[DRAFT] Sova: An EVM-Compatible Bitcoin Execution Environment

Kevin Kennis

Abstract

Sova is a new blockchain designed to add programmability to Bitcoin. While Bitcoin secures the most global economic value of any blockchain, applications and utility beyond value storage and exchange via $BTC are limited. This is because the Bitcoin network itself does not host a programmable virtual machine. Sova introduces a new EVM-compatible blockchain with a set of precompile-based VM extensions, combined with validator-level bindings to Bitcoin Core, that enables a smart contract execution environment to interact with Bitcoin and its native assets, such as Runes and Ordinals. Using pooled security from Ethereum mainnet, this network can quickly bootstrap its economic security to the requisite level for enforcing honest, user-initiated operation on Bitcoin and interaction with Bitcoin-native assets.

1 Background

1.1 Bitcoin as a Settlement Layer

One of a blockchain's functions is as global state synchronization machine, allowing multiple entities to subscribe to and coordinate the creation of updates to the state ledger. Different blockchains represent state in different ways; while Ethereum represents value in an account-based manner, using trie storage for the state associated with each particular account, Bitcoin uses a model such that the global state on the Bitcoin chain can be described completely by the set of all Bitcoin UTXOs[1]. A Bitcoin UTXO contains an amount of value (denominated in satoshis, the base unit of $BTC), and rules for who can spend the value.

The spending rules for Bitcoin UTXOs are expressed through Bitcoin Script[1], a non-Turing complete stack-based scripting language. Each UTXO has what is called a "redeem script", which must return true for a transaction spending that UTXO to be successful. Script can express rules that require specific signers for a given transaction input (`OP_CHECKSIG`), or multiple signers (`OP_CHECKMULTISIG`). Through use of various machine instructions (called "opcodes") which do not affect transaction security, authors of Bitcoin transactions can also push arbitrary unstructured data to the Bitcoin blockchain, a practice known as "inscribing" data to the chain[20].

Before inscriptions, data was often posted to the Bitcoin blockchain in (`OP_RETURN`) outputs, however this method was considered nonstandard, space-constrained, and data posted in outputs was subject to pruning[18]. Inscriptions, which emerged after the 2021 Taproot upgrade for Bitcoin[22], have proven a more robust and more accepted way to post data to the Bitcoin blockchain. While inscriptions provide a format for which data-storage opcodes should look like, Taproot allows complex Script rules to be written with sublinear effects to on-chain transaction costs, reducing the fee overhead of using inscriptions. Used together, Taproot and Inscriptions formed the technical foundation for the emergence of native forms of value on Bitcoin besides $BTC itself.

There are a number of factors which make inscribed tokens more scarce than similar abstractions on other blockchains. First, each inscription must be tied to a specific UTXO, which must contain at least one satoshi of value; Bitcoin's intrinsic supply cap also imposes a supply cap on the number of UTXOs that can be inscribed. Second, inscribing data has a transaction cost correlated to the amount of data that needs to be inscribed. These factors of cost, complexity, and asymptotic finite supply all combine to create an ecosystem for on-chain assets with different economic and scarcity fundamentals compared to EVM, SVM, and Move-based asset ecosystems.

These capabilities, and limitations, suggest the potential for a specific type of non-native asset ecosystem on top of Bitcoin. Higher transaction costs for inscriptions enforce an ecosystem of assets that favor quality over quantity; creating an asset on Bitcoin requires an additional level of thoughtfulness and financial commitment compared to more scalable chains.

Where assets of value emerge, so does finance; the self-emergent nature of economies around systems of value is an inherent property of human coordination. As the proliferation of assets of value on the Bitcoin blockchain increases, so does the demand for a native economy.

There is also a strong argument that data inscription is good for the long-term health of the Bitcoin network. Bitcoin has known long-term security budget concerns: as the block subsidy continues to decrease through successive halvings, the profitability of proof-of-work decreases as constant hashpower inputs earn smaller and smaller block rewards.

For Bitcoin's long-term survival, miners must make lost block reward value up through transaction fees; given that $BTC itself has been outcompeted as preferred medium of exchange by tokens on higher-throughput blockchains, it's unlikely that Bitcoin's store of value function alone will provide these needed fees. A vibrant ecosystem of native assets which are stored and transacted on the Bitcoin blockchain is one of the network's best bets at creating demand for blockspace to fund the long-term security budget.

1.2 Programmability on Bitcoin

Bitcoin programmability is limited by design; historically, the Bitcoin community has favored robustness over flexibility. Given Bitcoin's position as the oldest and largest blockchain, it's likely that its functionality will continue to ossify, rather than moving in the direction of supporting native general-purpose execution. While proposals like BitVM[13] suggest the possibility of more native programmability, their path to adoption by the Bitcoin community and implementation timelines remain indeterminate.

In terms of technologies currently supported on Bitcoin, Script has historically provided limited flexibility, mostly allowing abstractions such as time-locked or multisignature signing schemes. However, the set of Script-supported opcodes is limited by design, and there is a large gap between Script capability and general purpose execution. Working within these limitations, the Bitcoin community has developed a number of abstractions, both protocol-supported and protocol-extrinsic (based on convention), which have made it possible to define high-level assets and other data-defined abstractions on Bitcoin.

1.2.1 Partially-Signed Bitcoin Transactions

Partially-signed Bitcoin Transactions were introduced in BIP 174 and updated further in BIP370[4 5]. These transactions allow forms of multi-party transaction coordination. Two main capabilities of PSBTs are:

- Allowing multiple parties to asynchronously sign multisignature inputs.
- Allowing multiple parties to each sign individual inputs to a given transaction which spends multiple inputs from multiple addresses.
- Enabling the transportation of partially-signed transactions without revealing key material to either observers or subsequent signers.

These capabilities can provide the underlying substrate for multi-counterparty financial primitives, such as multisignature wallets, shared escrow accounts, and atomic swaps of currency (by matching inputs and outputs and have each counterparty co-sign).

1.2.2 Inscriptions

Inscriptions are a way of introducing data to the Bitcoin blockchain, by using opcodes with no other defined meaning to define a delineated space in which the transaction writer can post arbitrary data. Inscriptions were introduced in January 2023 by Casey Rodarmor[19], as a new convention, since the defined opcodes do not affect the semantics of a transaction.

Since Bitcoin uses a UTXO-based accounting model, inscribed data must be attached to a transaction output, such that the redeem script for the given output includes the inscribed data. Given that the importance of such an output is not derived from the inscribed data, inscribed outputs often have a value of a single satoshi. Due to this, inscriptions are often called "sats".

There is nothing in the Bitcoin blockchain itself that parses, interprets, or relies on the any inscribed data; rather, inscriptions are "conventional", similar to the use of `OP\_RETURN` in earlier iterations of Bitcoin programmability such as the Omni protocol[7]. It is the responsibility of wallet designers and application developers to detect when a wallet's spendable outputs might be inscribed, and how to interpret the inscribed data. Akin to common data-transfer protocols such as HTTP, inscriptions often define a MIME type[20].

Given that inscriptions support arbitrary data, they also can be used to express NFT-like or token-like asset abstractions, in the same manner that ERC20 tokens are defined by a balance ledger in an account-based model. In this way, the inscribed data itself can have an agreed-upon value and be used for economic activity. Potential applications exist beyond tokens and other units of value; for instance, a social network protocol can use inscriptions to record a user's social graph.

1.2.3 Ordinals

While inscriptions are a method of posting data to the Bitcoin blockchain, if such data is to be used as an asset supporting economic activity, that asset must be transferable and often divisible. However, UTXOs on Bitcoin cannot be "re-used"; by design of the Bitcoin protocol, any Bitcoin transaction always fully spends all value in any UTXO defined as input, effectively destroying the unspent.

Ordinal Theory is a solution to this, and allows the same piece of inscribed data to be tracked across multiple UTXOs and transactions in perpetuity. Ordinal theory relies on the fact that across the entire Bitcoin blockchain, UTXOs are unique and can be numbered: by the block the given satoshi came into existence and its index in that block. As long as its unit of value is kept in a single satoshi and not combined into a larger output, it retains its unique ordinal number.

Like inscriptions themselves, Ordinals are conventional; wallet and application designers must agree to follow the same rules of Ordinal Theory in order to agree on a common ownership ledger over data inscribed to Bitcoin.

1.3 Looking Forward

While there is no demand for general-purpose execution capability from miners and developers, there is demand from asset owners. In Ethereum, a significant portion of total on-chain value is locked in decentralized finance applications. As the total market for Bitcoin-native assets grows larger, so will the demand for on-chain utility and thus programmability.

2 Sova: an EVM for Bitcoin

Sova proposes a novel way to build programmability for Bitcoin assets: an EVM[14]-compatible blockchain with precompiles that bind to the Bitcoin chain. Application developers can write protocols using existing smart contract languages and frameworks (Solidity), and use global BTC bindings to handle asset settlement when required. While assets and related asset state, such as ownership, remain native to the Bitcoin blockchain, all protocol-related state and execution logic is managed by Sova. This gives users and developers the best of both worlds, allowing minimal switching costs for EVM developers to build on Bitcoin, and for users to engage with or enter the Bitcoin ecosystem for novel applications.

Sova's fundamental innovation is a set of new *precompiles* for the Sova EVM that allow on-chain interaction with Bitcoin through EVM smart contracts. The Sova precompiles will allow smart contract authors to:

- Read Bitcoin balances by address and UTXO
- Read owners of specific inscription IDs
- Construct and sign partially-signed Bitcoin transactions (PSBTs)

- Co-sign multi-party interactions with a key tied to the smart contract ("Network Signing")

- Broadcast fully signed Bitcoin transactions

Sova's smart contract environment is meant so that logic can be written on top of a Bitcoin settlement layer; as such, Sova supports Bitcoin precompiles that exist as analogues to EVM settlement actions, such as `ERC20#transfer`, `ERC20#transferFrom`, and `ERC721#transferFrom`.

2.1 Precompile Functionality

Sova provides a read-write interface to Bitcoin, within its smart contract environment, through the use of *precompiles*. Precompiles are built-in, pre-deployed global EVM smart contracts, with which any other smart contract can interact[21].

Use of precompiles to interact with underlying consensus layers or separate blockchains has been previously deployed at scale in Ethermint-based chains such as Evmos[9]. Using precompiles in this manner requires updates to the virtual machine and validator code. For Sova, validators are required to connect to a Bitcoin full node, and use that node when processing transactions that make use of precompiles. For instance, query-based precompiles require the execution environment to read a value from the Bitcoin blockchain and return it inside the EVM, and write-based precompiles produce event logs that block builders must ingest and parse in order to interact with the underlying Bitcoin blockchain. A full list of precompiles is available in Appendix A, and as the demand for new low-level Bitcoin abstraction grows, precompiles can be added to the network via hard fork.

Validators must publish their latest block header to ensure that they are using the latest canonical version of the Bitcoin blockchain. In addition, validators must connect to the Sova Execution Queue, a singleton service for broadcasting effects of precompile contracts onto the Bitcoin chain. It is validators' responsibility to interpret interactions with the precompiled contracts according to the design and rules of the network. Transaction building and submission for execution is further discussed in Section 2.4 below.

2.2 Client Signing

Since any transaction which uses a write-based Bitcoin precompile will result in the construction of a Bitcoin transaction, the Sova network must ensure that any constructed transaction can be fully signed. End-users interacting with Sova smart contracts that may create asset transfers should pre-sign inputs approving transfers, akin to `ERC0#approve`.

Bitcoin-native assets involved in Bitcoin transactions invoked by Sova precompiles may be controlled via different ownership schemes: assets can be owned directly by the user, be owned directly by the network, or be owned by a multisignature redeem script with signing keys controlled by a mix of end-user counterparties and network keys. In the first and last cases, any Bitcoin-level transaction will necessarily require end-user signatures.

Pre-signing inputs and transmission of partially-signed transactions is possible with Bitcoin PSBTs. Smart contract authors should design flows that require end-user signatures such that when initiating such a flow, one of the smart contract's function's parameters is the pre-signed, bytes-encoded PSBT required for asset settlement. The PSBT should be fully pre-signed for any inputs which are either directly controlled by the user or require a user signature on a multisignature redeem script. Smart contracts can decode bytes-encoded PSBTs into `struct` objects that can then be validated against transaction semantics.

In order to generate pre-signed signatures for Sova smart contract function arguments, the Sova ecosystem will provide SDK functionality that can be built into EVM-based wallets. The Sova ecosystem may design an environment-specific wallet for such a purpose, with a synchronized public/private keypair across the Bitcoin and Sova EVM environments. Users will be able to pre-sign any inputs required for underlying Bitcoin transactions using their private key, after which the wallet software can build the EVM-format transaction for the required smart contract interaction. The wallet software can then prompt the end-user to again sign the fully EVM-format transaction using the same private key.

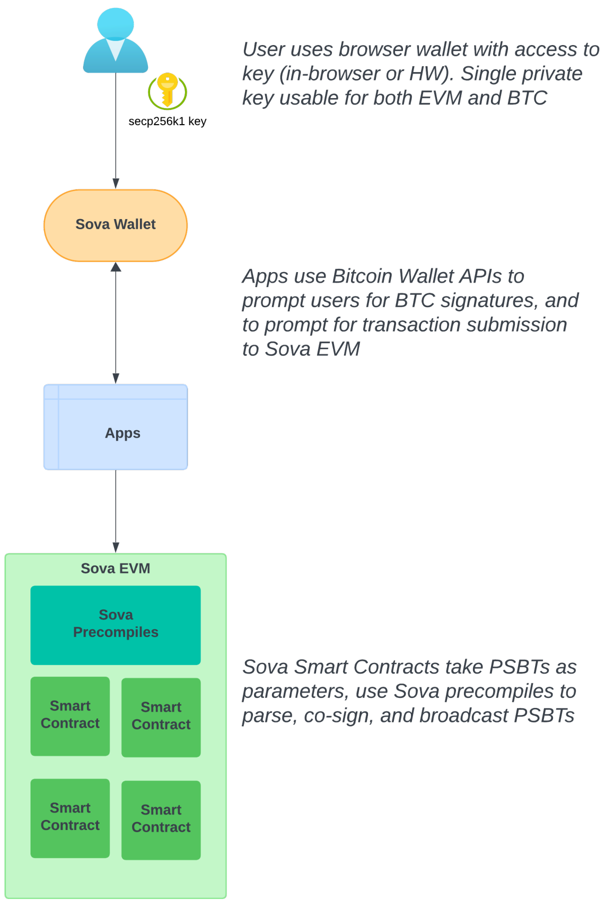
secp256k1 key
User uses browser wallet with access to key (in-browser or HW). Single private key usable for both EVM and BTC
Sova Wallet
Apps use Bitcoin Wallet APIs to prompt users for BTC signatures, and to prompt for transaction submission to Sova EVM
Apps
Sova EVM
Sova Precompiles
Smart Contract
Smart Contract
Smart Contract
Smart Contract
Sova Smart Contracts take PSBTs as parameters, use Sova precompiles to parse, co-sign, and broadcast PSBTs

Figure 1: *Sova Client Signing Flow*

2.2.1 Coin Selection

In situations where clients must pre-generate and pre-sign Bitcoin transaction signatures, client software to facilitate this process must also efficiently generate the needed transactions. One longstanding challenge of Bitcoin transaction generation is the concept of *unspent selection*[3]. Unspent selection is the process of selecting *which* spendable inputs to use from a client wallet in a generated transactions.

Some unspents may be more desirable than others among multiple lines; in general, using fewer inputs to a transaction leads to lower transaction fees, with each additional input increasing the transaction size on average by 60 vBytes[15]. However, long-term fee optimization over multiple transactions also involves change output consolidation; creating many change outputs may lead to the need to use more inputs in future transactions.

Client transaction-building software should be flexible and support multiple unspent selection algorithms, both naive (e.g. largest first) and complex. A more complex unspent selection algorithm that can generally balance long-term fee optimization concerns is branch-and-bound[17].

2.3 Network Signing

Like in Ethereum-based protocols, in many use cases the protocol itself must custody assets and participate in settlement. In an account-based computation model, the smart contract iself owns the asset natively, and the smart contract's code defines how and under what conditions assets can be transferred or approved for transfer.

To make this possible for assets natively held on Bitcoin, each deployed smart contract must itself control a Bitcoin private key, and should be able to leverage that key via Sova precompiles.

The feature of each smart contract controlling a network key it can sign with is called *network signing*.

With these extensions to the base EVM, Sova smart contracts can control asset ownership outright and govern transfer of those assets according to smart contract code, analogous to smart contract asset ownership capabilities on the EVM.

Keys used for network signing are defined by a base *network key*, held be a BIP32 validator wallet. When a smart contract is deployed to a given hexadecimal address on the Sova EVM, the corresponding Bitcoin address derived from the same corresponding public/private keypair will be a network key. This can be enabled through a modification to the EVM's smart contract deployment process to always deploy to a deterministic address.

When deploying an EVM smart contract, the smart contract's address is usually determined by a combination of the deployer's address and account nonce. In the Sova EVM, the smart contract's address is determined by a *network nonce*. This nonce is globally incremented on each smart contract deployment. In addition to defining the address that a smart contract is deployed to, the network nonce also defines the final component of an HD derivation path from the root network extended public key. The key controlled by the smart contract then corresponds to this key generated by its globally-unique derivation path. Since nonces are global and smart contracts are public, there is a clear bidirectional mapping between an EVM smart contract address and its corresponding Bitcoin public key. By using public derivation paths, a smart contract's address can be computed without access to the network private key itself, such that block builders can assign newly-deployed smart contracts to the correct addresses without accessing private key material. This does not require any reflection of the smart contract deployment on the Bitcoin blockchain.

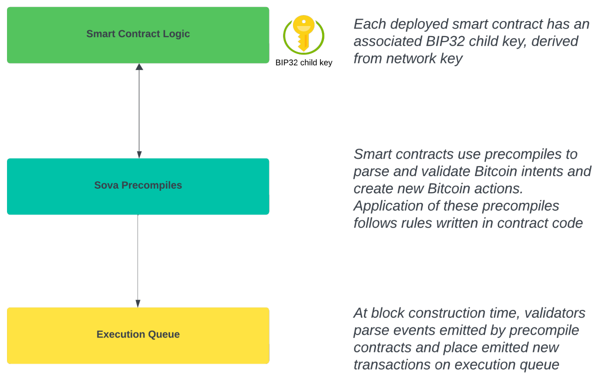
Smart Contract Logic
BIP32 child key
Each deployed smart contract has an associated BIP32 child key, derived from network key
Sova Precompiles
Smart contracts use precompiles to parse and validate Bitcoin intents and create new Bitcoin actions. Application of these precompiles follows rules written in contract code
Execution Queue
At block construction time, validators parse events emitted by precompile contracts and place emitted new transactions on execution queue

Figure 2: *Sova Network Signing Flow*

2.4 Key Resilience

The existence of a global network key, shared by validators, creates obvious attack vectors for assets held by Sova smart contracts. If the root HD key were compromised, any asset held by a smart contract on the network could be stolen.

In order to mitigate this attack vector, the ownership arrangements specific to each Sova smart contract can be defined by Bitcoin-native smart contract signature schemes. These schemes should be defined such that the network key corresponding to a given smart contract holds one of m required signing keys for a multisignature redeem script, but one or more counterparties to the application (end users) must also provide signatures in order to fully sign any related settlement transaction. For instance, in an Ordinals lending protocol, a smart contract may define a 2-of-3 escrow address, with the borrower, the lender, and the network each serving as a particular signer. When the borrower wants to repay a loan to recover collateral, the network can cooperate by providing its signature given that the borrowers has fulfilled the terms of the loan (signed a PSBT for repayment of owed funds).

Another advantage of multisignature schemes for smart contracts which own assets is resiliency to network liveness failures. In the case that the Sova network failed and was no longer able to provide its signature to recover an asset, end-user signers can independently construct a transaction to remove the asset from the multisignature address. In the Ordinals Lending use case described above, in the absence of network participation a borrower and lender can constitute the necessary quorum to recover assets from the escrow wallet.

2.5 Sova Execution Queue

While Sova blocktimes follow the 12-second slots defined by Ethereum proof of stake, Bitcoin proof-of-work blocktimes are unpredictable and can be ten minutes or more. This imposes a constraint on composability and sequencing for Sova transactions that use Bitcoin write precompiles.

In the EVM, interaction with a Bitcoin write precompile emits a logged event; block builders then must subscribe to logged events from the Bitcoin precompile contract, and perform requisite actions on Bitcoin to reflect the intent of the EVM trans-

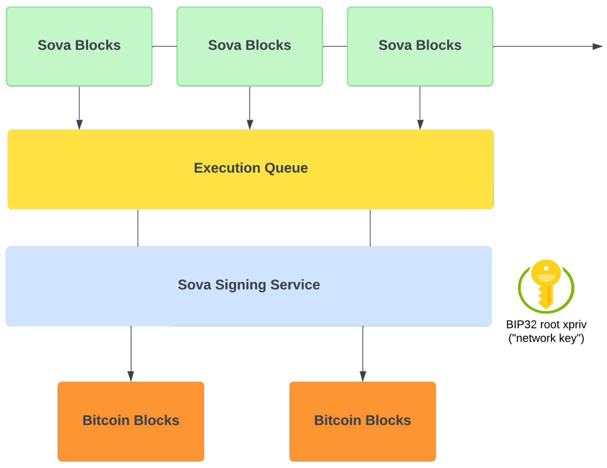

Figure 3: *Sova Execution Queue. When building blocks, validators parse precompile logs and publish corresponding transactions to the execution queue. The Sova signing service asynchronously pulls transactions from the execution queue, signs unsigned inputs associated with the network key, and broadcasts them to Bitcoin. Validators are subject to slashing if invalid transactions are submitted to the queue.*

action.

When a Bitcoin interaction in a Sova transaction requires a new Bitcoin transaction, transactions must be constructed by the validator and placed in the global Sova Execution queue. The Sova Execution Queue contains a sequenced set of unsigned and unbroadcasted transactions awaiting signature from the relevant network key.

When a block is constructed, a validator must build and enqueue *all* corresponding Bitcoin transactions generated by the Sova EVM execution layer, and must enqueue them correctly. Validators who fail to enqueue transactions or do so incorrectly are subject to slashing. Honest validator behavior can verified before providing network key signatures, with misbehaving validators slashed before incorrect transactions are broadcasted to Bitcoin.

Once any corresponding Bitcoin transaction has been placed on the Sova Execution Queue, the relevant Sova EVM transaction has been fully processed and blocks can be published. This means that EVM transactions may confirm without their corresponding effects having taken place on the Bitcoin blockchain. In order to prevent race conditions, double-spends, and state inconsistency, the execution queue and subsequent Sova blocks interacting with dependent assets must be sequenced according to specific constraints.

When a transaction is placed on the Sova execution queue, all inputs associated with the transaction should be flagged. Global input flags are shared among the validators and available to all block builders processing transactions and populating the execution queue. When an input is flagged, it may not be an input to any other transaction placed on the execution queue. Flags can be pruned either by the Sova sequencer as execution queue transactions confirm, or after an amount of time after which the transaction should have a high probability of a reorg-resistant number of confirmations (e.g. 24 hours).

2.5.1 Transaction Requirements

Adding transactions to the execution queue may fail under a number of conditions:

- The computed inputs do not exist.
- The computed inputs have been flagged during previous additions to the network queue.
- The computed unsigned inputs cannot be signed by a derivation of the network key.

When a corresponding transaction can be constructed but not added to the network queue given the conditions above, the corresponding Sova EVM transaction should fail and no EVM state changes may occur.

In addition to honest construction of Bitcoin transaction to reflect the semantics of the Sova EVM transaction, validators and/or block builders must ensure for all Bitcoin transactions that:

- The transaction follows the BIP370 format for partially signed bitcoin transactions[5].
- All inputs not owned by the network key have been signed.
- Fees are not zero and have been set to a reasonable value - the Sova sequencer may reject transactions above certain sat/vB ratios and slash the validator.

- Change outputs have been determined and properly accounted for.

Failure to meet this set of constraints may result in validators/block proposers being slashed. The Sova node itself supplies fee calculation and unspent selection algorithms, which can be tuned by node operators.

When a transaction is successfully placed on the execution queue, it should contain the following data:

- The underlying partially-signed Bitcoin transaction
- The txid of the corresponding Sova VM transaction
- The block number in which the corresponding Sova VM transaction was confirmed
- The Ethereum validator key of the validator who placed the transaction on the execution queue

Ordering within the execution queue is decided by the validator who publishes the execution log (see Section 3.1). However, the execution queue should enforce a strict sequential block ordering such that Bitcoin transactions from later Sova EVM blocks cannot precede those in earlier Sova EVM blocks.

2.6 Sequencing Service

In order to ensure the security of network key material and efficient economics for Bitcoin network operation, the Sova network uses a singleton sequencing service which is responsible for controlling the network key, signing inputs of transactions in the Bitcoin execution queue, and broadcasting those transactions.

Sequencing Service key material is a critical component of the security profile of the network and must be protected against both destruction and exfiltration. The sequencing service must control a root extended master private key, and publish the corresponding public key for derivation by block builders and validators. In order to protect against physical and machine-level security considerations, the network master key should be broke into shards, each of which should be placed in physically distributed trusted execution environments.

Processing a transaction on the execution queue constitutes the following steps:

1. Reading the Bitcoin transaction and verifying its semantics against the corresponding Sova VM transaction.
2. Ensuring all transaction requirements are met (defined in 2.4.1 above).
3. Validating the sequencer has enough BTC to pay fees for the transaction.
4. Distributing the transaction to sequencer nodes to apply threshold signatures with each shard.
5. Broadcasting the transaction to the Bitcoin blockchain.

If any transaction requirements above are not met, the sequencing service may slash the validator. Slashing messages must be signed with a unique hardened derivation of the root network master key.

In addition to broadcasting transactions as-is, the Sequencing Service may also be able to perform efficient batching of transactions when possible. This is a potential area of further research.

3 Network Security

3.1 Consensus

The Sova network itself uses Proof-of-Stake (PoS) consensus bootstrapped using pooled security. The network uses a turn-based sequencing regime government by validator stakes on Ethereum mainnet. Validators must run Sova network full nodes as well as Bitcoin nodes, in order to access Bitcoin state when interacting with query precompiles.

Like other proof-of-stake mechanisms and EVM-based blockchains, Sova is block-based, with an epoch-based proposer selection system. At the beginning of each epoch, a validator committee is randomly selected and assigned block proposer slots for the epoch. In each slot, the assigned validator is responsible for building and publishing a block consisting of public transactions from the mempool.

In addition to a sequence of transactions required for publishing a block as enforced by all EVMs, the Sova network also enforces the publishing of a separate list of transactions for the Sova Execution Queue. When a block is published, the corresponding Bitcoin transactions can be considered to have been enqueued for eventual processing by the sequencing service; after block publication, they cannot leave the queue. Validators are responsible for a *complete* and *accurate* publishing of enqueued transactions for the execution queue, given the corresponding Sova EVM transactions included in the block.

In summary, a validator is subject to the following block publishing conditions:

- A validator *must* publish a block during its assigned slot.
- Transactions must be honestly populated from the mempool in order of fees.
- Interactions with Bitcoin-query precompiles must show honest results; this can be enforced through inspecting subsequent state changes.
- Each interaction with Bitcoin-write precompiles must result in a corresponding transaction for the Sova Execution Queue.
- Interactions with Bitcoin-write precompiles must be honestly translated into corresponding Bitcoin transactions according to smart contract state and event logs.

Blocks published on Sova require three additional pieces of data in addition to normal block data:

- ***Bitcoin synchronization data***. The block header of the current tip of the Bitcoin chain at the time of block publication.
- ***Query log***. An ordered mapping, keyed by Sova EVM transaction ID, of transactions which interacted with Bitcoin-query precompiles, the corresponding query request made to the Bitcoin node, and the values returned to the EVM.
- ***Execution log***. An ordered mapping, keyed by Sova EVM transaction ID, of transactions which interacted with Bitcoin-write precompiles, the corresponding Bitcoin transaction that was constructed, and the transaction's unique ID on the execution queue.

One area of further research is if it is possible to reduce Bitcoin full node requirements on all validators by using a form of proposer-builder separation, or PBS[8]. In initial designs, validators already have the choice of accessing a hosted Bitcoin node instead of having to run a Bitcoin node directly alongside the Sova stack. In the future, it may reduce validator overhead to have a separate Bitcoin transaction execution service that can encode and place Bitcoin transactions on the Sova execution queue; at the cost of decentralization, the sequencing service can be responsible for this step. If Sova introduces a form of PBS specific to building Bitcoin transactions, validators who publish those transactions as part of block propagation would remain economically for the content of those transactions and completeness of transactions with regard to the precompile events logged in the block. Determining the best trade-offs with regard to validator overhead, decentralization, and enforceable honest operation is an area of further research. Given that a form of proto-PBS is used among 60% of the Ethereum mainnet network today via MEV-Boost[10], such a solution should evolve along similar timelines to mainnet-enshrined PBS.

3.2 Pooled Security

In a proof-of-stake system honest validator operation must be enforced through economic security[11]. Sova suffers from a cold-start problem in that at launch, its network token $SOVA may be less valuable than the underlying Bitcoin-native assets the network secures. In order to quickly bootstrap economic security to acceptable levels, the Sova network can leverage pooled security through restaking on Ethereum mainnet.

Restaking is a shared security protocol introduced by EigenLayer on Ethereum mainnet, and introduced and adopted by multiple teams. It is currently live on Ethereum mainnet as of April 2024[6]. Pooled security and restaking allow a group of stakers to subscribe to validation in a service external to the Ethereum blockchain itself; by allowing their stakes to be slashed by the service under predefined conditions, they align incentives towards honest operation of that external service. Stakers may subscribe to multiple services at once with the same staked value, hence the term “restaking”. Finally, users who wish to stake but not operate their service may delegate their stakes to parties actively involved in operation of the service, or "operators".

Using a restaking protocol to bootstrap economic security on the Sova chain has multiple advantages. First, Sova can leverage an existing community of operators who have previous experience running Ethereum-based infrastructure. Second, stakers can stake assets other than $SOVA; for instance, by allowing $ETH staking and slashing for Sova validators at launch, Sova can leverage existing, restaked stakes and build economic security from an asset with a much larger market capitalization and thus a larger security budget.

Using a restaking protocol with a dual-staking model allows Sova to inherit stronger security on Day 1, such that it can economically secure Bitcoin and Bitcoin-native assets on Day 1. As the ecosystem grows and demand for Sova blockspace increases, the Sova network can transition to a model wholly secured by $SOVA, and evolve from an L2 with economic security derived from mainnet to an L1 with native $SOVA staking securing the network on its own.

3.3 Slashing Conditions

The Sova EVM is subject to the same slashing conditions as Ethereum proof-of-stake[2], along with additional slashing conditions related to Bitcoin-specific interaction.

Additional slashing conditions are described below:

- ***Failure to keep Bitcoin node synchronized.*** Blocks published on Sova must contain the Bitcoin block header at the time of publication. Publishing a different block header from the canonical chain may result in slashing. Honest publications of the canonical chain which are later re-orged away will not be slashed.

- ***Failure to return accurate Bitcoin-query results.*** Blocks published on Sova must contain a "query log"; this is a mapping of EVM transactions which interact with Bitcoin-query precompiles, the corresponding queries made to the Bitcoin node by the validator, and the response data returned to the EVM. Inaccurate response data is subject to slashing. This slashing mode should incur a high penalty, as it may lead to inaccurate state transitions on the Sova EVM chain.

- ***Failure to include Bitcoin-write transactions.*** Blocks published on Sova must contain a "execution log"; this is a mapping of EVM transactions which interact with Bitcoin-write precompiles. If a published EVM transaction included a call to a Bitcoin-write precompile, but a corresponding Bitcoin transaction was not published in the execution log, the validator is subject to slashing. Validators can also be subject to slashing for publishing an entry in the execution log with an inaccurate execution queue ID.

- ***Mishandled Bitcoin-write transactions.*** Bitcoin-write precompiles should clearly define the semantics of what should happen on Bitcoin when the corresponding EVM transaction is processed. If the execution log publishes a transaction that does not accurately match the semantics of the Bitcoin-write precompile, the validator is subject to slashing. This slashing mode should incur a high penalty, as it may lead to unapproved transfers of assets on the Bitcoin blockchain.

- ***Failure to validate Bitcoin-write transactions.*** Transactions that are built, included in the execution log, and sent to the Execution queue must be fully signed for all inputs not owned by the network. If a transaction is not fully signed, it should not be included on the execution queue, and is a similar class of failure as the failure to include the transaction at all. Validators must validate signatures at block-building time and cause EVM transactions to fail if external inputs are not signed.

- ***Bitcoin input double-spending.*** As described in Section 2.5, inputs that are placed on the Sova execution queue are flagged until confirmation. In some cases, an EVM transaction may attempt to operate on flagged inputs. These transactions should be caused to fail by the validator; if a validator includes and honestly processes such a transaction, and adds duplicate inputs to the Sova Execution Queue, they are subject to slashing.

3.3.1 Slashing Proofs

In a both a restaking protocol and a chain-native staking protocol, validators must provide proof that one of the above slashing conditions were met in order to slash the publisher of a previous block. Each validator of the Sova network has the full amount of data available to them during the time of block publication in order to determine if any of the Bitcoin-specific slashing conditions have been met.

In order to do so, validators who suspect dishonest activity should "replay" the block, and ensure that the generated query log and execution log match the publish query log and execution log, in terms of query inclusion, query results, execution inclusion, and execution semantics. In the case where generated logs do not match publish logs, validators can publish a proof of the "correct" block to challenge the original validator. Once a challenge block has been published, the entire validator set can vote on which block to canonically include. The publisher of the disqualified block, whether the original publisher or the challenger, will then incur the slashing penalty. Slashing penalties will be defined by the class of failure and the number of instances of failure in the block, with no hard cap on slashing (up to 100% of stake can be slashed).

3.3.2 Finality

Given the slashing conditions above, and the absolute requirement of synchronization between Sova EVM state transitions and Bitcoin blockchain effects, Sova cannot provide per-slot finality. Finality can be affected by both Bitcoin block reorganization and Sova block reorganization, subject to block challenges described above.

Bitcoin finality is probabilistic[16]; as such, Sova inherits this constraint and can only provide probabilistic finality as well. Bitcoin block reorganization is handled by the Sova Sequencing Service; previously-confirmed transactions which are re-orged away can be tracked by the Sequencing Service and re-added to the execution queue with highest priority. At the time of re-adding these transactions to the execution queue, included inputs should be re-flagged such that subsequent EVM transactions cannot include these inputs. Since Bitcoin-level reorganization handling is internal to the Sequencing Service, Bitcoin reorgs need not precipitate reorganizations of the Sova EVM chain.

In some situations, the Sequencing Service must force a Sova EVM chain reorganization. In addition to flagging transaction inputs enqueued in Sova transactions, the Sequencing Service must monitor the Bitcoin mempool and check for extrinsic input conflicts. If the Sequencing Service were to submit a transaction on an input that had been already spent, it would be eventually be rejected; in this scenario, the Sova EVM block containing the originating transaction with the relevant Bitcoin-write precompile must be reorged such that the EVM transaction fails.

Block challenges which result in slashing, on the other hand, can involve reorganizations to the Sova EVM chain. To enforce the absolute synchronization requirement, successfully challenged blocks should be replaced, and the correct block should constitute the new chain tip. This *must* happen before transactions from that block's execution log are published and broadcasted by the sequencing service, otherwise Sova EVM state risks falling out of sync with underlying Bitcoin state. This imposes a short challenge period and a requirement that the Sequencing Service not broadcast transactions from the execution queue until the challenge period is complete.

This design imposes a challenge given that users desire quick confirmations, or minimal time spent waiting in the execution queue. However, smaller challenge periods impose costs on validators to verify blocks, challenge blocks, and conduct votes in real time. One mitigating factor is the relatively long average blocktime of the Bitcoin blockchain itself (10 minutes). While Bitcoin blocktimes are probabilistic and may be much shorter, an equivalent challenge period of 5 minutes provides a balance between a high probability of same-block inclusion for end-users, with a relative slack in required processing time for an automated block verification service. Blocks for which challenges are raised can have their waiting period raised to a maximum of one hour, or until a challenge is resolved by validators. This gives validators greater leeway to poll for open challenges and re-verify blocks when needed at a pace which does not impose high real-time costs. In this model, a challenge which has not been voted successful within 1 hour defaults to failure.

3.4 MEV Mitigation

Given the finality constraints described above, it's likely that economic activity on Sova creates the opportunity for Maximum Extractable Value, or MEV[12].

Given that Sova is an EVM chain, Sova will present similar MEV opportunities to those on other EVM chains with economic activity, such as Ethereum mainnet. However, state synchronization with Bitcoin through the Execution Queue, finality constraints, and the smart contract deployment flow described in Section 2.4 create new classes of possible MEV.

3.4.1 Contract Deployment Front-running

Given that deployed contract addresses can be precomputed based on the global nonce, parties may be motivated to try to deploy their contract to a specific nonce, or manipulate the nonce other contracts are deployed under. It is possible that given a pending deployment to a certain address, that deployment could be front-run such that the originally intended nonce gets pushed one back, and the smart contract associated with the target nonce is replaced by one containing malicious code.

3.4.2 Execution Queue Front-running

Front-running transactions in the Execution Queue is possible given that the transactions in the queue are not yet broadcasted. While the Sova network prevents the double-spending of inputs within its slashing conditions, its possible that malicious users could choose to double-spend inputs enqueued in Sova, via methods extrinsic to Sova. If Sova Sequencing Service transactions are rejected due to double-spends, it can force Sova EVM reorgs to the point of chain halt.

One way to mitigate Sova reorgs due to double-spends is to place locks on specific storage slots affected by Bitcoin transactions which are pending confirmation. For example, let's say Bitcoin transaction A is submitted to the Sova network and consequently updates particular storage slots in the Sova state. Before transaction A can be confirmed, it is double-spent outside the context of the Sova chain effectively making the storage slot updates invalid. When subsequent Sova transactions come along and need to update this storage slot, the slot can be return to its previous value and allow the new transaction to use the previous state effectively canceling out the double spent transaction's affect on the network.

Outside of network-level halting attackers precipitated by users, the economic protocols enabled by Sova can themselves create front-running opportunities: both garden-variety within-block MEV, and across multiple blocks with assistance from other Bitcoin-native protocols outside of Sova. In the former case, it's important that Sova protocols include common mitigation mechanisms such as offer deadlines, slippage protection, and signature expiry dates.

In the latter case, mitigation is more nuanced given that other Bitcoin-native protocols can exist outside Sova. For instance, an observer of the execution queue might notice buying activity for one particular Bitcoin-native asset in a set of transactions on the Bitcoin-native execution queue. This creates an arbitrage opportunity for the observer, who can use external marketplaces such as Magic Eden to buy up assets in question before the Sova-native transactions broadcast and affect asset prices across all markets.

3.4.3 Mitigations

Like on Ethereum itself, there are promising lines of research to minimize externalities and user loss caused by MEV. These include private mempools (or private Execution Queues) and internalization of MEV by validators. Both of these techniques as well as other lines of research can and should be implemented on the Sova network.

4 Economics

The native token of the Sova blockchain is $SOVA. Compute on the Sova blockchain is paid for with this token; like in other EVM-based systems, users must pay $SOVA transaction fees when submitting transactions to the Sova blockchain. These fees are then collected by validators when a given transaction is included in a block.

Validators earn $SOVA both through transaction fees and staking inflation. Inflation will be set to a number that allows proper incentive alignment for validators, encourages honest operation, and brings enough $SOVA into circulation to eventually support economic security solely through staked $SOVA. As described in Section 3.2, in early stages of the network economic security can be provided by both $SOVA and $ETH.

Beyond end-user and validator economics, the network keys must hold enough $BTC to be able to pay fees to the Bitcoin network. End-users can specify higher fees to be taken from pre-signed inputs in order to incentives faster inclusion both on the Bitcoin network, and for Sova validators (since the network signer will pay a smaller fee and be subsidized by the end-user).

In order to make the Sequencing Service economically viable, the $BTC spent during the network key signing process must be offset by per-transaction fees paid by validators. As such, per-transaction fees will be split between validators and Sequencing Service. The Sequencing Service must be responsible for capitalization any earned fees in a way that can ensure sufficient ongoing balances of $BTC to ensure network operation. Failure of the sequencer to be able to pay Bitcoin fees would equate to a liveliness failure of the network, further underscoring the need for the key resilience schemes described in Section 2.4.

In summary, the requirements and economics of the principals of the network can be described as below:

- ***Users:*** pay per-transaction fees in $SOVA in return for the utility the network provides. Opt-in to providing $BTC fees directly to sequencer for transaction prioritization.
- ***Validators:*** earn $SOVA staking rewards and a portion of per-transaction fees in $SOVA in return for the computational services required by validation and providing economic security to the network through staked assets.
- ***Sequencer:*** earn a portion of per-transaction fees in $SOVA in return for the application of network keys to operate the network and the subsidy of $BTC fees for transactions signed and broadcast by the sequencer.

Given these requirements, it is clear to see the required components of the economic model of the network:

- ***Users*** must receive more utility from the network than their per-transaction costs.
- ***Validators*** must earn more in staking rewards and transaction fees than overhead costs (and opportunity cost) to validate the network. Pooled security helps reduce opportunity cost.
- The ***Sequencer*** must earn more in transaction fees and other subsidies than overhead costs and $BTC cost to participate in input signing and transaction broadcasting services.

More economic research is needed to determine the optimal inflation rates, transaction fee models, and methods of minimizing operation costs for validators and the sequencer. In a long-term economically viable network, each party responsible either supplying block space or providing demand from it must stand to make a profit for participating.

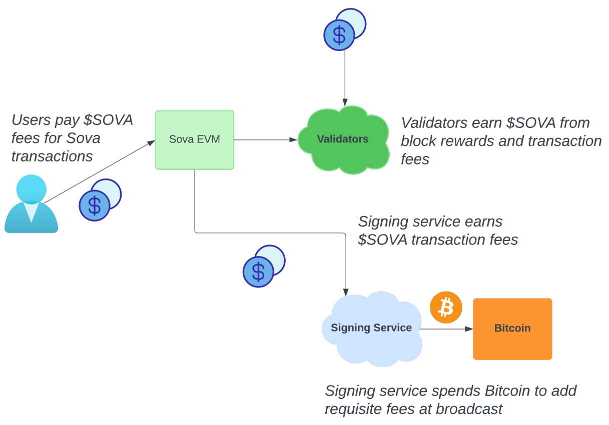
Users pay $SOVA fees for Sova transactions
Sova EVM
Validators
Validators earn $SOVA from block rewards and transaction fees
Signing service earns $SOVA transaction fees
Signing Service
Bitcoin
Signing service spends Bitcoin to add requisite fees at broadcast

Figure 4: *$SOVA/$BTC Economic Flows*

References

[1] A.M. Antonopoulos. *Mastering Bitcoin*. O'Reilly, 2014. URL: https://books.google.com/books?id=h_zToAEACAAJ.

[2] aslikaya. Proof-of-stake rewards and penalties. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/rewards-and-penalties/#penalties.

[3] Bitcoinops. Coin selction. URL: https://bitcoinops.org/en/topics/coin-selection/.

[4] Ava Chow. Bip-0174. URL: https://github.com/bitcoin/bips/blob/master/bip-0174.mediawiki.

[5] Ava Chow. Bip-0370. URL: https://github.com/bitcoin/bips/blob/master/bip-0370.mediawiki.

[6] EigenLayer. Mainnet launch announcement: Eigenlayer ∞ eigenda. URL: https://www.blog.eigenlayer.xyz/mainnet-launch-eigenlayer-eigenda/.

[7] JR Wallet et al. Omni protocol specification. URL: https://github.com/OmniLayer/spec.

[8] Ethereum roadmap: Proposer-builder separation. URL: https://ethereum.org/en/roadmap/pbs/.

[9] Evmos. Evmos: Evm extensions. URL: https://docs.evmos.org/develop/smart-contracts/evm-extensions.

[10] Flashbots. Mev-boost in a nutshell. URL: https://boost.flashbots.net/.

[11] Interchain. Understanding the basics of a proof-of-stake security model. URL: https://blog.cosmos.network/understanding-the-basics-of-a-proof-of-stake-security-model-de3b3e160710.

[12] Marius Kjærstad. Maximal extractable value (mev). URL: https://ethereum.org/en/developers/docs/mev/.

[13] Robin Linus. Bitvm. URL: https://bitvm.org/.

[14] minimalsm. Ethereum virtual machine. URL: https://ethereum.org/en/developers/docs/evm/.

[15] Murch. What are the sizes of single sig and 2-of-3 multisig taproot inputs? URL: https://bitcoin.stackexchange.com/questions/96017/what-are-the-sizes-of-single-sig-and-2-of-3-multisig-taproot-inputs.

[16] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[17] Gholamreza Ramezan, Manvir Schneider, and Mel McCann. A survey on coin selection algorithms in utxo-based blockchains, 2023. URL: https://arxiv.org/abs/2311.01113, arXiv:2311.01113.

[18] BitMEX Research. The op$_returnwarsof2014–dappsvsbitcointransactions.URL:$
.

Casey Rodarmor. Inscribing mainnet. URL: https://rodarmor.com/blog/inscribing-mainnet/.

Casey Rodarmor. Ordinal theory handbook. URL: https://docs.ordinals.com/.

smlXL. Precompiled contracts. URL: https://www.evm.codes/precompiled.

N. Wuille, P. Jonas and A. Towns. Taproot. URL: https://bitcoinops.org/en/topics/taproot/.

5 Appendices

5.1 Appendix A: List of Precompiles

5.1.1 Query Precompiles

1. `getRawTransaction(bytes32 txid): Transaction`

   Returns full transaction information about the specified txid, including inputs and outputs.

2. `getTxOut(bytes32 txid, uint256 n): Unspent`

   Returns information about the nth output of the specified txid.

3. `analyzePsbt(string calldata psbt): Transaction`

   Decodes base64-encoded PSBT into a `Transaction` struct. Can be used to verify that pre-signed PSBTs match settlement instructions defined by the calling smart contract.

4. `isInputQueued(bytes32 txid, uint256 n): boolean`

   Returns whether a given unspent is already part of a transaction in the Sova execution queue. Unspents that return `true` should not be included in newly-queued transactions created by the contract.

5. `joinPsbtSigs(Transaction[] memory psbts): Transaction`

   Joins multiple PSBTs with pre-existing signatures that sign the same transaction into a single Bitcoin transaction with all signatures applied.

6. `combinePsbts(Transaction[] memory psbts): Transaction`

   Joins multiple PSBTs which are independent of each other into a single Bitcoin transaction.

7. `inscriptionOwnerOf(uint256 inscriptionId): address`

   Checks the owner of a specified inscription, specified by ordinal ID. While on Bitcoin, ownership is represented by a Bitcoin address, this function returns the equivalent Ethereum address.

8. `inscriptionOwnerOf(uint256 runeId, address owner): uint256`

   Checks the balance of a specified rune for a specified wallet. While the function parameter is an Ethereum address, the balance is checked for the equivalent Bitcoin address.

9. `createMultisig(address[] memory signers, uint256 m): (address, bytes)`

   Creates a new m-of-n multisig address with the specified addresses as signers. While the function parameter is a list of Ethereum addresses, the equivalent Bitcoin addresses are designated as signers. Returns the newly-created address and redeem script.

5.1.2 Write Precompiles

1. `sendBTC(uint256 sats, address to): void`

Send the specified amount of satoshis to the specific address. The network will perform coin selection and ordinal sats will not be used as inputs. The transaction will fail if the addresses signable by the smart contract do not have sufficient balance.

2. `sendOrdinal(uint256 inscriptionId, address to): void`

   Send the specified ordinal (a single-satoshi UTXO) to the specified address. The transaction will fail if the owner of the specified ordinal is not an address associated with the smart contract.

3. `sendRune(uint256 runeId, uint256 amount, address to): void`

   Send the specified amount of a given rune to the specified address. The transaction will fail if the addresses signable by the smart contract do not have sufficient balance of the specified rune.

4. `sendTransaction(bytes tx): void`

   Broadcast a transaction to the Bitcoin network. This precompile does not trigger any network signing; the transaction must be fully pre-signed. The transaction will fail if any inputs are unsigned.

5. `signAndSendPsbt(bytes psbt): void`

   Trigger network signing of a partially-signed PSBT and broadcast it to the Bitcoin network. This should be used for most smart contract flows which require signing from multiple counterparties (see Appendix B and C below). The transaction will fail if a ny unsigned inputs require signatures from keys not controlled by the network.

5.2 Appendix B: A Bitcoin-Native AMM (Use Case 1)

A Bitcoin-native AMM requires coordination between an unbounded number of counterparties and a single liquidity pool, where tokens for the specified market are held. The example below describes a pool where the one asset is BTC and the other asset is a rune. Rune-to-rune AMMs are similarly possible with slight modifications to transaction flow.

5.2.1 Deployment

- Segregate a single network key to hold assets on both sides of the LP.
- Etch a new Rune to represent the LP token of the pool.
- Deploy a Uniswap-style smart contract to Sova, with modified settlement functions as described below, with contract references to the rune ID of each asset (or `0xBBB` for Bitcoin, and a contract reference to the LP token rune ID.

5.2.2 Adding Liqudity

1. User constructs a PSBT with 3 outputs. The first 2 outputs should be sends of correlated amounts of each pool asset to the LP address, with the last output being the sending of LP tokens to the user. The user's wallet software should source inputs for each output and prompt the user to sign the inputs sourcing the LP pool assets, leaving inputs for the LP token unsigned (to be signed by the network). Amounts to be signed can be computed by reading state from the smart contract. Open question: how to prevent the transaction from failing due to slippage?
2. User calls `addLiquidity(bytes psbt)`. In the smart contract's execution logic, the contract must verify that all user inputs are signed, and that the input amounts and LP token output amount in the PSBT match the reserve ratios defined by the contract.
3. Smart contract updates contract state for LP token balances, reserve ratios, the the total liquidity of the pool.
4. Smart contract calls `signAndSendTranasction` to send the user's PSBT to the execution queue. Once the transaction confirms off the execution queue, the LP pool will receive the correlated token amounts and the user will receive the LP token. The LP token acts as a redemption receipt for their funds (see "Removing Liquidity").

5.2.3 Trading

1. User constructs a PSBT with 2 outputs. The first output should be the token the user wishes to receive in the trade, with the user as the output address. The second output should be the token the user will provide, with the LP pool as the receiving address. This must correlate to the two assets of the LP pool. The amounts must match the current price of the pool as defined by the reserve ratio of the pool, which can be queried from the smart contract.
2. User calls `swap(bytes psbt, uint256 minAmountOut)`. In the smart contract's execution logic, the contract must verify that all inputs are signed, and that the input amounts and LP token output amount in the PSBT match the reserve ratios defined by the contract. If pricing has changed since the user constructed the original PSBT, the smart contract can replace the PSBT's output for the output token with an updated amount. If the pricing has changed and the updated amount is less than `minAmountOut`, the transaction should fail.
3. The smart contract should update internal contract state for reserve ratios and pricing.

4. Smart contract calls `signAndSendTranasction` to send the user's PSBT to the execution queue. Once the transaction confirms off the execution queue, the LP pool will receive the token swapped from and the user will receive the token swaped for.

5.2.4 Removing Liquidity

1. User constructs a PSBT with 3 outputs. The first 2 outputs should be sends of correlated amounts of each pool asset to the user's, with the last output being the sending of LP tokens to the LP address. The user's wallet software should source inputs for each output and prompt the user to sign the inputs sourcing the LP token, leaving inputs for the individual assets unsigned (to be signed by the network). Amounts to be signed can be computed by reading state from the smart contract.

2. User calls `removeLiquidity(bytes psbt)`. In the smart contract's execution logic, the contract must verify that all user inputs are signed, and that the input amounts and LP token output amount in the PSBT match the reserve ratios defined by the contract.

3. Smart contract updates contract state for LP token balances, reserve ratios, the the total liquidity of the pool.

4. Smart contract calls `signAndSendTranasction` to send the user's PSBT to the execution queue. Once the transaction confirms off the execution queue, the LP pool will receive the LP token and the user will receive the correlated token amounts.

5.3 Appendix C: Bitcoin-Native Peer-to-Peer Lending (Use Case 2)

A peer-to-peer lending protocol requires coordination between two counterparties (the borrower and the lender), as well as a collateral escrow mechanism controlled by the protocol itself. The example below describes a lending protocol for ordinals, but the same architecture can be used for any peer-to-peer lending arrangement regardless of collateral asset and funding asset. The terms implied are fixed-term loans with a pre-defined stable interest rate.

5.3.1 Deployment

- Segregate a derivation path under the smart contract's derivation path to be used for escrow addresses generated by the smart contract.
- Deploy a P2P lending protocol smart contract to Sova, such as that built by Arcade.xyz. Like in the AMM use case, settlement functions use Bitcoin precompiles instead of native EVM token transfer functionality. **Note:** Modified settlement protocols can also support using BTC-native as collateral with EVM-native funding currencies or vice versa.

5.3.2 Origination

The origination steps initializes a loan based on terms agreed upon by both counterparties. Collateral is set to an escrow address, with release of collateral conditioned on smart contract logic (see "Repayment" and "Default" steps).

1. Loan origination requires coordination between both counterparties. At origination time, one counterparty will be the **signer** and one the **originator**. The **originator** will initiate the origination transaction, and must provide a signature from the **signer**. Either of the borrowing or lending counterparty may play either of the signer or originator roles.

   The signer will sign an EIP712 signature to the loan terms as well as construct a PSBT with 2 outputs. The first output will send the funding currency from the lender to the borrower, and the second output will send the collateral from the borrower to escrow.. To avoid race conditions and possible collisions, the escrow address must be pre-generated and marked as used by the smart contract. The signer should sign whatever inputs correspond to their responsibilities in settlement. Amounts to be specified and signed in the PSBT can be derived from the loan terms.

2. The originator should co-sign the origination PSBT, such that the PSBT is fully signed for submission to the network. Then, the originator calls `initializeLoan(struct LoanTerms, bytes signature, bytes psbt)`. The smart contract must verify that the loan terms match the terms signed in the EIP712 signature. It must also verify that the outputs in the PSBT match the loan terms, and that the signer of the signature has also signed the relevant inputs in the PSBT. Finally, the smart contract must verify that the escrow address specified in the PSBT's outputs is 2-of-3 multisignature address, with the counterparties controlling 2 keys and the last key being the pre-generated escrow address controlled by the smart contract, as specified in Step 1.

3. The smart contract stores the loan state and associated terms in smart contract storage, and performs settlement via Bitcoin interaction. Smart contract calls `sendTransaction` to send the counterparties' transaction to the execution queue. Once the transaction confirms off the execution queue, the borrower will receive the loan funds and the escrow address will receive the loan collateral.

5.3.3 Repayment

In the repayment steps, the borrower pays owed funds to the lender, and receives their collateral back from the escrow address.

1. The borrower should construct a PSBT with 2 outputs: the first output should send the loan principal and accrued interest to the lender, and the second input should send collateral from the escrow address to the borrower. The borrower should pre-sign both the inputs for repayment as well as provide a partial signature for the withdrawal from the escrow multisig.
2. The borrower calls `repayLoan(bytes psbt)`. The smart contract must verify that the loan is active and that the escrow wallet still holds the collateral (the borrower has not defaulted). It must also verify that the outputs in the PSBT match the owed principal and interest as calculated by the smart contract. If the repayment amount is not sufficient, the transaction should fail.
3. Given a valid repayment, the smart contract should update stored loan state and initiate settlement on Bitcoin. The smart contract calls `signAndSendTransaction` to send the borrower's PSBT to the execution queue, pending a co-signature from the networkon the withdrawal from the multisig. Once the transaction confirms off the execution queue, the borrower will receive their collateral and the lender will receive their owed principal and interest.

5.3.4 Default

Default can occur if the borrower has not repaid funds before the loan's due date. In the default flow, the lender coordinates with the smart contract to recover the collateral from escrow.

1. The lender should construct a PSBT with a single output: the sending of the collateral from the escrow wallet to the lender. The lender should partially-sign the multisignature input from the escrow wallet for the withdrawal of collateral.
2. The lender calls `claimLoan(bytes psbt)`. The smart contract must verify that the loan is active and that the escrow wallet still holds the collateral (the borrower has not repaid). It must also verify that the due date of the loan has passed. If the escrow no longer holds the collateral or the due date has not passed, the transaction should fail.
3. The smart contract should update stored loan state and initiate settlement on Bitcoin. The smart contract calls `signAndSendTransaction` to send the lender's PSBT to the execution queue, pending a co-signature from the networkon the withdrawal from the multisig. Once the transaction confirms off the execution queue, the lender will receive the collateral, and the borrower will have no further loan obligation.