The Bit-Fusion Bridge Framework

Bitfinity Team

July 10, 2024

Abstract

Bit-Fusion is an open-source framework that allows developers to create decentralised bridges between an EVM network and Bitcoin, Bitcoin Runes, and other EVM networks, thereby enabling EVM chains to use the new assets from the Bitcoin ecosystem. In this sense, Bit-Fusion can be thought of as a Bitcoin L2 builder: it provides the necessary infrastructure to bridge assets from Bitcoin with decentralised security. Bit-Fusion bridges are accessible through a bridging widget that downstream applications can integrate to provide a nearly native user experience. The technology underpinning Bit-Fusion includes Chain-Key cryptography, decentralised integrations with the web via HTTP Outcalls, as well as a network of on-chain Bitcoin nodes to create a robust, decentralised framework for secure bridging.

Contents

1 BitFusion Background Assumptions

1.1 Introduction

The Bit-Fusion framework serves as a foundational infrastructure for Bitcoin Layer-2 solutions. It incorporates a threshold Elliptic Curve Digital Signature Algorithm (tECDSA) for transaction signing, interfaces directly with a network of Bitcoin nodes, and provides bridging capabilities for a diverse array of assets from the Bitcoin ecosystem, including Runes and BTC.

Bit-Fusion leverages committee-based security assumptions, which, despite being an addition to those of the base layer, we argue do not prevent it from being categorized within the family of Layer-2 solutions. The current discourse on Layer-2 solutions predominantly revolves around deriving security entirely from the base chain. However, we contend that this approach is practically infeasible for upgradable Layer-2s by presenting arguments suggesting an impossibility result.

This paper argues for the necessity of committee-based security in Layer-2 solutions and presents Bit-Fusion's approach to addressing this challenge, by building on Chain-Key, the only provably secure protocol for tECDSA that guarantees liveness under the model of partial asynchrony. This scheme supports committees comprising 50 or more members, comporable to the size of the consensus set for modern proof-of-stake blockchains as in SUI or Cosmos.

1.2 Security Assumptions for Upgradable Layer-2s

Roughly speaking, the prevailing view in the blockchain community is that a Layer-2 should not introduce unwarranted security assumptions beyond those inherent to the base chain. This implies that compromising a Layer-2 would require compromising the base chain itself. Typically, Zero-Knowledge Layer-2 solutions post mathematical proofs to the base chain to validate correct state transitions, purportedly tying their security to the consensus mechanisms of the base chain.

However, when examining upgradable Zero-Knowledge Layer-2 protocols, the perspective shifts. The necessity for upgradability arises from the inherent risks of bugs and the evolving nature of protocols. On base chains like Ethereum, network forks can update Ethereum Virtual Machine (EVM) clients to correct bugs or implement changes in the state transition function $F : s \to s'$. However, on Layer-2,

modifying the state transition function F requires changes to the Layer-2's verifier smart contract, necessitating approval by a designated committee or entity, perhaps via a threshold signature. This arrangement presupposes the committee's integrity, as it must not exploit its power for actions like double-spending or fund misappropriation.

Thus, the practical upgradability of Zero-Knowledge Layer-2 solutions ultimately depends on the security of protocols managed by an entity external to the base chain. Without this external governance, any upgrade to the Zero-Knowledge Layer-2 and its proof-verification smart-contracts on the base-chain, would necessitate a network fork of Ethereum. However, frequent forks for on-chain smart contracts are unlikely to be accepted by Ethereum network participants.

In the Bitcoin network, Zero-Knowledge (ZK) Layer-2 solutions are completely infeasible. This is due to Bitcoin's scripting language limitations, which do not allow the Bitcoin base layer to verify ZK proofs. Presently, all Bitcoin Layer-2 solutions, whether upgradable or not, rely on committee-based security assumptions.

1.3 A New Standard for Layer2s

Upgradable Layer-2 solutions, essential for maintaining security, inherently require the use of committees. Our framework, Bit-Fusion, aims to set a new standard for decentralized bridging by redefining Layer-2 solutions. Prominent institutions like Fireblocks and BitGo secure vast assets using threshold signatures and Multi-Party Computation (MPC) technologies, demonstrating the viability of MPC in Layer-2 solutions without compromising security.

The primary challenge has been scaling the threshold group to ensure MPC security comparable to that of proof-of-stake blockchains. Bit-Fusion addresses this challenge by utilizing Chain-Key cryptography, which incorporates a threshold-ECDSA scheme where 40-50 nodes hold key shares. This configuration ensures a high level of decentralization and security, comparable to modern proof-of-stake blockchains.

2 BitFusion

In this section, we will provide a high-level overview of some of the important components used in the BitFusion framework, which are used within bridging flows for different blockchain networks.

2.1 Components Overview

1. **Bridge Canister:** BitFusion utilizes a canister called the bridge canister to orchestrate the bridging flow. This canister serves as a decentralized web-server and inherits robust security properties such as distributed consensus and fault tolerance from the Internet Computer protocol. Canisters are equipped with a timer, or scheduling mechanism, to execute tasks periodically. Specifically, the canister's timer can monitor events on a designated blockchain and respond appropriately to detected events through http outcalls or the BTC adapter canister. The bridge canister can also issue requests for the subnetwork to sign transactions via tECDSA.

   The bridge periodically runs a task scheduler, called the BridgeRuntime, responsible for orchestrating the sequence of bridging operations. The BridgeRuntime listens to EVM events and schedules tasks to perform various operations. It manages the progression through the sequence of operations by scheduling new operation tasks until the bridging process completes. The BridgeRuntime allows users to restart an operation task in case of failures. It is also possible to query the current state of the bridge by operation ID or retrieve a list of bridge operations by token ID.

2. **Bridge and Wrapped Token Factory:** The framework deploys a bridge smart contract which can act as a token factory and owner for wrapped ERC-20 tokens, whilst also receiving

and escrowing base tokens. The bridge verifies the tECDSA signatures as signed messages before minting or burning any wrapped tokens.

Mintable ERC-20 tokens: Each wrapped token can be minted and burned by the bridge smart contract.

3. **tECDSA canister API:** This is the implementation of threshold ECDSA protocol by Groth and Shoup. The BitFusion bridge canister can make tECDSA requests through the management canister interface. Please see the appendix for more details on this.
4. **Bitcoin Adapter:** The Bitcoin adapter canister is the decentralised gateway interface to the Bitcoin network that the bridge canister calls into.
5. **Chain-Key Bitcoin:** Chain-Key Bitcoin is a wrapped Bitcoin token on the Internet Computer, realised under the ICRC-1 and ICRC-2 standards. To perform the wrapping, a minter canister communicates with the Bitcoin adapter scanning for UTXOs sent to the canister's tECDSA address, and then mints the corresponding amounts on the ICRC-2 ledger canister.
6. **HTTP Outcalls:** Through this API, the blockchain network can all collectively make an HTTP request and come to consensus on the result. This is essentially a more decentralised and flexible API for oracles, that enables a canister to request data actively rather than wait for data to be posted to the blockchain through an oracle service. It is used when fetching data from external chains, although not in the BTC bridging flow which directly relies on the BTC network and is more secure.

2.2 Discussion on Threshold ECDSA

This protocol ensures that private keys are never fully reconstructed but remain distributed among subnet replicas, enhancing security by making it incredibly difficult for an attacker to ex-filtrate key-shares from the nodes on the network. It has the following dimensions:

1. **Distributed Key Generation:** Executed on specific subnets to generate new threshold ECDSA keys, ensuring that the private key components are distributed and securely stored.
2. **Key Re-Sharing:** Periodically re-shares keys within the current subnet or across to a new subnet, enhancing security against adaptive attacks by rendering previously obtained key shares obsolete. On the IC, the fresh key-shares are re-shared approximately every 10 minutes, giving an attacker a window of 10 minutes to exfiltrate on third of the subnet's key-shares.
3. **Signing Operations:** The implementation supports efficient signing operations by leveraging the distributed key shares across the subnet nodes, thus ensuring secure and verifiable transactions.

The threshold ECDSA protocol is uniquely, one of the few protocols provably secure and live under the partially asynchronous model, which is an appropriate model for communciations on the blockchain. More details can be found in the original paper by Shoup and Groth.

2.3 Discussion on the Bitcoin Network

The Bitcoin adapter canister is the decentralised gateway interface to the Bitcoin network that the bridge canister calls into. To provide this API, there is a subnet, where each node has access to its own Bitcoin node. The subnet syncs the Bitcoin blockchain, and comes to consensus on the current UTXO set. The current UTXO set is then made available to the rest of the network through the BTC adapter APIs. Other canisters can call into this canister and query information on the UTXO set. It is planned to open up the APIs so that the the history of block-headers is also made available tot he network, which would allow for a way to build a fully decentralised indexer from the block-data, which would pave the way for Runes Indexers to be built fully on-chain.

2.4 On-Chain Ordinal and Runes Indexers

By adapting the Bitcoin adapter, we can retrieve the block header information from the Bitcoin blockchain and validate it through consensus. Once extracted, the header hashes are paired with the actual block data provided through an Oracle or HTTP outcalls. This mechanism ensures the authentication of the parsing and interaction with genuine Bitcoin blockchain blocks with a high degree of security.

A canister operates a fork of the ord repository, utilizing the same logic to parse blocks and extract Runes. Only the modules related to parsing Runes from blocks have been integrated into the on-chain canister parser. This indexer processes all Rune block data and serves the information through a canister API. This result is significant, demonstrating that interaction with Indexers can be achieved in a completely decentralized manner. The work to crate the on-chain indexer was initially undertaken by the developers at Omnity and their repo can be found here.

3 The BitFusion Flow

The bridge canister completes the bridging flow through a sequence of *operations*, which are tasks that must complete successfully to execute a bridging transaction, such as depositing or withdrawing tokens from the bridge. After an operation has completed successfully, the associated state is stored in the canister's stable memory. There are predefined operations for common parts of the bridging process, as well as custom operations that can be implemented by the user when implementing a particular bridge. All bridges share the same methods for minting and burning tokens on the Bitfinity EVM, and as such these operations are implemented by default which a developer can use through a trait interface. For the operations implemented by default, the following figures outline the steps involved as part of both the deposit and withdraw flows.

3.1 Deposit Flow

Figure 1 below shows the standard flow for depositing tokens on the base chain, and minting wrapped tokens. Before executing these steps, tokens must be locked on the base chain, and custom methods must be implemented for the bridge to recognise this, explained in subsequent sections.

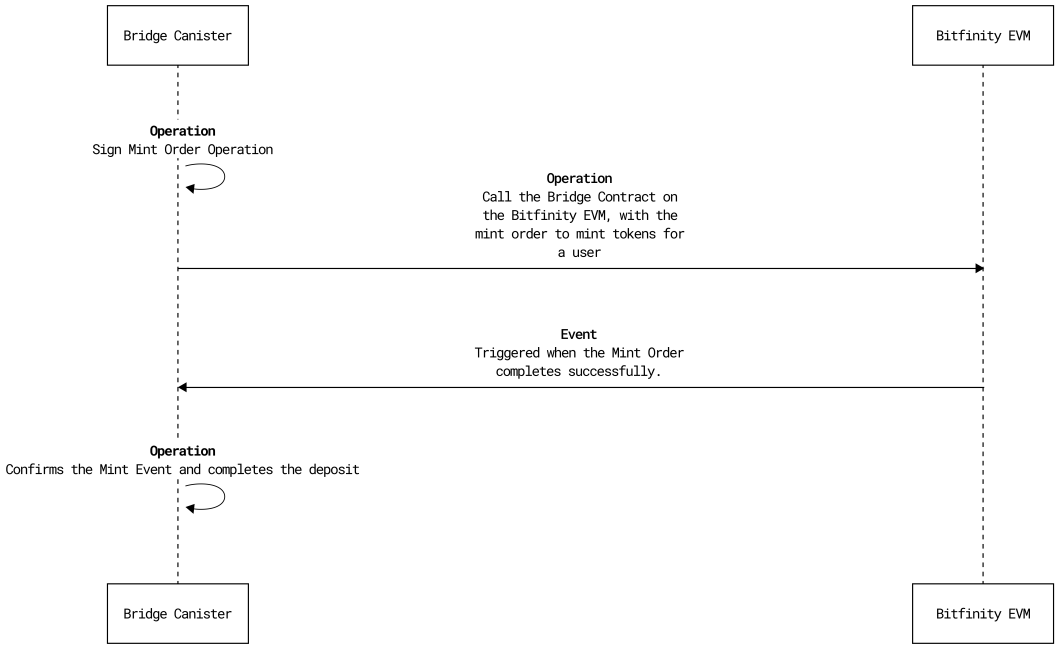


Figure 1: Overview of the default implemented flow for depositing base tokens and receiving wrapped tokens

The sequence of operations and events are explained in detail below. Note, these operations and events are imlemented by defaults as part of the bridge trait.

1. **Sign the Mint Order (Op)**: This is the first operation after tokens have been locked, and the canister bridge has detected this. This operation signs byte data that can be read by the bridged BftBridge smart contract to mint bridged tokens to the appropriate EVM address.

   The next step is to

2. **Send the Mint Transaction (Op):** This operation issues a transaction on the Bitfinity EVM to mint tokens, by calling the mint function on the BFTBridge smart-contract, with the signed mint order as data. Once this transaction is confirmed, wrapped tokens will be minted on-chain for the user.

3. **WrappedTokenMinted (Event):** Created by the EVM in a log when a wrapped token is minted. Implemented by default, the event handler for this event schedules a confirmation operation, a terminal operation that removes the operation data from storage.

4. **Confirm Mint Event (Op)**: This operation is triggered when the BridgeRuntime catches the corresponding Minted Event on the Bitfinity EVM by reading the logs. Developers can override this behaviour in the implementation of the Canister's Bridge trait.

3.1.1 Other Points of Consideration - Fees, DOS Protection

When the Bitfinity bridge canister mints tokens for the user, as part of the transaction the bridge will also request a fee: by transferring a small amount of Bitfinity tokens as payment to cover the

transaction costs. This fee is paid for by the user, who as part of the bridging flow, must approve the bridge canister to spend a small amount of Bitfinity tokens before commencing with the bridging. Though not explicitly shown in flow, this is done as part of a one-time setup, and will cover fees for all bridging operations for the relevant bridge.

Though not part of the standard flow, it will be seen later on in this paper that many of the deposit flows make use of a **NotificationEvent**. This is a generic event that returns untyped data, which can be parsed by the bridge canister. Typically we use this event for two purposes in the bridging flows: (1) to parse the recipient addresses intended to receive wrapped tokens; (2) as a means of DOS protection, as generating this event proves that the user has paid a gas fee before commencing any bridging operations.

3.2 Withdrawal Flow

Figure 2 below shows the flow for withdrawing tokens on the base chain. In this flow, after burning wrapped tokens, the bride canister will listen for the event and schedule an operation to transfer tokens to the user's designated wallet on the base chain. The operation to transfer base tokens will be implemented by the developer, and must successfully parse the user's base-chain wallet from the event.

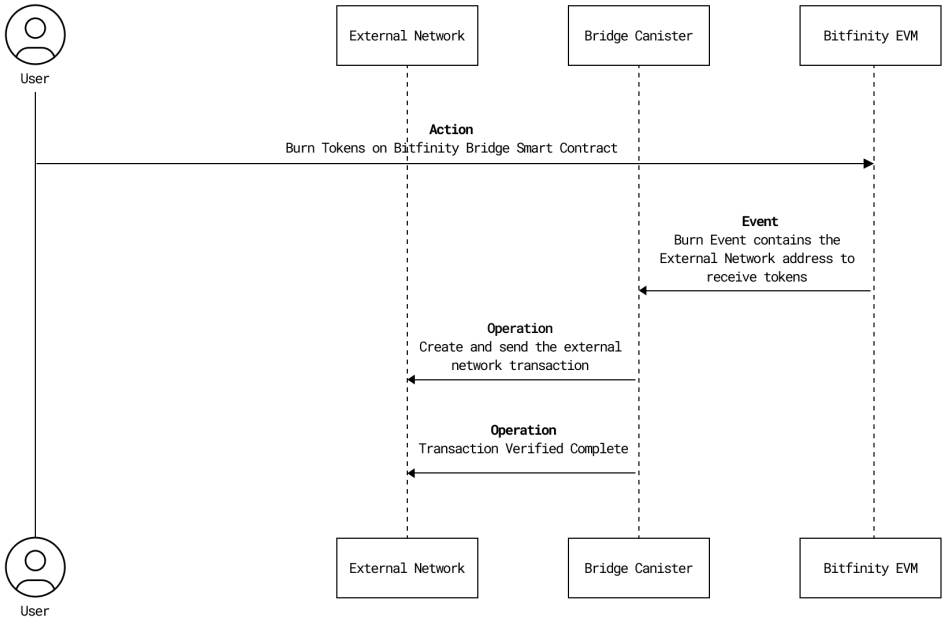


Figure 2: Overview of the default implemented flow for withdrawing base tokens by burning wrapped tokens

The sequence of operations form the figure, where the operations to issue and verify a transaction on the external network must be implemented by the downstream bridge, are described below.

1. **WrappedTokenBurned (Event)** Created by the EVM in a log when a wrapped token is burnt. Also implemented by default, the eventhandler save the burn information into the canister's stable

storage and schedules the transfer operation.

2. **Transfer Base Token (Op)**: The bridge canister issues a transfer of base tokens on the external token to the address saved within the burn information.
3. **Transaction Verified (Op)**. This operation verifies the transaction has either completed successfully or failed. In the case of failure, we can revert back to the prior stage to enable retries initiated by the user.

Having outlined the core components of the BitFusion framework, the following section describes the concrete implementation of BitFusion bridges across different blockchain ecosystems, including Bitcoin, Bitcoin Runes, Ethereum, and the Internet Computer (ICP). We will focus mainly on the deposit flow for these different networks, as the withdrawal flow is quite self explanatory and involves just implementing a way for the bridge to sign, send and verify transactions on each network.

4 Bridging Bitcoin

4.1 Background on ckBTC

Canisters can read the state of the Bitcoin blockchain through a network of Bitcoin nodes running on the Internet Computer protocol. Bitcoin that is wrapped on the Internet Computer is called Chain-Key Bitcoin. It makes use of the following canister components:

- Minter: `mqygn-kiaaa-aaaar-qaadq-cai`
- Ledger: `mxzaz-hqaaa-aaaar-qaada-cai`
- Bitcoin network adapter through management canister
- Threshold signing API through management canister

Blocks and transactions are retrieved and processed directly from the Bitcoin network. With the block data available through a network interface, the ckBTC Minter canister maintains the current Unspent Transaction Output (UTXO) set of the complete Bitcoin network. The UTXO information is offered to canisters via an interface on the minter, thereby enabling canisters to access information like the balance and UTXOs of Bitcoin addresses. When the ckBTC minter canister registers a transaction in BTC from or to a designated tECDSA Bitcoin wallet that it controls and reserved for a particular user, it mints ICRC tokens on the ckBTC Bitcoin ledger (a wrapped token) for the user in question.

The network uses threshold cryptography so that private keys are never entirely stored on any single node. The secret shares are held by nodes within a high-replication subnet. These secret shares are periodically reshared among the nodes to protect against a possible compromise. Once re-shared, the previously-valid shares become obsolete, rendering them useless to any malicious actor who might have obtained them. The subnet delegates ownership of specific Bitcoin addresses to specific canisters, which allows for these canisters, for instance the ckBTC minter to sign Bitcoin transactions.

These components ensure end-to-end decentralised security for bridging Bitcoin.

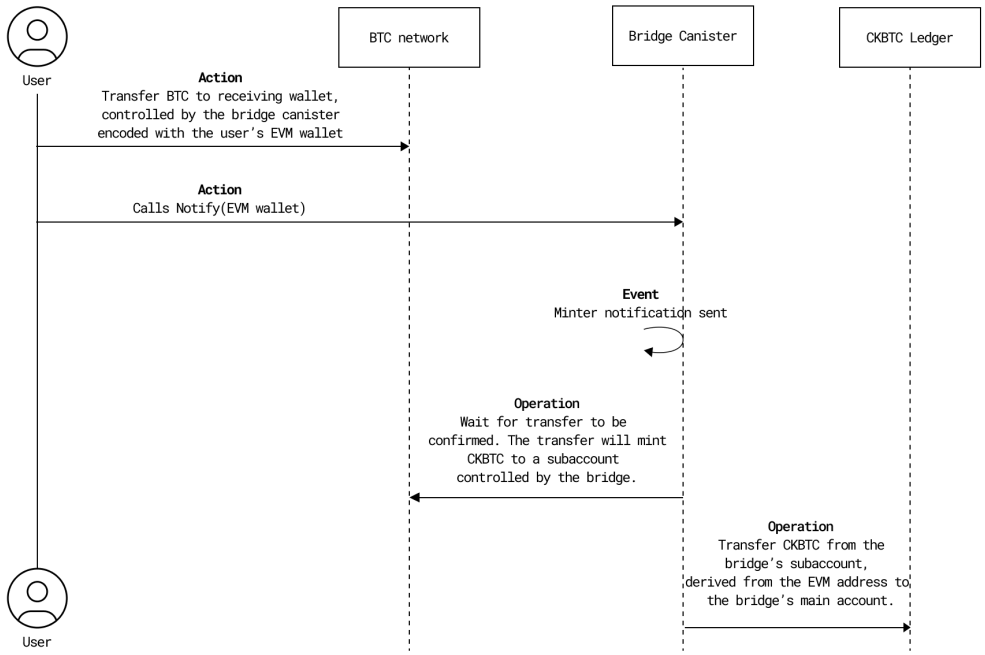
4.2 BitFusion Bitcoin Deposit Flow

BitFusion uses ckBTC to facilitate transfers of Bitcoins to the EVM. The deposit process starts by the user sending BTC to the BTC address of their deposit account. This deposit address is an encoding of the user's EVM address as a Bitcoin address, which the tECDSA signing network will recognise

as belonging to the ckBTC minter canister. More speificially the address is encoded based on the following function:

```
Address =
ckBTC_Minter_Map_to_BTC_Address[
    Account {
        owner: btc_bridge_canister_id,
        subaccount: wallet_eth_address_bytes
    }
]
// MaptoBTCAddress is a function on the ckBTC ledger that finds a designated
    Bitcoin address, given a Canister account identifier and subaccount.
```

After this action has been performed, the user calls the notifyMinter method on the BftBridge contract with the encoded EVM wallet address as an argument. The initial steps can be seen in figure 3 below, followed by the full sequence of operations.



...

Standard Flow Deposit

Figure 3: Overview of the Deposit Flow for Bitcoin

The sequence of operations after the user actions are described in detail in the following:

1. **Transfer Confirmed (Event)** The bridge will wait until enough confirmations are received on the Bitcoin network for the ckBTC minter to proceed with the mint.

2. **Mint CKBTC (Event)** On receiving the notification event, the BtcBridge periodically requests ckBTC minter to mint ckBTC into deposit subaccount, which is implemented as a custom operation.

3. **Transfer to Main Account (Event)**: After ckBTC tokens are minted, the BtcBridge canister transfers the tokens into its main account and creates a mint order for wrapped tokens.

4. **Continuation of Standard Flow** From here the standard flow to mint wrapped tokens on the EVM is undertaken.

4.3 Bitcoin Withdrawal Flow

To withdraw BTC from the Bitcoin network, we follow the template implementation described by 2. The withdrawal transfer operation is implemented by calling the appropriate method on the ckBTC ledger, which exposes an API that burns the ckBTC and sends the UTXOs to the designated user on the bitcoin network. The tECDSA subnet signs and spends a UTXO, on behalf of the ckBTC minter, which is then broadcast to the Bitcoin network through the adapter interface.

5 Bridging Runes

When bridging Runes tokens, the bridge canister communicates directly with the Bitcoin network through the Bitcoin Adapter to sign and sent transactions. We need to communicate directly with the Bitcoin network because Runes transactions must include certain metadata in the *op_return* flag of a Bitcoin transaction to be valid. To index Runes from the Bitcoin network, the bridge canister does not parse UTXOs on its own but relies on a separate, configurable Indexer canister to parse the Runes contents of the UTXOs. The Indexer canister can either query a list of off-chain indexers, considering a transaction as valid only when all indexers on the list agree on the Rune contents of a UTXO. However, the indexer also has the option to interface with an On-Chain Runes indexer, such as the one built by Omnity.

Through the unique technology at our disposal, Omnity has created a decentralized on-chain indexer for Runes. This was done by porting over the code for indexing a Runes transaction from the CK-ORD library and running it in a canister (a decentralized web-server). This canister has access to Bitcoin block data through the BTC adapter, which has been updated to store header hashes. You can find more information ont he implementation by Omnity here here.

Each EVM wallet is associated with an address on the Bitcoin network controlled by the bridge canister. The canister receives its master key from the IC ECDSA threshold signing mechanism and then uses the wallet address to generate a derivation path to derive a child key for each wallet.

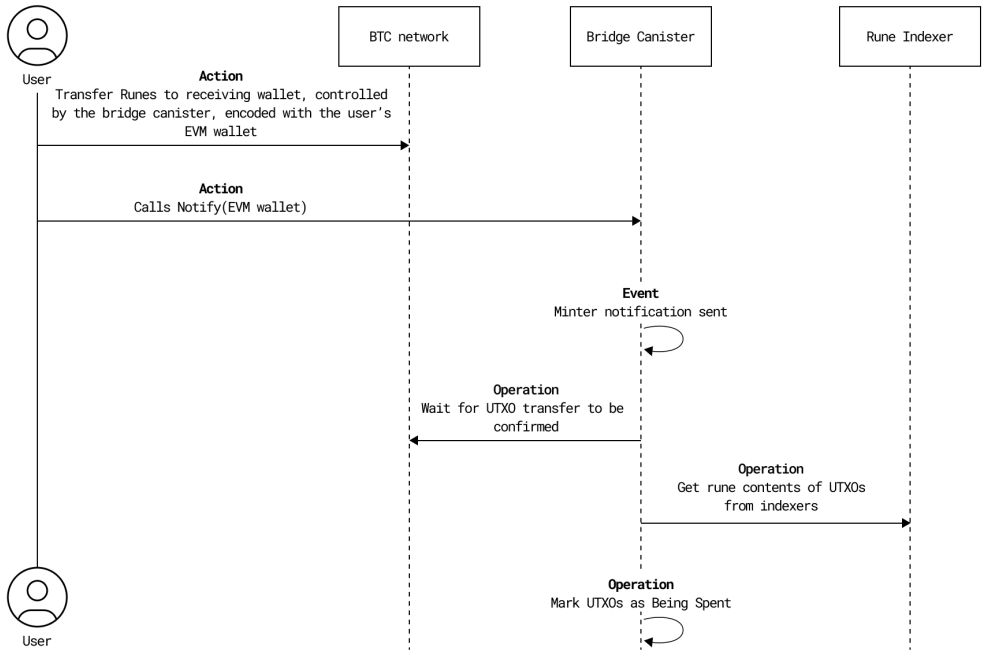
5.0.1 BitFusion Runes Deposit Flow

To deposit Runes tokens into the bridge, a user must issue a transaction containing Runes to the designated recipient address. The address is encoded as follows, with the only difference from the ckBTC encoding being that this address will be owned directly by the Runes Canister Bridge.

```
Address =
Bitcoin_Adapter_Map_to_BTC_Address[
    Account {
        owner: rune_bridge_canister_id,
        subaccount: wallet_eth_address_bytes
    }
```

```
]
// Bitcoin_Adapter_Map_to_BTC_Address is a function on the Bitcoin Adapter
    that creates a designated Bitcoin receiving address, given a Canister
    account identifier and subaccount.
```

Once the Runes have been sent to this recipient address, a user can invoke the notifyMinter method in the BftBridge contract to commence the sequence of briging operations. In figure 4 below we display the sequence of bridging operations.



...

Standard Flow Deposit

Figure 4: Overview of the Deposit Flow for ICRC tokens

Upon receiving the minter notification and saving the information on the transfer, the bridge canister will undertake the following sequence of operations.

1. **Waits for the UTXO to be Confirmed (Operation):** The bridge monitors the incoming UTXOs from the Bitcoin network, and waits until they are confirmed by the Bitcoin network.
2. **Retrieve Runes Contents from Indexers (Operation)** Once the transactions are confirmed, a call is made to the Rune indexing canister to retrieve the Runes contents for the transaction.
3. **Mark UTXOs as Spent (Operation)**: Once confirmation is reached, the UTXO ID is marked by the canister as spent/minted to prevent double spending in future deposit requests.

4. **Continuation of Standard Flow** From here we follow the standard flow to create the mint order and mint wrapped tokens on the Bitfinity EVM.

5.1 Runes Withdrawal Flow

The process to withdraw Runes is slightly more involved than for other flows. To withdraw Runes tokens from the bridge, the user must first provide fee payment in BTC to facilitate the creation of a withdrawal transaction. This is accomplished by sending a standard BTC transaction (no Runes) to the user's designated address. Once the transaction has been mined into a block, it becomes immediately available for the withdrawal process.

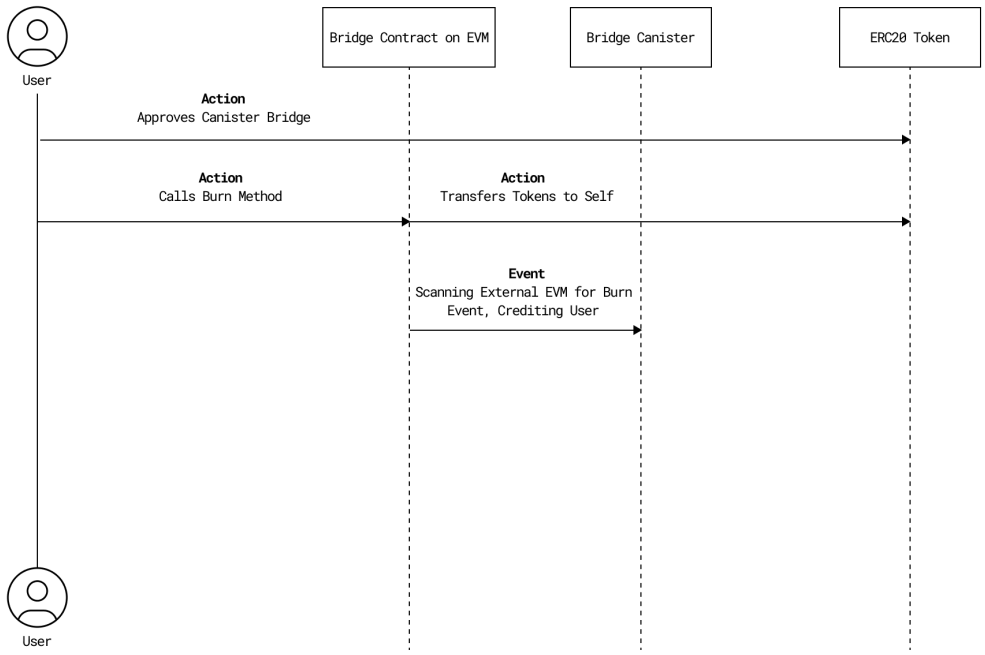
After the user initiates the burning of wrapped tokens, this will prompting the canister to create a Withdrawal. This transaction utilizes UTXOs containing Runes sufficient to cover the requested withdrawal amount, in addition to those provided as a fee by the user. Any unspent Runes are returned to the main address of the bridge canister, whilst the unspent BTC is sent back to the user's recipient address. See the figure below for an outline of the process.

To withdraw runes from the bridge, the user needs to provide BTC fee to create a withdrawal transaction. This is done by sending a regular BTC transaction to the user's designated address. After the transaction is mined into a block, it can be used by the withdrawal process right away. The user burns wrapped tokens and the canister creates a transaction, using as inputs enough UTXOs containing runes to cover the requested withdrawal amount, and the provided fee UTXOs. Leftover runes are returned to the main address of the bridge canister, and the leftover BTC are returned to the user's transit address.

6 Bridging ERC20 Tokens

Bridging to another EVM chain is relatively straightforward. To bridge ERC20 tokens from an external chain, a user must approve the BFT bridge contract deployed to the external chain to spend tokens. Then the user will call the lock method on the smart contract, which will trigger the bridge to lock the the tokens in escrow. Once this action has taken place, and the mint notification event is created and the flow proceeds as per the standard flow.

For the bridge to work in a decentralised manner, the BFT bridge canister must also have a way of querying the EVM with minimsied trust assumptions. One option is to make use of multiple RPC gateways (oracles) to the external EVM chain, ensuring all clients agree before bridging over funds. NB, this is not needed on Bitfinity because when querying from a canister, the query is authenticated through consensus. In fact, it is possible to query the Ethereum chain trustlessly through the Helios client, which is an Ethereum light client node implementation that has also been implemented as a canister. Helios can verify the proof of stake signatures authenticating blocks on Ethereum, and use these proofs to construct the hash headers on the blockchain. Once the proofs have been obtained, the client can query full nodes and authenticate this data against the header hash chain to ensure that the queries it receives are correct.



...

Standard Deposit Flow

Figure 5: Overview of the Deposit Flow for EVM tokens

6.1 External EVM Withdrawal Flow

To withdraw tokens to an EVM network, we follow the template implementation described by 2. The withdrawal transfer operation is implemented by issuing an ERC20 transfer of tokens and broadcasting this to the external network. This transaction can then verified either by a Helios client running on the network, or by querying oracles.

7 Account Abstraction: Future Directions

Bit-Fusion addresses many challenges in cross-chain interoperability, but it does not yet provide a near-instant bridging experience between chains. Achieving this requires higher-level primitives, like Chain-Abstraction. Chain-Abstraction introduces a higher-layer pseudo-transaction object called a UserOperation. Users send UserOperation objects into a separate mempool, where bundlers package these objects into a single transaction by calling a special contract. This transaction is then included in a block.

Account abstraction transitions from the current model of externally owned accounts (EOAs) to smart contract wallets with arbitrary verification logic. This shift improves wallet designs and reduces complexity for end users.

A smart-wallet secured by a threshold signature scheme can initiate a bridging transaction through a "Relay Pool." Users interact with this pool, where relayers with pre-bridged tokens send the user their bridged assets. In return, the relayer receives funds from the smart-wallet. This method allows for near-instantaneous cross-chain transfers without waiting for finality on the originating chain.For a detailed explanation, refer to the video here. Since Bit-Fusion is a bridge framework, relayers can use it to transfer their assets, making the proposed solution to instantaneous bridge transfers integratable with Bit-Fusion.

In conclusion, Bit-Fusion lays the groundwork for secure and decentralized bridging. Future directions involving Chain Abstraction and smart-wallets promise to enhance speed and user experience. These innovations can work in conjunction with Bit-Fusion's robust and flexible bridging framework.

8 BitFusion - A Modular and Extensible Bridge Framework for Bitcoin

The Bit-Fusion framework significantly advances the integration of the Bitcoin ecosystem with EVM networks by ensuring secure and decentralized asset transfers. Through the use of Chain-Key cryptography, a robust threshold Elliptic Curve Digital Signature Algorithm (tECDSA), and by directly interfacing with a network of nodes on the Bitcoin network, Bit-Fusion maintains the high security standards essential for Bitcoin. BitFusion closes the gap between the security of bridging solutions and proof-of-stake blockchain.

Bit-Fusion's use of a decentralized Runes indexer, deployed to a canister, in the bridging flow enhances security and reliability when introducing new tokens on the Bitcoin blockchain. It solves an open question: how can decentralised applications interface with these new tokens on Bitcoin with a high degree of security.

Bit-Fusion's modular and extensible architecture offers a high degree of flexibility, allowing developers to create novel bridge applications. Demonstrations of three implementations in this paper highlight the framework's potential, and future innovations are anticipated as the community builds upon it.

In summary, Bit-Fusion establishes a new benchmark for cross-chain interoperability, providing a robust, secure, and adaptable solution for bridging assets within the Bitcoin ecosystem and beyond.

References

[But21] Vitalik Buterin. Erc-4337: Account abstraction using alt mempool. https://eips.ethereum.org/EIPS/eip-4337, 2021.

[Dfi22] Dfinity. The internet computer for geeks. https://internetcomputer.org/whitepaper.pdf, 2022.

[Dfi23] Dfinity. Threshold ecdsa primer. https://internetcomputer.org/docs/current/references/t-ecdsa-how-it-works, 2023.

[Fou] Ethereum Foundation. Scaling zk rollups. https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[JG22] Victor Shoup Jens Groth. On the security of ecdsa with additive key derivation and presignatures. https://eprint.iacr.org/2021/1330.pdf, 2022.

[Pas22] Mario Pastorelli. Icrc1 and icrc2 github readme. https://github.com/dfinity/ICRC-1/tree/main/standards, 2022.

[Roy23] Uma Roy. Are intents, suave, account abstraction, cross-chain bridging all the same thing? https://www.youtube.com/watch?v=G0nFyq9DDPw, 2023.

[VB22] Dankrad Feist Vitalik Buterin. Danksharding - eip4844. https://eips.ethereum.org/EIPS/eip-4844, 2022.

[Vog15] Fabian Vogelsteller. Erc20: Token standard. https://eips.ethereum.org/EIPS/eip-20, 2015.

[Woo24] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger paris version 71beac3. https://ethereum.github.io/yellowpaper/paper.pdf, 2024.