Fluence Core: An Efficient Trustless Computation Platform

[WIP]

Alexander Demidko, Michael Voronov, and Alex Pyshnenko

Fluence Labs

Contents

1 Introduction

With the growing number of decentralized applications a need in cost-efficient, fast, and secure data processing, indexing, and querying becomes remarkably evident.

In the centralized environment it is a solved problem to process or query data even when there is a lot of it. For example, distributed streaming platforms such as Kafka [1], distributed computing frameworks such as Spark [2] [3], or distributed data stores such as Druid [4] allow to ingest, process and query petabytes of data using commodity hardware. Databases such as PostgreSQL and low-latency data stores such as Redis or RocksDB let developers to achieve sub-second responses for sophisticated queries issued to the stored data.

However, it is still hard to achieve similar results in the decentralized ecosystem. While blockchains such as Ethereum [5] are naturally designed to perform arbitrary computations, they are too expensive to handle any nontrivial amounts of data. Every node in Ethereum network has to replicate and process the same data, which makes the network secure but not too much cost-efficient. Furthermore, the current Ethereum design does not scale well and is limited to ≈ 15 transactions per second.

Cost efficiency is not the only property we might care about when building decentralized applications: latency is another important feature. It takes several seconds to propagate a block in Ethereum, and usually we need to wait for few additional blocks to gain enough confidence that the state change will not be reverted. Yet for lots of user-facing web or mobile applications we expect reaction time within a second.

Ideally, a decentralized data processing platform should allow building applications with throughput, latency, and cost efficiency comparable to the ones we see in the centralized world. This platform should also guarantee correctness of computations and state transitions even if some of the nodes in the network are malicious. While there has been a significant amount of work in the blockchain community, none of existing approaches reach the ideal state. Instead, every solution has its own set of tradeoffs. Below we discuss few existing approaches and consider what tradeoffs they make.

Related work. Blockchain sharding is one of the scaling approaches where a pool of network nodes is split across multiple shards, and each shard is behaving similar to a mini-blockchain. Nodes assigned to a specific shard repeat the same computation and vote for performed state transitions. For a state transition to be accepted, it has to receive a majority ($\frac{1}{2}$ or $\frac{2}{3}$) of votes in the shard. Once there are enough votes, the state transition might be committed to the root blockchain.

Normally, the number of the nodes assigned to a single shard is in the order of hundreds, which provides significant security. For example, if an adversary is controlling $\frac{1}{6}$ nodes in the network and a shard contains 100 nodes then the probability that more than half of the nodes in the shard are malicious is less than 10^{-6}. However, sharding has an obvious drawback: it still has much lower cost efficiency compared to the one we see in the centralized environment.

Another approach, TrueBit [6] [7] greatly improves blockchains scalability and cost efficiency by employing a concept called *verification game*. Every computation in TrueBit is performed off-chain by a single node at first, but is verified by few other nodes later. Any verifier which does not agree with the original computation result is free to submit a dispute to an external judge. Of course, the judge must be absolutely credible, which is why the judge role is played by an Ethereum smart contract. This smart contract is not able to repeat the entire computation: instead the disagreeing parties must narrow down the computation to find the first instruction which has produced mismatching results. This instruction is then reexecuted by the smart contract to determine which party made a mistake and penalize it.

Essentially, TrueBit trades security for scalability and cost efficiency. Reduced redundancy greatly improves efficiency: instead of thousands nodes performing the same computation there are only a handful. However, because the number of the nodes verifying the computation is low, it might happen that an adversary controls all of them. In this case the adversary will be able to place incorrect results into the system. The chance of this happening is low but non-negligible: for an adversary controlling $\frac{1}{6}$ nodes in the network and 5 verifiers per computation the probability that all verifiers are malicious is $\approx 1\%$ [6].

Latency is another property of TrueBit we need to consider. In TrueBit, a significant amount of time should pass for the client to receive computation results. The reason is that the TrueBit protocol heavily relies on timeouts for performing and verifying computations. Moreover, the computation definition as well as computed results are posted to Ethereum blockchain, which means that the client has to wait for at least few dozen seconds to get a result.

Our approach. We believe that a meaningful share of use cases requires low-latency and cost-efficient computations but can tolerate slightly reduced security.

In this paper we present Fluence Core, a trustless computation platform that allows to achieve latency of few seconds for request processing and cost efficiency similar to traditional cloud computing.

Fluence provides moderate guarantees that results delivered to the client are correct, but warrants that served results will always *eventually* get additionally verified. If during the additional verification it is found that results were produced incorrectly, offending network nodes lose their deposits, which forces most nodes to behave honestly and greatly increases overall network security.

In the next section we provide a brief Fluence overview.

2 Overview

The Fluence network consists of nodes performing computations in response to transactions sent by external clients. Algorithms specifying those computations are expressed in the WebAssembly [8] bytecode; consequently, every node willing to participate in the network has to run a WebAssembly virtual machine.

Independent developers are expected to implement a backend package handling client transactions in a high-level language such as C/C++, Rust, or TypeScript, compile it into one or more WebAssembly modules, and then deploy those modules to the Fluence network. The network then takes care of spinning up the nodes that run the deployed backend package, interacting with clients, and making sure that client transactions are processed correctly.

Two major layers exist in the Fluence network: the real-time processing layer and the batch validation layer. The former is responsible for direct interaction with clients; the latter, for computation verification. In other words, real-time processing is the *speed layer* and batch validation is the *security layer*. The network also relies on Ethereum (as a secure metadata storage and dispute resolution layer) and Swarm [9]/Filecoin [10]/Arweave [11]/other decentralized storage solution (as a data availability layer) (*Fig. 1*).

In the rest of this section, we discuss the components that constitute the Fluence network, the motivation to have two different processing layers, and finally, the network economy model.

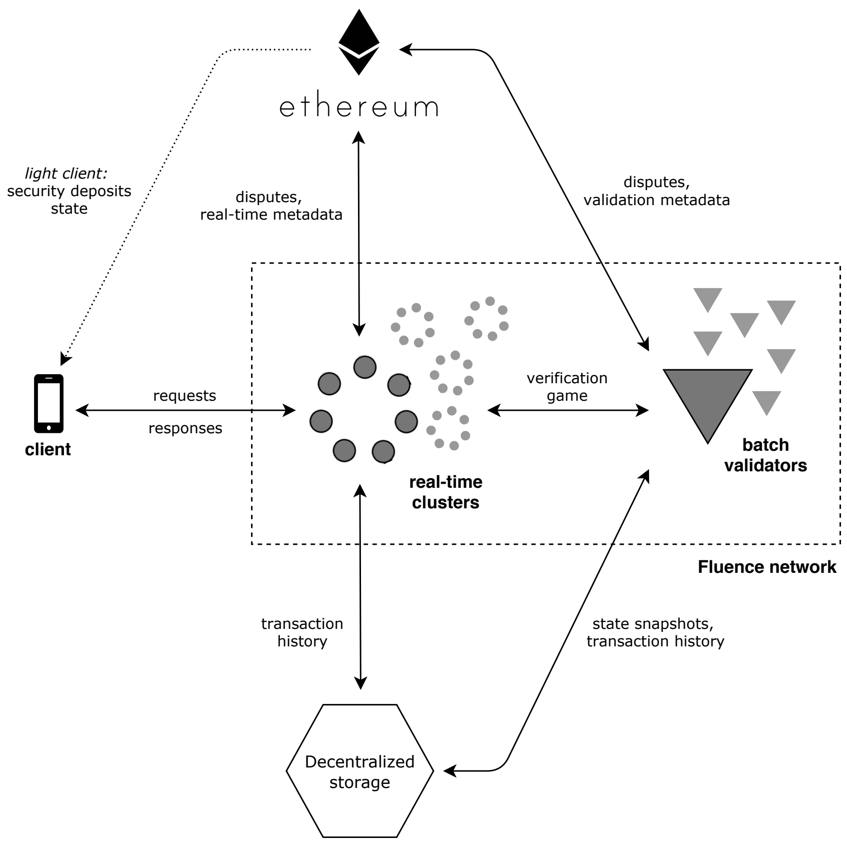


Fig. 1: An overview of the Fluence network architecture.

2.1 Components

Real-time processing layer. The real-time processing layer consists of multiple real-time clusters, which are stateful and keep locally the state required to serve client requests. Each cluster is formed by a few real-time worker nodes that are responsible for running particular backend packages and storing related state data. Workers in real-time clusters use Tendermint [12] [13] to reach BFT consensus and an interim metadata storage (built on top of a DHT such as Kademlia [14] [15]) to temporarily store consensus metadata before it is compacted and uploaded to the Ethereum blockchain.

To deploy a backend package to the Fluence network, the developer first has to allocate a cluster to run the package. Once the package is deployed, those functions that are exposed as external can be invoked by client transactions. If the package is no longer needed, the developer is able to terminate the cluster.

Developers possess significant control over real-time clusters: they are able to specify the desired cluster size and how much memory each node in the cluster should allocate to store the state. If one of the workers in the cluster is struggling, the developer who has allocated the cluster can replace this worker with a more performant one.

Real-time clusters are able to promptly respond to client requests, but those responses carry only moderate security guarantees when a significant fraction of network nodes are malicious. Because real-time clusters are formed by just a few worker nodes, they can tolerate only few malicious nodes, which leaves a non-trivial chance that a real-time cluster might be completely dominated by attackers. Therefore, an additional level of verification is required for computations performed by real-time clusters.

Batch validation layer. To keep real-time clusters in check, the batch validation layer separately verifies all performed computations. This layer is composed of independent batch validators, which are stateless and have to download the required data before performing verification. In order to support this, every real-time cluster is required to upload the history of received transactions and performed state transitions to the decentralized storage. Because Tendermint organizes transactions into blocks that each carry the hash of the state obtained after the previous block execution, real-time clusters upload transactions to the decentralized storage in blocks as well.

Later on, batch validators replay fragments of transaction history, which are composed of one or more blocks, and challenge state transitions that they have deemed incorrect through the dispute resolution layer. If one of the state transitions is not correct, it takes only a single honest validator to challenge this and penalize the real-time cluster that performed the transition.

Developers do not have any control over batch validators beyond deciding how much budget is carved out for batch validation – i.e., how many batch validations should happen for the fragment of transaction history once it is uploaded to the decentralized storage. Furthermore, the batch validator that verifies any specific history fragment is chosen randomly out of all batch validators in the network in order to prevent possible cartels.

Batch validators compact the transaction history and reduce the decentralized storage space usage by using intermediate state snapshots. Once a transaction history fragment has been verified a sufficient number of times, it is dropped, leaving only the corresponding snapshot.

Dispute resolution layer. We have already mentioned that batch validators are able to dispute state transitions. This ability is not exclusive to batch validators: a real-time worker can submit a dispute if it disagrees with another real-time worker on how the state should be updated. However, such disputes normally arise only between workers that belong to the same cluster – other real-time workers simply do not carry the required state.

No matter which node has submitted the dispute, it is resolved with the aid of an external authority. The Fluence network uses a specially developed Ethereum smart contract named *Arbiter* as this authority. Because Ethereum is computationally bounded and thus unable to repeat the entire computation to verify state transitions, a *verification game* mechanism [6] is used to find the first WebAssembly instruction that produced the diverging states. Only this instruction with the relevant portion of the state is then submitted to the Arbiter contract, which then makes its final decision as to which node performed the incorrect state transition.

Every node in the network is required to put down a significant security deposit before performing computations. If it is found that a node has behaved incorrectly, its deposit is slashed. Assuming that potential adversaries are financially restricted, this reduces the number of cases where a client might receive an incorrect response.

Data availability layer. Receipts mechanism (described for Swarm in [9]) is used to make sure that the fragments of transaction history uploaded by the real-time clusters do not disappear before the batch validators replay and verify them. A receipt is a confirmation from the decentralized storage node that it is responsible for the specific uploaded data. If the decentralized storage node is not able to return the data when requested, its deposit is slashed, which prevents the decentralized storage from losing potentially incriminating data. In other words, receipts mechanism prevents data withholding attacks.

Secure metadata storage. Deposits placed by Fluence network nodes, decentralized storage receipts issued for transaction history fragments, and metadata entries related to the batch validation and real-time cluster compositions are stored in the Ethereum blockchain. For the sake of simplicity in this paper, we will assume that the Arbiter contract holds this data in addition to its dispute resolution responsibilities.

2.2 Motivation

To understand the reason behind having two noticeably different layers, we need to recall their properties. Real-time workers are stateful, which considerably improves response latencies because they do not have to download the required state data to perform computations.

As an example, assume that we are building a decentralized SQL database that should support an indexed access to data. Complex queries such as the one listed below often require traversal of multiple indices, which are often implemented as B-trees.

```
SELECT
  DATE(ts) AS date,
  AVG(gas_price * gas_used) AS tx_cost
FROM transactions tx
WHERE
  tx.to IN (SELECT address
         FROM contracts
         WHERE name = 'CryptoDolphins')
GROUP BY DATE(ts)
```

Example query to blockchain data.

To traverse a B-tree, we need to sequentially fetch its nodes that satisfy the query conditions. It is not possible to retrieve the required B-tree nodes all at once because the next node to fetch can be determined only by matching the parent B-tree node against the query. If an index is stored externally in a decentralized storage, this means that multiple network roundtrips must be performed between the machine performing the computations and the data storage, which significantly increases latency.

Other algorithms, especially those requiring irregular access to data, could benefit from storing their data locally as well. However, data locality significantly increases the economic barrier to joining a real-time cluster because a worker willing to participate in the cluster has to download the current state first. Consequently, this motivates workers to remain in the cluster and thus cluster compositions do not change much over time.

This means that malicious real-time workers in the cluster might form a cartel to produce incorrect results. Without batch validation, every node in the real-time cluster knows it will be verified only by its peers, which are known in advance because clusters are tightly connected. Consequently, malicious nodes can, for example, exploit the following strategy: use a special handshake to recognize other malicious nodes in the cluster and start producing incorrect results if they account for at least $\frac{2}{3}$ of the total number of nodes (so they can reach BFT consensus without talking to the rest of the cluster); otherwise, work honestly. This strategy is virtually impossible to catch without external verification.

Furthermore, because real-time clusters are supposed to be small enough to be cost-efficient, the probability that malicious nodes will take over a cluster is significant. For example, for a network where 10% of all nodes are malicious, a real-time cluster that consists of 7 workers independently sampled from the network has approximately a $1.8 \cdot 10^{-4}$ chance (§7) to have at least $\frac{2}{3}$ malicious nodes.

To counteract this, the batch validation layer provides external verification. Nodes performing batch validation are chosen randomly, which means real-time nodes do not know *beforehand* which validator will be verifying them and thus cannot collude with the validator in advance.

Batch validation also decreases the probability that an intentional mistake made by a real-time cluster will never get noticed. Assume that in the same network where 10% of all nodes are malicious, we spun a real-time cluster of 4 workers and allocated a budget for 3 batch validations. In this setup, a mistake can go unnoticed only if malicious actors comprise at least $\frac{2}{3}$ of the real-time workers and all of the batch validators that verified the transaction history. The chance of this happening is $\approx 3.7 \cdot 10^{-6}$ (§7), which is two orders of magnitude less than in the case where the entire budget was spent on the real-time cluster only.

We also expect that in the presence of batch validators, the fraction of malicious nodes in the network will drop significantly below 10%, because every time a malicious action is caught, the node that performed it loses its deposit and thus leaves the network.

2.3 Incentive model

Fluence uses a concept similar to Ethereum gas to track computational efforts. With few exceptions, every WebAssembly instruction has a predefined associated cost, which is named *fuel* to avoid confusion with Ethereum gas. The fuel required to perform a computation is roughly proportional to the total sum of fuel amounts assigned to instructions in the computation execution trace (§3.2).

When it comes to storage usage accounting, Fluence rules differ significantly from Ethereum. Ethereum instructions for interacting with persistent storage require the client to pay a one-time fee – which is significant – to compensate for a certain amount of future expenses that will be incurred in storing the information. For example, `SSTORE` (the instruction that is used to save a word to the persistent storage) costs 20000 gas, which is a few orders of magnitude more than the cost of basic instructions such as `POP`, `ADD`, or `MUL`, which require 2–5 gas [16].

While this approach has been working fairly well for Ethereum, we think that adapting it to the Fluence network – the aim of which is achieving cost-efficiency comparable to traditional clouds – is problematic. In conventional backend software, it is common to update an on-disk or in-memory state without worrying that the performed update costs considerably more than other operations. In order to easily port existing software to Fluence, we need to provide developers a method that does not require them to fundamentally modify the code being ported.

Another reason to reconsider storage accounting is that execution of WebAssembly instructions and provision of data storage are quite different. If we say that network nodes are compensated for the performed *work*, then the total difficulty of processed instructions indeed defines an amount of the work performed. However, allocating a megabyte of storage is not *work* – it is *power*. Only after a node has kept a megabyte of data in storage for a certain time can we estimate how much work it has performed: $work = power \cdot time$.

When a client pays a one-time upfront fee to upload their data, there is no way for the network node responsible for its storage to know how long the data will be stored. No matter how large the upfront fee is, it is possible that expenses required to store the data will exceed this fee, leaving the financial burden on the node. This means that a different storage accounting approach must be developed for the Fluence network, which we propose and discuss below.

Rewards accounting. To counteract the aforementioned issues, various storage rent fees were proposed for Ethereum, including requiring clients to pay a fee to renew their storage every time they issue a transaction [17]. However, to bring the developer experience as close as possible to traditional backend software, in the Fluence network, the developer is the only party finally responsible for compensating network nodes.

Fluence nodes are compensated for the computational difficulty of executed WebAssembly instructions and for the storage space allocated for a specific period of time. Because different hardware might need different time to execute the same program, computational difficulty is used as a substitution for time. In other words, once a block of client transactions is processed and the fuel φ required to process it is counted, this fuel is transformed into the standard time t_{std} by multiplying it by the network-wide scaling constant $c_{time/fuel}$:

$$t_{\text{std}} = c_{\text{time/fuel}} \cdot \varphi \tag{2.1}$$

To estimate the total node reward Υ, two more scaling constants are introduced: $c_{\Upsilon/\text{fuel}}$ converts spent units of fuel into the network currency; $c_{\Upsilon/\text{spacetime}}$ does the same with the unit of storage space allocated for the unit of time. does the same with storage space per time. Assuming that the size of the allocated storage space is denoted by ω, the total node reward is computed as:

$$\Upsilon = c_{\Upsilon/\text{fuel}} \cdot \varphi + c_{\Upsilon/\text{spacetime}} \cdot \omega \cdot t_{\text{std}} \tag{2.2}$$

It should be noted that contrary to the system used by Ethereum where a client is able to choose a different gas price for every transaction, the scaling constants $c_{\text{time/fuel}}$, $c_{\Upsilon/\text{fuel}}$, and $c_{\Upsilon/\text{spacetime}}$ are fixed for the entire Fluence network. One reason for this design is that batch validators are selected randomly and are not able to choose the computations they are going to verify.

By allowing clients or developers to choose their compensation level, batch validators might be forced to perform complex computations for an unreasonably low reward. To prevent this, scaling constants are periodically updated, similar to how mining difficulty changes in Ethereum. If there is not enough supply, the network-wide compensation level increases; conversely, if there is not enough demand, the compensation level drops.

Dummy transactions. Because time is counted only for performed computations, simply storing the state without processing transaction blocks does not ensure any compensation to real-time workers. Therefore, in cases where incoming client transactions are rare, it is possible that the block creation rate will be low and thus low-demand backends will spend lots of time storing the state between the blocks. This time will never be compensated; workers running such low-demand backends might spend far more resources to store the state than their total compensation will be.

To offset this, real-time workers are allowed to send dummy transactions to themselves; the fuel required to process such dummy transactions is accounted for in the same way as the fuel required to process client transactions. This way, even if the client transaction volume is low, real-time workers will be compensated proportionally to the (real-world) time they have been running a certain backend deployment.

Batch validators, however, are not affected by this issue because they do not have to *wait* for incoming transactions and new blocks. A batch validator replays a fragment of transaction history at the maximum rate it is able to perform; once it completes the processing of the fragment, it moves to the next fragment. Additionally, for the same amount of work (which is defined by used fuel), real-time workers and batch validators are compensated evenly, which makes both options equally attractive to miners. Therefore, no special mechanism to recompense batch validators exists in the Fluence network.

Because different hardware can process transactions at different rates, it might happen that a very fast real-time cluster will be able to produce and process dummy transactions so fast that the compensation from the developer will become unexpectedly high. To mitigate this, in addition to being able to set the size of the storage space ω, developers have the ability to set the maximum fuel amount $\varphi_{\max}$ that real-time workers are allowed to spend per unit of time.

This allows a developer to budget how much will be spent on computations performed by the Fluence network in the next day, week, or month. Additionally, it lets real-time workers plan how much capacity they should allocate for transaction processing performed by the backend deployed by that developer.

We should note that it is possible for ill-disposed real-time workers to process *only* self-created dummy transactions – and none sent by clients. While we do not discuss a detailed mechanism to combat such behavior in this paper, we should note that a developer monitoring the use of the deployed backend will be able to notice a drop in the ratio between client and dummy transactions. In this case, the developer can either replace misbehaving real-time workers with other network nodes, or reduce the amount of fuel $\varphi_{\max}$ that the real-time cluster is allowed to spend per unit of time.

Client billing. As we have previously mentioned, developers are exclusively responsible for paying out rewards to the Fluence network nodes. Because miners' compensation is proportional to used fuel and allocated storage space, developers are directly incentivized to write efficient backends.

Generally, clients are not responsible for making any payments to network nodes. Nevertheless, different external monetization schemes are possible for reimbursing developers. The Fluence network does not prescribe an exact monetization scheme; however, it might provide some of the most common schemes through extension packages.

For example, one developer might allow only those clients from a whitelist to interact with the deployed backend, charging a flat rate to add a client to that whitelist; another developer might charge clients a fixed fee per each submitted transaction. It might also be possible for clients to pay no explicit fee while the developer uses their personal funds to cover the miners' expenses.

3 Virtual machine

As we have already discussed, both real-time workers and batch validators perform user-defined computations. In addition, to resolve disputes through the verification game, the Arbiter contract needs to run at least some portion of those computations as well. This brings us the need for a simple, secure, efficient, and reliable virtual machine. Below, we consider what properties a virtual machine should have to match Fluence requirements.

- *Step-by-step execution.* In the verification game mode, a contested machine should be able to stop the program execution at any instruction to compare the virtual machine state with that of the challenger. The Arbiter contract should also be able to execute a single virtual machine instruction without exceeding the gas limit.
- *Internal state access.* The virtual machine should allow easy access to its internal data structures. This is required for the verification game: disputing parties need to compare their virtual machine states when searching for the instruction that produced diverging states. Once this instruction is found and is ready to be executed by the Arbiter contract, the Arbiter contract will require certain pieces of the virtual machine state to serve as input data for this instruction.
- *Deterministic execution.* Because real-time workers need to achieve consensus before returning results to a client, every worker in a real-time cluster should perform identical state transitions. Moreover, the verification game requires that the same sequences of instructions produce the same states; otherwise it would be impossible to penalize a node that has performed an incorrect computation.
- *Performance.* The virtual machine should have low overhead so software written for it has efficiency comparable to native counterparts. While verification game mode is expected to happen rarely enough that reduced performance will be tolerable, normal execution requires low latencies and high throughput.
- *Sandbox isolation.* Because Fluence nodes have to execute code published by untrusted developers, there should be a solid protection that will prevent maliciously crafted code from escaping the virtual machine and accessing sensitive data.

– *Ecosystem and integration.* The virtual machine should be surrounded by a mature ecosystem. It should be possible to use different programming languages to develop code for the virtual machine, and there should also exist high-quality documentation and toolchains. Finally, it should be easy to port existing code to the virtual machine.

WebAssembly. To satisfy the aforementioned requirements, Fluence uses WebAssembly as the underlying virtual machine. Essentially, WebAssembly is just a specification of an abstract stack-based virtual machine that leaves a lot of freedom for particular implementations.

Existing implementations such as Asmble [18] can be easily adapted to support step-by-step execution and access to the internal virtual machine state from the host environment (see Appendix A). Additionally, WebAssembly specification leaves only a few possible options for nondeterministic behavior, which can easily be altered to adapt the virtual machine for use in the Fluence network.

Code ported to WebAssembly quite often can be executed at a speed only a few times slower than the native version: the benchmark published by *Jangda et al.* [19] reported 1.4×–1.9× average slowdown with WebAssembly versus the corresponding native code when embedded in a Firefox or Chrome browser.

One of the major WebAssembly target use cases was to be embedded into other applications (e.g., browsers). This led to the design where the virtual machine is carefully isolated from the rest of the host application and operating system in a sandboxed environment [20]. Furthermore, because reduction rules are defined for each operation, it is possible to verify WebAssembly code for correctness before the execution. This provides a solid level of security and makes it possible to run WebAssembly code that has been received from untrusted sources [21].

Finally, the ecosystem around WebAssembly can be considered reasonably mature. Four major browsers support WebAssembly bytecode execution [8]. Languages such as C/C++ [22], Rust [23], and TypeScript [24] can be compiled into WebAssembly modules, which allows projects as complex as Doom 3 to run in a browser when compiled into WebAssembly [25].

Further in this section, we introduce the virtual machine abstraction and discuss key concepts such as fuel accounting and deterministic execution. The reference virtual machine implementation is discussed in Appendix A.

3.1 Gateway interface

Virtual machine state. Memory, stack, and other WebAssembly structures such as global variables and tables constitute the virtual machine state VM. The program counter `pc`, the executed instructions counter eic, and the fuel counter `tf` are also included in VM, so:

$$\mathsf{VM} := (\mathsf{pc}, \mathsf{mem}, \mathsf{stack}, \dots, \mathsf{eic}, \mathsf{tf}) \tag{3.1}$$

The program counter `pc` points to the WebAssembly instruction that should be executed next by the virtual machine and is equal to `-1` when the virtual machine should stop. The executed instructions counter eic is equal to the number of instructions that the virtual machine has executed since its initialization. Finally, the fuel counter `tf` is equal to the amount of fuel spent since the virtual machine's initialization.

In Ethereum, the EVM state is split into volatile memory and persistent storage. The volatile memory is cleared after processing a transaction block, whereas the persistent storage data survives across blocks. On the contrary, in Fluence, all changes made to the virtual machine state are retained after the block processing is completed.

A composite hash of the virtual machine state H_{VM} is computed as follows. First, hashes are computed for each component in the state (such as memory, stack, and program counter). For components with a substantial size (such as memory or stack) this is done by building a Merkle tree and using the obtained Merkle root: $\mathsf{H}_{\text{MEM}} = \mathit{merkle}(\mathsf{mem})$. For smaller components, the hash can be computed directly: $\mathsf{H}_{\text{PC}} = \mathit{hash}(\mathsf{pc})$. Finally, the hash of the virtual machine state is obtained by building a Merkle tree over the individual components' hashes:

$$\mathsf{H}_{\text{VM}} := \mathit{merkle}(\mathsf{H}_{\text{PC}}, \mathsf{H}_{\text{MEM}}, \ldots, \mathsf{H}_{\text{TF}}) \tag{3.2}$$

With this construction, it is possible to prepare a proof that a part of the virtual machine state corresponds to the specific state hash. For example, two Merkle proofs are required to prove that a memory chunk corresponds to H_{VM}. The first should demonstrate that the chunk corresponds to H_{MEM}, and the second should demonstrate that H_{MEM} corresponds to H_{VM}.

Virtual machine operations. A virtual machine is always initialized with an empty state VM_0. A single function in a WebAssembly program Ψ can be marked as a gateway function by the developer. External code is able to invoke that gateway function and modify the virtual machine state by passing an input L (which, in the case of Fluence, is a sequence of client transactions) as an argument:

$$\mathsf{VM}_{i+1} = \mathit{vmcall}(\Psi, \mathsf{VM}_i, \mathsf{L}) \tag{3.3}$$

For the new state VM_{i+1} the virtual machine is required to compute an updated state hash $\mathsf{H}_{\text{VM}_{i+1}}$. The complexity of this operation depends of what part of the original state has changed. If it was a merkelized state component such as memory, the virtual machine needs to recompute hashes of modified chunks in the state component and recompute hashes of the nodes on the paths from modified chunks to the Merkle root (*Fig. 2*).

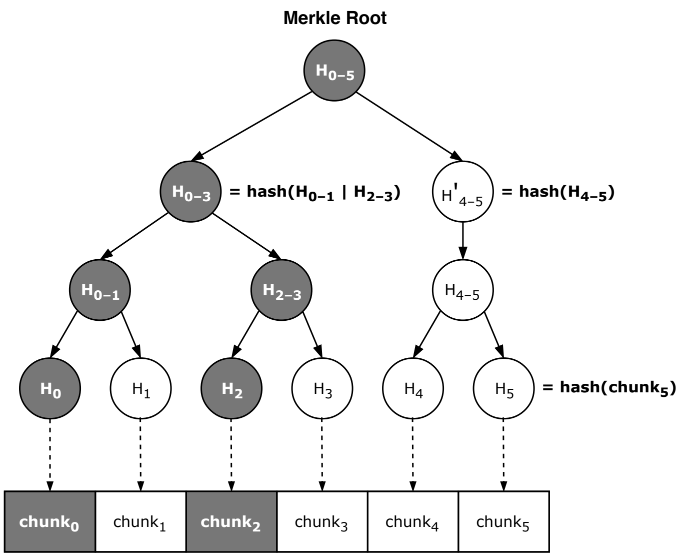


Fig. 2: Merkle root update path.

The complexity of the hash operation is linear relative to the size of the data being hashed. Therefore, for a binary Merkle tree composed of n chunks, out of which d were modified, the worst-case complexity of the Merkle root update is $\mathcal{O}(u \cdot \mathsf{d} + v \cdot \mathsf{d} \cdot \log_2 n)$, where u is the chunk size and v is the hash size. Assuming that the chunk size, the hash size, and the number of chunks are all constant, we obtain that the Merkle root update time is linear in d.

If a primitive state component (e.g., the program counter) was modified, its hash can be recomputed in constant time. Therefore, the total time to recompute the state hash H_{VM} is linear in the number of modified memory or other merkelized component chunks.

3.2 Fuel accounting

The only time that fuel can be spent by the node running the virtual machine is when a gateway function is called. Assume that invoking *vmcall*() with the input L for the program Ψ in the state VM generates an *execution trace* of n WebAssembly instructions $\mathsf{Tr}_{\Psi,\mathsf{VM},\mathsf{L}} = (\mathsf{ins}_1, \mathsf{ins}_2, \ldots, \mathsf{ins}_n)$, which are sequentially executed by the virtual machine.

Also assume there exists a function *fuel*() that for any instruction ins in the WebAssembly instruction set returns how much fuel this instruction requires. In this case, we can compute the amount of fuel spent to process instructions in the execution trace as a sum of fuel amounts required for each individual instruction:

$$\mathit{fuel}(\mathsf{Tr}_{\Psi,\mathsf{VM},\mathsf{L}}) = \sum_{j=1}^{n} \mathit{fuel}(\mathsf{ins}_j) \tag{3.4}$$

Additionally, the node running the virtual machine has to recompute the state hash after each gateway function invocation. As shown in §3.1, the complexity of the state hash update linearly depends on the number of modified state chunks.

Computational resources required to perform the state hash update can be significant. Imagine that a developer has produced a code executing 10^6 `i32.store` WebAssembly instructions, where each instruction writes to a different memory chunk. If the chunk size is fixed at `4 KB`, this means that a node will need to recompute the hash of the `4 GB` memory. However, computing the hash of a `4 KB` chunk might be a few orders of magnitude slower than storing 4 bytes with the `i32.store` instruction. This means that the node needs to be additionally reimbursed for the state hash recomputation.

To compensate network nodes, fuel is also charged for dirtying memory chunks. In the beginning of the *vmcall*() invocation, all chunks are marked as clean. If a WebAssembly instruction modifies a clean chunk, this chunk is marked as dirty and retains this state until the end of the gateway function call. The total fuel required to process the gateway function call is then computed as the sum of the fuel required to process instructions in the execution trace and the fuel required to recompute the state hash:

$$\mathit{fuel}_{\text{vmcall}}(\mathsf{Tr}_{\Psi,\mathsf{VM},\mathsf{L}}) = \sum_{j=1}^{n} \mathit{fuel}(\mathsf{ins}_j) + \mathsf{d} \cdot \mathsf{c}_{\text{fuel/chunk}} \tag{3.5}$$

In the formula above, d is the total number of chunks made dirty, and $\mathsf{c}_{\text{fuel/chunk}}$ is the network-wide constant specifying how much fuel should be charged for recomputing the chunk hash.

Denote by d_j the number of chunks that the execution of the instruction ins_j marked dirty. Then equation (3.5) can also be rewritten in the following form, which is useful for fuel disputes (discussed in §4.2):

$$\mathit{fuel}_{\text{vmcall}}(\mathsf{Tr}_{\Psi,\mathsf{VM},\mathsf{L}}) = \sum_{j=1}^{n} \mathit{fuel}_{\text{dirty}}(\mathsf{ins}_j), \text{ where} \tag{3.6}$$

$$\mathit{fuel}_{\text{dirty}}(\mathsf{ins}_j) = \mathit{fuel}(\mathsf{ins}_j) + \mathsf{d}_j \cdot \mathsf{c}_{\text{fuel/chunk}} \tag{3.7}$$

3.3 Deterministic execution

There are three major sources of nondeterminism in WebAssembly: noncanonical `NaN` payloads in floating point arithmetic, possible resource (e.g., memory or stack) exhaustion, and external function invocation [8].

NaN payloads. WebAssembly specification states only two possible ways of nondeterministic behavior in respect to `NaN` values. First, floating point operations might produce results with nondeterministic sign bits. Second, a nondeterministic output might be produced if any input to an operation was not canonical.

To make floating point operations deterministic, we strengthen the specification and stipulate that floating point operations must produce only canonical `NaN` results with the sign bit set to `1`. This means that 32-bit `NaN` values should always have the form `111111111000...0001`.

Resource exhaustion. To deal with resource exhaustion we demand that the developer has to specify the required memory amount beforehand. When the virtual machine is launched, it allocates the entire memory at once. Now, any instruction referencing an address within the allocated range must be processed correctly, whereas any instruction referencing outside of the range must trap. The WebAssembly `memory.grow` instruction, which extends the available memory set, must always return `-1` (unable to allocate more memory), and the `memory.size` instruction must return the memory size set by the developer.

A similar approach is taken with stack. Stack overflow cannot be noticed from within WebAssembly, which means there will not be an actual disagreement while reaching consensus or during the dispute resolution process. However, stack overflow might crash the virtual machine preventing it from obtaining computation results. To avoid this, the developer also must specify the required stack size before the submitted package is accepted by the Fluence network.

Other resources, such as global variables or tables, are dealt with in a similar manner.

External functions invocation. Generally, user space software interacts with the external world through system calls. In Fluence, available system calls are expressed as WebAssembly host functions, so when a deployed backend package makes a system call, this call is served by the wrapping node process. Only those calls explicitly allowed in the protocol will be processed – the rest will result in a trap. In addition, allowed system calls will always be processed deterministically.

For example, assume that a developer wants to be able to invoke the `time`() function from the backend package. It is not possible to return the current system time as a result of this call because the time will be different for different real-time workers and batch validators.

Instead, the `time`() function can be implemented as follows. The fuel spent since initialization is tracked as a part of the virtual machine state. The virtual machine "clock" is incremented by `1 ms` every time `1000000` fuel is spent. This way, every node in the network has the same notion of "time". Ethereum smart contracts performing dispute resolution can apply the same logic as well.

4 Verification game

It might happen that two network nodes do not agree on how the state transition should have been performed. In other words, two nodes produced different virtual machine states when they applied the same transaction block to the same initial state.

In some cases, this means that the produced states have different virtual machine memories; in other cases it means that the states have different metadata, for example, different fuel or program counters. Because the virtual machine should behave fully deterministically in our protocol, this means that at least one of the nodes is malicious or made a mistake.

Any network node is able to submit a dispute if it does not agree with another party. To resolve the disagreement, we use a protocol similar to the interactive verification game described by TrueBit in [6]. As is done in TrueBit, in Fluence we introduce an external computationally and spatially bound *judge*, which in the Fluence case is the Arbiter contract capable of executing a single WebAssembly instruction without spending too much gas. The update of the virtual machine state constructed by the Arbiter contract is taken as the final judgment. However, the Arbiter contract is unable to execute more than a few instructions and is unable to load more than a tiny fragment of the virtual machine state.

To find the first WebAssembly instruction where the virtual machine states began to diverge, a k-ary search is used. The disputing nodes must split their computations into k sequential fragments and compare the virtual machine states obtained at the end of each fragment. The fragment that has identical initial virtual machine states but different terminal states is further split into k smaller fragments and so on, until eventually a single instruction is found to have been executed differently by the nodes.

This instruction is then submitted to the Arbiter contract along with the proof that it really belongs to the deployed WebAssembly backend package. Parts of the virtual machine state that are required for the instruction execution (e.g., certain chunks of the virtual machine memory) are submitted to the Arbiter contract as well. Once the Arbiter has executed the instruction, it figures out which node has performed the incorrect state update and penalizes it.

Below we describe the protocol in more detail. We begin with a simple case where both nodes agree on the number of instructions in the execution trace and on the amount of fuel used to process the transaction block. This construction is quite similar to the one described by TrueBit, albeit adjusted to the WebAssembly virtual machine outlined in §3.

We also discuss how to address those cases where execution trace lengths used by disputing nodes do not match – in other words, when one node has stopped earlier than another. Finally, we discuss how a node can dispute the amount of fuel required by the computation, which is useful in two cases.

First, because network nodes are compensated in proportion to the amount of fuel they have spent, dishonest nodes might attempt to overstate this amount. Therefore, a protocol for adjudicating fuel disputes and penalizing overstating nodes is required to avoid excessively charging developers.

Second, malicious real-time clusters might declare that the computation has used too little fuel. In this case batch validators will not be properly compensated for completing the computation. Moreover, the time limit allocated for a batch validator to complete the computation is proportional to the fuel that was declared as spent by the real-time cluster (§6.4). This might prevent batch validators from being able to dispute an invalid computation if the ability to stop early and dispute the amount of spent fuel does not exist.

4.1 Protocol

Assume that two nodes A and B applied the same input L to virtual machines in the same state VM_i. Assume that both virtual machines were running the same program Ψ, but the nodes obtained different terminal states $\left[\mathsf{VM}_{i+1}\right]^{\mathrm{A}}$ and $\left[\mathsf{VM}_{i+1}\right]^{\mathrm{B}}$.

Denote by $[n]^{\mathrm{A}}$ the number of instructions that were executed by node A to process the input, and by $[n]^{\mathrm{B}}$ – the number of instructions executed by node B.

In the rest of this section, we will denote by $\mathsf{VM}_{i,m}$ the virtual machine state obtained after processing the first $m \leqslant n$ instructions, and by $\mathsf{H}_{\mathrm{VM}_{i,m}}$ – the hash of this state. Note that according to this notation, $\mathsf{VM}_{i,n} = \mathsf{VM}_{i+1}$ and $\mathsf{H}_{\mathrm{VM}_{i,n}} = \mathsf{H}_{\mathrm{VM}_{i+1}}$.

Dispute initiation. We assume that the Arbiter knows the Merkle hash H_Ψ of the WebAssembly program: $\mathsf{H}_\Psi = \mathit{merkle}(\Psi)$, which was provided by the developer when the backend package was initially deployed. By submitting an instruction along with the Merkle proof, any node can prove to the Arbiter that the instruction is located at a specific place in the original WebAssembly program Ψ. We also assume that the Arbiter knows the hash of the initial state $\mathsf{H}_{\mathrm{VM}_i}$ and the corresponding executed instructions counter eic_i, which both nodes agree on.

For the dispute to be initiated, one of the nodes, which we assume to be node A, needs to submit a dispute transaction with the terminal state hash $\left[\mathsf{H}_{\mathrm{VM}_{i+1}}\right]^{\mathrm{A}}$ and the executed instructions counter $\left[\mathsf{eic}_{i+1}\right]^{\mathrm{A}}$ to the Arbiter. Once the Arbiter receives the dispute transaction, it requires node B to submit its version of the state hash $\left[\mathsf{H}_{\mathrm{VM}_{i+1}}\right]^{\mathrm{B}}$ and the executed instructions counter $\left[\mathsf{eic}_{i+1}\right]^{\mathrm{B}}$ within a certain timeout. Both nodes must also provide Merkle proofs that submitted executed instructions counters correspond to the terminal state hashes.

The Arbiter then evaluates the received data. If $\left[\mathsf{H}_{\mathrm{VM}_{i+1}}\right]^{\mathrm{A}} \neq \left[\mathsf{H}_{\mathrm{VM}_{i+1}}\right]^{\mathrm{B}}$, then the nodes do not agree on the terminal states and the dispute resolution procedure must be performed.

If $\left[\mathsf{eic}_{i+1}\right]^{\mathrm{A}} = \left[\mathsf{eic}_{i+1}\right]^{\mathrm{B}}$, then the nodes agree on the number of executed instructions $n = \mathsf{eic}_{i+1} - \mathsf{eic}_i$ and the general dispute resolution protocol described in §4.1.1 can be used. Otherwise, nodes do not agree on the number of executed instructions and the execution trace length dispute resolution procedure (described in §4.1.2) should be used instead.

4.1.1 General dispute resolution

Diverging instruction index search. If $\left[\mathsf{eic}_{i+1}\right]^{\mathrm{A}} = \left[\mathsf{eic}_{i+1}\right]^{\mathrm{B}}$, the goal of the Arbiter is to force A and B to collaboratively find the smallest instruction index x such that virtual machine states diverged by executing instructions $[\mathsf{ins}_x]^{\mathrm{A}}$ and $[\mathsf{ins}_x]^{\mathrm{B}}$ respectively:

$$\begin{aligned}&\forall j < x : \left[\mathsf{H}_{\mathrm{VM}_{i,j}}\right]^{\mathrm{A}} = \left[\mathsf{H}_{\mathrm{VM}_{i,j}}\right]^{\mathrm{B}}\\ &\left[\mathsf{H}_{\mathrm{VM}_{i,x}}\right]^{\mathrm{A}} \neq \left[\mathsf{H}_{\mathrm{VM}_{i,x}}\right]^{\mathrm{B}}\end{aligned} \tag{4.1}$$

In the beginning, the Arbiter knows only that $x \in (0, n]$. To narrow down this interval, the Arbiter requires each node to provide $(k-1)$ hashes (where k is the predefined protocol constant) of the intermediate virtual machine states:

$$\begin{aligned}&\mathsf{H}_{\mathrm{VM}_{i,\sigma(1)}}, \mathsf{H}_{\mathrm{VM}_{i,\sigma(2)}}, \ldots, \mathsf{H}_{\mathrm{VM}_{i,\sigma(k-1)}}, \text{where}\\ &\sigma(j) = \left\lceil \frac{j \cdot n}{k} \right\rceil\end{aligned} \tag{4.2}$$

Comparing the provided hashes, the Arbiter is able to find the first $l < k$ such that:

$$\begin{aligned}&\left[\mathsf{H}_{\mathrm{VM}_{i,\sigma(l)}}\right]^{\mathrm{A}} = \left[\mathsf{H}_{\mathrm{VM}_{i,\sigma(l)}}\right]^{\mathrm{B}}\\ &\left[\mathsf{H}_{\mathrm{VM}_{i,\sigma(l+1)}}\right]^{\mathrm{A}} \neq \left[\mathsf{H}_{\mathrm{VM}_{i,\sigma(l+1)}}\right]^{\mathrm{B}}\end{aligned} \tag{4.3}$$

Consequently, now the Arbiter knows that the target instruction index $x \in \big(\sigma(l),\, \sigma(l+1)\big]$ and is thus able to repeat the search iteration by narrowing down the reduced interval. Eventually, the Arbiter finds the target x.

Instruction re-execution. Once the target instruction index x is found, the Arbiter needs to execute the instruction ins_x to resolve the dispute. Because the nodes might not necessarily agree on which instruction it should be, the Arbiter requires them to provide the program counter $\mathsf{pc}_{i,x-1}$ corresponding to the virtual machine state $\mathsf{VM}_{i,x-1}$. Additionally, the nodes must provide the Merkle proof that the program counter matches the state hash $\mathsf{H}_{\mathrm{VM}_{i,x-1}}$, which both nodes have previously agreed on.

After that, the Arbiter requires disputing nodes to provide the instruction ins_x and the Merkle proof that it matches the hash H_Ψ of the WebAssembly program Ψ and is located in the program at the offset specified by the program counter $\mathsf{pc}_{i,x-1}$. If any of the Merkle proofs are incorrect, the node that provided such proof automatically loses the dispute.

To independently execute the instruction, the Arbiter also needs chunks $(\eta_1, \eta_2, \ldots, \eta_p)$ of the virtual machine state $\mathsf{VM}_{i,x-1}$, which the instruction ins_x will be reading from or writing to. The Arbiter requires disputing parties to provide these chunks with Merkle proofs of correspondence to the last known good state $\mathsf{H}_{\mathrm{VM}_{i,x-1}}$. By executing ins_x, the Arbiter obtains updated chunks $(\eta_1^*, \eta_2^*, \ldots, \eta_p^*)$ that it substitutes into the provided Merkle proofs to compute the new virtual machine state hash $\left[\mathsf{H}_{\mathrm{VM}_{i,x}}\right]^*$.

Now, the Arbiter compares the authentic state hash $\left[\mathsf{H}_{\mathrm{VM}_{i,x}}\right]^*$ with the hashes $\left[\mathsf{H}_{\mathrm{VM}_{i,x}}\right]^{\mathrm{A}}$ and $\left[\mathsf{H}_{\mathrm{VM}_{i,x}}\right]^{\mathrm{B}}$ provided by the disputing nodes. The node that provided a different state hash is considered to be the violator and loses its security deposit.

4.1.2 Execution trace length dispute resolution

If $\left[\mathsf{eic}_{i+1}\right]^{\mathrm{A}} \neq \left[\mathsf{eic}_{i+1}\right]^{\mathrm{B}}$, the Arbiter computes the number of instructions executed by each node during the computation:

$$[n]^{\mathrm{A}} = \left[\mathsf{eic}_{i+1}\right]^{\mathrm{A}} - \mathsf{eic}_i \tag{4.4}$$

$$[n]^{\mathrm{B}} = \left[\mathsf{eic}_{i+1}\right]^{\mathrm{B}} - \mathsf{eic}_i \tag{4.5}$$

Then the Arbiter requires each party to provide the virtual machine state hash after executing $r = min([n]^{\mathrm{A}}, [n]^{\mathrm{B}})$ instructions.

If the provided state hashes do not match $(\left[\mathsf{H}_{\mathrm{VM}_{i,r}}\right]^{\mathrm{A}} \neq \left[\mathsf{H}_{\mathrm{VM}_{i,r}}\right]^{\mathrm{B}})$, then it is a dispute that can be resolved by following the general dispute resolution procedure described in §4.1.1. Indeed, both A and B agree that the number of instructions required to produce mismatching states $\left[\mathsf{VM}_{i,r}\right]^{\mathrm{A}}$ and $\left[\mathsf{VM}_{i,r}\right]^{\mathrm{B}}$ is equal to r.

Otherwise, both parties have provided the same state hash $\mathsf{H}_{\mathrm{VM}_{i,r}}$, and the Arbiter has to figure out whether the computation should halt after producing the virtual machine state $\mathsf{VM}_{i,r}$. To do so, the Arbiter requires A and B to provide the program counter $\mathsf{pc}_{i,r}$ with the Merkle proof of its correspondence to $\mathsf{H}_{\mathrm{VM}_{i,r}}$. If the provided program counter $\mathsf{pc}_{i,r}$ is equal to `-1`, it means that the computation should halt, and the node with the bigger number of instructions loses; otherwise this node wins.

4.2 Fuel disputes

The amount of fuel φ spent to advance the state from VM_i to VM_{i+1} should be equal to the difference between initial and terminal fuel counters:

$$\varphi = \mathit{fuel}_{\mathrm{vmcall}}(\mathsf{Tr}_{\Psi,\mathsf{VM}_i,\mathsf{L}}) = \mathsf{tf}_{i+1} - \mathsf{tf}_i \tag{4.6}$$

Because tf_{i+1} belongs to the terminal virtual machine state VM_{i+1}, if A and B disagree on how much fuel was spent during the computation, they can resolve this dispute by following the general procedure described in §4.1. However, if nodes are strictly performing the computation sequentially, there exists a special case that allows the second node to challenge the amount of spent fuel declared by the first node without entirely completing the computation.

Special case. *Assume that* A *has completed the computation first and then* B *is verifying it. Denote by* $[\varphi]^{\text{A}}$ *the amount of fuel that* A *claims was spent to complete the computation. If* $[\varphi]^{\text{A}}$ *is less than the true fuel amount* $[\varphi]^{*}$ *required by the computation,* B *can dispute that while spending not more than* $[\varphi]^{\text{A}} + \varepsilon$ *fuel, where* ε *is negligible.*

Assume that in the end of the computation, A has submitted the hash of the terminal virtual machine state $\left[\mathsf{H}_{\text{VM}_{i+1}}\right]^{\text{A}}$ to the Arbiter contract. Assume that A has also submitted to the Arbiter the executed instructions counter $\left[\mathsf{eic}_{i+1}\right]^{\text{A}}$ and the fuel counter $\left[\mathsf{tf}_{i+1}\right]^{\text{A}}$ with Merkle proofs of correspondence to $\left[\mathsf{H}_{\text{VM}_{i+1}}\right]^{\text{A}}$.

Based on this data, the Arbiter is able to compute the number of instructions $[n]^{\text{A}}$ that A claims were used to complete the computation: $[n]^{\text{A}} = \left[\mathsf{eic}_{i+1}\right]^{\text{A}} - \mathsf{eic}_i$.

Assume that B processed x instructions while performing the computation and reached the first virtual machine state $\left[\mathsf{VM}_{i,x}\right]^{\text{B}}$ such that the amount of fuel spent by B to reach it is already higher than declared by A :

$$\begin{aligned}&\forall j < x : \left[\mathsf{tf}_{i,j}\right]^{\text{B}} \leqslant \left[\mathsf{tf}_{i+1}\right]^{\text{A}} \\ &\left[\mathsf{tf}_{i,x}\right]^{\text{B}} > \left[\mathsf{tf}_{i+1}\right]^{\text{A}}\end{aligned} \tag{4.7}$$

In this case, B can immediately initiate a special case fuel dispute. To do so, B needs to send to the Arbiter contract the number of executed instructions x, intermediate state hashes $\left[\mathsf{H}_{\text{VM}_{i,x-1}}\right]^{\text{B}}$ and $\left[\mathsf{H}_{\text{VM}_{i,x}}\right]^{\text{B}}$, and intermediate total spent fuel amounts $\left[\mathsf{tf}_{i,x-1}\right]^{\text{B}}$ and $\left[\mathsf{tf}_{i,x}\right]^{\text{B}}$.

If $[n]^{\text{A}} < x$, then because $x \leqslant [n]^{\text{B}}$, A and B would end with a different number of instructions required to complete the computation. In this case, B initiates an execution trace length dispute, which is resolved through the procedure described in §4.1.2. Note that in this case the minimal common prefix length $r = min([n]^{\text{A}}, [n]^{\text{B}})$ is equal to $[n]^{\text{A}}$, and B does not need to finish the computation.

If $[n]^{\text{A}} \geqslant x$, the Arbiter requests from A the state hash $\left[\mathsf{H}_{\text{VM}_{i,x-1}}\right]^{\text{A}}$ obtained after executing $x-1$ instructions. A situation where $\left[\mathsf{H}_{\text{VM}_{i,x-1}}\right]^{\text{A}} \neq \left[\mathsf{H}_{\text{VM}_{i,x-1}}\right]^{\text{B}}$ is a general dispute and is resolved as described in §4.1.1. Note that in this case also, B does not need to finish the computation.

Consequently, now we can assume that $[n]^{\text{A}} \geqslant x$ and $\left[\mathsf{H}_{\text{VM}_{i,x-1}}\right]^{\text{A}} = \left[\mathsf{H}_{\text{VM}_{i,x-1}}\right]^{\text{B}}$.

In this case, the Arbiter has to re-execute the instruction ins_x and check if $\left[\mathsf{tf}_{i,x}\right]^{\text{B}}$ is equal to $\left[\mathsf{tf}_{i,x-1}\right]^{\text{B}} + \mathit{fuel}_{\text{dirty}}(\mathsf{ins}_x)$. This is done in a way similar to the one described in §4.1.1. If the equality holds, it means that the spent fuel amount should be more than declared by A, and B wins the dispute; otherwise, B loses.

Note that B has performed the computation only up to the state $\left[\mathsf{VM}_{i,x}\right]^{\text{B}}$. The fuel that B has spent to reach this state can be expressed as:

$$[\varphi_{i,x}]^{\text{B}} = \sum_{j=1}^{x} \mathit{fuel}_{\text{dirty}}([\mathsf{ins}_j]^{\text{B}}) \tag{4.8}$$

Because the fuel $[\varphi_{i,x-1}]^{\text{B}}$ that B has spent to reach the state $\left[\mathsf{VM}_{i,x-1}\right]^{\text{B}}$ is less (4.7) than the total spent fuel $[\varphi]^{\text{A}}$ declared by A, using (3.6) we obtain:

$$[\varphi_{i,x}]^{\text{B}} = [\varphi_{i,x-1}]^{\text{B}} + \mathit{fuel}_{\text{dirty}}([\mathsf{ins}_x]^{\text{B}}) \leqslant [\varphi]^{\text{A}} + \varepsilon \tag{4.9}$$

Here ε is the maximum possible cost of execution of a single WebAssembly instruction and the rehashing of chunks it made dirty. Because a single instruction can write only to a few chunks, this is exactly what we were aiming for in this special case.

5 Real-time processing layer

The real-time processing layer acts as a *speed layer* in the Fluence network and is the only layer that clients directly interact with. It is split into multiple independent real-time clusters, that are controlled by individual developers. Workers in each real-time cluster run the same backend package and hold the same state; therefore, the replication level for the cluster is equal to the number of workers in it. Each real-time worker is composed of three major components: the Tendermint consensus engine process, the state machine, and the WebAssembly virtual machine (*Fig. 3*).

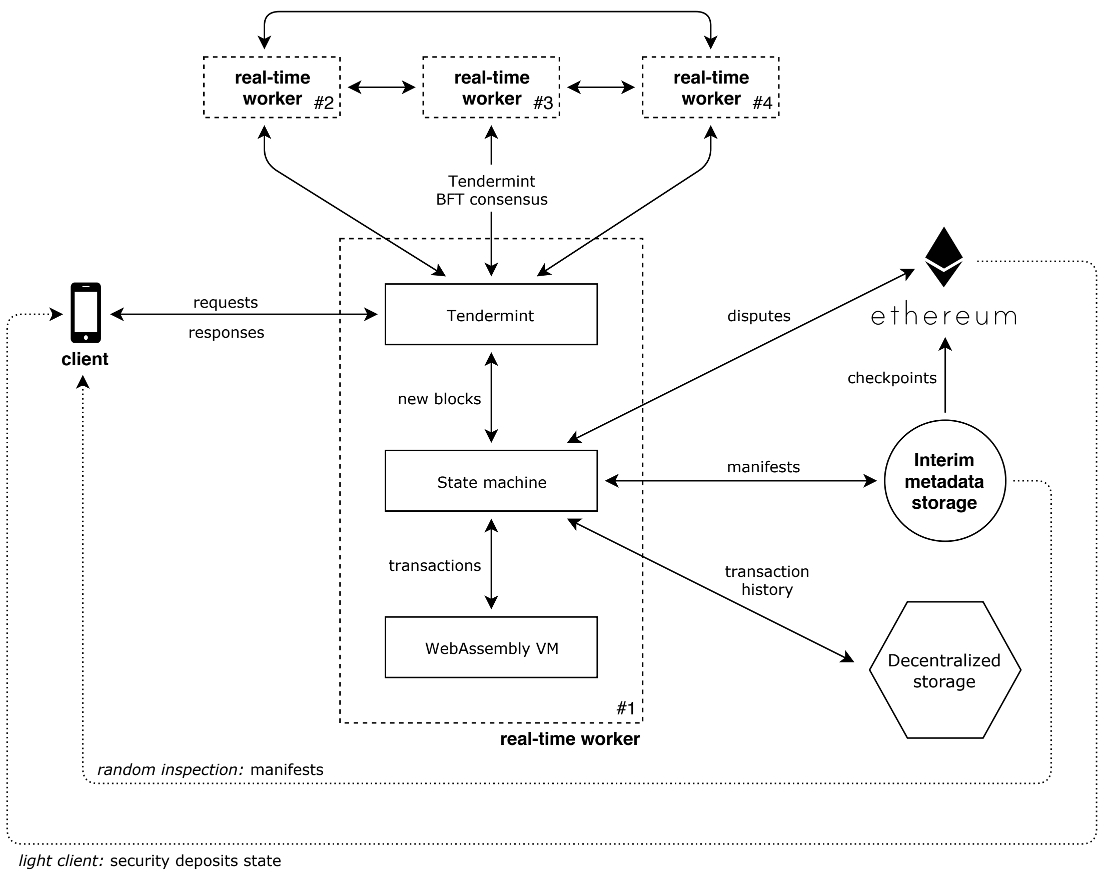


Fig. 3: Real-time layer architecture overview.

Consensus engine. Tendermint is responsible for the connectivity between cluster workers and the BFT consensus over transaction order and state transitions. To satisfy BFT consensus requirements, the number of workers in a cluster is expected to be of form $3k + 1$. We expect that most real-time clusters will be modestly sized, typically with 4–16 workers in any given cluster. However, single-worker clusters are also possible – although in this case, if the only real-time worker produces incorrect results, there will be no other workers to challenge this.

Every client request comes in as a transaction issued to one of the Tendermint endpoints. Tendermint combines incoming transactions into blocks and replicates those blocks across the cluster. Each real-time worker applies blocks to the internal virtual machine and tries to reach consensus with other workers in the cluster on how the state should advance. Every worker that accepts the state transition bears full responsibility if the transition turns out to be incorrect. If a real-time worker does not agree with the state transition, it is free to immediately raise a dispute, which is resolved through the verification game mechanism (§4).

State machine. The state machine glues everything together: it receives new transaction blocks from Tendermint, forwards them to the virtual machine and gets back the new virtual machine state hash, uploads transaction blocks to the decentralized storage and consensus metadata to the interim metadata storage, and, finally, computes and returns to Tendermint the new *application hash.*

Every new transaction block B_i that the state machine receives is first handed over to the virtual machine that actually performs the computation. By applying the block to the virtual machine state VM_i, the new state VM_{i+1} and its hash $\mathsf{H}_{\text{VM}_{i+1}}$ are produced:

$$\mathsf{VM}_{i+1} = \mathit{vmcall}(\Psi, \mathsf{VM}_i, \mathsf{B}_i) \tag{5.1}$$

To expose the transaction history for batch validation, real-time clusters store transaction blocks and related metadata to the decentralized storage. The history is used not only by batch validators to inspect virtual machine state transitions but also to restore real-time clusters if any worker in a cluster (or even the entire cluster) goes down.

Once the virtual machine processes the block, the state machine uploads the block to the decentralized storage, which returns a receipt R_i confirming that the block was successfully stored. Then the state machine combines the decentralized storage receipt with the new virtual machine state hash $\mathsf{H}_{\text{VM}_{i+1}}$ to produce the new application hash $\mathsf{H}_{\text{APP}_{i+1}}$:

$$\mathsf{H}_{\text{APP}_{i+1}} = \mathit{hash}(\mathsf{H}_{\text{VM}_{i+1}} \| \mathsf{R}_i) \tag{5.2}$$

The application hash is included in the next block B_{i+1}, so in order to reach consensus, the real-time cluster workers will either have to agree on the new virtual machine state hash $\mathsf{H}_{\text{VM}_{i+1}}$ and the decentralized storage receipt R_i, or open a verification game dispute through the Arbiter contract.

Interim metadata storage. The state machine also uploads the block metadata (hereinafter: *manifest*) to the interim metadata storage, which periodically compacts stored manifests and uploads compaction results to Ethereum. The interim metadata storage runs on top of a distributed hash table: each manifest is replicated to multiple interim metadata storage nodes (hereinafter: *metadata nodes*) that are responsible for the cluster that created the manifest.

Metadata nodes store the chain of the most recent manifests uploaded by each cluster, and verify that the newly uploaded manifests do not create a fork or gap in the chain and are properly signed by real-time workers. Once the manifest chain reaches a certain size, metadata nodes upload the last manifest in the chain as a checkpoint to the Arbiter contract and then truncate the chain. This way, the real-time processing layer avoids overloading Ethereum while making sure that the critical metadata is not lost.

Interim metadata storage is able to survive a great number of malicious nodes. If a malicious metadata node has checkpointed a wrong (but valid) manifest, it takes only a single honest metadata node out of all those serving the same real-time cluster to initiate a dispute through the Arbiter contract. Similarly, if a malicious real-time cluster has uploaded an invalid manifest, it takes a single honest metadata node serving that cluster to dispute it. Finally, if a malicious real-time cluster has created a fork in the chain, it takes two honest metadata nodes serving the cluster that have different chain versions to notice and dispute that.

Client interactions. It should be noted that while clients are able to send transactions to real-time clusters without too much difficulty, receiving and verifying responses is much more involved. Once a transaction sent by the client is processed, the real-time cluster returns to the client just a few chunks of the new virtual machine state.

The client then must download the necessary block manifests from multiple randomly selected interim metadata nodes and then verify that: (1) the transaction was included in a new block; (2)

returned chunks correspond to the new virtual machine state hash; (3) the new virtual machine state hash corresponds to the application hash stored in the next block; (4) real-time workers have reached consensus over both blocks; and finally, (5) both blocks were properly stored in the decentralized storage.

The client needs to interact with multiple metadata nodes because some of them might be malicious and not check the stored manifest chain validity or not checkpoint manifests to Ethereum. Additionally, the client needs to check that security deposits posted by real-time workers and metadata nodes the client interacted with are in possession of the Arbiter contract. Only after performing these checks can the client be certain that incorrect state transitions will eventually be discovered by batch validation and the workers that performed incorrect transitions penalized.

Cluster formation. To allocate a new real-time cluster, a developer first has to prepare and upload to the decentralized storage a backend package with the domain-logic WebAssembly code. The package must include a specially designated gateway function that takes a block of transactions as a single argument, which is called every time a new block is propagated. Next, the developer sends a transaction specifying the number of workers in the cluster, the desired virtual machine memory size, and the hash of the backend package to the Arbiter contract, which then takes care of the cluster formation.

Real-time cluster composition is supposed to be stable and thus should not change much over time. Before joining the cluster, a worker must post a significant security deposit to the Arbiter contract. The deposit is returned only after a cooling-off period, during which the correctness of the state transitions performed by this worker may be disputed. The relative stability of real-time clusters means there are less expensive state resynchronizations when workers join or leave.

5.1 Consensus engine

To synchronize workers, real-time clusters use Tendermint as the consensus engine. Tendermint combines transactions into blocks, where each block additionally carries certain metadata that allows clients to verify the correctness of real-time cluster operations. A simplified Tendermint block definition is provided below:

$$\begin{aligned}
&\mathsf{B} := (\mathsf{header}, \mathsf{txs}, \mathsf{votes}) \qquad (5.3)\\
&\mathsf{header} := (\mathsf{height}, \mathsf{H}_{\textsc{prev}}, \mathsf{H}_{\textsc{app}}, \mathsf{H}_{\textsc{txs}}, \mathsf{H}_{\textsc{votes}})\\
&\mathsf{txs} := (\mathsf{tx}_1, \mathsf{tx}_2, \ldots, \mathsf{tx}_n)\\
&\mathsf{votes} := \{\mathsf{vote}_{\mathrm{r}_1}, \mathsf{vote}_{\mathrm{r}_2}, \ldots, \mathsf{vote}_{\mathrm{r}_m}\}
\end{aligned}$$

Here, height denotes the index of the block in the Tendermint blockchain stored by a real-time cluster. The hash of the previous block header is denoted by $\mathsf{H}_{\textsc{prev}}$, and the application hash obtained by the real-time worker as the result of the previous block execution is denoted by $\mathsf{H}_{\textsc{app}}$. Transaction j in the block is denoted by tx_j, and the vote of the real-time worker rt_s for the previous block header is denoted by $\mathsf{vote}_{\mathrm{r}_s}$. The block header also contains hashes of the block's list of transactions and list of votes, denoted by $\mathsf{H}_{\textsc{txs}}$ and $\mathsf{H}_{\textsc{votes}}$ respectively.

The secret key of the real-time worker rt_s is denoted by $\mathsf{sk}_{\mathrm{rt}_s}$, and the number of worker in the real-time cluster by γ. In a correctly functioning Tendermint blockchain, the following set of equations should hold true:

$$\begin{aligned}
&\forall i : \\
&\quad \mathsf{B}_i \,.\, \mathsf{header} \,.\, \mathsf{height} = i \\
&\quad \mathsf{B}_{i+1} \,.\, \mathsf{header} \,.\, \mathsf{H}_{\text{PREV}} = \mathit{merkle}_{\text{tm}}(\mathsf{B}_i \,.\, \mathsf{header}) \\
&\quad \mathsf{B}_i \,.\, \mathsf{header} \,.\, \mathsf{H}_{\text{TXS}} = \mathit{merkle}_{\text{tm}}(\mathsf{B}_i \,.\, \mathsf{txs}) \\
&\quad \mathsf{B}_i \,.\, \mathsf{header} \,.\, \mathsf{H}_{\text{VOTES}} = \mathit{merkle}_{\text{tm}}(\mathsf{B}_i \,.\, \mathsf{votes}) \\
&\quad \forall s \in [1..m] : \mathsf{B}_i \,.\, \mathsf{vote}_{\mathrm{r}_s} = \mathit{sign}_{\text{tm}}(\mathsf{sk}_{\mathrm{rt}_s}, \mathsf{B}_i \,.\, \mathsf{header} \,.\, \mathsf{H}_{\text{PREV}}) \\
&\quad |\, \mathsf{B}_i \,.\, \mathsf{votes} \,| \geqslant \left\lceil \frac{2}{3} \cdot \gamma \right\rceil
\end{aligned} \tag{5.4}$$

Client interaction protocol. To issue a request to one of the real-time clusters, a client needs to send a transaction tx to the Tendermint endpoint of one of the cluster workers. Upon receiving the transaction, Tendermint will:

- Replicate the transaction to other workers in the cluster.
- Compose a new block B_i from this and other incoming transactions.
- Establish consensus of at least $\lceil \frac{2}{3} \rceil$ workers in the cluster over the order of transactions in block B_i and this block metadata. Votes cast by the cluster workers during the consensus process will be stored in block B_{i+1}.
- Pass block B_i to the state machine and memorize the state machine's output – application hash ${\mathsf{H}_{\text{APP}}}_{i+1}$. The application hash will also be stored in block B_{i+1}.

To verify that the cluster has reached consensus over transaction order in block B_i and that the processing of this block has changed the application state, the client needs to wait for two more blocks: B_{i+1} and B_{i+2} (*Fig. 4*).

Votes stored in block B_{i+1} confirm that the transaction order in block B_i was accepted by the real-time cluster. In addition, votes stored in block B_{i+2} confirm that application hash ${\mathsf{H}_{\text{APP}}}_{i+1}$ (which is stored in block B_{i+1}) was also accepted by the real-time cluster.

Finally, the client needs to verify that transaction tx was really included in block B_i. This is performed by examining hash ${\mathsf{H}_{\text{TXS}}}_i$ of transactions stored in block B_i and the Merkle proof of tx inclusion supplied by the real-time cluster.

5.2 Transaction history storage

For the purposes of this paper, we assume that the decentralized storage exposes the *upload*() function such that for each uploaded byte sequence bs it returns receipt R (which contains the decentralized storage hash H_{ST} of the byte sequence) and signature stsign of the decentralized storage node w responsible for storage:

$$\mathsf{R} := (\mathsf{H}_{\text{ST}}, \mathsf{stsign}) \tag{5.5}$$

$$\begin{aligned}
&\forall\, \mathsf{R} = \mathit{upload}(\mathsf{bs}) : \\
&\quad \mathsf{R} \,.\, \mathsf{H}_{\text{ST}} = \mathit{merkle}_{\text{st}}(\mathsf{bs}) \\
&\quad \mathsf{R} \,.\, \mathsf{stsign} = \mathit{sign}_{\text{st}}(\mathsf{sk}_{\mathrm{w}}, \mathsf{R} \,.\, \mathsf{H}_{\text{ST}})
\end{aligned} \tag{5.6}$$

Tendermint blocks are stored in the decentralized storage as multipart *bundles*: block metadata and transactions are stored separately in the decentralized storage and are connected by storage receipts. To upload a block to the decentralized storage, the real-time cluster first uploads list of

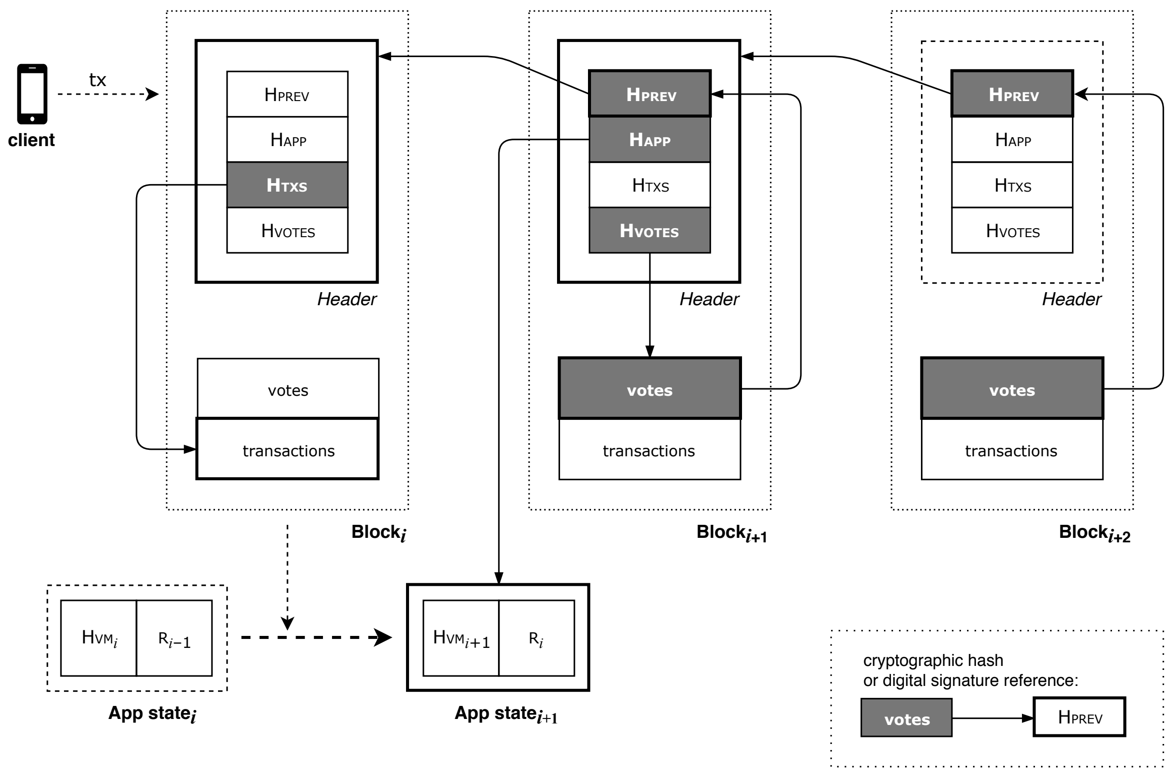


Fig. 4: Blocks required to verify Tendermint consensus over tx processing.
Only the references required for the verification are depicted.

transactions txs_i and collects receipt $\mathrm{R}_{\mathrm{TXS}_i}$. Then the cluster uploads to the decentralized storage the block manifest M_i that holds the Tendermint block header $\mathsf{B}_i\,.\,\mathsf{header}$, workers' votes $\mathsf{B}_i\,.\,\mathsf{votes}$, the virtual machine state hash $\mathrm{H}_{\mathrm{VM}_i}$, and receipts $\mathrm{R}_{\mathrm{TXS}_i}$ and $\mathrm{R}_{\mathrm{PREV}_i}$ issued by the decentralized storage for the list of transactions txs_i and the previous block manifest M_{i-1}:

$$\begin{aligned}
&\mathsf{M}_i = (\mathsf{B}_i\,.\,\mathsf{header}, \mathsf{B}_i\,.\,\mathsf{votes}, \mathrm{H}_{\mathrm{VM}_i}, \mathrm{R}_{\mathrm{PREV}_i}, \mathrm{R}_{\mathrm{TXS}_i}),\ \text{where} \qquad (5.7)\\
&\mathrm{R}_{\mathrm{PREV}_i} = \mathit{upload}(\mathsf{M}_{i-1})\\
&\mathrm{R}_{\mathrm{TXS}_i} = \mathit{upload}(\mathsf{txs}_i)
\end{aligned}$$

This way, it becomes possible to fetch the entire chain of blocks stored in the decentralized storage by having the decentralized storage receipt point to the last block in the chain (*Fig. 5*).

Mismatched hashes attack. It should be noted that the decentralized storage hash of the stored content might not correspond to the Tendermint hash. For example, a malicious real-time worker could upload a different transaction list to the decentralized storage than it previously voted for during the consensus process:

$$\begin{aligned}
&\mathsf{B}_i\,.\,\mathrm{H}_{\mathrm{TXS}} = \mathit{merkle}_{\mathrm{tm}}(\mathsf{txs}_i^1)\ \ \&\ \ \mathrm{R}_{\mathrm{TXS}_i}.\,\mathrm{H}_{\mathrm{ST}} = \mathit{merkle}_{\mathrm{st}}(\mathsf{txs}_i^2),\ \text{where} \qquad (5.8)\\
&\mathsf{txs}_i^1 \neq \mathsf{txs}_i^2
\end{aligned}$$

This might happen because Merkle trees employed by the decentralized storage and Tendermint might be different in chunk size or in used hash function. In other words, $\mathit{merkle}_{\mathrm{tm}}(\mathsf{txs})$ is not necessarily equivalent to $\mathit{merkle}_{\mathrm{st}}(\mathsf{txs})$.

It is possible to notice and challenge such an inconsistency; however, a challenging party needs to download the entire content to which two mismatching hashes are claimed to belong. Consequently, no hash consistency checks are performed by clients: instead, it is expected that batch validators will perform the verification and submit a dispute if an inconsistency is found.

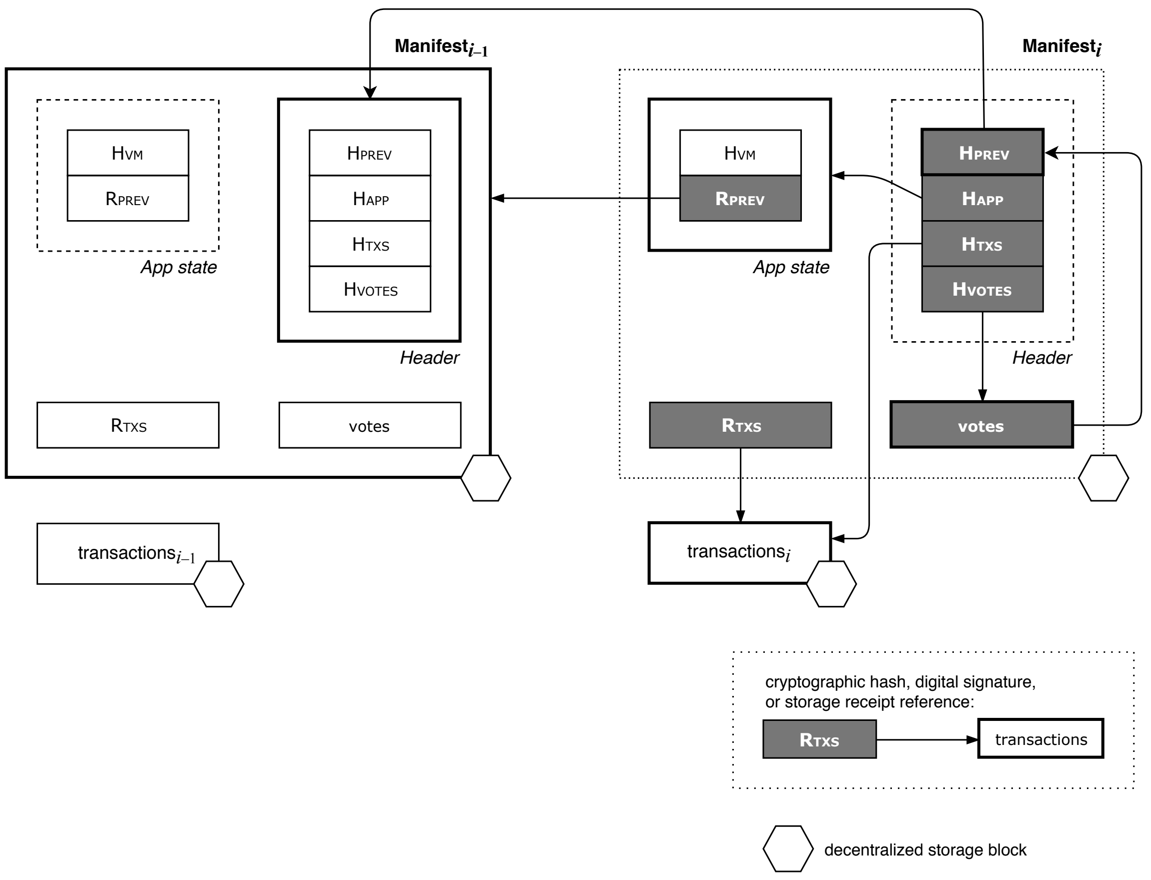


Fig. 5: Decentralized storage block layout.

5.3 Interim metadata storage

Merely uploading blocks to the decentralized storage is not sufficient to ensure that the transaction history will be verified by batch validators. To fetch the transaction history chain, batch validators require the last block receipt, which should be stored in a trustworthy entity. If the entity storing receipts is not reliable, a malicious cluster might conspire with it to replace genuine receipts pointing to the incriminating history with bogus ones pointing to a fake but consistent history.

The Arbiter contract could be the entity storing block receipts. In this case, once a block is uploaded to the decentralized storage, the real-time cluster would also send a transaction with the block manifest to the Arbiter contract. The Arbiter contract would verify that the manifest height is an exact increment of the previously stored manifest height and that the two manifests are linked correctly. This would prevent forks or gaps in the manifest chain, and a malicious cluster would not be able to hide incriminating history. A client in this case would only need to verify that the Arbiter contract has received the required manifests to ensure that state transitions performed by the real-time cluster will be verified by batch validators.

However, due to performance reasons, it is not possible to upload every block manifest to Ethereum. To keep real-time operations latency low, real-time clusters are expected to produce new blocks with a sub-second delay. For a certain number of clusters, this could potentially generate a load that the current Ethereum design is unable to accommodate. To reduce the load, only one in every several block manifests is checkpointed to Ethereum; the rest of the manifest chain is cached in the interim metadata storage, which is built on top of the Kademlia DHT.

Kademlia overview. Kademlia [14] is a distributed hash table (DHT) that arranges records based on their proximity to DHT node identifiers. Node identifiers and record keys use 160-bit values; XOR distance metric is used to define proximity. For a node with the identifier id and the record with the key $\mathsf{\kappa}$, the distance is calculated as $d_{\oplus}(\mathsf{id}, \mathsf{\kappa}) = \mathsf{id} \oplus \mathsf{\kappa}$. A record in Kademlia is stored only on those nodes that are closest to it according to the XOR distance metric.

Nodes in Kademlia keep information about peers in their lookup tables as well. For each $i \in [0 \mathrel{..} 159]$, every Kademlia node stores a list of peers that have a distance between 2^i and 2^{i+1}. This allows (starting from a random DHT node) for looking up a record by using its identifier with $\mathcal{O}(\log(n))$ complexity, where n is the total number of nodes in the Kademlia network. S/Kademlia [15] – an adaptation of Kademlia for a decentralized environment where some DHT nodes might be malicious – allows this lookup with $\mathcal{O}(w \cdot \log(n))$ complexity, where w is the number of independent lookup paths.

Kademlia is able to preserve records during network reorganization events: if a new node joins Kademlia, it receives replicas of the records it is supposed to store from its neighbors; if a node leaves Kademlia, the replication level of the records it was storing is restored by republishing them to other DHT nodes.

Metadata storage organization. A tail of the manifest chain that each cluster produces is stored by metadata nodes that are closest to that cluster in XOR distance metric (*Fig. 6*). It should be noted that a single metadata node can store multiple independent manifest chains produced by different real-time clusters.

To verify that a manifest produced by the real-time cluster was correctly received by metadata nodes serving this cluster, the client randomly picks multiple metadata nodes and requests the manifests from each of those nodes. If the returned manifests are the same for each metadata node, the client assumes that at least a few metadata nodes out of those selected are honest, and accepts the manifest as properly uploaded to the interim storage. (We discuss security guarantees provided by the interim metadata storage in greater detail in §7.)

Because the efforts required to store manifests is negligible compared to the efforts required to perform actual computations, it is expected that practically every Fluence node will store a few manifest chains in addition to real-time processing or batch validation responsibilities. Furthermore, Fluence nodes are economically incentivized to participate in the interim metadata storage – for correctly checkpointed manifests, metadata nodes receive a certain compensation.

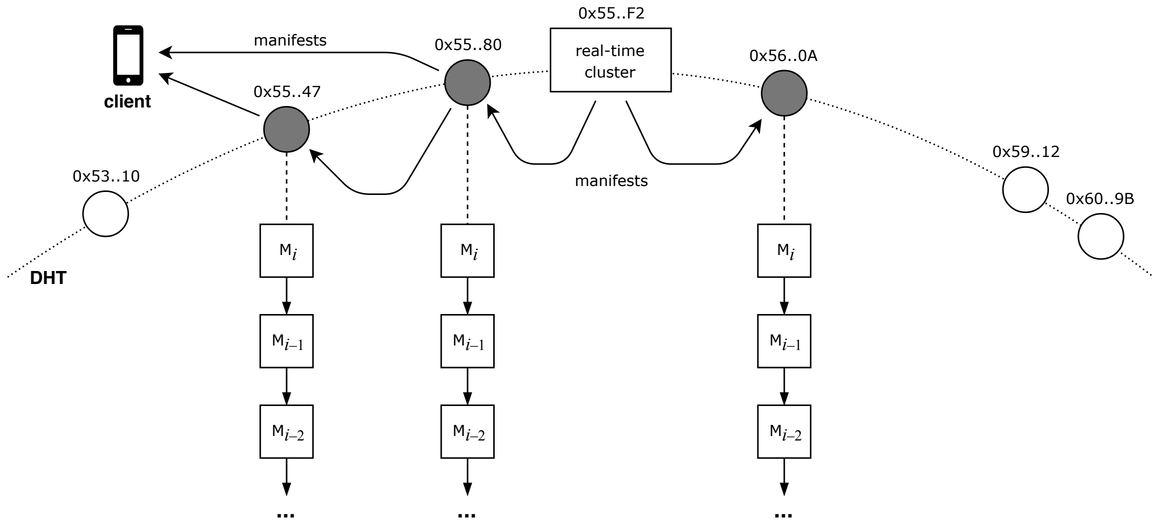


Fig. 6: Metadata nodes holding a particular cluster manifest chain.

Manifests verification. Every time a new manifest M_i is uploaded to a metadata node, the node verifies that the manifest is valid and sequenced correctly with the latest manifest M_{i-1}, which means that the following set of conditions holds:

$$\begin{aligned}
&\mathsf{M}_i\,.\,\mathsf{header}\,.\,\mathrm{H}_{\mathrm{APP}} = \mathit{hash}(\mathsf{M}_i\,.\,\mathsf{header}\,.\,\mathrm{H}_{\mathrm{VM}} \,\|\, \mathsf{M}_i\,.\,\mathrm{R}_{\mathrm{PREV}}) \\
&\mathsf{M}_i\,.\,\mathsf{header}\,.\,\mathrm{H}_{\mathrm{VOTES}} = \mathit{merkle}_{\mathrm{tm}}(\mathsf{M}_i\,.\,\mathsf{votes}) \\
&\forall s : \mathsf{M}_i\,.\,\mathsf{vote}_{\mathrm{r}_s} = \mathit{sign}_{\mathrm{tm}}(\mathsf{sk}_{\mathrm{rt}_s}, \mathsf{M}_i\,.\,\mathsf{header}\,.\,\mathrm{H}_{\mathrm{PREV}}) \\
&|\,\mathsf{M}_i\,.\,\mathsf{votes}\,| \geqslant \left\lceil \frac{2}{3}\cdot\gamma \right\rceil \text{, where } \gamma \text{ is the number of workers in the real-time cluster}
\end{aligned} \tag{5.9}$$

$$\begin{aligned}
&\mathsf{M}_i\,.\,\mathsf{header}\,.\,\mathsf{height} = \mathsf{M}_{i-1}\,.\,\mathsf{header}\,.\,\mathsf{height} + 1 \\
&\mathsf{M}_i\,.\,\mathrm{R}_{\mathrm{PREV}}.\,\mathrm{H}_{\mathrm{ST}} = \mathit{merkle}_{\mathrm{st}}(\mathsf{M}_{i-1}) \\
&\mathsf{M}_i\,.\,\mathsf{header}\,.\,\mathrm{H}_{\mathrm{PREV}} = \mathit{merkle}_{\mathrm{tm}}(\mathsf{M}_{i-1}\,.\,\mathsf{header})
\end{aligned} \tag{5.10}$$

Any metadata node is able to challenge a case by submitting the necessary evidence to the Arbiter contract if the manifest M_i is invalid, not sequenced correctly with the previous manifest M_{i-1}, or if another manifest M'_i also sequenced with the manifest M_{i-1} exists. To find the real-time workers that produced the invalid manifest M_i, two more manifests (M_{i+1} and M_{i+2}) are required (*Fig. 7*).

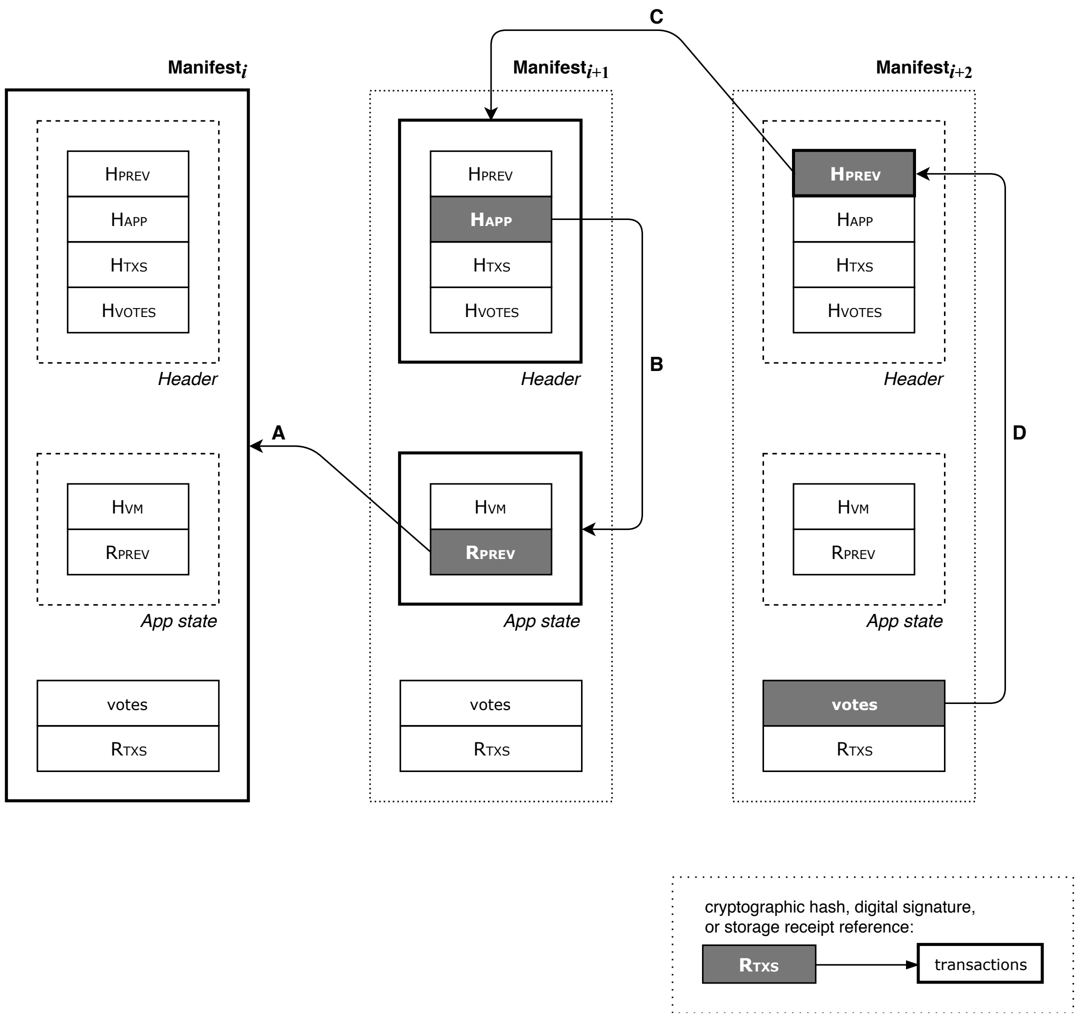


Fig. 7: Manifests linkage.
Only the references confirming that the real-time cluster has accepted the manifest M_i are depicted.

A) The manifest M_{i+1} carries the previous manifest receipt $\mathsf{M}_{i+1}\,.\,\mathrm{R}_{\mathrm{PREV}}$ such that:

$$\mathsf{M}_{i+1}\,.\,\mathsf{R}_{\text{PREV}}.\,\mathsf{H}_{\text{ST}} = \mathit{merkle}_{\text{st}}(\mathsf{M}_i) \tag{5.11}$$

B) The manifest M_{i+1} also carries the block header $\mathsf{M}_{i+1}\,.\,\mathsf{header}$ such that:

$$\mathsf{M}_{i+1}\,.\,\mathsf{header}\,.\,\mathsf{H}_{\text{APP}} = \mathit{hash}(\mathsf{M}_{i+1}\,.\,\mathsf{header}\,.\,\mathsf{H}_{\text{VM}} \,\|\, \mathsf{M}_{i+1}\,.\,\mathsf{R}_{\text{PREV}}) \tag{5.12}$$

C) The manifest M_{i+2} carries the previous header hash $\mathsf{M}_{i+2}\,.\,\mathsf{header}\,.\,\mathsf{H}_{\text{PREV}}$ such that:

$$\mathsf{M}_{i+2}\,.\,\mathsf{header}\,.\,\mathsf{H}_{\text{PREV}} = \mathit{merkle}_{\text{tm}}(\mathsf{M}_{i+1}\,.\,\mathsf{header}) \tag{5.13}$$

D) The manifest M_{i+2} also carries the votes of the real-time workers $\mathsf{M}_{i+2}\,.\,\mathsf{votes}$ such that:

$$\forall s : \mathsf{M}_{i+2}\,.\,\mathsf{vote}_{\text{r}_s} = \mathit{sign}_{\text{tm}}(\mathsf{sk}_{\text{rt}_s}, \mathsf{M}_{i+2}\,.\,\mathsf{header}\,.\,\mathsf{H}_{\text{PREV}}) \tag{5.14}$$

By following the references described in (5.11), (5.12), (5.13), and (5.14), it is possible to show that votes stored in the manifest M_{i+2} confirm the correctness of the manifest M_i. Consequently, metadata nodes never treat a manifest as *accepted* until two subsequent manifests correctly displaying real-time workers' votes are received.

If the manifest M_i is not valid, the workers that voted for it are penalized. If there is a gap in the chain – if the manifest M_i is not sequenced correctly with the manifest M_{i-1} (for example, if their heights are not consecutive) – workers that voted for both manifests are penalized. The same applies in the case where there is a fork in the chain, such as if there are two manifests M_i and M'_i sequenced after the manifest M_{i-1}.

Manifest checkpointing. Every ρ-th manifest is checkpointed to the Arbiter contract. Any metadata node is able to checkpoint the manifest; however, it might happen that one metadata node has uploaded a checkpoint M_x that another metadata node storing the same manifest chain does not agree with (*Fig. 8*).

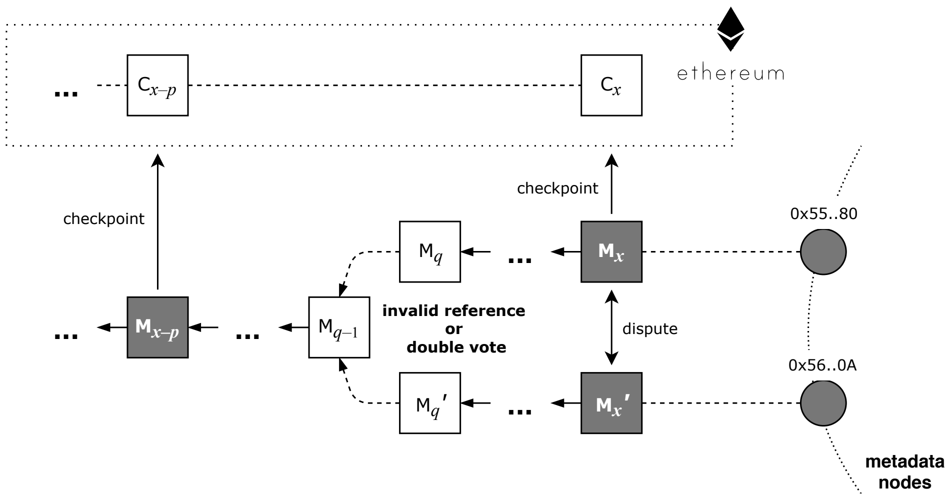


Fig. 8: Manifests checkpoint dispute.

Two cases are possible in this situation:

A) At least one of the metadata nodes has an invalid manifest chain in which one or more manifests are invalid or incorrectly sequenced. Consequently, this node is malicious – otherwise it would have complained about this issue to the Arbiter contract.

B) A malicious real-time cluster has created a fork in the manifest chain and uploaded different (but perfectly correct) chain versions to different metadata nodes.

In the case of a disagreement, the second metadata node can submit a dispute by uploading its version of the checkpoint manifest M'_x. To resolve the dispute, an approach similar to the verification game is used: disputing nodes perform a binary search for the first manifest with an index $q \in [x - \rho + 1 \,..\, x]$ for which they have a disagreement. Clearly, the search can be performed with $\mathcal{O}(\log(\rho))$ complexity and produces two manifests $\mathsf{M}_q \neq \mathsf{M}'_q$.

Manifests M_q, M'_q, and M_{q-1} are uploaded by disputing nodes to the Arbiter contract, which performs the final dispute resolution. If one of the manifests M_q, M'_q is invalid or incorrectly sequenced with the manifest M_{q-1}, then the metadata node that accepted such manifest is penalized. Otherwise, it is obvious that there is a fork in the manifest chain, and the real-time cluster that produced the fork is penalized.

5.4 Client workflow

The normal request-response client workflow is depicted in *Fig. 9* and can be described roughly as follows:

1. The client sends the transaction tx to the real-time cluster, which includes this transaction in the block B_i.
2. Each real-time worker delivers the block B_i to the virtual machine, which updates the virtual machine state from VM_i to VM_{i+1}.
3. Each real-time worker splits the block B_i into the manifest M_i and the transactions list txs_i, which are then uploaded by the real-time cluster to the decentralized storage. The decentralized storage stores the data and returns the receipt R_i issued for the manifest M_i.
4. The real-time cluster uploads the manifest M_i to the interim metadata storage. The interim metadata storage waits for two more manifests (M_{i+1} and M_{i+2}) to verify and accept the manifest M_i.
5. The receipt R_i and the new virtual machine state hash $\mathsf{H}_{\textsc{vm}_{i+1}}$ are combined to produce the new application hash $\mathsf{H}_{\textsc{app}_{i+1}}$, which will be included in the next block B_{i+1}.
6. The client waits until blocks B_{i+1}, B_{i+2}, B_{i+3}, B_{i+4}, and B_{i+5} are created and their manifests are also uploaded to the interim metadata storage.
7. The metadata storage accepts manifests M_i, M_{i+1}, M_{i+2}, and M_{i+3}.
8. The client requests from the real-time cluster the chunk η of the virtual machine state VM_{i+1}.
9. The client requests from the interim metadata storage manifests M_i, M_{i+1}, M_{i+2}, and M_{i+3} and confirms with the metadata nodes that these manifests are accepted.
10. The client verifies that the transaction tx was included in the block B_i by checking the Merkle proof of tx inclusion into the transaction list hash $\mathsf{H}_{\textsc{txs}_i}$, which is stored in the manifest M_i.
11. The client verifies that the chunk η corresponds to the new virtual machine state VM_{i+1} by checking the Merkle proof of η inclusion into the virtual machine state hash $\mathsf{H}_{\textsc{vm}_{i+1}}$, which is stored in the manifest M_{i+1}.
12. The client verifies that the real-time cluster correctly voted for manifests M_i and M_{i+1}. For the manifest M_i, this is performed using manifests M_{i+1} and M_{i+2}; for the manifest M_{i+1}, using manifests M_{i+2} and M_{i+3}.
13. Finally, the client verifies security deposits of the real-time cluster workers, metadata nodes, and the decentralized storage nodes that participated in the process.

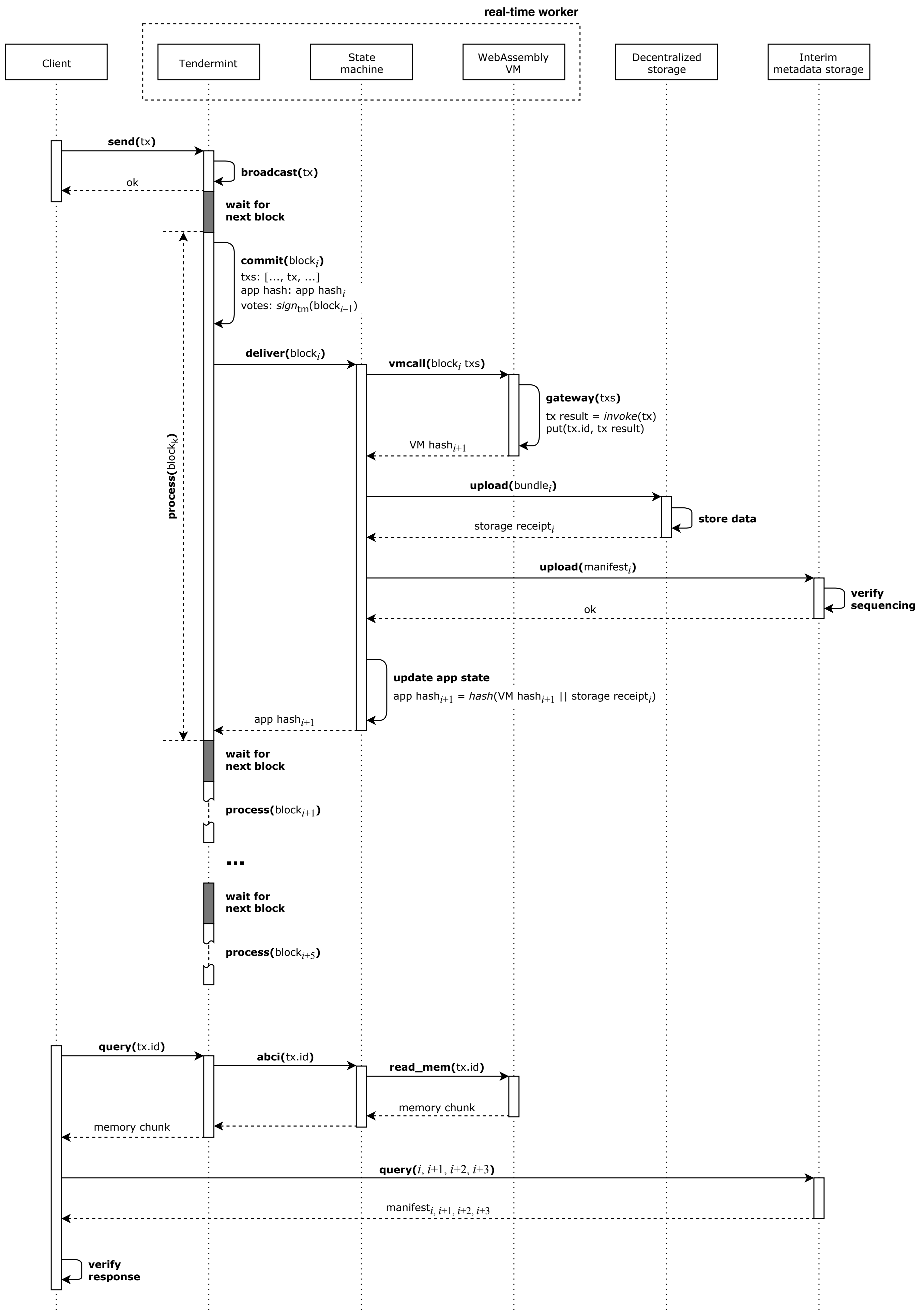


Fig. 9: Normal client workflow.

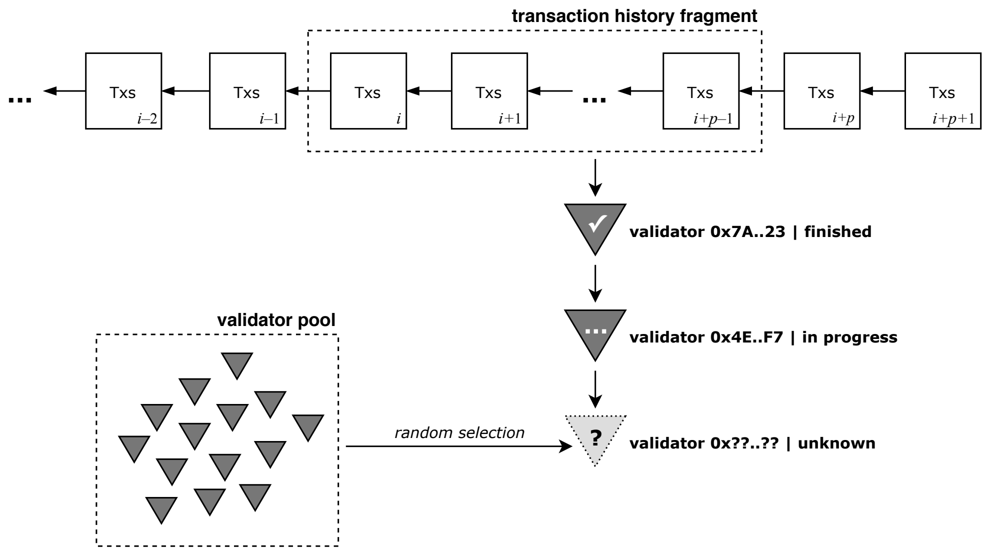
Once the described request-response procedure is completed, the client should have confidence that the submitted transaction was included in the new block and that the received response corresponds to the new state that the cluster claims to be the block processing result. The client should also be certain that the evidence necessary to penalize a maliciously acting cluster is stored in the decentralized storage and the interim metadata storage.

6 Batch validation layer

In the Fluence network, batch validators form the *security layer*, the fundamental purpose of which is to repeat computations performed by real-time clusters and then confirm or disprove correctness of virtual machine state transitions.

In order to achieve that, the history of transactions processed by real-time clusters is split into fragments, where checkpointed block manifests indicate fragment boundaries. Each fragment is verified independently by one or more batch validators.

Validation structure. Batch validators for each fragment are randomly chosen from the pool of all available validators in the network. This way, a real-time cluster is unable to predict which validators will be verifying its operations. Moreover, a batch validator verifying a transaction history fragment does not know which batch validator will be reverifying its decisions.



transaction history fragment
Txs $i-2$
Txs $i-1$
Txs i
Txs $i+1$
Txs $i+p-1$
Txs $i+p$
Txs $i+p+1$
validator 0x7A..23 | finished
validator 0x4E..F7 | in progress
validator pool
random selection
validator 0x??..?? | unknown

Fig. 10: Batch validation overview.

Batch validations of the same transaction history fragment are always performed sequentially. Each batch validator is required to deliver a proof of independent execution (PoIE) similar to the one described by *Drake* in [26]. This is done to force batch validators to perform computations independently instead of simply accepting results produced by the real-time cluster or confirmed by the previous batch validator.

Neither real-time clusters nor batch validators know in advance the number of the batch validations that will be performed for any specific history fragment. Instead, once one batch validation is completed, the Arbiter contract probabilistically decides whether another validation should be performed. This way, a batch validator never knows if it is the last one in the validation sequence, so a lazy validator is not able to skip the computation without potential consequences.

This design ensures that no ahead-of-time collusion to accept an incorrect history fragment is possible between malicious real-time clusters and batch validators or between malicious batch validators. Consequently, a misbehaving node has to carefully weigh the potential profit obtained by propagating incorrect results against the general probability of being caught and losing a security deposit.

While decisions to perform a validation are made non-deterministically, developers do haves control over the probability by which the Arbiter contract should decide to set up another batch validation. By manipulating this probability, the developer is able to change the expectation of the number of validations for a particular real-time cluster, which gives the developer control over how the budget is spent.

External state storage. Because batch validators are chosen completely at random, a batch validator will verify a number of different transaction history fragments over the course of its active lifetime. Those fragments will be produced by different real-time clusters, which means that batch validators will not be able to cache real-time cluster states locally.

In principle, by replaying the transaction history from the genesis block, batch validators could restore the virtual machine state; however, with an ever-growing transaction history the state restore process will eventually require an impractical amount of time.

Instead, before verifying a fragment of transaction history, a batch validator downloads the virtual machine state preceding the fragment from the decentralized storage. Then, the validator downloads and applies the history fragment to the virtual machine state. During this process, the validator verifies that each time a transaction block from the history is applied, the results correspond to what was submitted by the real-time cluster.

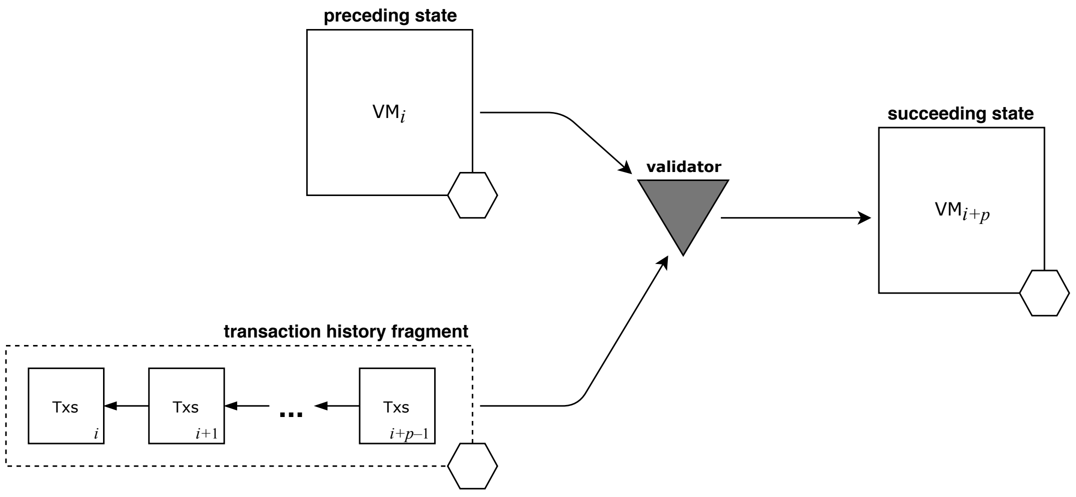


Fig. 11: External state generation.

As a side effect of the history processing, the batch validator generates a new virtual machine state and uploads it to the decentralized storage. This new virtual machine state is used for the validation of subsequent history fragments.

Blocks forming transaction histories and related virtual machine states are not stored in the decentralized storage for an indefinite amount of time because this would cause quadratic storage expenses, assuming the rate of incoming transactions is constant. Instead, only the most recent portion of a transaction history is kept: once a history fragment has been validated enough times, it will be dropped from the decentralized storage. Additionally, the checkpointed manifest that corresponds to that history fragment will be marked as fully verified, and will never be validated again.

6.1 Validator selection

The Fluence network maintains a pool of active batch validators in the Arbiter contract. Once a validator is ready to perform validations, it sends a registration transaction to mark itself as active. To let validators mark themselves as inactive, the Arbiter contract allows sending a deregistration transaction.

Every active validator is obliged to perform the validation if it receives a corresponding request; in the case that a validator fails to complete the validation, it is charged a certain fee, and another validation attempt is made with a different validator.

As we mentioned in §5, every $\mathsf{\rho}$-th manifest is checkpointed to Ethereum. Assume that the manifest M_h is marked as fully verified and the network is validating the next fragment of transaction history corresponding to the manifest $\mathsf{M}_{h+\rho}$.

The real-time cluster identifier is denoted by id, and the number of already performed attempts (including unsuccessful attempts) to validate the history fragment is denoted by v. The probability with which the Arbiter contract decides to perform the validation is denoted by $\mathsf{p_v}$, and the number of active validators in the pool is denoted by N.

To decide whether another batch validation should be performed for the manifest $\mathsf{M}_{h+\rho}$, the Arbiter contract first computes the validation seed w, and then uses this seed to generate a random $x \in [0, 1)$:

$$\begin{aligned} w &= hash(rand() \parallel \mathsf{id} \parallel h + \rho \parallel v) \\ x &= rand(w) \end{aligned} \tag{6.1}$$

If $x < \mathsf{p_v}$, then the validation will be performed. In this case, the Arbiter contract chooses a validator bt_j from the pool of all available validators, where:

$$j = hash(w) \bmod N \tag{6.2}$$

If $x \geqslant \mathsf{p_v}$, then no more validations will be performed, and in this case the manifest $\mathsf{M}_{h+\rho}$ is marked as fully verified by the Arbiter contract. The Arbiter contract also allows the decentralized storage to drop the corresponding transaction history fragment.

6.2 Validation process

Initialization. To validate the fragment of history corresponding to $\mathsf{M}_{h+\rho}$, a validator needs to download the previous virtual machine state snapshot and transaction blocks corresponding to the history fragment.

The previous state snapshot VM_{h-1} is stored in the decentralized storage; its storage receipt is attached to the previous checkpoint C_h in the Arbiter contract. The required transaction blocks can also be downloaded from the decentralized storage by loading the first manifest $\mathsf{M}_{h+\rho}$ attached to the checkpoint $\mathsf{C}_{h+\rho}$ and then following storage links from the manifest $\mathsf{M}_{h+\rho}$ up to and including the manifest M_{h-1} (*Fig. 12*).

Note that the virtual machine state $\mathsf{VM}_{h+\rho-1}$, which the real-time cluster produces by applying the block $\mathsf{txs}_{h+\rho-2}$ to the state $\mathsf{VM}_{h+\rho-2}$, is confirmed by the votes in the manifest $\mathsf{M}_{h+\rho}$ (see equation 5.12, 5.13, and 5.14). Consequently, manifests $\mathsf{M}_{h+\rho-1}$ and $\mathsf{M}_{h+\rho}$ in the validated manifest chain during this particular validation are used as metadata only: transactions corresponding to these two manifests are used for the next history fragment validation.

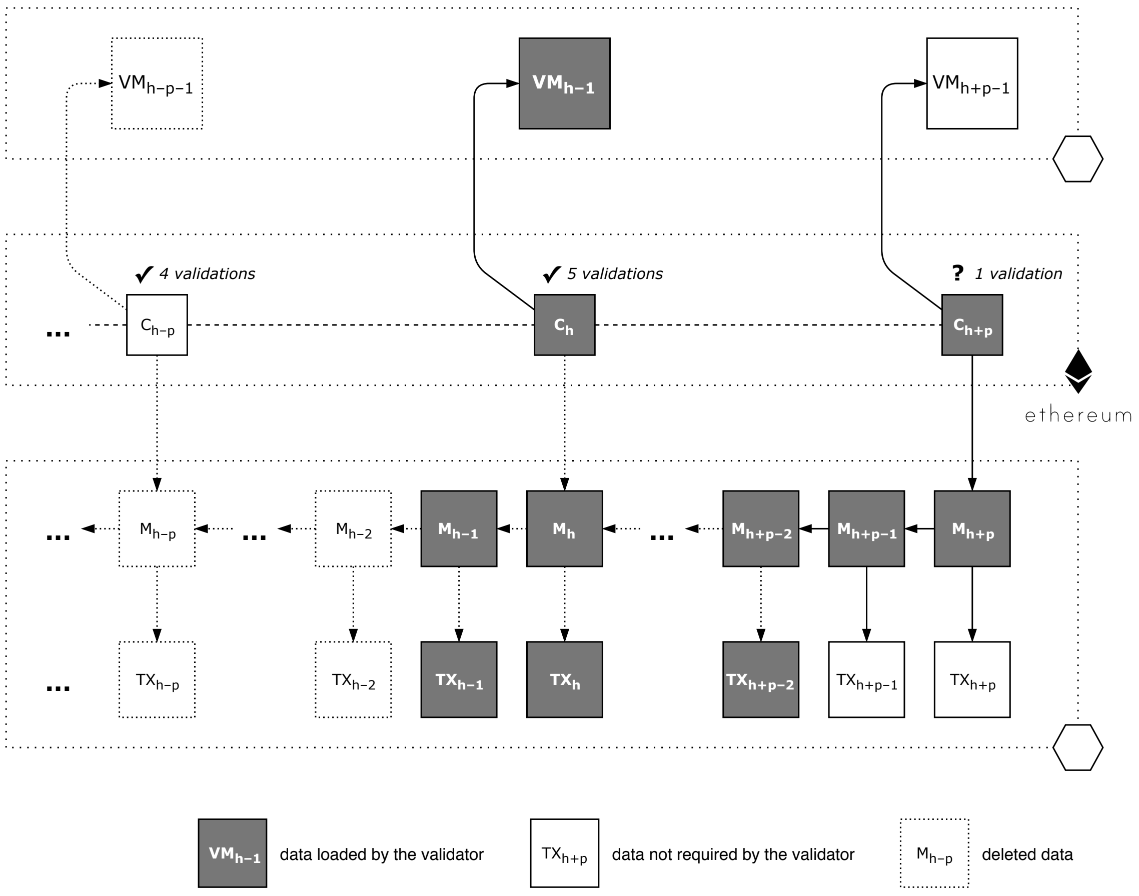


Fig. 12: Data required to validate the checkpoint $\mathsf{C}_{h+\rho}$.

Computation verification. Starting from the virtual machine state VM_{h-1}, the validator consequently downloads and applies transaction blocks $\mathsf{txs}_{h-1}, \mathsf{txs}_{h}, \ldots, \mathsf{txs}_{h+\rho-2}$. This way, the validator successively produces virtual machine states $\mathsf{VM}^*_{h}, \mathsf{VM}^*_{h+1}, \ldots, \mathsf{VM}^*_{h+\rho-1}$.

If the hash $\mathsf{H}_{\text{VM}}{}^*_{h+\rho-1}$ of the virtual machine state $\mathsf{VM}^*_{h+\rho-1}$ does not match the hash $\mathsf{H}_{\text{VM}}{}_{h+\rho-1}$ stored in the manifest $\mathsf{M}_{h+\rho-1}$, the validator opens a history fragment dispute against the real-time workers that cast $\mathsf{M}_{h+\rho}\,.\,\mathsf{votes}$.

To initiate the dispute, the validator submits to the Arbiter contract the manifest $\mathsf{M}_{h+\rho-1}$ (which contains the contested virtual machine state hash $\mathsf{H}_{\text{VM}}{}_{h+\rho-1}$) and the state hash $\mathsf{H}_{\text{VM}}{}^*_{h+\rho-1}$ computed by the validator. The smart contract already possesses challenged real-time workers' votes as they are stored as a part of the checkpoint $\mathsf{C}_{h+\rho}$.

The history fragment dispute consists of two phases. The first phase is similar to the verification game described in §4. The only difference is that instead of searching for the diverged WebAssembly instruction, the validator and one of the challenged real-time workers are searching for the first manifest M_x, where $x \in [h\,..\,h+\rho-1]$ is such that:

$$\mathsf{H}_{\text{VM}}{}^*_{x-1} = \mathsf{M}_{x-1}\,.\,\mathsf{H}_{\text{VM}}$$
$$\mathsf{H}_{\text{VM}}{}^*_{x} \neq \mathsf{M}_{x}\,.\,\mathsf{H}_{\text{VM}}$$

Once the manifest M_x is found, a standard verification game dispute over the transition from the virtual machine state VM_{x-1} to the state VM_x is initiated. In this case, the transaction block txs_{x-1} represents the input L to the virtual machine state.

If the challenged real-time worker loses the dispute, its deposit is slashed and the validator proceeds to challenge the next real-time worker that voted for the contested virtual machine state. Otherwise, if the validator loses the dispute, its deposit is slashed instead.

Finalization. If the validator agrees with the virtual machine state hash $\mathsf{H}_{\mathsf{VM}_{h+\rho-1}}$, it sends a confirmation to the Arbiter contract. Additionally, the validator uploads the corresponding virtual machine state $\mathsf{VM}_{h+\rho-1}$ to the decentralized storage (unless the state was already uploaded by one of the previous validators).

It is worth noting that once the validator has confirmed that it agrees with the state $\mathsf{VM}_{h+\rho-1}$, it is responsible for the correctness of that state. If one of the subsequent batch validators of the same transaction history fragment produces a different state $\mathsf{VM}'_{h+\rho-1}$, it will challenge not only the real-time workers that signed the contested state, but also all of the batch validators that confirmed this state.

6.3 Proof of independent execution

One of the issues that commonly arise when verifying outsourced computations in trustless networks is the so-called *verifier's dilemma* [27]. Applied to our case, it means that a lazy validator might skip replaying the fragment of transaction history and simply agree with results that were submitted by the real-time cluster and confirmed afterward by the validators preceding the lazy one. In this case, the lazy validator is gambling on the correct prior processing of the history fragment.

If the fragment was processed incorrectly, and one of the next validators discovers and challenges this, the lazy validator faces deposit slashing. However, with a growing number of preceding validators the chance of losing a deposit becomes low enough to make the option of skipping the verification attractive.

This is because, in such cases, the chance that at least one of preceding validators has actually verified the computation continually grows. Therefore, the probability that the fragment was processed incorrectly but has gone unchallenged reduces to almost zero. Consequently, at some point, the potential for profit attained by collecting verification rewards for doing nothing free outweighs the expected deposit loss.

6.3.1 TrueBit forced errors approach and its PoIE modification

Forced errors mechanism. To force verifiers to genuinely perform computations, TrueBit uses the "forced errors" approach [6]. With a certain probability, a solver originally performing the computation secretly submits incorrect results. If a verifier accepts such results, the solver uncovers that the computation was intentionally performed incorrectly, and the verifier loses its deposit.

While "forced errors" might work well for the batch-only environment, it does not seem feasible to use this approach for the hybrid Fluence network. A real-time cluster responds to clients before batch validators verify the corresponding transaction history, so if a "forced error" comes into effect, clients start receiving invalid results. To avoid misleading clients, the real-time cluster could reveal to them that this "forced error" is targeted at specific transaction blocks, but this would compromise the security of the TrueBit protocol.

PoIE modification. A modification of the TrueBit protocol proposed by *Koch and Reitwießner* [7] does not rely on the "forced errors" mechanism. Instead, for each computational task it chooses multiple solvers, each of which is required to deliver its personalized "proof of independent execution" (*Drake* [26]) along with the computed results.

To build the proof, each solver uses a different secret seed. After a certain period of time passes, solvers reveal their results and challenge each other using a verification game if their computed results differ. Furthermore, solvers are also able to challenge incorrect proofs of independent execution submitted by other solvers. A solver that loses the challenge has its deposit slashed, while the winner receives a certain fraction of the slashed deposit sum.

In addition to selected solvers, an outside challenger (which any network entity can be) can repeat the computation and contest solvers if they provided incorrect solutions or proofs of independent execution. In this case, the outside challenger is also rewarded with a fraction of the deposit that belonged to the slashed solver.

Because selected solvers know each other ahead of time, it is possible that they might form a "malicious cartel" that tries to push invalid results through. For example, malicious solvers selected to perform a certain computation might collude (or belong to the same party) and produce invalid results only if all solvers that were selected are malicious; otherwise, they will act honestly. In this case, the only protection is provided by outside challengers (which are not known to selected solvers) independently verifying computation results.

However, because outside challengers are rewarded only when they discover an invalid result, it is possible that the expected profit for them will be low or even negative if most nodes in the network are honest. In this case, there might be no outside challengers to repeat computations and contest invalid results submitted by malicious cartels.

The protocol proposed in [7] deals with this issue by allowing cartel participants to challenge cartel results by covertly acting as outside challengers. Because the verification game winner receives some fraction of the slashed deposits, it might be profitable for a cartel participant to challenge results produced by the cartel if the potential reward is higher than the deposit this participant stands to lose.

However, if the outside-world reward for *pushing a counterfeit solution through* is high enough, it might outweigh the reward offered by slashed deposits. In this case, it will not be advantageous for malicious nodes to reveal the conspiracy. Consequently, it is possible that invalid results produced by a malicious cartel will be accepted, unless some entities deploy *pro bono* challengers to keep the network secure.

6.3.2 Fluence PoIE protocol for batch validation

We propose a further modification to the TrueBit protocol that we believe should remove the possibility of forming cartels without controlling a predominant fraction of network nodes.

As we have discussed, batch validations of the same transaction history fragment happen strictly sequentially in the Fluence network. The decision to perform another validation is made probabilistically by the Arbiter contract, and the next validator is selected randomly out of all available validators. This means that validators do not know whether they will later be reverified, or which entity will be performing the reverification.

If a validator agrees with results produced by the real-time cluster, it must submit a proof of independent execution along with the agreement confirmation. This proof is constructed similar to the one described by *Drake* [26]: during the program execution, virtual machine state hashes are computed at randomly sampled moments, and then obtained state hashes are combined into a single hash. The random seed that determines the sampling is different for each validation, so different validators produce different proofs. Additionally, the random seed is generated for a validation only once the previous validation completes. This way the preceding validator does not know which seed will be used by the next validator and can not precompute and share the proof with it.

Each validator is required to produce its own proof of independent execution and to verify the proof produced by the preceding validator. If a validator finds that the preceding validator's proof is incorrect, it can dispute this proof in a way similar to the verification game mechanism. If the validator wins the dispute, it receives a certain fraction of the deposit that belonged to the losing validator, while the rest of this deposit is burned.

Formalized PoIE construction. Proofs of independent execution are constructed by batch validators as follows. Assume that the batch validator bt_j is j-th in the sequence of validators verifying the transaction history fragment. Once the previous validator bt_{j-1} commits its validation results, the Arbiter contract publicly generates a random validation seed λ_j that the validator bt_j uses to compute its personalized proof of independent execution.

Assume that the preceding virtual machine state for the transaction history fragment that the j-th validator needs to verify is VM. Assume that by sequentially applying transaction blocks, an honest validator should produce the succeeding state VM'.

Denote by n the total number of WebAssembly instructions that need to be executed by an honest virtual machine to reach the state VM' from the state VM. In this section, we will denote by VM_m the virtual machine state obtained after processing the first $m \leqslant n$ instructions, and will denote the hash of this state by $\mathsf{H}_{\mathsf{VM}_m}$. According to this notation, $\mathsf{VM}_n = \mathsf{VM}'$ and $\mathsf{H}_{\mathsf{VM}_n} = \mathsf{H}_{\mathsf{VM}'}$.

For the validation seed λ_j, the validator bt_j computes an initial sampling index $T_{j,0} = hash(\lambda_j) \bmod \mathsf{P}$, where P is the sampling period. The validator begins to execute the WebAssembly program; once it reaches the state $\mathsf{VM}_{T_{j,0}}$ (assuming that $T_{j,0} \leqslant n$), it stops and computes the corresponding state hash $\mathsf{H}_{\mathsf{VM}_{T_{j,0}}}$.

Then the validator computes the next sampling index $T_{j,1} = T_{j,0} + (\mathsf{H}_{\mathsf{VM}_{T_{j,0}}} \bmod \mathsf{P})$. The validator continues the program execution, and if $T_{j,1} \leqslant n$, once it reaches the state $\mathsf{VM}_{T_{j,1}}$ it computes the state hash $\mathsf{H}_{\mathsf{VM}_{T_{j,1}}}$. Repeating this procedure until the end of the program execution, the validator obtains the sequence of state hashes:

$$\begin{aligned} &(\mathsf{H}_{\mathsf{VM}_{T_{j,0}}}, \mathsf{H}_{\mathsf{VM}_{T_{j,1}}}, \ldots, \mathsf{H}_{\mathsf{VM}_{T_{j,k}}}), \text{ where} \\ &T_{j,m} = T_{j,m-1} + (\mathsf{H}_{\mathsf{VM}_{T_{j,m-1}}} \bmod \mathsf{P}) \text{ for } m \geqslant 1 \end{aligned} \tag{6.3}$$

Once the execution is complete, the validator bt_j computes the proof of independent execution π_j:

$$\pi_j = merkle(\mathsf{H}_{\mathsf{VM}_{T_{j,0}}}, \mathsf{H}_{\mathsf{VM}_{T_{j,1}}}, \ldots, \mathsf{H}_{\mathsf{VM}_{T_{j,k}}}) \tag{6.4}$$

This way, a sequence of batch validators $\mathsf{bt}_1, \mathsf{bt}_2, \ldots, \mathsf{bt}_v$ produces a sequence of independent execution proofs $\pi_1, \pi_2, \ldots, \pi_v$. Each validator bt_j produces the proof π_j and verifies (by recomputing) the proof π_{j-1} produced by the previous validator. If the proof π^*_{j-1} recomputed by the validator bt_j does not match the proof π_{j-1} produced by the validator bt_{j-1}, the validator bt_j opens a dispute.

Analysis. For a high enough sampling period P, generation and verification of independent execution proofs is negligible compared to the main computation. Therefore, if the validator bt_j is honest and actually performed the computation and was able to generate the correct proof π_j, then there is almost a zero economic incentive for it to skip the verification of the proof π_{j-1} produced by the previous validator.

If the validator bt_j is lazy, it does not know whether there will be a subsequent validation and which validator will be selected until bt_j completes the computation. This means that the lazy validator bt_j is not able to reach an agreement with the subsequent validator bt_{j+1} that the latter will skip the verification of the proof π_j until bt_j commits the proof.

However, once the validator bt_{j+1} finishes its proof, it is possible that it will try to blackmail bt_j by demanding payment for silence if the proof π_j turns out to be incorrect. In the case of a winning dispute, bt_{j+1} receives a certain fraction of the deposit that belonged to bt_j; the rest of this deposit is burned. Therefore, we expect that bt_j will have to pay bt_{j+1} at least as much as what bt_{j+1} is entitled to receive after winning the dispute. For the right deposit size, this sum should be much higher than the computational resources required to produce a correct proof, so we expect that rational validators will try to submit correct proofs instead of gambling on later collusion with subsequent validators.

Another option for the lazy validator bt_j is to collude with the validator bt_{j-1} (which it does know) to produce the proof π_j without repeating the computation, thereby reducing computational expenses and sharing in the profits that arise. To achieve that, the validator bt_{j-1} will need to cache hashes of the virtual machine states that are required to produce the proof π_j. However, because the seed λ_j is not known until bt_{j-1} completes the computation, bt_{j-1} will have to cache *all* state hashes for the duration of its computation. For nontrivial computations, this should be a few orders of magnitude more expensive than repeating the computation from scratch, which renders outsourcing the proof construction moot.

So far we have only considered independent validator incentives; however, outcomes are similar for a stingy entity that controls a fraction of network validators and tries to reduce aggregate computational expenses by sharing solutions. Unless this entity controls almost all network nodes, there is always a meaningful chance that a proof it produces will be verified by an uncontrolled validator. Therefore, the stingy entity has to produce correct proofs or it faces a deposit slashing. However, as we have already shown, it is not possible to perform the computation once and use it to build multiple proofs, so such an entity will have to perform the computation every time a validator controlled by it is selected.

A malicious entity that controls a fraction of the network batch validators and real-time workers and attempts to serve counterfeit responses to clients faces the same difficulties. Even if this entity gets to control a real-time cluster, it is not able to control the validators that later verify the transaction history. Therefore, if a certain number of the validators in the network are honest, it is not profitable for a malicious entity to serve clients invalid responses – unless the outside-world reward for doing so is remarkably high.

Comparing our PoIE protocol to the one proposed by *Koch and Reitwießner* [7], we think that our proposal eliminates the need for outside challengers, which are paid only if they catch a malicious or lazy validator and thus do not have a guaranteed income. In our protocol, a validator is always paid for the computation it has performed, and we believe that the amount of unpaid work a validator would need to perform to construct or verify a proof of independent execution can be made negligible compared to the efforts to perform the computation itself. We further believe that this modification should keep a healthy fraction of honest but rational validators in the network, thus keeping it secure.

On the downside, our protocol performs validations strictly sequentially, which might significantly increase the time it takes to verify a fragment of transaction history. We do not think this is a serious issue for the Fluence network because clients are served responses instantaneously by real-time clusters. However, a parallelized version of the validation protocol would reduce the delays before the validators' deposits release. Another downside of the protocol is its probabilistic nature: a developer does not know the exact number of validations that will be performed for a specific history fragment, which makes budget planning hard. In the long run, however, this is alleviated by the fact that the number of performed validations should be close enough to the expected value, which is controlled by the developer.

6.4 Timeouts

Because validations happen strictly sequentially, it is important not to let malicious, slow, or offline validators block the validation process by inactivity. Once selected, a validator is given a certain window of time to verify the solution and produce its proof of independent execution; if the validator fails to deliver the proof within this time, a fraction of its deposit is slashed. To avoid slashing the entire deposit in the case if the validator unintenionally goes offline, once the validator has failed to complete a few validations, it is forcibly marked as inactive and must be manually reconnected by its operator.

The time to deliver the proof is chosen based on the amount of spent fuel, which is declared by the real-time cluster for each manifest stored in the decentralized storage. For the fragment of transaction history corresponding to the checkpoint $\mathsf{C}_{h+\rho}$ and formed by manifests $\mathsf{M}_{h-1}, \mathsf{M}_{h}, \ldots, \mathsf{M}_{h+\rho-2}$, the fuel declared as spent by the real-time cluster can be computed as: $\varphi^{\text{rt}} = \mathsf{M}_{h+\rho-2} \,.\, \mathsf{tf} - \mathsf{M}_{h-1} \,.\, \mathsf{tf}$.

The time during which the batch validator must verify the computation and produce the proof of independent execution is computed as $t_{\pi} = k \cdot \mathsf{c}_{\text{time/fuel}} \cdot \varphi^{\text{rt}}$, where $\mathsf{c}_{\text{time/fuel}}$ is the network-wide constant and k is the scaling coefficient. In other words, the time limit is proportional to the amount of spent fuel that was declared by the real-time cluster.

We expect that for the correct $\mathsf{c}_{\text{time/fuel}}$ calibration and large enough k, the chance that a batch validator having as much computational ability as the one used for calibration will miss the timeout is negligible if the spent fuel amount φ^{rt} is declared correctly. However, it is possible that a malicious real-time cluster could declare the amount of spent fuel φ^{rt} as less than the actual amount of fuel φ^{*} required by the computation.

Nevertheless, in this case, it follows from §4.2 that the batch validator is able to dispute the amount of spent fuel by spending no more than $\varphi^{\text{rt}} + \varepsilon$, where ε is negligible. Consequently, the batch validator should be able to submit a dispute within the timeout t_{π}.

7 Security

TODO: under construction

8 Conclusion

TODO: under construction

Appendices

A Virtual machine implementation

A.1 Miners VM

To execute WebAssembly bytecode, reference Fluence node implementation uses a fork of Asmble – a WebAssembly to Java bytecode compiler. For each WebAssembly module, Asmble produces a separate JVM class file where each WebAssembly method is compiled into a JVM method and global variables are turned into JVM class fields. To invoke a WebAssembly method, a host simply needs to call the corresponding JVM class method. During execution the WebAssembly operand stack roughly maps the JVM stack, and WebAssembly memory can be accessed as a `ByteBuffer` field in loaded objects.

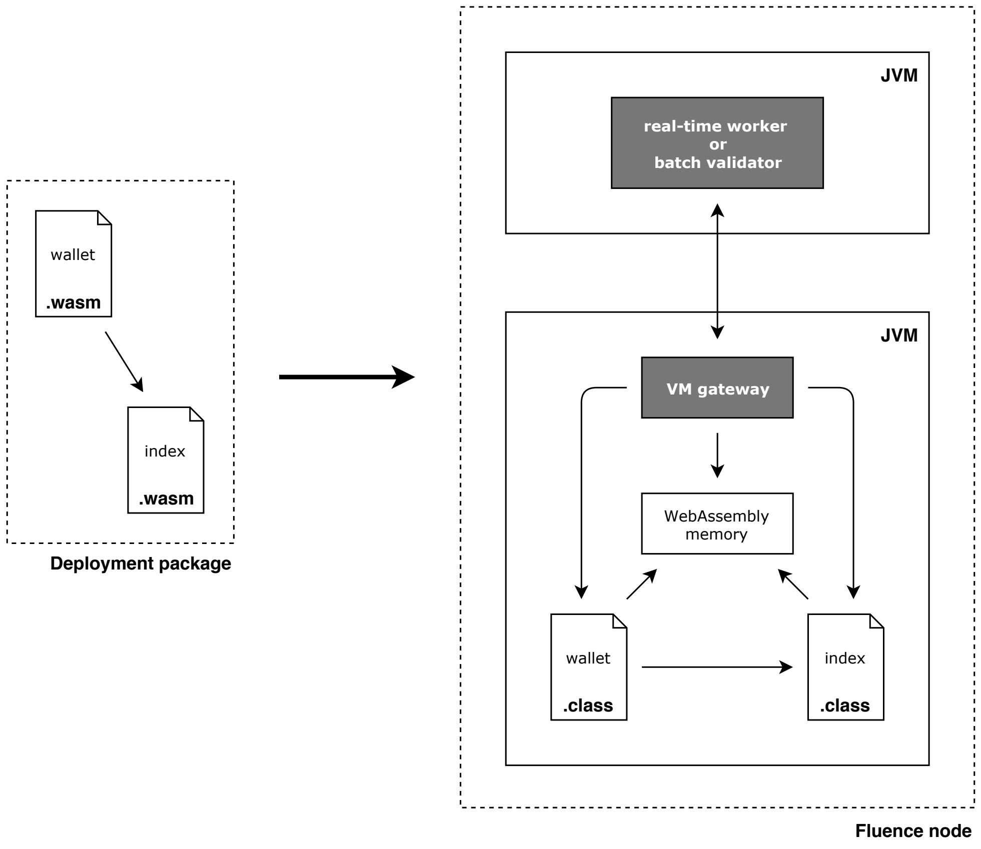


Fig. 13: Asmble integration with Fluence real-time workers / batch validators.

Generated bytecode is executed by a separate virtual machine which is also wrapped in a Docker container, so even if a malicious code escapes from the Asmble VM, it will not be able to access sensitive data stored on the node (*Fig. 13*).

It should be noted that Asmble can be replaced by other virtual machines such as WAVM or Wasmer [28] without much difficulty. In addition, single pass compilers such as V8 Liftoff potentially can be adapted as well to guard against compiler bombs. There were few reasons, however, to choose Asmble for the initial implementation.

First, most WebAssembly instructions can be directly translated into JVM opcodes without additional overhead. This, coupled with the JVM JIT compiler delivers performance similar to WAVM or Wasmer and much higher than internally build interpreters [29].

Second, for the purposes of verification game we need to be able to run the virtual machine in the interpreter mode: stop the execution at arbitrary instructions and access logical WebAssembly constructions such as memory or stack. To achieve that, in Asmble it is possible to extend the generated Java bytecode with shadow stack updates and conditional stops, so performance does not drop significantly during the dispute.

A.2 Arbiter VM

For dispute resolution through Ethereum blockchain we also need to run pieces of WebAssembly code in Ethereum smart contracts. However, this does not require a full scale WebAssembly virtual machine implementation in Solidity. Instead, only a single Wasm instruction execution is needed because the verification game efficiently narrows the trace of the computation. TrueBit has already implemented an on-chain WebAssembly interpreter which likely can be reused for our purposes [30].

A.3 Normal and dispute execution modes

Two types of execution are supported by the virtual machine implementation. The normal mode aims to achieve maximum performance in a situation when no disputes over computation results are present. In this case, the only overhead is in computing the fuel consumption and recalculating the hash of the virtual machine state after each transactions block.

When a dispute happens, the virtual machine switches into the dispute mode. Verification game requires that disputing nodes pass parts of the virtual machine memory and the operand stack to the Ethereum dispute resolution contract. While this would be trivial in the case of a WebAssembly interpreter, Asmble directly maps the WebAssembly operand stack to the JVM stack when compiling WebAssembly code. This approach significantly increases performance, but makes it difficult to extract stack contents in case of dispute. To support dispute mode, we use shadow stack which accumulates stack operands independently of JVM, but adds certain computational overhead.

Simplified, this can be described as follows. Consider a WebAssembly snippet that sums two integer values. First, two constants are pushed to the stack, then the summation operation reads two values from the top of the stack and pushes the result back to the stack:

```
i32.const 8       ;; wasm stack = [8]
i32.const 42      ;; wasm stack = [42, 8]
i32.add           ;; wasm stack = [50]
```

In the normal mode, this WebAssembly code can be directly translated into the following JVM bytecode:

```
bipush 8          ;; jvm stack = [8]
bipush 42         ;; jvm stack = [42, 8]
iadd              ;; jvm stack = [50]
```

However, in the dispute mode we need to explicitly add elements to the shadow stack. Assume that a JVM class `ShadowStack` provides methods that allow to push and pop values to and from the shadow stack:

```
class ShadowStack
{
  private Stack<Object> backingStack;

  // pushes the value to the top of the backing stack
  public void push(int value)

  // removes 2 values from the top of the backing stack
  public void remove2()
}
```

Using `ShadowStack`, it becomes possible to translate the previous WebAssembly snippet into the JVM bytecode, where the shadow stack emulates the WebAssembly stack:

```
;; corresponds to ‘i32.const 8‘
bipush 8          ;; jvm stack = [8]                  | shadow = []
dup               ;; jvm stack = [8, 8]               | shadow = []
aload_0           ;; jvm stack = [@shadow, 8, 8]      | shadow = []
invokevirtual #1  ;; jvm stack = [8]                  | shadow = [8]

;; corresponds to ‘i32.const 42‘
bipush 42         ;; jvm stack = [42, 8]              | shadow = [8]
dup               ;; jvm stack = [42, 42, 8]          | shadow = [8]
aload_0           ;; jvm stack = [@shadow, 42, 42, 8] | shadow = [8]
invokevirtual #1  ;; jvm stack = [42, 8]              | shadow = [42, 8]

;; corresponds to ‘i32.add‘
iadd              ;; jvm stack = [50]                 | shadow = [42, 8]
aload_0           ;; jvm stack = [@shadow, 50]        | shadow = [42, 8]
invokevirtual #2  ;; jvm stack = [50]                 | shadow = []
dup               ;; jvm stack = [50, 50]             | shadow = []
aload_0           ;; jvm stack = [@shadow, 50, 50]    | shadow = []
invokevirtual #1  ;; jvm stack = [50]                 | shadow = [50]
```

Another need present in the dispute mode, but not present during the normal execution is updating the executed instructions counter and the WebAssembly instruction pointer. While in the normal mode JVM maintans its own instruction pointer to step over instructions, in the dispute mode we need to track the WebAssembly instruction pointer to be able to deliver it to the dispute resolution smart contract. Because the JVM instruction pointer not necessarily matches the WebAssembly instruction pointer, we emulate the latter in a way similar to the shadow stack.

A.4 Fuel accounting

The technical difficulty with fuel accounting consists in that separately tracking fuel usage for each instruction might cause an overhead comparable, if not more, to executing the instruction itself. To reduce this overhead, in the normal execution mode the fuel is tracked per basic block. For each block of WebAssembly code that does not contain jumps and therefore can be processed without breaking the execution flow we can calculate how much fuel is needed in the compile time.

Consider the following code snippet:

```
int i = 0;
while (i < 10) {
  i++;
}
```

Directly translated into WebAssembly, it will look as follows:

```
00  block $0
01    loop $1

;;  <block A: begin>
02      get_local $0  ;; 2¢
03      i32.const 9   ;; 2¢
04      i32.gt_s      ;; 3¢
05      br_if $0      ;; 4¢, jumps to #12 if i > 9
;;  <block A: end>

;;  <block B: begin>
06      get_local $0  ;; 2¢
07      i32.const 1   ;; 2¢
08      i32.add       ;; 3¢
09      set_local $0  ;; 2¢
10      br $1         ;; 4¢, jumps to #02
;;  <block B: end>

11    end label $1
12  end label $0
```

In this listing the block `A` (instructions `02`, `03`, `04` and `05`) and the block `B` do not carry any branches or jumps except at the block end. This means it is possible to compute the fuel required for a block execution in compile time and advance the fuel counter by this number before executing the block. For the block `A` the required fuel amount will be $2 + 2 + 3 + 4 = 11¢$, for the block `B` – $13¢$.

TODO: fuel accounting in the dispute mode

A.5 Security

External defelopers can submit arbitrary WebAssembly code to Fluence network, which means that nodes will run untrusted code which might attempt to break out and attack the host system. To mitigate this, we base our defense on layers.

First, WebAssembly specification is designed the way that with proper validation, WebAssembly code will not be able to access host memory or run arbitrary code outside of the virtual machine. We expect that every node will perform validation before running, which can be performed in the linear time and is already supported by Asmble.

Second, Asmble produces class files which are then loaded by the JVM. JVM security manager allows to restrict accessed host functions and data, so even if a malicious code escapes from the WebAssembly VM, it will also need to break out of the JVM to access the host system.

Finally, miners nodes run JVM processes inside Docker containers. Docker allows to control resource usage with cgroups, access to other host processes with namespaces, and even restrict operations such as network or filesystem access from within a container. It can be made even more secure with operating system security modules such as AppArmor or SELinux. This means another layer of defense to break for malicious code.

TODO: gateway data injection into the virtual machine

TODO: separate offsets for function calls

B SDK

TODO: replace Go examples with Rust ones

So far we have treated the virtual machine as a black box that provides only one way to interact with its state – the gateway function. However, most of the time developers of web services have a need to let clients interact with different endpoints that expose domain-specific code. To access an endpoint, a client usually sends a request with a particular payload to the predefined URI, and then awaits for the response carrying the operation results.

To reduce the mismatch between the virtual machine interface and conventional web services composition, Fluence provides a WebAssembly wrapper (*Fig. 14*) that accepts a transactions block through the gateway, parses transactions from it, orders transactions, checks nonces and client authorizations, and, finally, locates requested API functions and invokes them. The wrapper also stores returned results in the virtual machine memory if the invoked API function returns any.

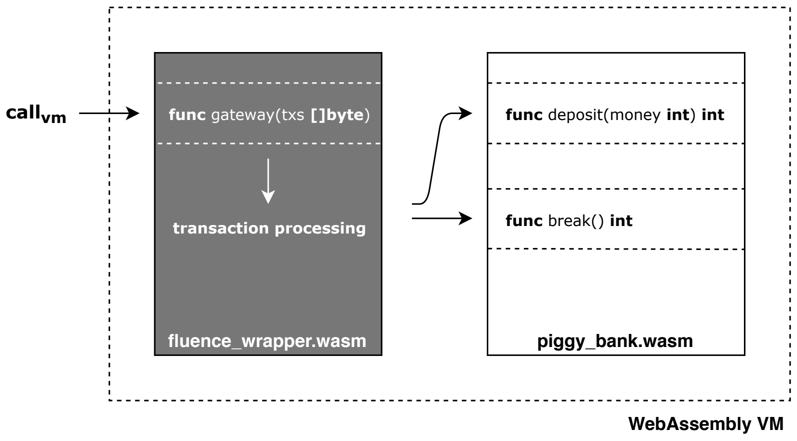


Fig. 14: Fluence WebAssembly wrapper.

The reason to implement transaction processing logic in WebAssembly is that the execution of a WebAssembly code is verified by batch validators and arising disputes are resolved through the verification game. If a malicious node accepts a transaction coming from an unauthorized client, the wrapper code that permitted such action is inspected and offending node is penalized.

However, if any part of transaction processing is located outside of the virtual machine, it is not possible to resolve disputes around, for example, transaction authorization or ordering without having its custom implementation in the Ethereum smart contract.

WebAssembly wrapper implementation also allows to easily swap or extend its logic if necessary. For example, some use cases might demand more complex transactions ordering than provided by the vanilla wrapper. In this case, a developer might implement a new wrapper and package it with the submitted code without the need to roll out a special smart contract for disputes resolution.

Because results returned by invoked API functions are stored in the virtual machine memory, it allows to create a Merkle proof verifying that results match the virtual machine state hash. Consequently, only a fragment of the virtual machine memory has to be sent to the client to return the function invocation results as a response (*Fig. 15*).

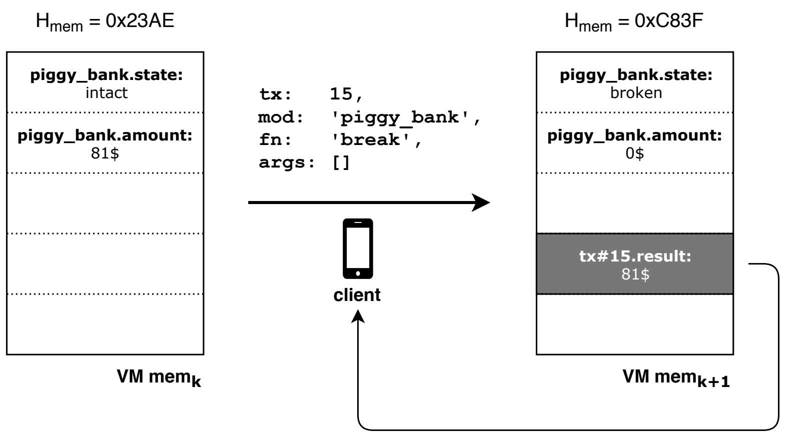


Fig. 15: Virtual machine memory layout.

References

[1] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A Distributed Messaging System for Log Processing, 2011. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf.

[2] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.

[4] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2595631. URL http://doi.acm.org/10.1145/2588555.2595631.

[5] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. URL https://github.com/ethereum/wiki/wiki/White-Paper.

[6] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains, 2017. URL https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf.

[7] Julia Koch and Christian Reitwießner. A Predictable Incentive Mechanism for Truebit, 2018. URL https://arxiv.org/abs/1806.11476.

[8] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with Webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062363. URL http://doi.acm.org/10.1145/3062341.3062363.

[9] Viktor Trón, Aron Fischer, Dániel A Nagy, Zsolt Felföldi, and Nick Johnson. Swap, Swear and Swindle: Incentive System for Swarm, 2016. URL https://swarm-gateways.net/bzz:/theswarm.eth/ethersphere/orange-papers/1/sw^3.pdf.

[10] Filecoin: A Decentralized Storage Network. URL https://filecoin.io/filecoin.pdf.

[11] Arweave lightpaper. URL https://www.arweave.org/files/arweave-lightpaper.pdf.

[12] Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, 2016.

[13] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. URL http://arxiv.org/abs/1807.04938.

[14] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. URL http://dl.acm.org/citation.cfm?id=646334.687801.

[15] Ingmar Baumgart and Sebastian Mies. S/Kademlia: A Practicable Approach Towards Secure Key-based Routing. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems - Volume 02*, ICPADS '07, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1889-3. doi: 10.1109/ICPADS.2007.4447808. URL http://dx.doi.org/10.1109/ICPADS.2007.4447808.

[16] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. URL https://ethereum.github.io/yellowpaper/paper.pdf.

[17] Vitalik Buterin. A simple and principled way to compute rent fees, 2018. URL https://ethresear.ch/t/a-simple-and-principled-way-to-compute-rent-fees/1455.

[18] Asmble. URL https://github.com/cretz/asmble.

[19] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code, 2019. URL https://arxiv.org/abs/1901.09056.

[20] WebAssembly security model, . URL https://webassembly.org/docs/security/.

[21] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security Chasms of WASM, 2018. URL https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf.

[22] Compiling a New C/C++ module to WebAssembly. URL https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm.

[23] Rust and WebAssembly. URL https://rustwasm.github.io/book/.

[24] AssemblyScript. URL https://github.com/AssemblyScript/assemblyscript.

[25] Doom 3 in WebAssembly. URL http://continuation-labs.com/d3wasm/.

[26] Justin Drake. Proof of independent execution, 2018. URL https://ethresear.ch/t/proof-of-independent-execution/1988.

[27] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying Incentives in the Consensus Computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813659. URL http://doi.acm.org/10.1145/2810103.2813659.

[28] Wasmer, . URL https://github.com/wasmerio/wasmer.

[29] A standalone WebAssembly VM benchmark, . URL https://medium.com/fluence-labs/a-standalone-webassembly-vm-benchmark-7a0f701da3d5.

[30] On-chain interpreter for WebAssembly written in Solidity, . URL https://github.com/TrueBitFoundation/webasm-solidity.