel and the L2 members want to upgrade their channel to be an L1 channel, but the rest of the existing L1 participants don't want to close the channel. This functions a lot like a splice, except it's a 1-in-2-out situation where each of the outputs are to channel addresses. And you'd need Reanchor nodes on *both* channel chains, except in the parent it'd be a "special" one that also removes the partition for the subchannel. If we get around to this we may have to introduce a new node type, not actually use Reanchor.

4. References and Links

Earlier presentation (linked earlier): See first page
In-progress implementation: https://gitlab.com/delbonis/roz

Lightning: https://lightning.network/lightning-network-paper.pdf
Plasma: https://plasma.io/plasma.pdf
Eltoo: https://blockstream.com/eltoo.pdf
Channel Factories:
https://www.tik.ee.ethz.ch/file/a20a865ce40d40c8f942cf206a7cba96/Scalable_Funding_Of_Blockchain_Micropayment_Networks%20(1).pdf

Talk Joseph Poon gave on Plasma ideas on Bitcoin: https://youtu.be/QkYXPJMqBNk?t=8166
(Not directly related, those ideas would require forks and a lot more research and development than I've done here.)

A splicing proposal by Rusty Russel (there's others but this is the one I based this off of):
https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-October/001434.html
(See replies, too.)

There's a lot of other fun stuff on the lightning-dev mailing list, I might have missed some other things there that I should have included here. Let me know if there's other things here that I should link.