Bitfinity Network: Asynchronous Tokens Enabling Sharding and Near-Native Bridge UX

Bitfinity

June 2024

Abstract

The Bitfinity Network is a new paradigm for Ethereum Virtual Machine (EVM) compute which includes EVM processors and asynchronous tokens, as part of a sharded blockchain network. By running on the Internet Computer, which uses fast-finality BLS threshold signatures to come to consensus, the Bitfinity EVM inherits high throughput and scalability. A single EVM processor can run hundreds of transactions per second and reach finality in one to two seconds.

Bitfinity introduces a novel computing paradigm for the EVM by allowing tokens to be deployed independently of the EVM, enabling greater parallelization and speed, essentially implementing a sharded blockchain. Bitfinity also provides a near-native UX for cross-chain bridging, along with new primitives to improve the security of cross-chain bridges. Through Bit-Fusion and Chain-key Cryptography, Bitfinity functions as a BTC L2 and supports bridging for many EVM chains, covered in detail in the Bit-Fusion whitepaper.

Contents

1 Scaling the EVM through Sharding

In the discourse on enhancing the Ethereum Virtual Machine's (EVM) scalability, it's crucial to examine different strategies for parallelization. The Bitfinity platform has adopted a distinctive approach towards this goal through sharding [1]. While traditional EVM-based blockchains operate on a purely synchronous model, which inherently limits their scalability to the capacity of a single thread of execution, other blockchain architectures, such as the Internet Computer, facilitate asynchronous operation of each smart contract—thus allowing them to function independently. Several other blockchain platforms, including SUI[2], Aptos, and Solana [3], focus on parallelizing computation at the processor level—thereby greatly enhancing scalability but still limiting it to the number of processors available on a single machine.

Bitfinity introduces an innovative method to scale, which balances fully asynchronous smart contracts with the benefits and safety of the synchronous EVM. The core of this approach involves treating tokens as independent units of concurrency. Under this model, tokens operate as fully independent actors realized as canisters (smart contracts on the Internet Computer) and are bridged just-in-time to EVM canisters, where transactions involving these tokens are processed synchronously, effectively implementing sharding.

This hybrid model leverages the asynchronous actor model [4] compute paradigm to enhance scalability without departing from the proven security and stability of the synchronous EVM architecture.

This approach represents a significant shift in how blockchain technologies manage concurrency and scalability, potentially setting a new precedent. Bitfinity is a linearly scalable network of EVMs. The rationale behind this assertion lies in the method by which processing capacity can be increased. Instead of relying on the finite number of cores within a single machine, which limits the extent to which parallel processing can be executed, Bitfinity's model enables scaling by simply adding more EVM instances. Each instance can run decentralized applications (Dapps) independently, thus effectively distributing the workload across multiple EVMs. This facilitates a significant improvement over traditional parallel processing methods in blockchain technologies.

In summary, Bitfinity's approach to sharding, which involves treating tokens as independent units of concurrency and processing them within a scalable network of EVM instances, offers a promising solution to the scalability challenges faced by traditional EVM-based blockchains. This innovative method enhances the efficiency and capacity of blockchain systems, paving the way for more robust and scalable decentralized applications.

[1] DankSharding - [VB22]
[2] Sui whitepaper, sections on Move and Narwhal and Tusk consensus - [laba]
[3] Solana white-paper, section on Parallel Merkle Trie - [Yak]
[4] Internet Computer Blockchain, section on actor model - [Dfi22]

2 Network Architecture and Assumptions

2.1 Partially Synchronous Communication Model

Bitfinity, through the IC [5], employs a partially synchronous communication model to achieve Byzantine fault tolerance, accommodating up to $f < n/3$ faulty nodes. This model is a practical compromise commonly used in Byzantine fault tolerant systems that do not depend on synchronous communication, as seen in other systems (e.g., [CL99, BKM18, YMR+18]). The partial synchrony allows for certain periods where delays are known but unbounded ([DLS88]), unlike purely asynchronous models where delays can be indefinite, often making effective consensus challenging.

In this model, the assumption of partial synchrony specifies that communication between replicas in a subnet experiences periods of synchrony interspersed with asynchronous communication. Specifically, the system assumes that there are predictable but finite intervals during which messages are delivered, a notion less stringent than the indefinite delays characterized in fully asynchronous systems (as discussed in [MXC+16]). This temporary synchrony is essential for achieving progress in consensus protocols (ensuring the so-called liveness property of the system), but it is not required for maintaining the correctness of consensus (the safety property) or for other aspects of the protocol stack.

2.2 Cross-Subnet Communication

The Internet Computer employs a sophisticated architecture that enables efficient and secure message processing across its distributed framework. Each subnet within the IC operates autonomously, processing messages independently. This decentralized approach not only facilitates robust parallel processing but also enhances the system's overall scalability and efficiency.

Central to the IC's security infrastructure are threshold signatures [6], a well-established cryptographic technique. The secret signing key of a subnet is divided into shares and distributed among all replicas within the subnet. This arrangement ensures that corrupt replicas cannot forge signatures, while honest replicas can generate valid signatures that align with the policies and protocols of the IC. This method of authentication allows individual outputs of one subnet to be verified by another subnet or external users by simply validating a digital signature against the public signature-verification key of the originating subnet. Notably, the public signature-verification key, obtainable from the Network Nervous System (NNS), remains constant throughout the lifetime of a subnet, even as its membership may change.

This robust mechanism is supported by an innovative distributed key generation (DKG) protocol that constructs a public signature-verification key and provisions each replica with a share of the corresponding secret signing key. The protocol is designed to operate effectively within the IC's fault and communication model, which assumes a degree of asynchronicity and potential Byzantine faults.

Furthermore, the IC implements proactive resharing of secrets at the beginning of each epoch. This process ensures that any new members in a subnet receive an appropriate share of the secret, while replicas that are no longer members lose access to their shares. This approach not only adapts to changes in subnet composition but also guards against potential security breaches from leaked shares, as reshared secrets render old shares obsolete.

2.3 Execution of messages under the Actor Model

In the realm of execution, the IC [7] processes each input in a sequential manner, directed to specific canisters based on the input queue. This methodical processing updates the state of the canister and may trigger further actions, such as adding messages to output queues or updating the ingress history with responses to previous messages. Each subnet accesses a distributed pseudorandom generator, deriving unpredictable bits from the threshold signatures. These pseudorandom bits are essential for the random beacon used in consensus mechanisms and the random tape used in execution, ensuring that the operations remain secure and unpredictable.

Cross-canister calls are modeled on an actor framework, guaranteeing that every message will receive a response. This system design not only facilitates smooth inter-canister communication across

[5] Internet Computer - [Dfi22]
[6] Threshold ECDSA - [JG22], [Dfi23]
[7] Internet COmputer, section on execution - [Dfi22]

subnets but also ensures that message authenticity and responses can be verified efficiently using BLS signatures. The architectural sophistication of the IC, coupled with its strategic implementation of threshold signatures and proactive security measures, provides a robust platform for deploying decentralized applications, maintaining high throughput, and ensuring system integrity and reliability.

3 The Bitfinity Network

3.1 Asynchronous actions for tokens and the EVM

The tokens on the Internet Computer (IC) adhere to the ICRC-1 and ICRC-2 standards [8], which incorporate methods such as transfer, approve, and transferFrom. These methods are modeled on the Ethereum standard but are adapted for asynchronous environments. As of the current implementation, up to 500 concurrent invocations of these methods can be sent to a canister, with processing completed within a few seconds. These three methods form the backbone of the just-in-time bridging mechanism employed by the Bitfinity EVM.

3.1.1 Just-in-Time Bridging

The concept of just-in-time bridging is central to the Bitfinity Network's operation. Due to the efficient processing times on the IC — where transfers are typically completed within a few seconds - bridging asynchronous ICRC-2 tokens to ERC20 [9] tokens on an EVM processor. The bridging process is mediated by a specialized bridging canister that facilitates communication between the EVM and tokens.

When a transfer is initiated on the IC, tokens are temporarily held by a bridge actor. This canister is responsible for minting the corresponding tokens as ERC-20 tokens on the EVM. This mechanism ensures interoperability between the IC's asynchronous ICRC-2 tokens and Bitfinity ERC20 tokens, whilst not impeding the scalability and independent operations of a single ICRC-2 canister.

3.1.2 Detailed Workflow

The workflow for bridging ICRC-2 tokens to the EVM involves several critical steps:

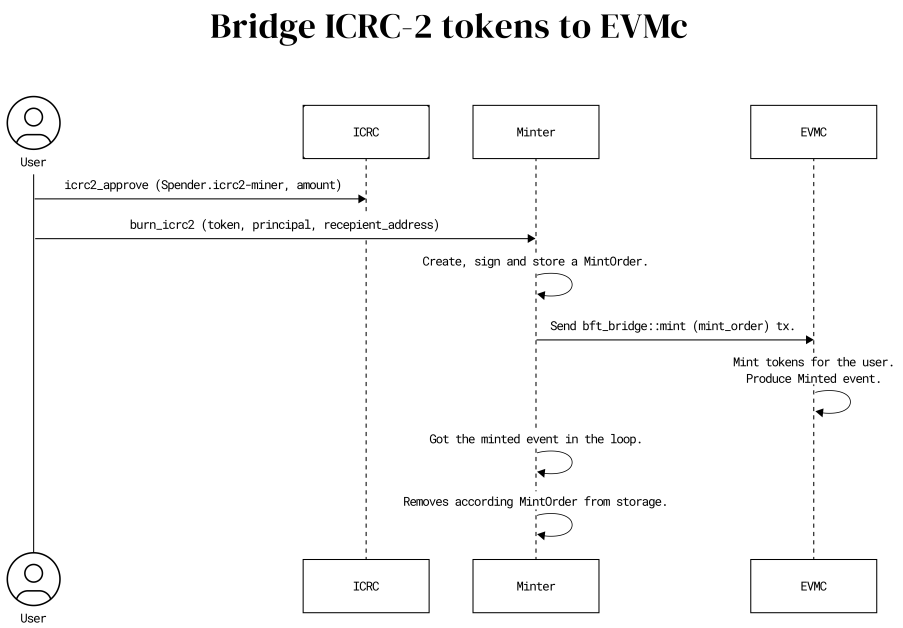

Figure 1: ICRC to EVM Flow

[8] ICRC standard proposals - [Pas22]
[9] ERC20 standard - [Vog15]

1. Prerequisites: Prior to bridging, BFT tokens must be paid to a Fee smart-contract
2. Approval: The user performs an approval operation for ICRC-2 token transfer, specifying the Minter canister principal as the spender parameter. The spender account is:

```
{
    owner = bridge_canister_principal,
    subaccount = receiver_eth_address . [0; 8] }
}
```

3. The user calls the notifyMinter method on the bft bridge smart-contract with deposit parameters (the encoded EVM address)
4. Mint Order Preparation: The Minter canister prepares a MintOrder based on the burn operation and sends it to the BFTBridge::mint() contract function on the EVM.
5. Mint Transaction Handling: If the mint transaction fails, the user can manually retrieve and resend the MintOrder to the BFTBridge::mint() endpoint. Once the transaction succeeds, a Minted event is created. If the mint transaction succeeds, fee tokens are burned.
6. Event Querying and Cleanup: The Minter canister queries the Minted event and removes the MintOrder from its memory.

Conversely, the workflow for bridging EVM tokens to ICRC-2 follows these steps:

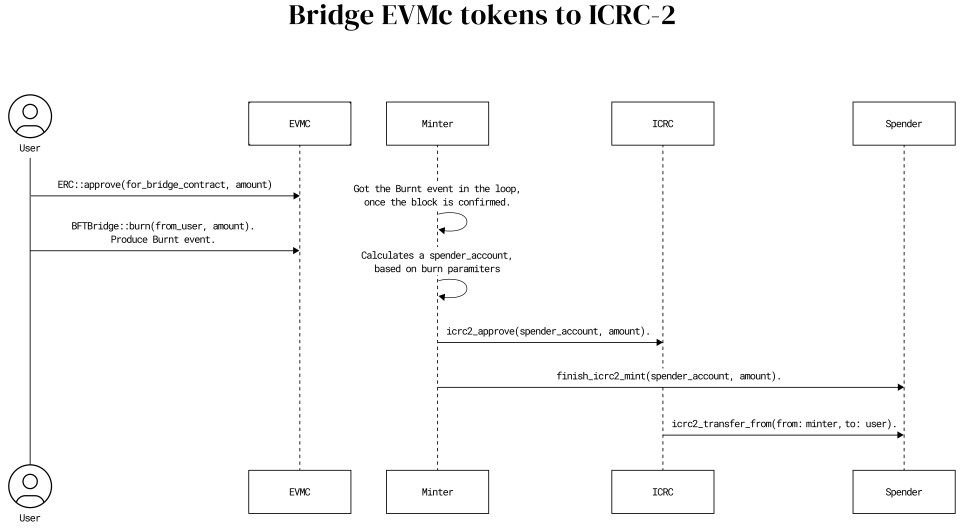

Figure 2: ICRC to EVM Flow

1. ERC-20 Approval: The user approves ERC-20 token transfer, specifying the BFTBridge address as the spender.
2. Burn Operation: The user performs the transfer using the BFTBridge::burn() function. Upon token burn, the BFTBridge contract creates a Burnt event.
3. ICRC-2 Minting: The Minter canister approves ICRC-2 token transfer to a special subaccount of the Spender canister, calculated uniquely for each burn operation.
4. Final Transfer: The Minter canister calls spender::finishicrc2mint(), prompting the Spender canister to perform the icrc2transferFrom() call, sending the tokens to the user.

3.1.3 Architectural Advantages

By leveraging the asynchronous actor model, realized as canisters on the Internet Computer (IC), the Bitfinity Network achieves linear scalability for both tokens and EVM processors. The integration of bridge canisters facilitates efficient and secure communication between EVM canisters and tokens, significantly enhancing the network's overall scalability. The fast finality of ICRC-2 token transfers supports just-in-time bridging, enabling the seamless use of tokens in decentralized applications (dApps) with minimal user friction. Additionally, the framework is extensible to tokens from other cross-chain networks, a concept explored in detail in the BitFusion paper. Designed with cross-chain interoperability in mind, the Bitfinity Network can seamlessly integrate various token standards, thereby broadening its utility and application.

In summary, the Bitfinity Network's adoption of the ICRC-1 and ICRC-2 standards, combined with its innovative just-in-time bridging mechanism, represents a sophisticated approach to scaling blockchain technology. This architecture not only optimizes performance and scalability but also ensures a high degree of interoperability across different blockchain ecosystems, while maintaining the tried-and-tested security of the EVM.

4 On-Chain Components

4.1 Introduction

The Bitfinity EVM architecture leverages the scalability and performance capabilities of the Internet Computer (IC) while maintaining compatibility with existing Ethereum infrastructure. This architecture comprises several critical components that work together to process Ethereum transactions efficiently, as displayed in figure 3. The entire flow for processing transactions end-to-end is also detailed in figure 4.

On-Chain Components

- EVM Core Canister
- Signature Verification Canister

Off-Chain Components

- Block Extractor
- Adapted Reth Node
- EVM Archiver

At the core of this architecture is the EVM canister, which serves as the primary entry point for the system and hosts the JSON RPC API through the canister's HTTP interface. This direct hosting eliminates the need for intermediaries like Infura. The EVM canister receives transaction requests from Ethereum clients, such as Metamask wallets, and IC agents. Upon receiving these requests, the canister pushes transactions into a pending transactions pool. Transactions are then batched and sequentially passed to an REVM-based executor, which processes them and updates the blockchain state. The blockchain state is maintained within a stable-storage data structure, using an adapted version of Parity's Merkle Trie [10], tailored for IC canister storage. Additionally, a limited history of blockchain data is retained on this canister and is accessible via the Ethereum JSON RPC API.

A separate canister is dedicated to signature verification. Before a transaction is added to the pool, the Signature Verification canister verifies the correctness of Ethereum signatures. Given that signature verification is computationally intensive and requires substantial CPU cycles, this task is offloaded to a dedicated canister. This separation of duties optimizes performance and ensures that the main EVM canister remains unburdened by heavy computations.

This architectural approach allows the Bitfinity EVM to achieve high efficiency and scalability, supporting robust transaction processing while keeping the on-chain data footprint low. It reduces

[10] Parity Technology - Merkle Trie Repo

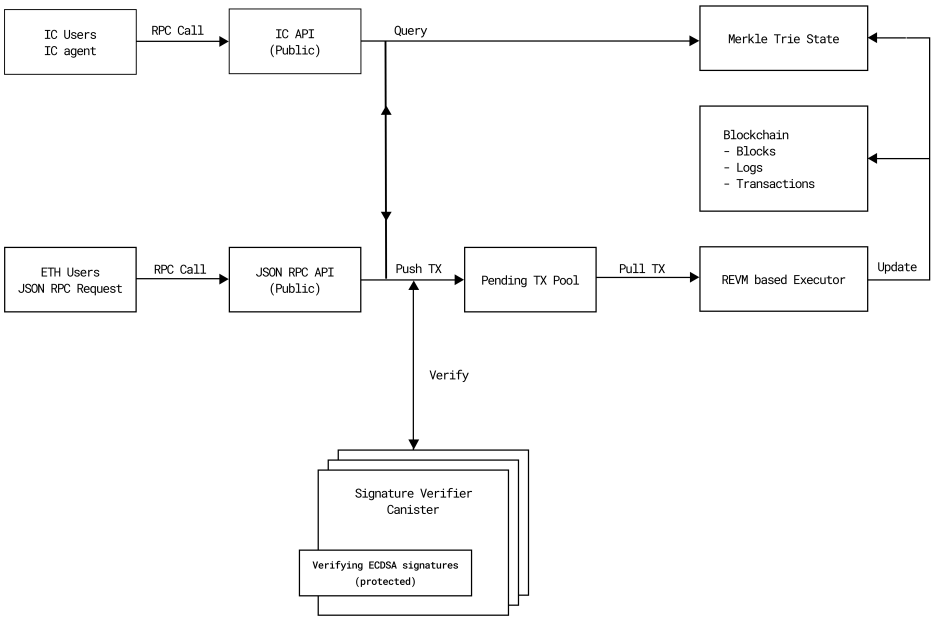

Figure 3: Bitfinity EVM Canister Architecture

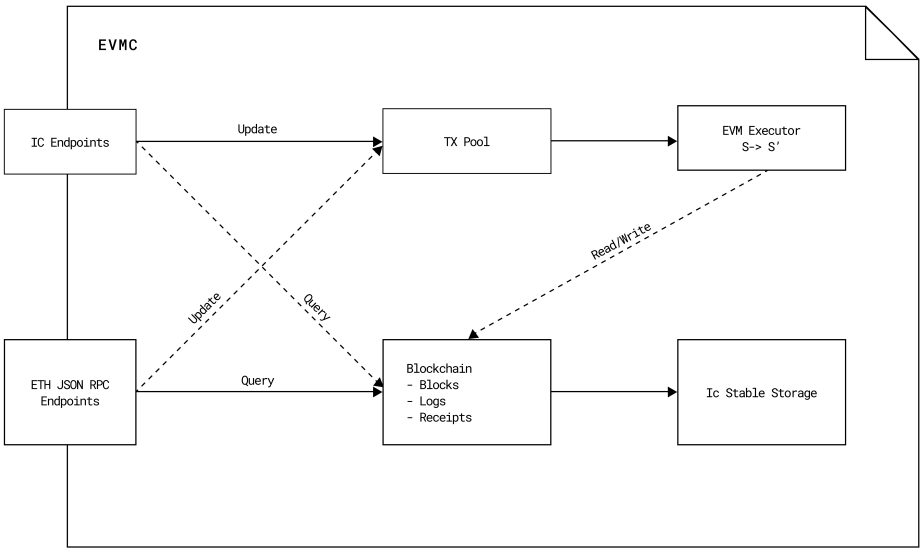

Figure 4: Request Flow Diagram

the need to explicitly run your own EVM nodes or RPC providers, as required in traditional EVM deployments, and enables seamless integration with the broader Ethereum ecosystem.

In the next sections, we outline the key on-chain components of the EVM architecture.

4.2 EVM Canister

The EVM canister implements all the Ethereum JSON RPC methods required by a fully compatible EVM node (but with a limited history of on-chain block data) and maintains the internal state of the blockchain, authenticated through a Patricia-Merkle Trie.

4.2.1 EVM Executor

The EVM executor is an implementation of the Ethereum Virtual Machine running inside the canister. It consists of code to execute Ethereum transactions, generate blocks and update the state DB, and is based on REVM[11]. The executor interfaces with a modified version of Parity's Patricia-Merkle Trie implementation which updates the canister storage (WASM-page based storage accessible through the "stable-storage" API as a backend to store the state and Merkle paths to the state.

4.2.2 Ethereum JSON RPC API

This is a full implementation of the Ethereum JSON RPC specification that allows the EVM API to receive requests from external Ethereum libraries (e.g. web.js), wallets (e.g. Metamask), and other clients.

Every time a call is received, the endpoint decodes its content and translates it into a "query" [non-state changing] or an "update" [state-changing] to be dispatched to the EVM executor. Every Ethereum JSON RPC method defined in the specification has a corresponding IC endpoint for use by IC-agents. Those endpoints allow interaction with the EVM blockchain from canisters and other clients on the Internet Computer, using the standard cross-canister communication methods. Before an IC canister can communicate with the EVM it has to register a public key[12] or alternatively use the threshold ECDSA API [13].

4.2.3 Blockchain Data

The Blockchain Data is a data structure that contains the most recent blocks. Our implementation stores the most recent 2 weeks of block data, and persists this data to stable-storage. It would be infeasible to store all the blockchain data on-chain, as this could run into terabytes of storage.

4.2.4 Transaction Pool

Transaction requests are enqueued in a persistent transaction Pool within the EVM canister. The transactions are then prioritized based on the gas fees and passed from the pool to the executor in the expected order.

Where possible, mature existing external libraries are used to make the best use of existing Ethereum libraries. Among the others, Bitfinity uses:

4.2.5 Signature Verification Canister

The Ethereum signatures on transactions are verified through the use of a dedicated canister. We use a separate canister for verifying signatures to optimise transaction throughput by performing these actions in a dedicated canister, as this action is relatively computationally expensive and could block the main thread of the EVM canister as it executes transactions. Due to the manner in which query calls are verified (verifying a signature involves no state-updates on the signature verification canister), signature verification can be parallelised through multiple concurrent calls, making this decomposition particularly efficient.

[11] A fast and flexible Rust implementation of the EVM. Refer to 4.3.2 for more details.
[12] See the IC agent docs
[13] For more details on tECDSA see: [Dfi23]

4.3 Open-Source Dependencies

In building the EVM, we have attempted as far as possible to reuse useful libraries implemented by others. Here we outline some of the useful open-source libraries that have been used.

4.3.1 Parity-tech's Patricia Merkle Trie

We have used an adapted version of Parity-Tech's Patricia Merkle Trie, which interfaces with a database that interfaces with the stable-storage APIs on the Internet Computer, and ultimately let us save data to WASM-pages, which is the backbone of the canister storage.

4.3.2 REVM as an EVM processor

REVM is a modern EVM written in Rust that is focused on speed, simplicity, compatibility, and stability. REVM is a relatively young project but it passes all the Ethereum Client Tests and it performs nearly as twice well as GETH, another reference EVM implementation.

4.3.3 Bitfinity's Canister SDK

The Bitfinity EVM is built on top of the company's canister-sdk and the IC's stable-storage libraries for effective state management on the Internet Computer. The canister-SDK provides a streamlined framework for developing canisters, which includes many useful features like unit-testing inter-canister calls, trait-based inheritance as well as libraries for using useful data-structures that uses WASM-pages as the underlying storage.

4.3.4 Foundry

Foundry is a fast, portable, and modular toolkit for Ethereum application development written in Rust, which provides a node, testing framework, and various EVM utilities and types. Bitfinity uses Foundry for testing.

5 Off-Chain Components

In addition to the on-chain components, several off-chain software applications are used to ensure data redundancy and accessibility for the full API.

A block extractor indexes and stores all block data, operating as a Docker daemon and processing blocks to store them on a cloud provider. A modified Reth node ingests blocks from the EVM's HTTP interface rather than a typical P2P network. The Reth node maintains a full history of the EVM data, serves the full JSON RPC API, and can forward transactions to the EVM canister for processing. Additionally, an EVM-archiver synchronizes with the full state of the EVM canister. Unlike the Reth node, the EVM-archiver operates in an environment similar to the EVM running on a canister, using MMapped files similar to the IC's stable storage with WASM pages.

All of these deployments can be deployed together through the use of infrastructure-as-code tools: Bitfinity relies on Terrafrom to automate its deployments.

5.0.1 Block Extractor

The Block Extractor is a crucial off-chain component designed to index and store all blockchain data efficiently, given the limitations to storing all this data on chain. Operating as a Docker daemon, the Block Extractor continuously polls the Bitfinity EVM, extracting and processing blocks in real-time. This process involves parsing the blockchain data and storing it on a database in a cloud provider, ensuring high availability and redundancy. By maintaining a comprehensive and up-to-date index of all block data, the Block Extractor supports querying for blocks through the JSON RPC API, enabling quick and reliable access to historical data.

5.0.2 Reth Node

The Reth Node is an adapted full node that ingests blocks from the EVM's HTTP interface or a block-extractor, diverging from the typical peer-to-peer network approach. This modified Reth node maintains a complete history of EVM data, ensuring that all transactional information is preserved and accessible. By leveraging the Reth framework, the node provides a full implementation of the Ethereum JSON RPC API, facilitating seamless interaction with Ethereum-based tools and applications. Additionally, the Reth Node can forward transactions to the EVM canister for processing, integrating tightly with a Bitfinity network EVM. Detailed instructions for building and running the Reth Node can be found in the Bitfinity Developer Documentation [14].

Developer documentation to build and run a node can be found on GitHub and in the Bitfinity Developer Documentation Website.

5.0.3 EVM Archiver

Unlike the Reth node, the EVM Archiver replicates an environment similar to the EVM on a canister, using memory-mapped (MMapped) files akin to the Internet Computer's stable storage with WebAssembly (WASM) pages. This setup allows for accurate local debugging and detailed examination of data structures and values from the main network.

6 Performance and Competitive Analysis

The EVM runs as a canister inside the Internet Computer blockchain and it is fully on-chain, having both code and data executed in web-assembly. As WASM is single-threaded, parallel computation on the EVM is achieved through a multi-canister architecture. Unlike Ethereum which is completely synchronous, the Internet Computer can increase transaction throughput by running distributed transactions in parallel.

In our EVM, we optimise transaction throughput by performing signature verification on a dedicated canister. The performance of a single EVM has been benchmarked between 250 TPS and 1000 TPS, depending on configuration.

Several Ethereum Virtual Machines (EVMs), such as SEI [15], have adopted parallel processing, which can achieve speeds up to five times faster than typical EVM implementations. This performance improvement is partly limited by the fact that many transactions involve the same state, often due to the design of popular tokens. For example, a widely used token like wETH might be involved in approximately 10% of all transactions, necessitating sequential processing for these cases. Thus, parallelism can offer a performance boost of up to tenfold.

To achieve further enhancements, one could redesign token structures to reduce interdependent states, as seen in platforms like Solana, Sui, and Aptos. However, this approach risks losing the inherent advantages of the EVM model.

A more practical solution involves running multiple EVMs concurrently, complemented by just-in-time token bridging. This strategy offers a scalable solution with linear and potentially unlimited growth.

Delivering a satisfactory user experience (UX) in this model is both feasible and promising, given recent work on intents and cross-chain bridging. [16] Building on this, implementing wallets that track tokens across multiple EVMs, along with leveraging account abstraction, can ensure seamless token usage across different EVM processors. Briefly, the idea is that a network of liquidity providers can be coupled with smart wallets (for instance, relying on a threshold signature scheme and a locking mechanism to prevent the double spend problem) can bring instantaneous liquidity cross-chain without having to necessarily wait for bridging operations and finality. These advancements, which we will generically terms as account-abstraction extensions, can be built on top of secure bridging primitives as covered in the Bit-Fusion whitepaper. Additionally, this approach presents a viable alternative to scaling a single EVM instance, offering a more scalable and flexible solution.

[14] Bitfinity Developer Documentation
[15] See Sei whitepaper: [Labb]
[16] Solving the Cross-chain UX issue with account abstraction [Roy23]

6.1 EVM equivalence and applicable tooling

The Bitfinity EVM implements the Ethereum JSON-RPC API. This is a full implementation of the Ethereum JSON-RPC specifications that allows the EVM API to receive requests from external Ethereum libraries (e.g. web.js), wallets (e.g. Metamask), and other clients. Every time a call is received, the endpoint decodes its content and translates it into a query or an update to be dispatched to the EVM executor. By supporting the Ethereum JSON-RPC API, Bitfinity EVM allows developers to use existing Ethereum tools and DApps (Decentralized Applications) without significant modifications. This compatibility simplifies the transition for projects migrating from Ethereum to Bitfinity, enabling them to leverage the robust Ethereum ecosystem. Developers can utilize a wide array of tools that are already available for Ethereum, such as Truffle, Remix, and MetaMask. This availability helps in accelerating development cycles, as developers do not need to learn new interfaces or write new integration code for these tools.

For comprehensive details on integrating and using Bitfinity EVM with Ethereum tools such as MetaMask, please refer to the official documentation provided by Bitfinity [17].

6.2 Security and Testing

6.2.1 Introduction and description of Tests

The code is well-tested, and we aim for high coverage across the core modules. By our estimates, we achieve a high coverage of almost 95% of the code-base.

The EVM canister successfully passes all the Ethereum common client tests [18], currently the gold-standard test-suite for Ethereum - which includes EVM execution, state tests.

We also leverage the Parity Tests and tests from a pre-existing test suite for our adapted Patricia Merkle-Trie implementation.

6.2.2 Audited - Quantstamp

An initial audit by Quantstamp [19] identified 14 high to medium severity issues in the Bitfinity EVM Canister. The development team responded promptly and effectively, resolving all issues. As of November 23, 2023, Quantstamp has verified that all high to medium severity findings have been addressed, with statuses of fixed, acknowledged, or mitigated. Fuzz tests conducted on key components showed no system crashes.

6.3 DOS - Attack Vectors and Mitigation

The Internet Computer (IC) supports three types of endpoints for its methods: query, update, and certified queries. Query calls to canisters are non-replicated, with responses issued by a single boundary node. Update calls pass through consensus, achieving network certification with a latency of a few seconds. Certified queries are replicated by multiple boundary nodes, offering reduced latency compared to update calls.

Currently, the IC's canisters can handle hundreds of update calls per second, and the network can manage virtually limitless numbers of query and certified query calls. The IC also has anti-DOS mitigation measures for query calls. Consequently, the bottleneck in our system lies in update calls to the JSON RPC interface.

We utilize the canister's inspect message, which incurs no fee to the canister, to prevent users without sufficient balances from calling any state-changing endpoints on the EVM. Additionally, all state-changing endpoints, such as submitting transactions, must go through the transaction pool, ensuring users are charged the corresponding fee.

7 Summary and Conclusion

The Bitfinity Network has successfully implemented a sharded blockchain leveraging asynchronous tokens and EVM processors, marking a significant advancement in blockchain technology. By treating

[17] Getting Started with Bitfinity
[18] see: ethereum/tests
[19] See Quantstamp

tokens as independent units of concurrency and utilizing just-in-time bridging mechanisms, Bitfinity enables rapid and efficient transaction processing. Our innovative architecture supports high throughput with transaction finality in just 1-2 seconds, setting a new benchmark for performance in sharded blockchains. The fast-finality of transfers ensures that just-in-time bridging does not impede user experience (UX).

Furthermore, we have addressed user experience (UX) challenges associated with sharded systems through the integration of smart wallets and account abstractions, ensuring seamless token usage across multiple EVM instances. This provides an intuitive and efficient user experience. The extensibility of our system allows for interoperability with other blockchain ecosystems, broadening its utility and adaptability. Just-in-time bridging can be extended to tokens from other ecosystems, such as Runes, BTC, or Ethereum, albeit with longer delays for cross-chain transactions.

Bitfinity stands at the forefront of blockchain innovation, offering a scalable, efficient, and user-friendly solution that sets new standards in the industry.

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.

[Kon22] Georgios Konstantopoulos. Introducing reth. https://www.paradigm.xyz/2022/12/reth, 2022.

[laba] Mysten labs. The sui smart contracts platform. https://docs.sui.io/paper/sui.pdf.

[Labb] Sei Labs. Sei: The layer 1 for trading. https://github.com/sei-protocol/sei-chain/blob/main/whitepaper/Sei_Whitepaper.pdf.

[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.

[SB] Evan Cheng Sam Blackshear. Move: A language with programmable resources. https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf.

[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.

[Yak] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. https://solana.com/solana-whitepaper.pdf.