BitScaler: Scaling Bitcoin for DeFi & Beyond

Casey Bowman
casey@portaltobitcoin.com
PortalToBitcoin.com

Sean Gilligan
sean@portaltobitcoin.com
PortalToBitcoin.com

Jose Storopoli
jose@storopoli.io
Universidade Nove de Julho

Manoj Duggirala
manoj@portaltobitcoin.com
PortalToBitcoin.com

August 12, 2024

Abstract

This paper introduces BitScaler, a novel framework aimed at enhancing Bitcoin's scalability for cross-chain applications, particularly within the realm of decentralized finance (DeFi). By integrating customized channel factories, Taproot, and a policy language, BitScaler facilitates the development of transparent, composable, secure, and optimized transaction scripts. The necessity for BitScaler arises from the challenge of implementing automated market makers (AMM) on Bitcoin in conjunction with the Lightning Network (LN). This design significantly reduces on-chain footprint and decreases LN fees by 90%. Furthermore, the simplifications introduced in the hub-and-spoke architecture contribute to the homogenization of liquidity that is typically siloed in payment channels, enabling more effective liquidity monitoring and programmability for DeFi applications. The innovative approach to non-custodial delegation mitigates the liveness requirement in the Lightning Network, thereby providing greater contract flexibility and user experience. Although these innovations were developed with a AMM use case in mind, they establish a generic framework that can be leveraged to create a wide range of applications. This paper also details the Rust-based implementation of channel factories, non-custodial delegation mechanisms, and policy templates, thereby expanding Bitcoin's potential to support a diverse array of DeFi applications.

Keywords Bitcoin · DeFi · Lightning · Channel Factory

1 Introduction

BitScaler[1] represents a significant advancement in Bitcoin scaling protocols, integrating Bitcoin primitives and payment channels to establish an off-chain, permissionless framework for expanding Bitcoin's applications. By implementing a modified version of the channel factory —a concept heralded as a key advancement for multi-layer Bitcoin scaling first proposed by (Burchert, Decker, and Wattenhofer 2018). BitScaler introduces the ability to create multiple

[1]See our implementation at https://github.com/Portaldefi/bitscaler

peer-to-peer channels from a single on-chain transaction, thus elevating scalability of Bitcoin to layer-3 or higher.

The BitScaler protocol extends this foundational concept with novel enhancements, including non-custodial delegation for facilitating Automated Market Makers (AMMs), (Hayden Adams 2020) on Bitcoin. By leveraging Taproot (BIP-341 2020) and policy language, BitScaler starts with a straightforward implementation, allowing for incremental improvements. Future iterations will incorporate advanced features such as MuSig2 (Nick, Ruffing, and Seurin 2020), FROST (Komlo and Goldberg 2020), DLCs (Dryja, n.d.), and Eltoo (Decker 2019), creating a comprehensive toolkit for sophisticated and permissionless off-chain contracts.

This paper revisits the core principles of the channel factory and details the innovative enhancements introduced by BitScaler. Leveraging current tools and frameworks, BitScaler aims to address Bitcoin's scalability challenges, providing a robust foundation for DeFi applications on Bitcoin. Those already familiar with channel factories may proceed directly to the subsequent sections detailing our enhancements and implementations and skip section 2.

2 Channel Factory Basics

2.1 Overview of the Lightning Network

The Lightning Network (LN) (Poon and Dryja 2016) is a layer-2 (L2) solution designed to economize on the cost and time involved in exchanging Bitcoin. It allows two peers to exchange transactions off-chain, settling their balances on-chain only occasionally. This minimizes the fees and confirmation times associated with on-chain transactions, which are part of the Bitcoin blockchain, defined as layer-1 (L1).

2.2 Establishing an LN Channel

To initiate an LN channel, an on-chain transaction is required, anchoring the channel with funds contributed by the two peers, locked to a 2-of-2 multisig address. Off-chain transactions are then exchanged privately between peers, by revoking old states (balances) and committing to new states, without broadcasting to the Bitcoin blockchain. The protocol's game-theoretic incentives ensure the fair movement of funds between peers, with penalties for non-cooperation.

2.3 Closing the LN Channel

When both peers agree, they can close the channel by signing a single on-chain transaction reflecting the final state of the channel. This transaction summarizes and settles all off-chain activity, releasing funds to each peer according to the agreed final state.

2.4 The Need for Channel Factories

Despite the efficiency of LN channels, opening and closing channels still incur on-chain costs & confirmation cycles. For example, the opening and the closing transactions of any one channel are still on chain. There is still that cost in time and fees. There are a number of situations where we would like to open and close channels nimbly. For example, when desired liquidity to one side or another is depleted, we may wish to rebalance some of our channels. We could combine in one transaction the closing of one channel with the opening of another, with splicing. But there are more efficient ways to do this, one of which is the model Christian Decker et al,

introduced in 2015, the concept of channel factories. These allow for the creation and closure of multiple channels from a single on-chain transaction, elevating the scalability to layer-3 (L3) or higher.

We do this with a channel factory. If there are n participants who would like to open and close channels amongst themselves, they may cooperate to create a set of alternative n-of-n multisig addresses and contribute Bitcoin funds to the factory in an initial on-chain transaction involving these addresses. This is called a hook transaction and lies in L1. From then on, they can create and close channels amongst themselves, keeping each channel's transactions entirely off-chain, including the transactions that fund these channels. The latter are called allocation transactions and lie in L2. They are each anchored in the hook transaction via one of the multisig addresses. Figure 1 below shows channel factory with subchannels.

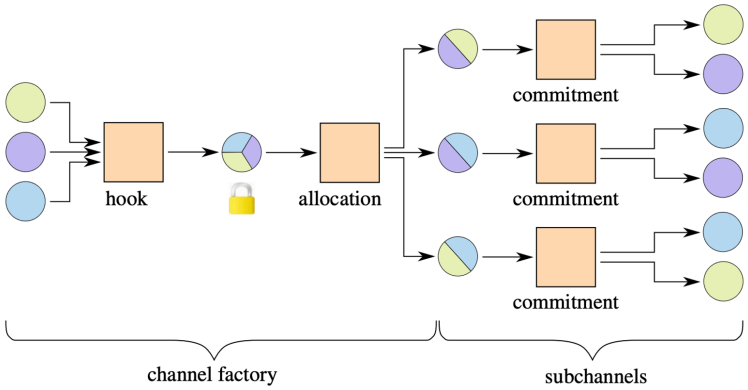

Figure 1: Channel Factory with Subchannels (Burchert, Decker, and Wattenhofer 2018)

Each allocation transaction funds a number of channels, each channel through one of its UTXOs. The transactions in each such channel are one layer up from the allocation transaction that funds it. These channels may take different forms. They may all be two-party channels, or they may include a number of multi-party channels. The channels may each rely on different constraints and penalties to keep their off-chain L3 transactions on the straight and narrow. One channel might use the LN protocol. Another might use a customized form of the LN protocol. Yet another might use Eltoo once it becomes practicable. Another might use the poor man's version of Eltoo in the meantime. Both two-party and multi-party channels (MPC) may use timelocks and invalidation trees to enable proper channel state transitions, as described in the original channel factory paper and in the paper introducing duplex micropayment channels (DMC) (Burchert, Decker, and Wattenhofer 2018; Decker and Wattenhofer 2015). Let's refer to the MPC version of the duplex channel as a multiplex channel. Invalidation tree can be seen below in Figure 2.

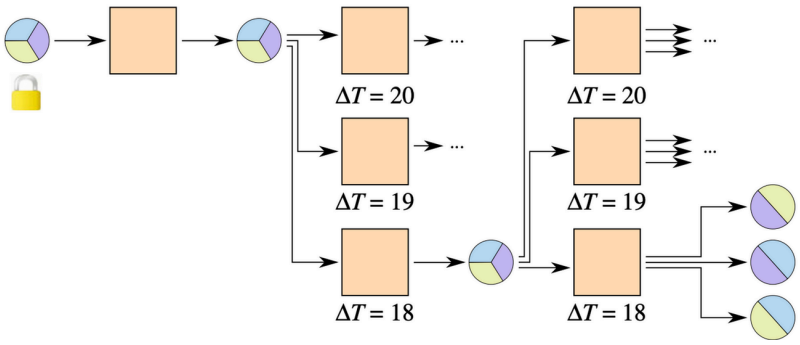

Figure 2: Invalidation Tree (Burchert, Decker, and Wattenhofer 2018)

In all cases, any off-chain L3 transactions must at all times be in a proper state to handle a non-cooperative participant (NCP), as with the L2 transactions in a regular LN channel. If the NCP is in the L3 channel, there must be transactions there that can go on-chain to settle the channel, dropping down to L1. If the NCP is outside that channel, the channel can alternatively drop down to L2, once the allocation transaction that anchors that channel goes on-chain.

The trick is that any current allocation transaction in L2 may be replaced with another by treating that layer as a multi-party, multiplex channel itself. Each allocation transaction will have a timelock associated with it. Its channels are subchannels of this root MPC. MPCs can even parent other MPCs in a channel factory through suballocation transactions, leading to higher layers, treated recursively.

If the participants cooperate, they may agree to rebalance the funds in their subchannels by creating a new allocation transaction with an earlier timelock, both relative to the hook transaction, using a sufficient time gap to allow the new transaction to be safely broadcast before the old one. Invalidation trees allow for a larger number of allocation transactions to be used in the MPC than would otherwise be possible if one were to merely decrement the timelock times. Strands of connected transactions in an invalidation tree are only valid within that subtree with the earliest timelocks.

If the participants continue to participate through the lifetime of the channel factory, they may agree to close the factory with a final closing on-chain transaction or a splice. In the case of a simple opening and closing of a channel factory, there would then be but two on-chain transactions covering all the activity that occurred through its lifetime. In the case of a splice, some participants might leave and others enter to begin another channel factory, while economizing by using only one transaction to both close the old factory and open a new one.

3 BitScaler

BitScaler extends channel factory design and implements several enhancements like non-custodial delegation and simplifications to make it more suitable for applications that are beyond economization of payment channels. A brief overview of various components are described in subsequent sections.

3.1 Taproot

The hook transaction in BitScaler uses a taproot (BIP-341 2020) script for its UTXO. Each branch of the taproot script consists of an n-of-n multisig with distinct keys and timelock. The

internal key is a NUMS. A new allocation transaction or node in the invalidation tree needs to have an input that unlocks the taproot script with the next (earlier) timelock. Figure 3 shows an example for taproot output for the hook transaction.

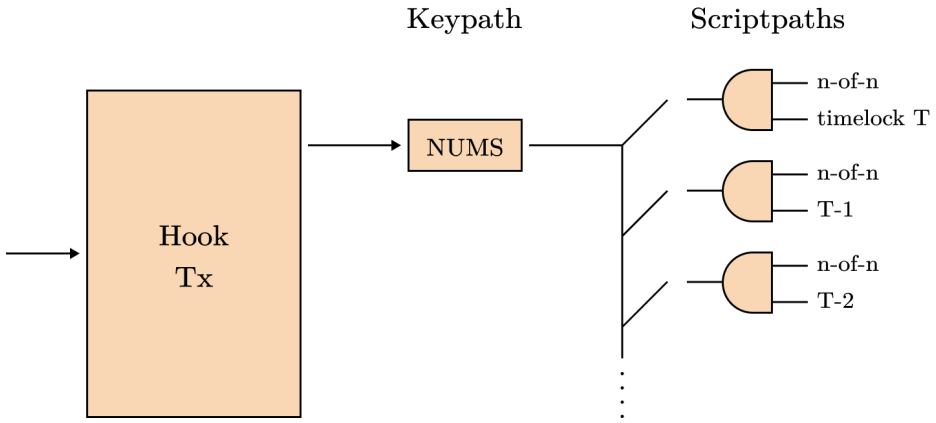

Figure 3: The taproot output of the hook transaction

3.2 Hub and Spoke

To serve the purpose of building an AMM, BitScaler uses only two-party sub-channels, each of which we use for atomic swaps.

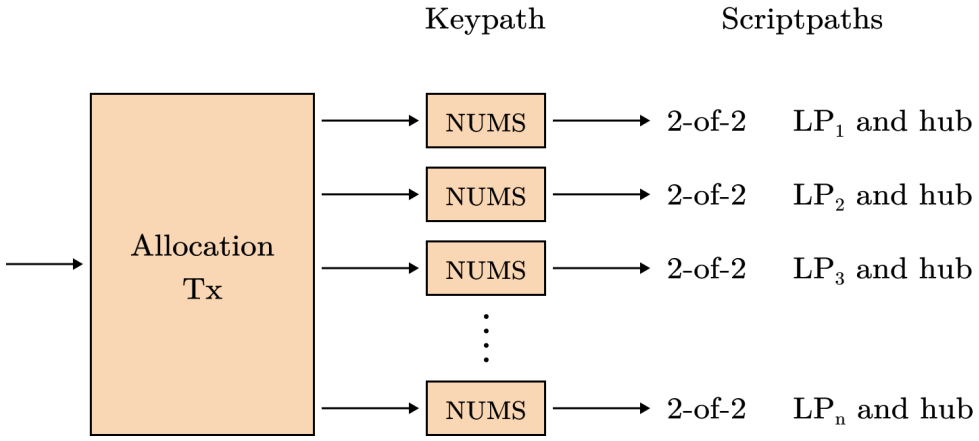

Figure 4: The taproot output of the allocation transaction

A set of liquidity providers (LPs) for the AMM are participants in a BitScaler channel factory. There is one additional participant controlled by a federation of validators. We arrange the two-party channels in a hub-and-spoke pattern (see Figure 5), where the hub corresponds to the validator federation and the spokes to the n LPs, making for a total of (n+1) participants. This design helps federation monitor all the liquidity locked in the channels, as well as help compose swaps with shared liquidity and homogenize liquidity across channels. Note that our hook transaction then includes (n+1)-of-(n+1) multisigs in its taproot scripts instead of n-of-n multisigs. Figure 4 shows taproot output of allocation transactions.

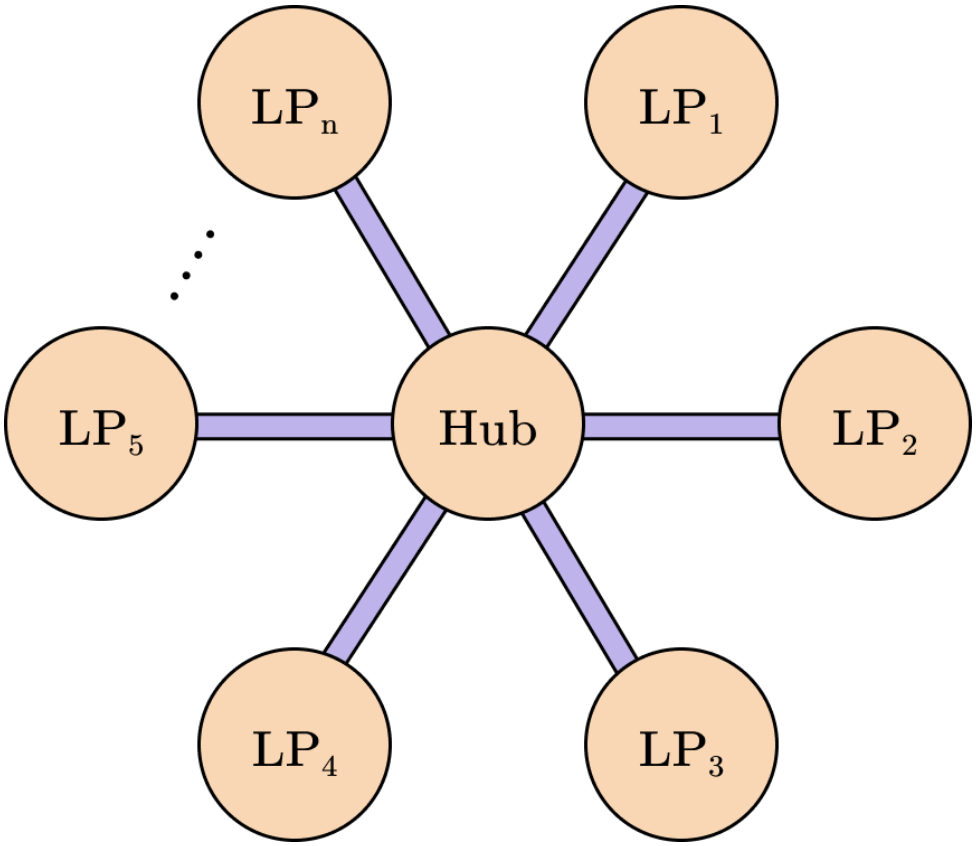

Figure 5: The hub and spoke pattern of the channels

Each subchannel is a duplex channel, with its own invalidation tree, like the MPC's invalidation tree, only consisting of staging transactions (see Figure 6) instead of allocation transactions.

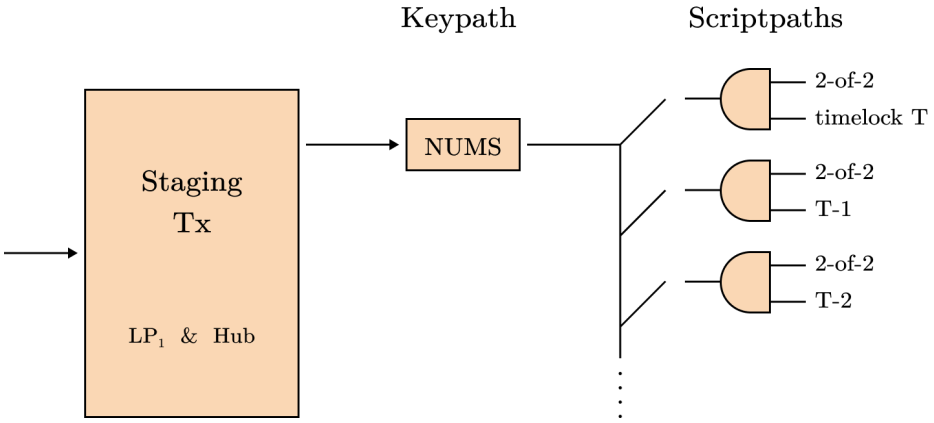

Figure 6: The taproot output of the staging transaction

3.3 Atomic Swaps

The commitment transactions in each subchannel may gather HTLC outputs as one part of an HTLC-based atomic swap between the hub and the spoke for that channel. This is how traders are able to trade off-chain against the AMM. An example of this is shown in Figure 7 below.

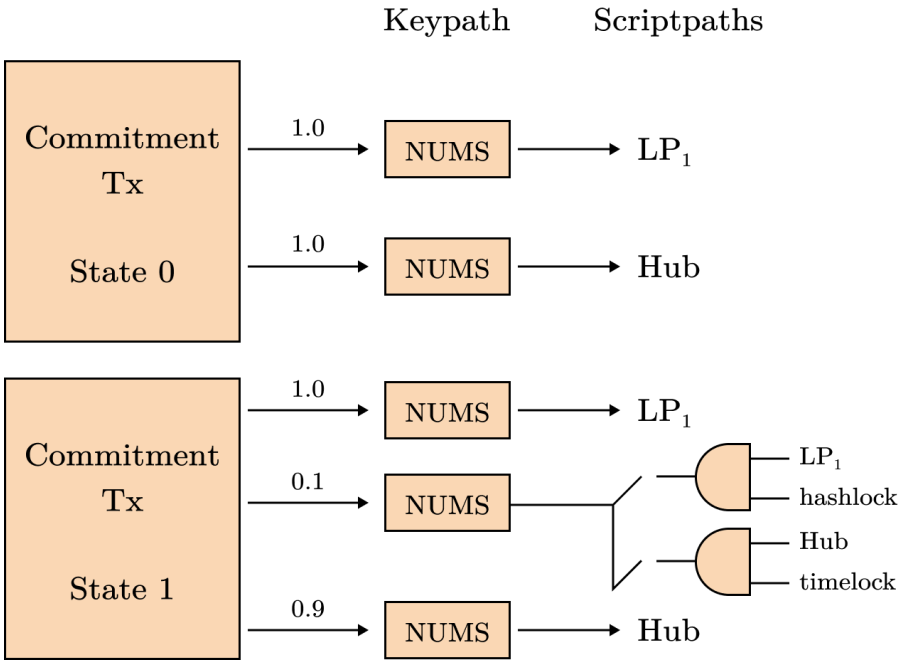

Figure 7: The taproot outputs of the commitment transactions including an HTLC output

3.4 Delegation

The invalidation tree for a subchannel may have subtrees which use the keys of a delegate for the LP. For proper resolution in the case of a non-cooperative participant, a tapscript branch is added to allow the delegator to claim funds earlier than the delegate. Figure 8 shows this taproot output of a delegation transaction. This design was chosen to enable greater flexibility around user experience, by participant LPs

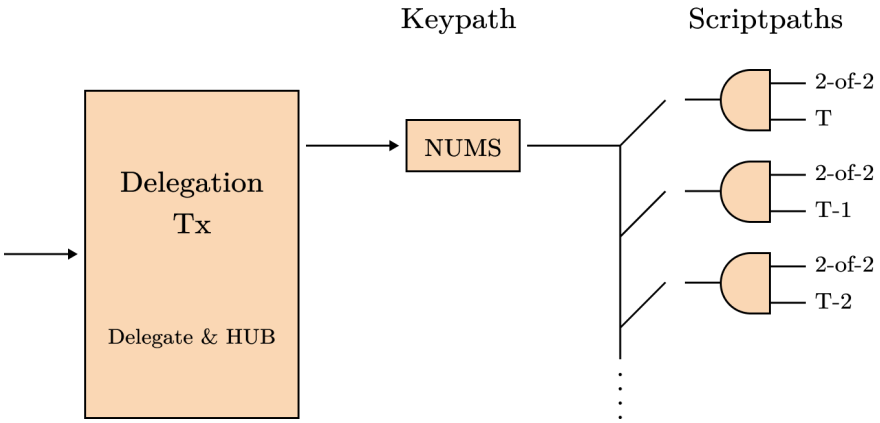

Figure 8: The taproot output of a delegation transaction, which has a staging transaction output as its input.

3.5 Policy language

Policy language allows the channel factory to easily compose sub-policies into larger policies used for the construction of taproot output descriptors ("Bitcoin Descriptors," n.d.) for the input and output scripts of its various transactions. Policies are compiled into miniscript strings (Wuille 2019), which then translate into Bitcoin scripts directly, one-to-one. Thanks to this compilation,

the scripts are not only optimized and secure, but also transparent to the participants. See policy language to miniscript below in Figure 9.

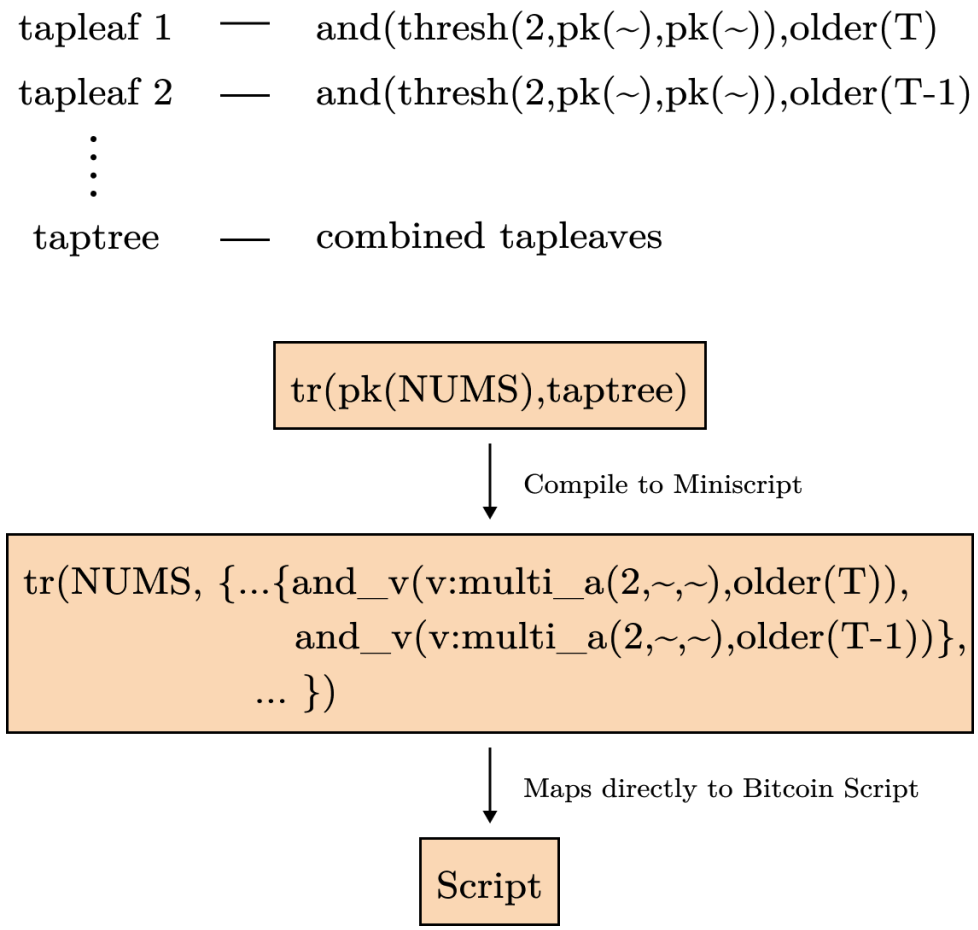

Figure 9: Policy language to miniscript via compilation

3.6 Channel Factory Wallet

BitScaler employs a variety of transaction types, each characterized by outputs defined by Taproot output descriptors. Multiple output descriptors are utilized within this framework. We use Bitcoin Development Kit (BDK) version 1.0.1 ("Bitcoin Development Kit," n.d.) for the wallet implementation, given that it supports wallets defined by only two output descriptors—one for external addresses and another for internal addresses—BitScaler leverages a collection of BDK wallets to manage keys for addresses and signing purposes. These wallets, referred to as sub-wallets, facilitate the necessary key management operations.

3.7 Policy templates

For sub-polices which require public keys, we use policy templates.

A policy template defines how a stream of policies may be generated by each participant and the channel factory itself, using a list of extended public keys from the participants.

Any output descriptor that is fed into a sub-wallet at its creation is in essence a miniscript expression compiled from a policy, which is fed generally into a miniscript function specific to an output type.

In our initial design we use taproot outputs. So the descriptors will use the miniscript function tr().

The policy is an easy abstraction to work with. The miniscript that a policy compiles to is a bit more tricky with non-obvious specifics added that are subtle; though overall it resembles the policy it's compiled from.

One distinction is essential: Policies are composable while Descriptors are not.

The protocol for gathering extended public keys will be based on a policy. In essence, each participant will be asked for one or more extended public keys, passing this list around for each participant to add to. The request will include each participant's user-id and user-label and will specify the derivation path above the extended public key and the policy template ids for which these keys are to be used.

For example, for any leaf in the invalidation tree for the hook transaction, we would have a taproot output with a number of taproot scripts, each with a different timelock sub-policy. For example, lets assume that there are five timelocks. Each timelock sub-policy is combined with a multisig sub-policy to form a taproot script policy. Each of these needs to use a distinct key.

Here's an example of a policy template that might be used in this case for the first timelock's associated multisig sub-policy:

```
thresh(4, <0,0>/0/*, <0,1>/0/*, <0,2>/0/*, <0,3>/0/*)
```

The expression with angle brackets `<i,j>` represents the jth extended public key of the ith participant.

For each new leaf of the invalidation tree, we can generate a new address per the asterisk. For five timelocks, we would need five such templates and five lists.

```
thresh(4, <0,0>/0/*, <0,1>/0/*, <0,2>/0/*, <0,3>/0/*)
thresh(4, <1,0>/0/*, <1,1>/0/*, <1,2>/0/*, <1,3>/0/*)
thresh(4, <2,0>/0/*, <2,1>/0/*, <2,2>/0/*, <2,3>/0/*)
thresh(4, <3,0>/0/*, <3,1>/0/*, <3,2>/0/*, <3,3>/0/*)
thresh(4, <4,0>/0/*, <4,1>/0/*, <4,2>/0/*, <4,3>/0/*)
```

Alternatively, we could use a single list with all five templates and generate the addresses in a different way, by adding to the depth of the derivation path.

```
thresh(4, <0,0>/0/0/*, <0,1>/0/0/*, <0,2>/0/0/*, <0,3>/0/0/*)
thresh(4, <0,0>/1/0/*, <0,1>/1/0/*, <0,2>/1/0/*, <0,3>/1/0/*)
thresh(4, <0,0>/2/0/*, <0,1>/2/0/*, <0,2>/2/0/*, <0,3>/2/0/*)
thresh(4, <0,0>/3/0/*, <0,1>/3/0/*, <0,2>/3/0/*, <0,3>/3/0/*)
thresh(4, <0,0>/4/0/*, <0,1>/4/0/*, <0,2>/4/0/*, <0,3>/4/0/*)
```

We expect to settle on standards for such derivation paths as we go.

For signing purposes, each participant may substitute in their own extended private key for the corresponding extended public key in a policy template.

3.8 Merkle tree synchronization

We manage the persistence of the off-chain transactions using Partially Signed Bitcoin Transactions (PSBTs) ("Partially Signed Bitcoin Transactions," n.d.) as the data structure. We add metadata so that we can retrieve PSBTs by participant, by channel, or by swap. At any

one time, there are a number of strands of transactions stretching from the hook transaction to a number of commitment and associated swap-related transactions added to HTLC outputs. To simplify our description, we loosen up and treat the commitment transactions as the leaves of the channel factory tree in the large, together with the swap-related transactions.

The strands, or thinned subtrees, with leaf transactions for the most current state of each channel are all that matter. The rest may be archived. A new sub-strand may be added and included only when its transactions are complete and ready for broadcast, all the way to the leaf transactions. Behind the scenes, we add them and formally only treat them as included when the statuses of all the transactions in the added sub-strand are complete. Handled in this way, the addition of off-chain transactions becomes atomic.

We refer to the strands of transactions that have been included and that have not been archived as active strands(Figure 10, 11).

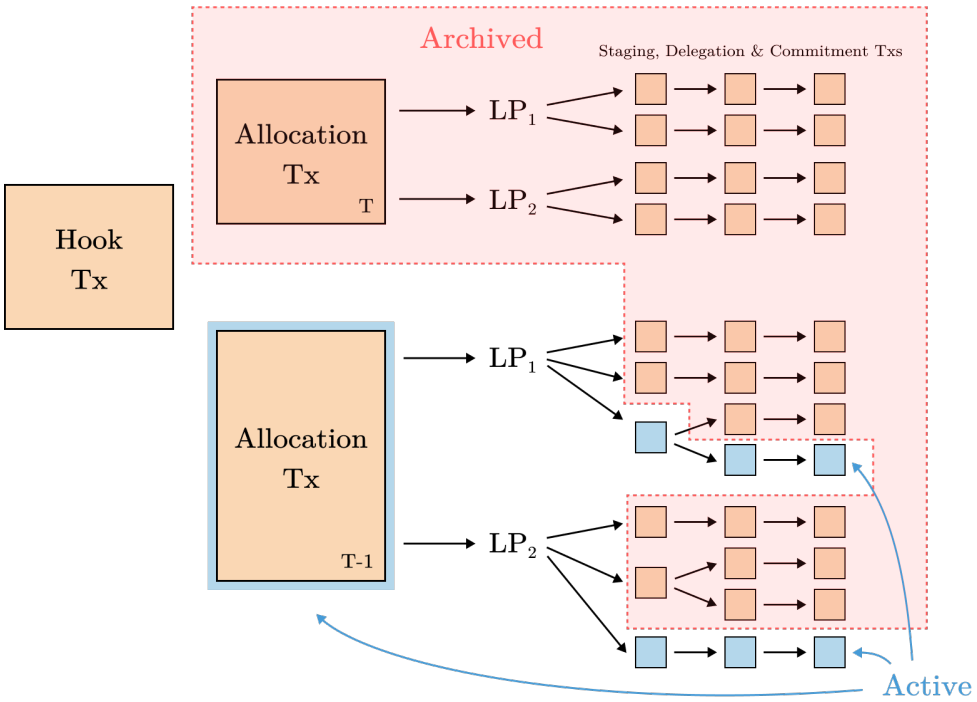

Figure 10: Active and archived strands of off-chain transactions

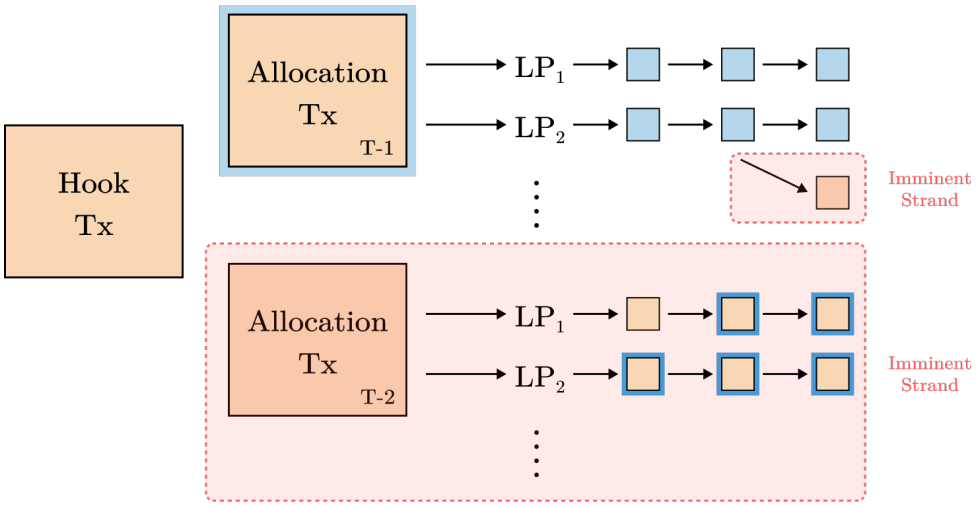

Figure 11: Active and imminent strands of off-chain transactions. Blue transactions are complete and ready for broadcast. Red transactions are incomplete.

We use merkle-trees to make sure that all participants are in sync with respect to the active strands. Moreover, we can hide other sub-strands from those who are not participating in the channels of that sub-strand, including only the bitcoin tallies for the substrand and the substrand's hash.

3.9 Communication

For communication between BitScaler and the AMM and between the participants of the BitScaler channel factories, we use websockets together with STOMP messages as implemented in the tokio-stomp crate ("Tokio, Asynchronous Applications with Rust," n.d.).

3.10 Statechain

One limitation of current BitScaler design is the factory participants are setup initially during hook transaction and any changes to that would require an on-chain transaction. This is required for pooling DeFi use cases that rely on expiry times like Liquidity Provisioning. However to make the framework more extensible and real-time, in the subsequent versions of the implementation, we plan on adding a statechain protocol. Statechains (Somsen, n.d.) enable transferring ownership of UTXOs in the hook transaction to any other participant who wants to enter an existing factory. Implementations like Mercury layer's blinded co-signing ("Mercury Layer's Blinded Co-Signing," n.d.) are proven this to work seamlessly and this can be adapted to BitScaler to offer a marketplace for participants, who could go in and out of a factory by transferring their UTXO ownership.

3.11 Applications

The current implementation of BitScaler is specifically designed to enable Automated Market Makers (AMMs) on Bitcoin without relying on sidechains, multisig custody of user funds, bridges, or permissioned vaults. However, the framework's building blocks are versatile and can be applied to various other applications. For instance, BitScaler can facilitate the creation of a permissionless Layer 2 solution that seamlessly integrates with the Lightning Network. Additionally, BitScaler can be combined with Discreet Log Contracts (DLC) to develop a non-custodial perpetual trading marketplace or to issue a stable balance token, similar to Ethena

("Synthetic Dollar," n.d.), on Bitcoin in a fully non-custodial manner. There are numerous other DeFi applications that can be built on Bitcoin using the BitScaler approach, without the need for new Bitcoin Improvement Proposals (BIPs) or the activation of new OP Codes. We are enthusiastic about the potential innovations the community will develop in the future.

3.12 Future Work

BitScaler is designed to evolve incrementally. We initiate with an implementation tailored to a specific use case involving cross-chain decentralized financial applications, with the intention of building upon this foundation incrementally. Once Eltoo becomes available, we anticipate transitioning our channel implementations to utilize Eltoo.

Additionally, we are considering the implementation of MuSig2 or FROST for multiplex channels. One potential feature that would facilitate this, for our purposes, involves the hypothetical use of a keypath-scriptpath combination in Taproot. Instead of unlocking when either the keypath or the scriptpath is satisfied, the Taproot output would unlock only when both conditions are met, contingent upon the toggling of a flag. In this scenario, the internal key would be the MuSig2 or FROST key, while the script tree would manage the remaining policy requirements, such as timelocks and other necessary conditions. This approach would necessitate a Bitcoin Improvement Proposal (BIP) and the support of the community. Should this not be feasible, we will explore alternative methods, particularly with FROST, including the use of ChillDKG: Distributed Key Generation for FROST (Tim Ruffing, n.d.). Furthermore, FROST's ability to nest could be advantageous in conjunction with the hub participant, which is governed by a federation of validators.

We also intend to investigate the use of Lightning Network (LN) sub-channels, either standard or custom, as required. Duplex channels appear simpler and better suited to our needs compared to LN sub-channels.

Looking ahead, we will explore other methods for conducting atomic swaps, such as adaptor signatures or PTLCs (Nick 2018), which employ adaptor signatures. While the current implementation of BitScaler is focused on specific use cases, we foresee numerous applications beyond this scope.

References

BIP-341. 2020. "Taproot: Segwit Version 1 Spending Rules". 2020. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

Burchert, Conrad, Christian Decker, and Roger Wattenhofer. 2018. "Scalable Funding of Bitcoin Micropayment Channel Networks". *Royal Society Open Science* 5 (July): 180089–90. http://dx.doi.org/10.1098/rsos.180089

Decker, Christian. 2019. "Eltoo & the Far Future". Chaincode Labs. September 18, 2019. https://www.youtube.com/watch?v=3ZjymCOmn_A

Decker, Christian, and Roger Wattenhofer. 2015. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In *Stabilization, Safety, And Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings 17*, 3–18

Dryja, Thaddeus. n.d. “Discreet Log Contracts”. https://adiabat.github.io/dlc.pdf

Hayden Adams, Dan Robinson, Noah Zinsmeister. 2020. “Uniswap V2 Core”. 2020. https://uniswap.org/whitepaper.pdf

Komlo, Chelsea, and Ian Goldberg. 2020. “FROST: Flexible Round-Optimized Schnorr Threshold Signatures”. 2020. https://eprint.iacr.org/2020/852

Nick, Jonas. 2018. “Multi-Hop Locks from Scriptless Scripts”. 2018. https://github.com/BlockstreamResearch/scriptless-scripts/blob/master/md/multi-hop-locks.md

Nick, Jonas, Tim Ruffing, and Yannick Seurin. 2020. “MuSig2: Simple Two-Round Schnorr Multi-Signatures”. 2020. https://doi.org/10.1007/978-3-030-84242-0_8

Poon, Joseph, and Thaddeus Dryja. 2016. “The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments”

Somsen, Ruben. n.d. “Statechains: Non-Custodial Off-Chain Bitcoin Transfer”. https://medium.com/@RubenSomsen/statechains-non-custodial-off-chain-bitcoin-transfer-1ae4845a4a39

Tim Ruffing, Jonas Nick. n.d. “Chilldkg: Distributed Key Generation for Frost”. https://github.com/BlockstreamResearch/bip-frost-dkg

Wuille, Pieter. 2019. “Miniscript”. 2019. https://bitcoin.sipa.be/miniscript

“Bitcoin Descriptors.” n.d.. https://github.com/bitcoin/bitcoin/blob/master/doc/descriptors.md

“Bitcoin Development Kit.” n.d.. https://bitcoindevkit.org/

“Mercury Layer's Blinded Co-Signing.” n.d.. https://github.com/commerceblock/mercurylayer/tree/dev/docs

“Partially Signed Bitcoin Transactions.” n.d.. https://github.com/bitcoin/bips/blob/master/bip-0174.mediawiki

“Synthetic Dollar.” n.d.. https://github.com/ethena-labs

“Tokio, Asynchronous Applications with Rust.” n.d.. https://github.com/tokio-rs/tokio