Flow: Separating Consensus and Compute
– Execution Verification –

Dr. Alexander Hentschel
alex.hentschel@dapperlabs.com

Maor Zamski   Dieter Shirley   Layne Lafrance
maor@dapperlabs.com   dete@dapperlabs.com   layne@dapperlabs.com

Abstract

Throughput limitations of existing blockchain architectures are well documented and are one of the most significant hurdles for their wide-spread adoption. In our previous proof-of-concept work, we have shown that separating computation from consensus can provide a significant throughput increase without compromising security [1]. In our architecture, Consensus Nodes only define the transaction order but do not execute transactions. Instead, computing the block result is delegated to compute-optimized Execution Nodes, and dedicated Verification Nodes check the computation result. During normal operation, Consensus Nodes do not inspect the computation but oversee that participating nodes execute their tasks with due diligence and adjudicate potential result challenges. While the architecture can significantly increase throughput, Verification Nodes still have to duplicate the computation fully.

In this paper, we refine the architecture such that result verification is distributed and parallelized across many Verification Nodes. The full architecture significantly increases throughput and delegates the computation work to the specialized Execution Nodes and the onus of checking it to a variety of less powerful Verification Nodes. We provide a full protocol specification of the verification process, including challenges to faulty computation results and the resulting adjudication process. Furthermore, we formally prove liveness and safety of the system.

Contents

1 Flow Architecture

In most traditional blockchains, each full node must perform every task associated with running the system. This process is akin to single-cycle microprocessors, where one instruction is executed per step. In contrast, modern CPU design leverages pipelining to achieve higher throughput and scaling.

Rather than asking every node to choose the transactions they will include in a block, compute the block's output, come to a consensus on the output of those transactions with their peers, and finally sign the block, appending it onto the chain, Flow adopts a pipelined architecture. In Flow, different tasks are assigned to specialized node roles: Collection, Consensus, Execution, Verification, and Observation. This design allows high levels of participation in Consensus and Verification by individuals on home internet connections while leveraging large-scale datacenters to do most of the heavy lifting of Execution. Actors participating in Consensus and Verification hold the other nodes accountable with crypto-economic incentives allowing Flow to gain massive throughput improvements without undermining the decentralization or safety of the network. In a system with Consensus and Execution nodes only, Flow achieved a throughput increase by a *factor* of 56 compared to architectures where consensus nodes also perform block computation [1].

In any design where actors other than the consensus nodes perform tasks, correct task execution is not covered by the safety guarantees of consensus. Therefore, the protocol must include dedicated components to ensure secure system operation, even in the presence of a moderate number of malicious participants. In our first white paper [1], we formally analyzed the security implications of delegating tasks to other actors than Consensus Nodes. The central result of our research was that transaction execution can be transferred to one group of nodes (Execution Nodes), and result verification to an independent group (Verification Nodes). The protocol must include the following components to ensure safety:

- The Verifiers must be able to appeal (formally: submit a challenge) to Consensus Nodes if they detect a protocol violation.
- Consensus Nodes must have the means to determine whether the challenger or the challenged is correct (formally: adjudicate the challenge).

When such mechanisms are included, the pipelined architecture is as secure as a blockchain where all tasks are executed by all consensus nodes [1]. In the present paper, we refine the architecture such that result verification is distributed and parallelized across many Verification Nodes. Furthermore, we specify the details of the different challenges and adjudication protocols for execution verification.

Core Architecture Principles

Above, we noted that nodes must be able to appeal to Consensus Nodes if they detect a protocol violation. For an appeal system to provide security guarantees that protect from Byzantine attacks, the system must have the following attributes.

- *Detectable*: A single, honest actor in the network can detect deterministic faults, and prove the error to all other honest nodes by asking them to recreate part of the process that was executed incorrectly.
- *Attributable*: All deterministic processes in Flow are assigned to nodes using a verifiable random function (VRF) [2]. Any detected error can be attributed to the nodes responsible for that process.

- *Punishable*: Every node participating in the Flow network must put up a stake, which is slashed in case the node is found to exhibit Byzantine behavior. Reliably punishing errors via slashing is possible because all errors in deterministic processes are detectable[1] and attributable.
- *Recoverable*: The Flow protocol contains specific elements for result verification and resolution of potential challenges. These elements serves to deter malicious actors from *attempting* to induce errors that benefit them more than the slashing penalty, as the probability of their erroneous result being committed is negligible.

Assumptions

- We solely focus on Proof of Stake blockchains, where all participants are known and each node is authenticatable through its signature.
- Nodes commit to (and stake for) participating in the network for a specific time interval, which we refer to as an Epoch. Epochs are system-wide. While nodes can participate over multiple Epochs, the end of an Epoch is a dedicated point in time for nodes to leave or join the system. Epochs are considered to last for about a week. Technical details for determining the length of an Epoch and a mechanism for Epoch changeover are left for future publications. In this paper, we consider the system running only within one epoch.

Furthermore, we assume
- The existence of a reliable source of randomness that can be used for seeing pseudo-random number generators. We require the random seed to be unpredictable by any individual node until the seed itself is generated and published. Possible solutions include DFINITY's Random Beacon [3] or proof-of-delay based systems [4].
- An aggregatable, non-interactive signature scheme, such as BLS signatures [5].
- Adequate compensation and slashing mechanics to incentivize nodes to comply with the protocol.
- Partially synchronous network conditions with message traversal time bounded by Δ_t. Furthermore, we assume that local computation time is negligible compared to message traversal time.
- In numerous places throughout this paper, we refer to fractions of nodes. This is a short-form of referring to a set of nodes which *hold the respective fraction of stake*. Formally, let $\mathcal{N}^{(r)}$ be the set of all nodes with role r and s_α the stake of node $\alpha \in \mathcal{N}^{(r)}$. A fraction of *at least* κ nodes (with role r) refers to any subset

$$\widetilde{\mathcal{N}} \subseteq \mathcal{N}^{(r)} \quad \text{such that} \quad \sum_{\tilde{\alpha}\in\widetilde{\mathcal{N}}} s_\alpha \geq \kappa \sum_{\alpha\in\mathcal{N}^{(r)}} s_\alpha, \tag{1}$$

for $0 \leq \kappa \leq 1$. For example, stating that "more than $\frac{2}{3}$ of Consensus Nodes have approved of a block" implies that the approving nodes hold more than $\kappa = \frac{2}{3}$ of the Consensus Nodes' accumulated stake.

---

[1] In Flow, nodes check protocol-compliant behavior of other nodes by re-executing their work. In most cases, verification is computationally cheap, with the noticeable exception of computing all transactions in a block. We describe in section 3.4 how verifying the block computation is distributed and parallelized such that each Verifier only has to perform a small fraction of the overall block computation.

1.1 Roles

Roles are a service a node can provide to the network. These are: Collector Role, Consensus Role, Execution Role, Verification Role, and Observer Role. We refer to a network node that performs the respective role as Collector Node, Consensus Node, etc. From an infrastructure perspective, the same hardware can host multiple roles in the network. However, the network treats individual roles as if they are independent nodes. Specifically, for each role a node stakes, unstakes, and is slashed independently. We furthermore assume that a node has it's own independent staking key for each of its roles, even if all roles are hosted on the same hardware. Figure 1 illustrates the messages which nodes of the individual roles exchange during normal operation.

1.1.1 Collector Role

The central task of Collector Role is to receive transaction submissions from external clients and introduce them to the network. Staked nodes are compensated through transaction fees and all roles require a minimum stake to formally participate in that role. When receiving a transaction, a Collector Node checks that the transaction is well-formed. By signing the transaction, the node guarantees to store it until the transaction's result has been sealed. (For details on block sealing, see section 3.5.)

Clusters: For the purpose of load-balancing, redundancy, and Byzantine resilience, Collection Nodes are partitioned into clusters. We require that Collection Nodes are staked equally. At the beginning of an Epoch, each Collection Node is assigned to exactly one cluster. Cluster assignment is randomized using the random beacon output.

Collections: The central task of Collector Nodes is to collect well-formed transactions from external clients and to batch them into collections. When a Collector Node sees a well-formed transaction, it hashes the text of that transaction and signs the transaction to indicate two things: first, that it is well-formed, and second, that it will commit to storing the transaction text until the Execution Nodes have finished processing it. By signing it, the Collector Node guarantees the transaction's storage and will subsequently be slashed (along with the rest of the cluster) if it doesn't produce the transaction.
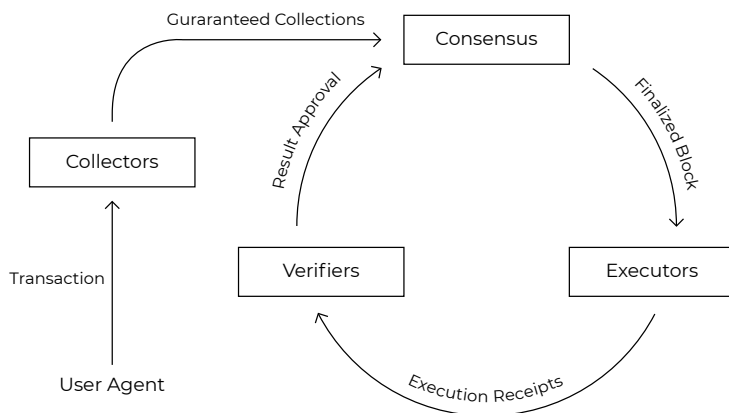

Figure 1: Overview of the node roles and messages they exchange. For simplicity, only the messages during normal operation are shown. Messages that are exchanged during the adjudication of slashing requests are omitted.

Collector nodes share all well-formed transactions they receive among their cluster and collaborate to form a joint collection of transactions. A cluster forms collections one at a time. Before a new collection is started, the current one is closed and send off to the Consensus Nodes for inclusion in a block. Further details on collections are provided in section 3.1.

Collector Nodes in one cluster must agree on the transactions included in the current collection and at what point to close the collection. The determination of when to close a collection is based on a number of factors including token economics, which is out of scope for this paper. This distributed agreement requires the nodes to run a consensus algorithm. Fortunately, the number of nodes in a cluster and the transaction volume to processed by one cluster is moderate[2]. Therefore, established BFT consensus algorithms, such as Tendermint [6, 7, 8], SBFT [9], or Hot-Stuff [10], are fully sufficient.

Security implication: As there are relatively few collectors in each cluster, a cluster may be compromised by malicious actors. The cluster could withhold the collection content that is referenced in a block but whose execution is still pending. The mitigation strategy (Missing Collection Challenge) for this attack is described in section 4.2.

1.1.2 Consensus Role

The Consensus Node's central tasks are the following:

Block Formation: Consensus Nodes form blocks from the collections. Essentially, Consensus Nodes maintain and extend the core Flow blockchain. In Flow, a block defines the transactions as well as the other inputs (incl. the random seed) required to execute the computation, but not the resulting computational state *after* block execution.

An agreement to accept a proposed block needs to be reached by many nodes which requires a Byzantine-Fault-Tolerant (BFT) consensus algorithm [11, 12]. While the specific design of the Flow's consensus system is still subject to active research, we restrict the design space to algorithms with the following guarantees.

- The algorithm is *proven BFT*, i.e., it maintains consistency among honest participants, as long as less than one-third of Consensus Nodes are Byzantine.
- *Safety* is guaranteed even in cases of network asynchrony. Per definition of safety [13, 14], Consensus Nodes might declare a block finalized at different points in time, depending on the amount of information they gathered from other Consensus Nodes. Nevertheless, a safe consensus algorithm guarantees that all honest nodes eventually will declare the same block as finalized.
- *Liveness* is guaranteed in partially synchronous network conditions [15]. As the Fischer-Lynch-Paterson (FLP) theorem states, a fault-tolerant, distributed consensus system cannot guarantee safety and liveness at the same time under asynchronous network conditions [16]. For Flow, we *prioritize safety* over liveness in case of a network split. Consistency of the world-state is more important than forward-progress of the network in extremely rare and adverse circumstances of a large-scale network split.

---

[2] For the mature system, we anticipate on the order of 20 to 50 nodes per cluster.
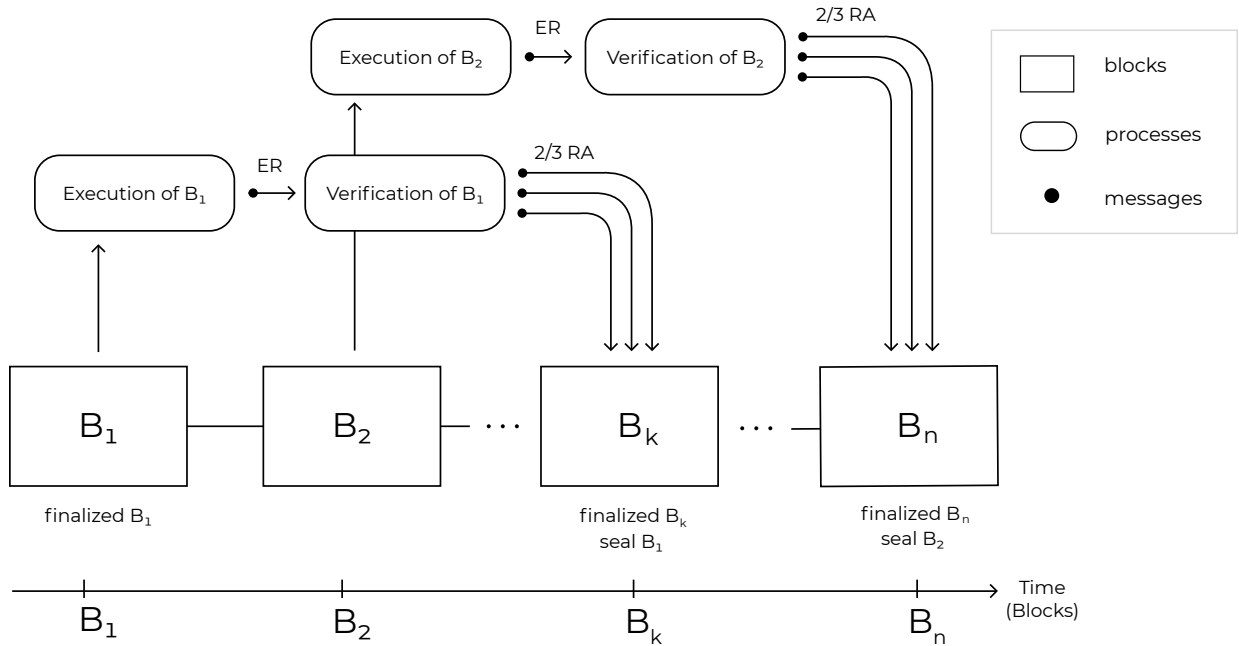
Figure 2: Illustration of the placement of Block Seals within the Flow blockchain. After block B_1 is finalized by the Consensus Nodes, the Execution Nodes process their transactions and issue Execution Receipts (see section 3.3), which are subsequently checked by the Verifier Nodes (see section 3.4). Provided that the checked parts of the computation result are found valid, a Verifier sends a Result Approval back to the Consensus Nodes. As soon as conditions for sealing block B_1 are satisfied (see section 3.5), the seal for the computation result of block B_1 is included in the next block B_k that is generated by the Consensus Nodes.

- A desired core feature of Flow's consensus algorithm is *deterministic finality*. Once a block is finalized by *any* Consensus Node, this commitment will never[3] be changed. Deterministic finality provides the significant advantage that dapp developers do not have to deal with the effects of chain reorganizations.
- *Sustainability*: we do not consider proof-of-work consensus systems due to their exorbitant energy consumption. We focus on Proof of Stake systems only, where computational costs are reduced to necessary elements: primarily cryptography and book-keeping.

Block Sealing: After a block has been computed and the resulting computational state verified, the Consensus Nodes publish a Block Seal for the respective block. A Block Seal contains a commitment to the resulting computational state after block execution. Furthermore, it proves that the computation has been verified by a super-majority of Verifier Nodes.

Consensus Nodes publish the block seals as part of the new blocks they finalize. As executing a block's transactions follows its finalization, the seal for the block's computational result cannot be included in the block itself. Instead, the seal is included in a later block. An illustration is shown in Figure 2.

Tracking Staked Nodes: Consensus Nodes maintain a table of staked nodes including the nodes' current stakes and their public keys. It is important to note that staking balances are tracked solely by the Consensus Nodes and are *not* part of the computational state. Whenever a node's stake

---

[3] Deterministic finality is guaranteed via BFT consensus, unless the system is under active attack of at least one-third of Byzantine actors.

changes, Consensus Nodes publish this update as part of their next finalized block.

During normal operations, staking and unstaking can only take place at the switchover from one Epoch to the next. However, involuntary changes of stake through slashing can occur within an epoch and are accounted for by all honest nodes as soon as they process the block containing the respective stake update.

Slashing: Consensus Nodes adjudicate slashing challenges and adjust the staking balances accordingly.

Security implication: The consensus committee is the central authority in the system. The consensus committee itself must adjudicated challenges against committee members. Safety and liveness are of this process are guaranteed through the BFT consensus algorithm. Flow uses a consensus algorithm with *deterministic finality*. Therefore, dapp developers do not have to deal with the additional complexity of chain reorganization. Our results hold for *any* BFT consensus algorithm with deterministic finality. HotStuff [10, 17] is the leading contender. However, we continue to assess other algorithms such as Casper CBC [18, 19, 20] or Fantômette [21].

1.1.3   Execution Role

Execution Nodes compute the outputs of all finalized blocks they are provided. They then ask the Collector Nodes for the collections containing transaction that are to be executed. With this data they execute the block and publish the resulting computational state as an Execution Receipt. For verification, the computation is broken up into chunks. The Execution Nodes publish additional information (see section 3.3) in the Execution Receipt about each chunk to enable Verification Nodes to check chunks independently and in parallel.

The Computation Nodes are primarily responsible for Flow's improvements in scale and efficiency because only a very small number of these powerful compute resources are required to compute and store the canonical state.

Security implication: Malicious Execution Nodes could publish faulty Execution Receipts. The protocol for detecting incorrect execution results is covered in section 3.4 and the adjudication process for the challenges in section 4.1.

1.1.4   Verification Role

Verification Nodes check the computation from the Execution Nodes. While each node only checks a small number of chunks, all Verification Nodes together will check all chunks with overwhelming probability. For each chunk, a Verification Node publishes a Result Approval, provided it agrees with the result. The Execution Receipt and the Result Approval are required for the block to be sealed.

Security implication: Like most blockchains, Flow has to address the Verifier's Dilemma [22]. In a system where workers are producing results and Verifiers are confirming result correctness, there is an incentive for Verifiers to approve results without expending the work of checking. This conflict of incentives is at the heart of the Verifier's Dilemma. It persists even the worker and Verifiers are *not* colluding, so additional Verifiers do not help. For Flow, we developed Specialized Proofs of Confidential Knowledge (section 2.1) to overcome the Verifier's Dilemma (section 3.4.2 for more details).

1.1.5 Observer Role

Observer Nodes relay data to protocol-external entities that are not participating directly in the protocol themselves.

1.2 Locality of Information

- Correctness of information is cryptographically verifiable using on-chain information
- Flow blockchain is, from a data perspective, *not* self-contained. For example, cryptographic hashes of the computational state are included in blocks, but the state itself is not. This implies that anyone can verify the integrity of any subset of the computational state using the hashes in the Flow blockchain (and merkle proofs which must be provided alongside the actual data). However, it is impossible to extract the state itself from the core blockchain.
- Computational state is local to Execution Nodes
- Reference to the information holder is guaranteed by the holder's signature

1.3 Computational State vs. Network Infrastructure State

An important conceptual difference in Flow is handling information that pertains to the computational state vs. that of the network infrastructure itself. While the computational state is held by the Execution Nodes, the network's infrastructure state is maintained by the Consensus Nodes.

To illustrate the difference, consider the situation where the nodes in the network do not change (nodes never leave, join, or change stake). However, the transactions executed by the system will modify register values, deploy smart contracts, etc. In this setting, only the computational state changes. Its integrity is protected by the verification process (see section 3.4 for details).

In contrast, let us now consider a situation where no transactions are submitted to the system. Blocks are still produced but contain no transactions. In this case, the system's computational state remains constant. However, when nodes leave or join, the state of the network infrastructure changes. The integrity of the network state is protected by the consensus protocol. To modify it, more than $^2/_3$ of Consensus Nodes must approve it.

1.3.1 Staking and Slashing

The network state itself contains primarily a list of all staked nodes which contains the node's staking amount and its public staking key. Updates to the network state are relevant for all nodes in the network. Hence, Consensus Nodes publish updates directly as part of the blocks they produce. Furthermore, slashing challenges are directly submitted to Consensus Nodes for adjudication. (For example, section 4.1 provides a detailed protocol for challenging execution results.) As Consensus Nodes maintain the network state, including staking balances, they can directly slash the stake of misbehaving nodes, without relying on Execution Nodes to update balances.

2 General Techniques

In this section, we describe methods and techniques used across different node roles.

2.1 Specialized Proof of Confidential Knowledge (SPoCK)

A SPoCK allows any number of provers to demonstrate that they have the same confidential knowledge (secret ζ). The cryptographic proof does not leak information about the secret. Each prover's SPoCK is specialized to them, and can not be copied or forged without possession of the secret.

The SPoCK protocol is used in Flow to circumvent the Verifier's Dilemma [22] (section 3.4.2). The protocol prevents Execution or Verification Nodes from copying Execution Results or Result Approvals from each other. Thereby, actors cannot be compensated for work they didn't complete in time. In Flow, the secret ζ is derived from the execution trace of the low-level execution environment (e.g., the virtual machine). Executing the entire computation is the cheapest way to create the execution trace, even when the final output of computation is known.

Formally, the SPoCK protocol provides two central guarantees:

1. An arbitrary number of parties can prove that they have knowledge of a shared secret ζ without revealing the secret itself.
2. The proofs can be *full revealed*, in an arbitrary order, without allowing any additional party to pretend knowledge of ζ.

The SPoCK protocol works as follows.

- Consider a normal blockchain that has a transition function $t()$ where $S' = t(B, S)$, with B being a block of transactions that modify the world state, S as the state before processing the block, and S' as the state after. We create a new function $\tilde{t}()$ that works the same way, but has an additional secret output ζ, such that $(S', \zeta) = \tilde{t}(B, S)$ for the same S, S', and B as $t()$. The additional output, ζ, is a value *deterministically* derived from performing the computation, like a hash of the CPU registers at each execution step, which can't be derived any more cheaply than by re-executing the entire computation. We can assume the set of possible values for ζ is very large.
- An Execution Node, Alice, publishes a signed attestation to S' (a merkle root of some sort), and responds to queries about values in S' with merkle proofs. Additionally it publishes a SPoCK derived from ζ.
- A Verifier Node, Bob verifies that S' is an accurate application of $t(B, S)$, and also publishes its own SPoCK of ζ.
- An observer can confirm that both SPoCKs are derived from the same ζ, and assume that Bob actually verified the output with high probability[4] and didn't just "rubber stamp" the result. This doesn't provide any protection in the case where Alice and Bob are actively colluding, but it does prevent a lazy node from "confirming" a result without actually knowing that it is correct.
- A SPoCK is created as follows:
  - Use ζ (or a cryptographic hash of ζ) as the seed for a deterministic key generation process, generating a public/private key pair (pk, sk).

---

[4]We assume that honest nodes will not accept unproved values for ζ, because they would be slashed if the ζ-value was incorrect. Therefore, the observer can assume that statistically more than $\frac{2}{3}$ of the SPoCKs have been obtained by truthful re-computation of the respective chunks.

– Use the private key `sk` to sign your public identity (such as a node `ID`), and publish the signature along with the deterministically generated public key `pk`:

$$\text{SPoCK} \quad Z = (\text{pk}, \text{sign}_{\text{sk}}(\text{ID})) \tag{2}$$

All observers can verify the signatures to see that both Alice and Bob must have access to the private key `sk`, but the signatures themselves don't allow recovery of those private keys. Alice and Bob must both have knowledge of the same underlying secret ζ used to generate the private key.

In order t seal a block, several SPoCKs have to be validated for each chunk. Benchmarks of our early BLS implementation indicate that all proofs for sealing a block can be verified on a single CPU core in the order of a second. Parallelization across multiple CPU cores is straight forward. Verification time can be further reduced by utilizing vectorized CPU instructions such as AVX-512 or a cryptographic coprocessor.

3 Transaction Processing Flow

In this section, we present a formal definition of the transaction processing flow. Figure 3 provides a high-level overview of the individual stages which we discuss in detail in sections 3.1 – 3.5. For conciseness and clarity, we focus on the core steps and omit protocol optimizations for conservation of bandwidth, run-time, etc. In section 3.6, we formulate Theorem 2 which proves that correctness of a sealed computation result is probabilistically guaranteed even in the presence of Byzantine actors.

To formally define the messages that nodes exchange, we use Protobuf-inspired[5] pseudo-code [23].
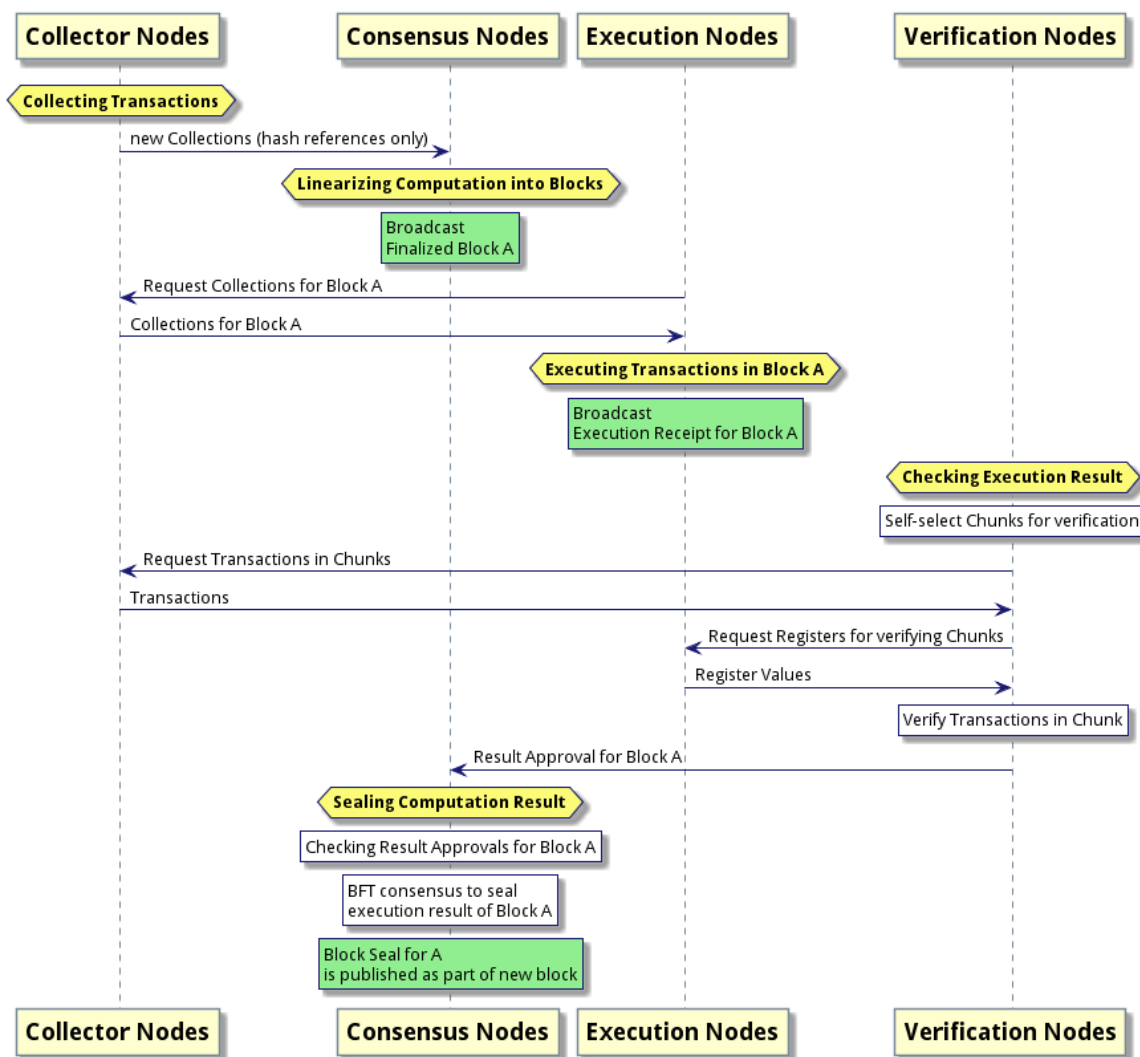
Figure 3: Transaction Processing Flow. The hexagon-shaped boxes indicate the start of the individual stages, which are discussed in sections 3.1 – 3.5. Arrows show message exchange between nodes with specific roles. Green boxes represent broadcast operations where the content is relayed to all staked nodes in the network (independent of their respective roles). White boxes are operations the nodes execute locally.

---

[5]Specifically, we use Protobuf with the exception of omitting the field numbers for the sake of conciseness.

3.1 Collector Role: Batching Transactions into Collections

As described in section 1.1.1, there are several clusters of collectors, where each cluster maintains one collection at a time. A cluster may decide to close its collection as long as it is non-empty. As part of closing the collection, it is signed by the cluster's collector nodes to indicate their agreement with the validity of its content and their commitment to storing it, until the block is sealed.

Definition 1 A **Guaranteed Collection** is a list of transactions that is signed by more than 2/3rds of collectors in its cluster. By signing a collection, a Collector Node attests:

- that all transaction in the collection are well-formed;
- the collection contains no duplicated transactions;
- to storing the collection including the full texts of all contained transactions.

A guaranteed collection is considered immutable. Once a collection is guaranteed by the collectors in its cluster, its reference is submitted to the Consensus Nodes for inclusion in a block (see Message 1).

```
1: message GuaranteedCollection {
2:     bytes collectionHash;
3:     uint32 clusterIndex;
4:     Signature aggregatedCollectorSigs;
5: }
```

Message 1: Collector Nodes send a `GuaranteedCollection` message to Consensus Nodes to inform them about a guaranteed Collection. For a collection to be guaranteed, more than $\frac{2}{3}$ of the collectors in the cluster (`clusterIndex`) must have signed it. Instead of storing the signatures individually, `aggregatedCollectorSigs` is an aggregated signature.

3.2 Consensus Role: Linearizing Computation into Blocks

Consensus nodes receive the `GuaranteedCollection`s from Collector clusters and include them in blocks through a BFT consensus algorithm outlined in section 1.1.2.

The structure of a finalized block is given in Message 2. The universal fields for forming the core blockchain are `Block.height`, `Block.previousBlockHash`. Furthermore, the block contains a `Block.entropy` as a source of entropy, which is generated by the Consensus Nodes' Random Beacon. It will be used by nodes that process the block to seed multiple random number generators according to a predefined publicly-known protocol. In Flow, a reliable and verifiable source of randomness is essential for the system's Byzantine resilience.

```
1: message Block {
2:     uint64 height;
3:     bytes previousBlockHash;
4:     bytes entropy;
5:     repeated GuaranteedCollection guaranteedCollections;
6:     repeated BlockSeal blockSeals;
7:     SlashingChallenges slashingChallenges;
8:     NetworkStateUpdates networkStateUpdates;
9:     Signature aggregatedConsensusSigs;
10: }
```

Message 2: Consensus Nodes broadcast `Block`s to the entire network. Their Block is valid if and only if more than $\frac{2}{3}$ of Consensus Nodes have signed. Instead of storing the signatures individually, we use store an aggregated signature in `signedCollectionHashes`.

The filed `Block.guaranteedCollections` specifies the new transactions that are to be executed next. During normal operations, Consensus Nodes only require the information provided by the `GuaranteedCollection` message. In particular, they work only with the collection hash (`GuaranteedCollection.collectionHash`) but do not need to inspect the collection's transactions unless an execution result is being challenged.

The `blockSeals` pertain to previous blocks in the chain whose execution results have just been sealed. A Block Seal contains a commitment to the resulting computational state after block execution and proves the computation has been verified by a super-majority of Verifier nodes. Section 3.5 describes in detail how Consensus Nodes seal blocks.

The fields `Block.slashingChallenges` and `Block.networkStateUpdates` are listed for completeness only. For brevity, we do not formally specify the embedded messages `SlashingChallenges` and `NetworkStateUpdates`. `Block.slashingChallenges` will list any slashing challenges that have been submitted to the Consensus Nodes (but not necessarily adjudicated yet by the Consensus Nodes). Slashing challenges can be submitted by any staked node. Some challenges, such as the Faulty Computation Challenge discussed in section 4.1), require the challenged node to respond and supply supplemental information. The filed `networkStateUpdates` will contain any updates network infrastructure state (see section 1.3 for details). Most significantly, staking changes due to slashing, staking and unstaking requests are recorded in this field. By including `slashingChallenges` and `networkStateUpdates` in a Block, their occurrences are persisted in the chain. As all staked nodes must follow recently finalized blocks to fulfill their respective roles, journaling this information in the block also serves as a way to publicly broadcast it.

3.3 Execution Role: Computing Transactions in Block

When the Execution Nodes receive a finalized Block, as shown in Message 2, they cache it for execution. An Execution Node can compute a block once it has the following information.

- The *execution result from the previous block* must be available, as it serves as the starting state for the next block. In most cases, an Execution Node has also computed the prior block and the resulting output state is known. Alternatively, an Execution Node might request the resulting state from the previous block from a different Execution Node.
- The Execution Node must fetch *all collections' text referenced in the block* from the Collector Nodes.

3.3.1 Chunking for Parallelized Verification

After completing a block's computation, an Execution Node broadcasts an Execution Receipt to the Consensus and Verification Nodes. We will discuss the details of the Execution Receipt in more detail in section 3.3.2. At its core, the Execution Receipt is the Execution Node's commitment to its final result. Furthermore, it includes interim results that make it possible to check the parts of the computation in parallel without re-computing the entire block. The execution result's correctness must be verified in a dedicated step to ensure an agreed-upon computational state. At the execution phase, the block computation is done by compute-optimised nodes. We therefore assume that a block re-computation by any other node, as part of the independent verification process, will always be slower. To address the bottleneck concern, we take an approach akin to split parallel verification [24]. We define the Execution Receipt to be composed of separate chunks, each constructed to be independently verifiable. A verification process (section 3.4) is defined on a subset of these chunks.

By breaking the block computation into separate chunks, we can distribute and parallelize the execution verification across many nodes. Let us consider a block containing Ξ chunks. Furthermore, let η be the fraction of chunks in a block that each Verification Node checks. The parameter-value for η is protocol-generated (see section 3.6 for details).

$$\text{Number of chunks each Verification Node checks} = \lceil \eta\, \Xi \rceil, \tag{3}$$

for $\lceil \cdot \rceil$ the ceiling function. A mature system with $\mathcal{V}$ Verification Nodes would re-compute in total $\mathcal{V}\lceil \eta\, \Xi \rceil$ chunks if all Verification Nodes were to fully participate. Hence, on average, each chunk is executed $\frac{\mathcal{V}\lceil \eta\, \Xi \rceil}{\Xi}$ times. For example, a mature system with $\mathcal{V} = 1000$ Verification Nodes could break up a large block into $\Xi = 1000$ chunks. With $\eta = 4.2\%$, each Verification Node would check $\lceil \eta\, \Xi \rceil = 42$ chunks and each chunk would be verified by $42 = \frac{\mathcal{V}\lceil \eta\, \Xi \rceil}{\Xi}$ different Verification Nodes (on average). The redundancy factor of 42 makes the system resilient against Byzantine actors (see section 3.7).

It is important that there is not single chunk that has a significant increased computation consumption compared to others in the block. If chunks had vastly different execution time, Verifiers assigned to check the long-running chunks would likely not be finished before the block is sealed (see section 3.5). Hence, Execution Nodes could attack the network by targeting the long-running chunk to introduce a computational inconsistency that is left unverified. This weakness is mitigated by enforcing chunks with similar computation consumption. Specifically, we introduce the following system-wide parameters.

$$\Gamma_{\text{Tx}} : \quad \text{upper limit for the computation consumption of one transaction} \tag{4}$$
$$\Gamma_{\text{chunk}} : \quad \text{upper limit for the computation consumption of all transactions in one chunk} \tag{5}$$
$$\text{with } \Gamma_{\text{Tx}} \ll \Gamma_{\text{chunk}} \tag{6}$$

Here, we assume the existence of a measure of computation consumption similar to gas in Ethereum.

Since there is a hard limit of computation consumption both on a transaction as well as a chunk, with the chunk's being significantly higher than the transaction's, the simple greedy Algorithm 1 will achieve the goal of similar computational consumption. Formally, let $\Gamma_{\text{chunk}} = n \cdot \Gamma_{\text{Tx}}$, for $n \geq 1$. Then, all chunks, except for the last one, will have a computation consumption c with the following properties.

- $(n-1)\Gamma_{\text{Tx}} < c$. If this was not the case, i.e., $c \leq (n-1)\Gamma_{\text{Tx}}$, Algorithm 1 would add more transactions to the current chunk (line 6), as the computation consumption for transactions is upper-bounded by Γ_{Tx}.
- $c \leq n\Gamma_{\text{Tx}}$, which is guaranteed by Algorithm 1, line 6.

Hence, all chunks, except for the last one, will have a computation consumption c

$$(1 - \tfrac{1}{n})\Gamma_{\text{chunk}} < c \leq \Gamma_{\text{chunk}}. \tag{7}$$

Choosing n large enough guarantees that all but the last chunk have similar computation consumption.

The last chunk could contain as little as a single transaction. Hence, its computation consumption $c_{\text{LastChunk}}$ could take any value $0 < c_{\text{LastChunk}} \leq \Gamma_{\text{chunk}}$. As opposed to $(1 - 1/n)\Gamma_{\text{chunk}} < c$ for any other chunk in the block. The last chunk being significantly smaller than $(1 - 1/n)\Gamma_{\text{chunk}}$ does not pose a problem for the following reason. For example, consider a node participating in a network

---
Algorithm 1 Chunking

Input:
  $\mathcal{T}$: List; element $\mathcal{T}[i]$ is computation consumption of transaction with index i in current block
Output:
  $\mathcal{C}$: List; element $(k, c) \equiv \mathcal{C}[i]$ represents chunk with index i in current block; k is index of first
    transaction in chunk; c is chunk's computation consumption

1: **Chunking**($\mathcal{T}$)
2:    $\mathcal{C} \leftarrow [\,]$ $\triangleright$ initialize $\mathcal{C}$ as empty list
3:    $c \leftarrow 0$ $\triangleright$ computation consumption of current chunk
4:    $k \leftarrow 0$ $\triangleright$ start index of current chunk
5:    **for** $i = 0, 1, \ldots, \mathtt{length}(\mathcal{T}) - 1$:
6:      **if** $c + \mathcal{T}[i] > \Gamma_{\text{chunk}}$: $\triangleright$ adding transaction with index i would overflow chunk
7:        $e \leftarrow (k, c)$ $\triangleright$ complete current chunk without transaction i
8:        $\mathcal{C}.\mathtt{append}(e)$
9:        $k \leftarrow i$ $\triangleright$ start next chunk at transaction i
10:       $c \leftarrow \mathcal{T}[i]$
11:      **else**: $\triangleright$ current chunk computation consumption of current chunk
12:        $c \leftarrow c + \mathcal{T}[i]$
13:    $e \leftarrow (k, c)$ $\triangleright$ complete last chunk
14:    $\mathcal{C}.\mathtt{append}(e)$
15:    **return** $\mathcal{C}$

Algorithm 1: separates transactions in a block into chunks. The algorithm presumes that computation consumption $\mathcal{T}[i] \leq \Gamma_{\text{Tx}}$ for all i.

that can process blocks with up to $\Xi = 1000$ chunks. For $\eta = 4.2\%$, a node would be required to have the capacity to process up to 42 chunks per block (as outlined above). Hence, for each block with $\Xi' \leq 42$, an honest Verifier would simply check the entire block. For blocks with more chunks, the workload is failry uniform across all Verifiers, even though each Verifier samples a subset of chunks for verification. This is because the large majority of all chunks have fairly comparable computation consumption as derived in equation (7).

3.3.2   The Execution Receipt

As Message 3 shows, the `Execution Receipt` is a signed `ExecutionResult`, which provides authentication of the sender and guarantees integrity of the message. The `ExecutionResult` encapsulates the Execution Node's commitments to both its interim and final results. Specifically, it contains a reference, `ExecutionResult.blockHash`, to the block whose result it contains and `ExecutionResult.finalState` as a commitment[6] to the resulting state.

Consider the situation where an Execution Node truthfully computed a block but used the computational state from a faulty Execution Receipt as input. While the node's output will likely propagate the error, it is important to attribute the error to the malicious node that originally introduced it and slash the malicious node. To ensure attributability of errors, `ExecutionResult`s specify

---
[6]We don't specify details of the `StateCommitment` here. It could either be the full state (which is likely much too large for a mature system). Alternatively, `StateCommitment` could be the output of a hashing algorithm, such as `Merkle Patricia Trie` used by Etherum [25].

the computation result they built on top of as `ExecutionResult.previousExecutionResultHash`. As we will show later in section 3.6, a faulty computation result will be rejected with overwhelming probability.

Lastly, the `ExecutionResult.chunks` contains the Execution Node's result of running the `Chunking` algorithm (`Chunk.startingTransactionIndex` and `Chunk.computationConsumption`). Also, a `Chunk` states the starting state (`Chunk.startState`) for the computation and the computation consumption for the first transaction in the chunk (`Chunk.`τ_0)

Solving the Verifier's Dilemma

The field `ExecutionReceipt.`Zs is a list of SPoCKs which is generated based on interim states from computing the chunks. Formally, for the ith chunk `VerificationProof.`$Zs[i]$ holds the SPoCK demonstrating knowledge of the secret derived from executing the chunk. As explained in section 3.4.2, the SPoCK is required to resolve the Verifier's Dilemma in Flow. Furthermore, it prevents Execution Nodes from copying `ExecutionResult` from each other, pretending their computation is faster than it is.

Given that there are several Execution Nodes, it is likely that multiple Execution Receipts are issued for the same block. We define consistency of Execution Receipts as follows.

```
1: message Chunk {
2:     StateCommitment startState;
3:     float τ₀;
4:     uint32 startingTransactionIndex;
5:     float computationConsumption;
6: }
```

```
1: message ExecutionResult {
2:     bytes blockHash;
3:     bytes previousExecutionResultHash;
4:     repeated Chunk chunks;
5:     StateCommitment finalState;
6: }
```

```
1: message ExecutionReceipt {
2:     ExecutionResult executionResult;
3:     repeated SPoCK Zs;
4:     bytes executorSig;
5: }
```

Message 3: Execution Nodes broadcast an `ExecutionReceipt` to the entire network after computing a block.

Definition 2 *Consistency Property of Execution Receipts*
Execution Receipts are **consistent** if and only if
1. their `ExecutionResults` are identical and
2. their SPoCKs attest to the same confidential knowledge.

3.4 Verification Role: Checking Execution Result

The verification process is designed to probabilistically guarantee safety of the computation result (see page 23, Theorem 2). A crucial aspect for the computational safety is that Verification Nodes verifiably *self-select* the chunks they check *independently* from each other. This process is described in the section below. In section 3.4.2, we describe how the selected chunks are verified.

3.4.1 Self-selecting chunks for verification

The protocol by which Verification Nodes *self-select* their chunks is given in Algorithm 2. While inspired by Algorand's random sortition [26], it has significantly reduced complexity.

Algorithm 2 ChunkSelfSelection

Input:

    η: fraction of chunks to be checked by a Verification Node

   er: Execution Receipt

   sk: Verifier Node's secret key

Output:

    L: List of chunk indices that are assigned to the Verifier

    p: proof of protocol-compliant selection

1: **ChunkSelfSelection**(η, er, sk)
2:     chunks $\leftarrow$ er.executionResult.chunks $\triangleright$ list of chunks from Execution Receipt
3:     $\Xi \leftarrow$ length(chunks) $\triangleright$ number of chunks in the block
4:     $p \leftarrow$ sign$_{\text{sk}}$(er.executionResult) $\triangleright$ sign Execution Receipt's ExecutionResult
5:     seed $\leftarrow$ hash(p) $\triangleright$ use signature's hash as random number generator seed
6:     $n \leftarrow \lceil \eta \cdot \Xi \rceil$ $\triangleright$ number of chunks for verification
7:     $L \leftarrow$ FisherYatesShuffle(chunks, seed, n) $\triangleright$ generate random permutation of chunks
8:     **return** L, p

Algorithm 2: randomly selects chunks for subsequent verification. In line 6, $\lceil \cdot \rceil$ is the ceiling operator. The function FisherYatesShuffle(list, seed, n) draws a simple random sample without replacement from the input list. The seed is used for initializing the pseudo-random-number generator.

ChunkSelfSelection has the following crucial properties.

(1) *Verifiability.* The seed for the random selection of the chunks is generated in lines $4 - 5$. Given the ExecutionReceipt, the Verifier's public key, and the proof p, any other party can confirm that p was generated according to protocol. Moreover, given p, anyone can re-compute the Verifier's expected chunk assignment. Thereby, the chunk-selection protocol is verifiable and deviating from the protocol is detectable, attributable, and punishable.

(2) *Independence.* Each Verifier samples its chunks locally and independently from all the other Verifiers.

(3) *Unpredictability.* Without knowledge of the Verifier's secret key sk, it is computationally infeasible to predict the sample. Formally, the computation of seed can be considered a verifiable random function [2].

(4) *Uniformity.* A Verifier uses Fisher-Yates shuffling [27, 28] to self-select the chunks it checks. The Fisher-Yates algorithm's pseudo-random number generator is seeded with the output of a cryptographic hash function. The uniformity of the seed and the uniformity of the shuffling algorithm together guarantee the uniformity of the generated chunk selection.

Corollary 1

Algorithm 2 samples a subset of $\lceil \eta \cdot \Xi \rceil$ chunks, for Ξ the number of chunks in the block and η the fraction of chunks to be checked by each Verification Node. The selection probability *uniform* and *independently and identically distributed* (i.i.d.) for all Verifiers. The sample is *unpredictable* without the Verification Node's secret key.

Independence and unpredictability are crucial for the system's security. A selection protocol without these properties might be abused by a malicious Execution Node (see section 4.3 for a detailed discussion).

Algorithm 3 ChunkVerification

Input: Λ: starting state for computing the chunk

         Txs: List of transactions in chunk

         τ_0: computation consumption of leading transaction in *this* chunk

         Λ': resulting state after computing the chunk as stated in the Execution Receipt

         τ_0': computation consumption of leading transaction in *next* chunk;

           or ∞ if this is the last chunk in the block

         c: chunk's computation consumption as stated in the Execution Receipt

Output: true: if and only if chunk passes verification

1: **ChunkVerification**(Λ, Txs, Λ', ntx, c)
2:    $\gamma \leftarrow 0$ $\triangleright$ accumulated computation consumption
3:    **for** t **in** Txs: $\triangleright$ for each transaction in chunk
4:      $\Lambda, \tau \leftarrow$ execute(Λ, t) $\triangleright$ execute transaction
5:      **if** t is first transaction in chunk:
6:        **assert that** $\tau_0 == \tau$ $\triangleright$ computation consumption for first transaction in chunk is correct
7:      $\gamma \leftarrow \gamma + \tau$ $\triangleright$ add transaction's computation consumption to γ
8:    **assert that** $\gamma == c$ $\triangleright$ computation consumption for entire chunk is correct
9:    **assert that** $c \leq \Gamma_{\text{chunk}}$ $\triangleright$ computation consumption does not exceed limit
10:    **assert that** $c + \tau_0' > \Gamma_{\text{chunk}}$ $\triangleright$ chunk is full: no more translations can be appended to chunk
11:    **assert that** $\Lambda == \Lambda'$ $\triangleright$ verify Execution Node's resulting state
12:    **return** true

Algorithm 3: verifies a chunk. The function execute(Λ, t) applies the transaction t to the computational state Λ and returns a pair of values: the resulting state (first return value) and transaction's computation consumption (second return value). The assert statement raises an exception if the condition is false.

Our approach to independent verification of chunks has similarities with traditional acceptance sampling theory [24, 29, 30], yet differs as our model assumptions are different. In contrast to traditional acceptance sampling, where physical items are tested, identical copies of our digital chunks can be checked in parallel by multiple Verifiers. Part of the novelty of our approach is that we elevate an acceptance sampling model with parallel sampling.

3.4.2 Verifying chunks

The verification protocol is designed to be self-contained, meaning any Execution Receipt can be verified in isolation. All required data is specified through the hashes in the execution receipt. Checking the correctness of a chunk requires re-computing the transactions in the chunk. The details of the verification protocol are given in Algorithm 3. The inputs τ_0 and τ_0' for the ChunkVerification algorithm are taken directly from the Execution Receipt. Λ, Txs, and Λ', have to be fetched from the Execution Nodes and checked to match the Execution Receipt (specifically: Λ, Λ') or the original block (specifically: Txs) via hashing[7]. Therefore, errors uncovered by the verification process can be attributed to the data provider and slashed.

The verification process is also given enforcement power, as we enable it to request slashing

---
[7] We use the term 'hashing' here to refer to the one-time application of a conventional hash function as well as iterative hashing schemes such as Merkle trees or Merkle Patricia trie.

1: **message** CorrectnessAttestation {
2:     **bytes** executionResultHash;   ▷ Hash of approved `ExecutionResult`
3:     **bytes** attestationSig;   ▷ Signature over `executionResultHash`
4: }

1: **message** VerificationProof {
2:     **repeated uint32** L;   ▷ list of chunk indices assigned to verifer
3:     **bytes** p;   ▷ proof to verify correctness of chunk assignment
4:     **repeated SPoCK** Zs;   ▷ for each assigned chunk: proof of re-computation
5: }

1: **message** ResultApproval {
2:     **CorrectnessAttestation** attestation;
3:     **VerificationProof** verificationProof;
4:     **bytes** verifierSig;   ▷ signature over all the above fields
5: }

Message 4: Verification Nodes broadcast a `ResultApproval` to all Consensus Nodes if all their assigned chunks pass verification.

against an Execution Node. A successful verification process results in a `ResultApproval` (Message 4) being broadcast by the Verifier to all Consensus Nodes. It is important to note that a `ResultApproval` (specifically `ResultApproval.attestation`) attests to the correctness of an `ExecutionResult`. Specifically, in `CorrectnessAttestation`, the Verifier signs the `ExecutionResult`, *not* the Execution Receipt. Per definition 2, multiple *consistent* Execution Receipts have identical `ExecutionResult`s. Hence, their correctness is simultaneously attested to by a single `ResultApproval` message. This saves communication bandwidth, as each Verifier Node issues only one `ResultApproval` for the common case that several Execution Nodes publish the same results by issuing consistent Execution Receipts.

The second important component is the `ResultApproval.verificationProof`, which proves that the Verification Node completed the assigned verification tasks. We designed this protocol component to address the Verifier's Dilemma [22]. It prevents the Verifier from just approving ExecutionReceipts, betting on honesty of the Execution Nodes, and thereby being compensated for work the Vertifier didn't do. The field `VerificationProof.L` specifies the chunk indices the Verifier has selected by running `ChunkSelfSelection` (Algorithm 2). As detailed in section 3.4.1 (property (1) *Verifiability*), protocol-compliant chunk selection is proven by `VerificationProof.p`, which holds the value for p returned by `ChunkSelfSelection`. The list `VerificationProof.Zs` contains for each assigned chunk a proof of verification. Formally, for each $i \in$ `VerificationProof.L`, `VerificationProof.Zs[i]` holds a SPoCK.

Each Verifier samples the chunks it checks independently (property (2)). Furthermore, statistically each chunk is checked by a larger number of nodes (e.g., on the order of 40 as suggested in section 3.7) Therefore, with overwhelming probability, all chunks are checked. (We formalize this argument in Theorem 2).

3.5 Consensus Role: Sealing Computation Result

Sealing a block's computation result happens after the block itself is already finalized. Once the computation results have been broadcast as Execution Receipts, Consensus Nodes wait for the ExecutionResults to accumulate matching Results Approvals. Only after a super-majority of Verifier Nodes approved the result and *no* FaultyComputationChallenge has been submitted (see section 4.1 for details), the ExecutionResult is considered for sealing by the Consensus Nodes. The content of a BlockSeal is given in Message 5. Algorithm 4 specifies the full set of conditions a BlockSeal must satisfy. Algorithm 4 enforces a specific structure in our blockchain which is illustrated in Figure 4.

Once a Consensus Node finds that all conditions are valid, it incorporates the BlockSeal as an element of Block.blockSeals into the next Block it proposes. All honest Consensus Nodes will check the validity of the BlockSeal as part of verifying the Block, *before* voting for it. Thereby the validity of a BlockSeal is guaranteed by the BFT consensus algorithm. Furthermore, condition (8) guarantees that any FaultyComputationChallenge has been received *before* the block is sealed. This gives rise to the following corollary:

Corollary 2

Given a system with partially synchronous network conditions with message traversal time bounded by Δ_t. A Block is only sealed if more than $\frac{2}{3}$ of Verification Nodes approved the ExecutionResult and no FaultyComputationChallenge was submitted and adjudicated with the result that the ExecutionResult is faulty.
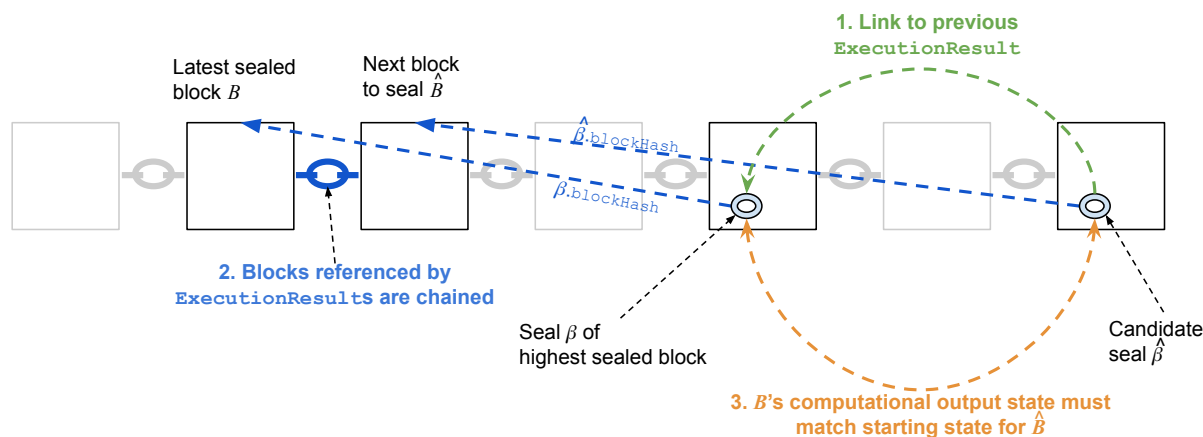

Figure 4: Validity conditions for a new BlockSeal according to Algorithm 4.

```
 1: message BlockSeal {
 2:     bytes blockHash;
 3:     bytes executionResultHash;
 4:     bytes executorSigs;                    ▷ Signatures from Execution Nodes over ExecutionResult
 5:     bytes attestationSigs;            ▷ Signatures from Verification Nodes approving the ExecutionResult
 6:     bytes proofOfWaiting;        ▷ output of VDF to prove waiting for a FaultyComputationChallenge
 7: }
```

Message 5: In order to seal a block (with hash `blockHash`), Consensus Nodes add a `BlockSeal` to the next block (field `Block.blockSeals`) they finalize. The field `executorSigs` is the aggregated signature of at least one Execution Node that published an Execution Receipt with a compatible `ExecutionResult`. Their `BlockSeal` is valid only if more than $\frac{2}{3}$ of Verifier Nodes have signed. Instead of storing the signatures individually, we use store an aggregated signature in `attestationSigs`.

Algorithm 4 Validity of Block Seal

Let β be the (accepted) `BlockSeal` for the *highest* sealed block, i.e., β is contained in a finalized block. A candidate `BlockSeal` $\hat{\beta}$ must satisfy the following conditions to be valid.

(1) $\hat{\beta}$.`executionResult.previousExecutionResultHash` must be equal to
β.`executionResultHash`

(2) Let
  - B be the block whose result is sealed by β, i.e., B is referenced by β.`blockHash`;
  - $\hat{B}$ be the block whose result is sealed by $\hat{\beta}$, i.e., $\hat{B}$ is referenced by $\hat{\beta}$.`blockHash`.

  $\hat{B}$.`previousBlockHash` must reference B.

(3) Let
  - E be the `ExecutionResult` that referenced (sealed) by β.`executionResultHash`;
  - $\hat{E}$ be the `ExecutionResult` that is referenced by $\hat{\beta}$.`executionResultHash`.

  The starting state for computing $\hat{B}$ must match the B computational output state, i.e.,

$$\hat{E}.\texttt{chunks}[0].\texttt{startState} == E.\texttt{finalState}$$

(4) $\hat{\beta}$.`attestationSigs` must contain more than $\frac{2}{3}$ of Verifier Nodes' signatures

(5) For each Verifier who contributed to $\hat{\beta}$.`attestationSigs`, the `VerificationProof` has been validated.

(6) No `FaultyComputationChallenge` against the `ExecutionResult` is *pending* (i.e., not yet adjudicated).

(7) No `FaultyComputationChallenge` against the `ExecutionResult` was adjudicated with the result that the `ExecutionResult` is faulty.

(8) β.`proofOfWaiting` proves a sufficiently long wait time Δ_t

Per axiom, we consider the genesis block as sealed.

Algorithm 4: protocol for determining validity of a `BlockSeal`. Figure 4 illustrates conditions (1) – (4).

3.6 Proof of Computational Correctness

Below, we prove in Theorem 1 that Flow's computation infrastructure has guaranteed liveness, even in the presence of a moderate number of Byzantine actors. Furthermore, Theorem 2 proves that block computation is *safe*, i.e., the resulting states in sealed blocks are correct with overwhelming probability. While safety is unconditional on the network conditions, liveness requires a partially synchronous network.

Theorem 1 *Computational Liveness*
Given a system with
- more than $\frac{2}{3}$ of the Consensus Nodes' stake is controlled by honest actors;
- and at least one honest Execution Node;
- and more than $\frac{2}{3}$ of the Verification Nodes' stake is controlled by honest actors;
- and partially synchronous network conditions with message traversal time bounded by Δ_t.

The computation and sealing of finalized blocks always progresses.

Proof of Theorem 1
- Assuming liveness of the consensus algorithm, finalization of new blocks always progresses.
- For a system with one honest Execution Node, there is at least one Execution Receipt with a correct `ExecutionResult`.
- Every honest Validator will approve a correct `ExecutionResult`. Hence, there will be Result Approvals by at least $\frac{2}{3}$ of the Verification Nodes.
- Malicious Verifiers might temporarily delay block sealing by raising a `FaultyComputation-Challenge`, which triggers condition (6) in Algorithm 4. However, the resolution process (see section 4.1 for details) guarantees that the `FaultyComputationChallenge` is eventually adjudicated and malicious Verifiers are slashed (Corollary 3). Therefore, malicious Verifiers cannot indefinitely suppress block sealing via condition (6) or even reach condition (7).
- Consequently, all honest Consensus nodes will eventually agree on the validity of the Block Seal.
- Assuming a consensus algorithm with guaranteed chain quality[8], an honest Consensus Node will eventually propose a block and include the Block Seal as prescribed by the protocol.
- Given that there are more than $\frac{2}{3}$ of honest Consensus Nodes, the block containing the seal will eventually be finalized (by liveness of consensus). □

Theorem 2 *Computational Safety*
Given a system with
- partially synchronous network conditions with message traversal time bounded by Δ_t;
- more than $\frac{2}{3}$ of the Consensus Nodes' stake is controlled by honest actors;
- all Verification Nodes are equally staked and more than $\frac{2}{3}$ of them are honest.

Let $\widetilde{N}$ denote the number of honest Verification Nodes and η the *fraction* of chunks each Verifier checks. The probability P_{error} for a computational error in a sealed block is bounded by

$$P_{\text{error}} \lesssim \Xi \cdot \left(1 - \eta\right)^{\widetilde{N}} \tag{8}$$

for large $\widetilde{N}$. Here, Ξ denotes the number of chunks in the Execution Receipt.

---

[8] Formally, chain quality of a blockchain is the ratio $\frac{\text{number of blocks contributed by honest nodes}}{\text{number of blocks contributed by malicious nodes}}$ [31].

Theorem 2 states that the probability of a computational error decreases *exponentially* with the number of Verifiers.

Proof of Theorem 2

Consider an Execution Receipt.

- For brevity, we denote the Execution Receipt's chunks as L , i.e.,

$$L = \texttt{ExecutionReceipt.executionResult.chunks}.$$

  Without loss of generality, we treat L as a *set* (instead of an ordered list), as this proof does not depend on the order of L's elements.
- Let Ξ be the total number of chunks in the Execution Receipt, i.e $\Xi = |L|$, where $|\cdot|$ denotes the cardinality operator.
- Let $\xi \in L$ denote a chunk.

As formalized in Corollary 1, each honest Verifier randomly selects a subset $S \subset L$ with $|S| = \lceil \eta \cdot \Xi \rceil$ by execution $\texttt{ChunkSelfSelection}$ (Algorithm 2). As each Verifier selects the chunks by Fisher-Yates shuffle, it follows that chunk ξ not being selected as the first element is $\frac{\Xi-1}{\Xi}$ and that ξ is not selected as the second element is $\frac{\Xi-2}{\Xi-1}$, etc. Hence, the probability that chunk ξ is *not checked* by *one specific verifer* is

$$\bar{p}_\xi = \frac{\Xi-1}{\Xi} \cdot \frac{\Xi-2}{\Xi-1} \cdot \ldots \cdot \frac{\Xi-n}{\Xi-(n-1)} = \frac{\Xi-n}{\Xi} = 1 - \frac{n}{\Xi}, \qquad \text{for } n := |S| = \lceil \eta \cdot \Xi \rceil. \qquad (9)$$

Let $\bar{P}_\xi$ be the probability that ξ is checked by *none* of the honest Verifiers.

$$\bar{P}_\xi = \bar{p}_\xi^{\widetilde{N}} = \left(1 - \tfrac{n}{\Xi}\right)^{\widetilde{N}} \qquad (10)$$

$$\leq \left(1 - \eta\right)^{\widetilde{N}} \qquad (11)$$

as $n = \lceil \eta \cdot \Xi \rceil \geq \eta \cdot \Xi$. Consequently, the probability of *the specific* chunk ξ not being checked decreases *exponentially* with the number of Verifiers $\widetilde{N}$. Figure 5 visualizes the *exact* probability $\bar{P}_\xi$ as a function of η as given in equation (10).

The probability that *all* chunks are checked by *at least* one honest Verifier is $(1 - \bar{P}_\xi)^{\Xi}$. Consequently, the probability an error in *any chunk in the block* remaining undetected is

$$P_{\text{error}} = 1 - (1 - \bar{P}_\xi)^{\Xi}. \qquad (12)$$

We assume that the system parameter η is chosen such that $\bar{P}_\xi \approx 0$ to ensure computational safety. Hence, we can approximate eq. (12) by its first order Taylor series in $\bar{P}_\xi$.

$$(1 - \bar{P}_\xi)^{\Xi} \simeq 1 - \Xi \cdot \bar{P}_\xi \qquad (13)$$

Inserting equations (13) and (11) into (12) yields

$$P_{\text{error}} \lesssim \Xi \cdot \left(1 - \eta\right)^{\widetilde{N}}, \qquad (14)$$

which proves equation (8) from the theorem. We have now shown that the probability of a faulty chunk in an Execution Receipt not being checked by an honest Verifier is bounded by (14). Furthermore, every honest Verifier will challenge any faulty chunk it is assigned to check by raising a $\texttt{FaultyComputationChallenge}$ (see section 4.1 for details). Corollary 2 guarantees that a block is only sealed if no correct $\texttt{FaultyComputationChallenge}$ was raised. Hence, the only way a block can be sealed with a faulty $\texttt{ExecutionResult}$ is if the faulty chunks are not checked by honest Verifers. Consequently, eq. (14) also bounds the probability of a faulty $\texttt{ExecutionResult}$ being sealed.

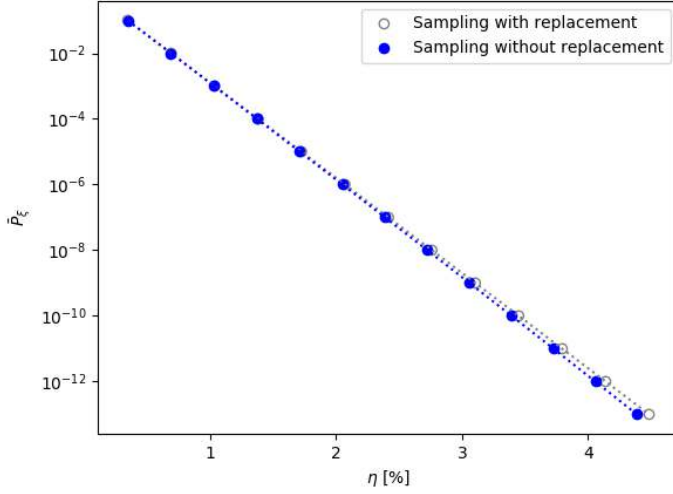<div style="text-align: right">□</div>

Figure 5: Probability $\bar{P}_\xi$ that a specific chunk is checked by *none* of the honest Verifiers. η is the *fraction* of chunks each Verifier selects for checking. The graph illustrates probabilities for $\widetilde{N} = 667$ honest Verification Nodes verifying an Execution Receipt with $\Xi = 1000$ chunks. The blue curve shows the dependency when Verifiers sample their chunks based on Fisher-Yates as specified in Algorithm 2, i.e., sample chunks from the Execution Receipt *without replacement*. For comparison, we show sampling with replacement.

3.7 Computational Load on Verification Nodes

Using equation (8), we can compute the required fraction η of chunks that each Verifier has to check to achieve a specific P_{error}. For the mature system under full load, we expect there to be 1000 Verification Nodes and each block to contain up to $\Xi = 1000$ chunks. Furthermore, we make the conservative assumption that only $\frac{2}{3}$ of the Verification Nodes are honest, i.e., $\widetilde{N} = 667$.

Let the probability for a malicious Execution Node to succeed with an attempt to introduce a compromised state into the blockchain be $P_{\text{error}} \leq 10^{-6}$. To achieve this, every Verification Node would need to check 32 chunks, i.e., execute $\eta = 3.2\%$ of the work of an Execution Node. To lower the probability even further to $P_{\text{error}} \leq 10^{-9}$, Verifiers only need to execute $\eta = 4.2\%$ of the work of an Execution Node.

This shows that distributing and parallelizing the verification of computation results is efficient. Furthermore, note that checking the chunks can be trivially executed in parallel.

4 Mitigation of Attack Vectors

In the following, we will discuss the most severe attacks on Flow. In this context, we would like to re-iterate that the staking balances are maintained by the Consensus Nodes as part of the network state (compare section 1.3). Hence, Consensus Nodes can adjudicate and slash misbehaving nodes directly. The resulting updates to the network state are published in the blocks (field `slashingChallenges` in message 2) without needing to involve Execution Nodes.

4.1 Adjudicating with Faulty Computation Results

In section 3.6, Theorem 2, we have shown that a faulty computation state will be challenged by a Verification Node with near certainty. Formally, a Verification Node submits a **Faulty Computation Challenge** (FCC), to the Consensus Nodes for adjudication. We start by introducing the necessary notation and then proceed with specifying the details of an FCC and the protocol for processing them.

Nomenclature (illustrated in Figure 6): For an Execution Receipt with Ξ chunks, the field `ExecutionReceipt.executionResult.chunks` holds the `StateCommitments` $[\Lambda_0, \Lambda_1, \ldots, \Lambda_\Xi]$. For $i \in \{0, 1, \ldots, \Xi - 1\}$, Λ_i denotes the starting state for the computation of the chunk with index i. Λ_Ξ is the final state at the end of the block (after all transactions are computed).

To denote the (interim) states *between individual transactions*, we extend the notation accordingly. Let the chunk at index i contain χ_i transactions. For $k \in \{0, 1, \ldots, \chi_i - 1\}$, $\Lambda_{i,k}$ denotes the input state for computing the transaction with index k within the chunk. Accordingly, Λ_{i,χ_i-1} is the state at the end of the chunk.

```
1:  message FaultyComputationChallenge {
2:      bytes executionReceiptHash;
3:      uint32 chunkIndex;
4:      ProofOfAssignment proofOfAssignment;
5:      StateCommitment stateCommitments;
6:      Signature verifierSig;
7:  }
```

Message 6: Verification Nodes send this message to Consensus Nodes to challenge a specific Execution Receipt (executionReceiptHash). The `FaultyComputationChallenge` is specific to a computational output state for one of the chunks, where `chunkIndex` is a zero-based index into `ExecutionReceipt.executionResult.chunks` (compare Message 3).

Note that Λ_{i,χ_i-1} simultaneously serves as the *starting state* for the next chunk at index $i + 1$. Hence, Λ_{i,χ_i} as well as Λ_{i+1} and $\Lambda_{i+1,0}$ refer to the same `StateCommitment`. The different nomenclatures are introduced for ease of notation only.

Definition 3
A well-formed **Faulty Computation Challenge (FCC)**, specified in Message 6, challenges one specific chunk at index `chunkIndex` of an Execution Receipt (referenced by `executionReceiptHash`). It provides the list

$$\text{stateCommitments} = [\Lambda_{\texttt{chunkIndex},0}, \Lambda_{\texttt{chunkIndex},1}, \ldots, \Lambda_{\texttt{chunkIndex},\chi_{\texttt{chunkIndex}}}]. \tag{15}$$

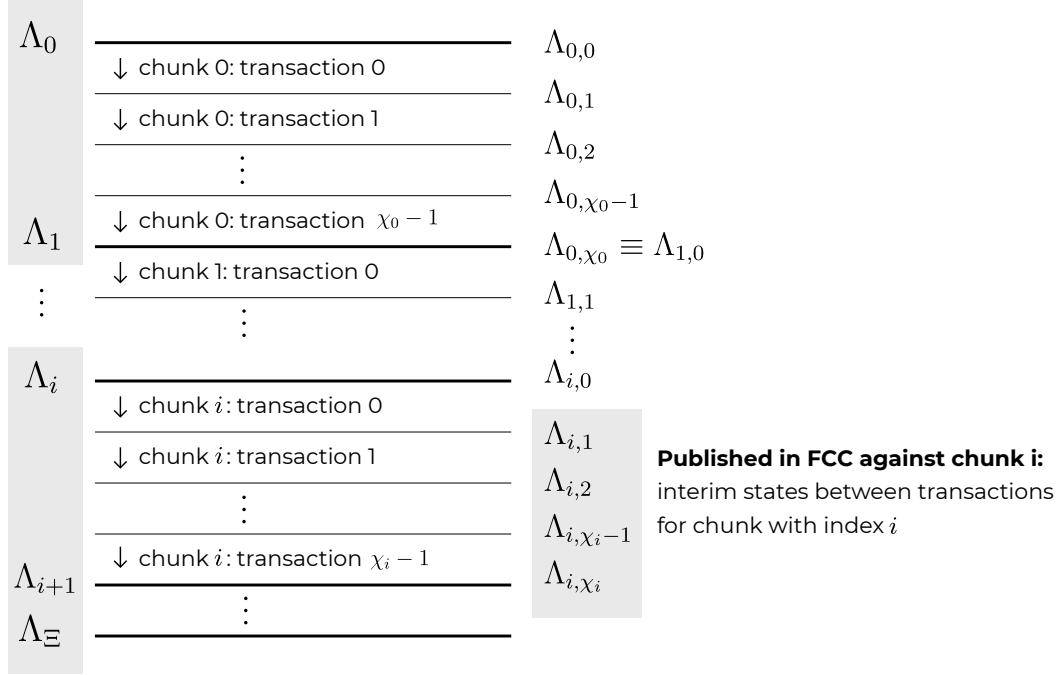**Published in Execution Receipt**
States between chunks

Figure 6: Illustration of nomenclature State commitments (e.g., hashes) are represented as vertical lines and denoted by Λ. The bold lines visualize the starting states $\Lambda_0, \Lambda_1, \dots, \Lambda_{\Xi-1}$ for the chunks, as well as the final state Λ_Ξ. Furthermore, $\Lambda_{i,k}$ is the *input* state for computing the transaction with index k within the chunk.

Definition 4 *Protocol for Adjudicating a Faulty Computation Result*
Let there be a Verifier Node V that is checking the chunk at index i and disagrees with the resulting state $\Lambda_{i+1} \equiv \Lambda_{i,\chi_i}$. In the following, we denote the Verifier's `StateCommitments` as $\widetilde{\Lambda}$ and the Execution Node's with Λ.

1. Verifier V broadcasts a `FaultyComputationChallenge` to the Consensus Nodes with
   - `FaultyComputationChallenge.chunkIndex` $\leftarrow i$
   - `FaultyComputationChallenge.stateCommitments` $\leftarrow [\widetilde{\Lambda}_{i,1}, \dots, \widetilde{\Lambda}_{i,\chi_i}]$
2. Consensus Nodes publish the FCC in their next finalized block (field `slashingChallenges` in message 2)
3. The challenged Execution Node has a limited time to broadcast a `FaultyComputationResponse` (Message 7) to the Consensus Nodes. Time is measured using a verifiable delay function [32].

1: **message** FaultyComputationResponse {
2:   **bytes** FaultyComputationChallengeHash;
3:   **StateCommitment** stateCommitments;
4:   **Signature** executerSig;
5: }

Message 7: A challanged Execution Node broadcast a `FaultyComputationResponse` to the Consensus Nodes.

(a) Should the Execution Node not respond, it is slashed. Consensus Nodes will include a corresponding notification in the next block (field `networkStateUpdates` in Message 2) that also includes the output of the VDF as proof of waiting. In this case, adjudication ends with the Execution Node being slashed.

(b) To prevent being slashed, the Execution Node must disclose all interim states in the chunk by submitting a `FaultyComputationResponse` with

- `FaultyComputationResponse.stateCommitments` $\leftarrow [\Lambda_{i,1}, \ldots, \Lambda_{i,\chi_i}]$

to the Consensus Nodes. In case the Execution Node sends a `FaultyComputationResponse`, the protocol continues with step 4 below.

4. Consensus Nodes now compare the `stateCommitments` from both parties element-wise and find the *first* mismatch. Let the first mismatch occur at index ℓ, i.e.

$$\Lambda_{i,\ell} \neq \widetilde{\Lambda}_{i,\ell} \tag{16}$$

$$\text{for all } l \in [0, 1, \ldots, \ell - 1] : \quad \Lambda_{i,l} = \widetilde{\Lambda}_{i,l}. \tag{17}$$

Essentially, both parties agree that, starting from state $\Lambda_i \equiv \Lambda_{i,0}$, the computation should lead to $\Lambda_{i,\ell-1}$ as the input state for the next transaction. However, they disagree on the resulting state *after* computing this next transaction.

5. Consensus Nodes request state $\Lambda_{i,\ell-1}$ from either party. Furthermore, by resolving the texts of collections in the block, Consensus Nodes obtain the transaction with index ℓ in the chunk, whose output is disputed.

6. Consensus Nodes use $\Lambda_{i,\ell-1}$ as input state for computing transaction with index ℓ in the chunk. Consensus Nodes now compare their locally computed output state with $\Lambda_{i,\ell}$ and $\widetilde{\Lambda}_{i,\ell}$.

7. Any party who published a computation result that does not match the values computed by the Consensus Nodes is slashed. Consensus Nodes will include a corresponding notification in the next block (field `networkStateUpdates` in message 2).

Informally, Definition 4 describes a protocol by which a Verifier Node can appeal to the committee of Consensus Nodes to re-compute a specific transaction whose output the Verifier does not agree with. To avoid being slashed, the challenged Execution node must provide all information that is required for the Consensus Nodes to re-compute the transaction in question. Nevertheless, there is no leeway for the Execution Node to provide wrong information as honest Consensus Nodes will verify the correctness based on previously published hash commitments:

- Consensus Nodes request transaction texts of collections. The collection hashes are stated in blocks, which allow verification of the collection texts.
- Consensus Nodes request the last interim state $\Lambda_{i,\ell-1}$ in the chunk that both parties agree on. A hash of this state was published by both the Verifier and the challenged Execution Node. This allows the Consensus Nodes verify state variables (e.g., via Merkle proofs).

Furthermore, the described protocol is executed by *all* honest Consensus Nodes. The resulting slashing is included in a block and hence secured by BFT consensus. Assuming a super-majority of honest Consensus Nodes, it is guaranteed that the misbehaving node is slashed. The following Corollary 3 formalizes this insight.

Corollary 3

Given a system with
- more than $\frac{2}{3}$ of the Consensus Nodes' stake is controlled by honest actors;
- and partially synchronous network conditions with message traversal time bounded by Δ_t.

The following holds.
- If an honest Verifier node is assigned to verify a chunk that has a faulty computation result, the Execution Node who issues the corresponding Execution Receipt will be slashed.
- If a dishonest Verifier Node challenges a correct computation result, the Verifier will be slashed.

4.2 Resolving a Missing Collection

As message 2 and 1 show, a block references its collections by hash but does not contain the individual transactions texts. The transactions texts are stored by the Collector Node cluster which build the collection and is only required when Execution Nodes want to compute the blocks' transactions. Hence, a cluster of Collector nodes could withhold the transaction texts for a guaranteed collection. While block production is not impacted by this attack, block execution halts without access to the needed transaction texts.

When an Execution Node is unable to resolve a guaranteed collection, it issues a **Missing Collection Challenge** (MCC). An MCC is a request to slash the cluster of Collector Nodes (Message 1, line 4) who have guaranteed the missing collection. MCCs are directly submitted to Consensus Nodes.

Definition 5 *Protocol for Resolving a Missing Collection*
1. An Execution Node determines that the transaction texts for a `GuaranteedCollection` from the current block are not available as expected. The protocol does not dictate how this determination is reached, but the obvious implementation is assumed (ask the Guarantors, wait for a response, ask other Execution Nodes, wait for a response).
2. The Execution Node broadcasts an MCC to all Collector and Consensus Nodes. The Consensus Nodes record the challenge in the next block, but do not otherwise adjudicate the challenge at this stage.
3. Any honest Collector Node, which is *not* a member of the challenged cluster, sends a request to κ randomly selected Guarantors to provide the missing Collection. If the request is answered, the requesting Collector Node forwards the result to the Execution Nodes.
4. If the Guarantor does not respond within a reasonable time period Δ_t, the requesting Collector Node will sign a Missing Collection Attestation (MCA), including the hash of the collection in question. Time is measured using a verifiable delay function [32] and the MCA contains the VDF's output as proof of waiting. The MCA is broadcast to all Consensus and Execution Nodes.
5. An honest challenged Guarantor will respond with the complete Collection text to any such requests.
6. If the Execution Nodes receive the collection, they process the block as normal. Otherwise, they wait for more than $\frac{2}{3}$ of the Collector Nodes to provide MCAs.
7. Appropriate MCAs must be included in all Execution Receipts that skip one or more Collections from the block.
8. Every MCC will result in a small slashing penalty for each Execution Node and each challenged Guarantor. Even if the MCC is resolved by finding the Collection, each of these actors must pay the fine, including the Execution Nodes that did not initiate the MCC. This is designed to prevent the following edge cases:

- Lazy Guarantors that only respond when challenged: without a fine for challenged Guarantors, even in the case where the collection is resolved, there is no incentive for Guarantors to respond without being challenged.
- Spurious MCCs coming from Byzantine Execution Nodes: without a fine for Execution Nodes, there is a zero-cost to generating system load through unjustified MCCs.
- Don't punish the messenger: all Execution Nodes must be fined equally so that there is no disincentive to be the first Execution Node to report a problem.

9. If an Execution Receipt containing an MCC is sealed, ALL guarantors for the missing Collection are subject to a large slashing penalty (equal to the minimum staking requirement for running a Collector Node).

Discussion

- The slashing penalty for a resolved MCC should be small enough that it doesn't automatically eject the Collector Nodes from the network (by dropping them below the minimum staking threshold), but must be significant enough to account for the fact that resolving an MCC is very expensive.
- Resolving an MCC is very expensive. Each Collector Node will request the Collection from one Guarantor, so each Guarantor will have to respond to many requests or risk being slashed. Each Execution Node will be flooded with copies of the Collection if it is available. We are operating under the theory that if MCCs have a very high probability of being resolved correctly (Lemma 3), spurious MCCs should be very rare specifically because of the described penalties.
- If most Execution Nodes are Byzantine and raise spurious MCCs, but at least one Execution Node is honest and generates complete Execution Receipts, a correct Execution Receipt will be sealed (assuming an honest super-majority of Collector Nodes and Consensus Nodes). Furthermore, the Execution Nodes who raised the spurious MCC will be slashed.
- If most Guarantors of a particular Collection are Byzantine, and refuse to provide Collection data, but at least one Guarantor is honest, the Collection will be provided to an honest Execution Node and executed properly.
- A cabal of 100% of Execution Nodes acting maliciously can halt the network by not executing new blocks. Nevertheless, no faulty state can be introduced into the network by such a denial of service attack.
- In order for an attacker to obtain the ability to introduce an error into the computation state (with non-negligible probability), the attacker would need to control a Byzantine fraction of 100% of Execution Nodes *and* more than $\frac{2}{3}$ of Verifier Nodes.

Theorem 3 *Liveness of Collection Text Resolution*

Given a system with
- more than $2/3$ of the Consensus Nodes' stake is controlled by honest actors;
- more than $2/3$ of the Collector Nodes' stake is controlled by honest actors;
- and partially synchronous network conditions with message traversal time bounded by Δ_t.

The system can be configured such that any guaranteed collection is available with probability close to 1.

Proof of Theorem 3

For this proof, we assume that Byzantine nodes collaborate to the maximum extent possible to

prevent a collection from being resolved. Unless there are protocol-level guarantees, we consider the most favorable conditions for the Byzantine nodes. Let us assume that the Byzantine nodes successfully prevented a collection from being resolved, i.e., more than 2/3 of the collectors issued an MCA. Let

- N the total number of collector nodes, $\widetilde{N}$ the number of honest collectors, and $\bar{N}$ the number of Byzantine collectors;
- n_{cluster} the size of the collector cluster that produced the missing collection, $\tilde{n}_{\text{cluster}}$ the number of honest collectors in the cluster, and $\bar{n}_{\text{cluster}}$ the number of Byzantine collectors in the cluster;
- n_{g} the number of guarantors of the missing collection, $\tilde{n}_{\text{g}}$ the number of honest guarantors, and $\bar{n}_{\text{g}}$ the number of Byzantine guarantors.

We consider a system configuration with $N = \widetilde{N} + \bar{N} = 1000$ collector nodes where $\widetilde{N} = 2\lfloor N/3 \rfloor + 1$. In Flow, the clusters are created by randomly partitioning the collectors into clusters of size n_{cluster} via Fisher-Yates shuffling. Hence, the probability of drawing a cluster with $\bar{n}_{\text{cluster}}$ Byzantine actors is given by the hypergeometric distribution

$$\mathcal{P}_{n_{\text{cluster}}, N, \bar{N}}(\bar{n}_{\text{cluster}}) = \frac{\binom{\bar{N}}{\bar{n}_{\text{cluster}}} \binom{N - \bar{N}}{n_{\text{cluster}} - \bar{n}_{\text{cluster}}}}{\binom{N}{n_{\text{cluster}}}}. \tag{18}$$

For a collection, at least $n_{\text{g}} = 2\lfloor n_{\text{cluster}}/3 \rfloor + 1 = \bar{n}_{\text{g}} + \tilde{n}_{\text{g}}$ guarantors are required. The number of Byzantine guarantors $\bar{n}_{\text{g}}$ could take any value in $0, 1, \ldots, \bar{n}_{\text{cluster}}$. There could be more Byzantine nodes in the cluster than required to guarantee the collection, i.e., $n_{\text{g}} < \bar{n}_{\text{cluster}}$. In this case, we assume that only the minimally required number of n_{g} Byzantine nodes would guarantee the collection to minimize slashing.

$$\bar{n}_{\text{g}} \in \left\{ 0, 1, \ldots, \min\left(n_{\text{g}}, \bar{n}_{\text{cluster}}\right) \right\}. \tag{19}$$

As each honest guarantor increases the probability of a collection being successfully retrieved, we assume that the Byzantine nodes only involve the absolute minimum number of honest nodes to get the collection guaranteed:

$$\tilde{n}_{\text{g}} = n_{\text{g}} - \bar{n}_{\text{g}} = 2\lfloor n_{\text{cluster}}/3 \rfloor + 1 - \bar{n}_{\text{g}} \tag{20}$$

When an honest collector that is *not* a guarantor receives a MCC, it selects κ guarantors and requests the collection from them. We assume that only honest guarantors would answer such a request. The probability for a correct node to receive no answer when inquiring about the missing collection, i.e., to issue in MCA, is

$$\mathcal{P}_{\kappa, n_{\text{g}}, \tilde{n}_{\text{g}}}(0) = \frac{\binom{n_{\text{g}} - \tilde{n}_{\text{g}}}{\kappa}}{\binom{n_{\text{g}}}{\kappa}}. \tag{21}$$

Furthermore, every Byzantine node that is *not* a guarantor of the collection would issue an MCA to increase the chances that the Missing Collection Challenge is accepted. Hence, there are $(\bar{N} - n_{\text{g}})$ MCAs from Byzantine nodes. For a collection to be considered missing, the protocol require $\frac{2}{3}N$ collectors to send MCAs. Consequently, the minimally required number of MCAs from honest nodes is

$$r = \widetilde{N} - (\bar{N} - \bar{n}_{\text{g}}). \tag{22}$$
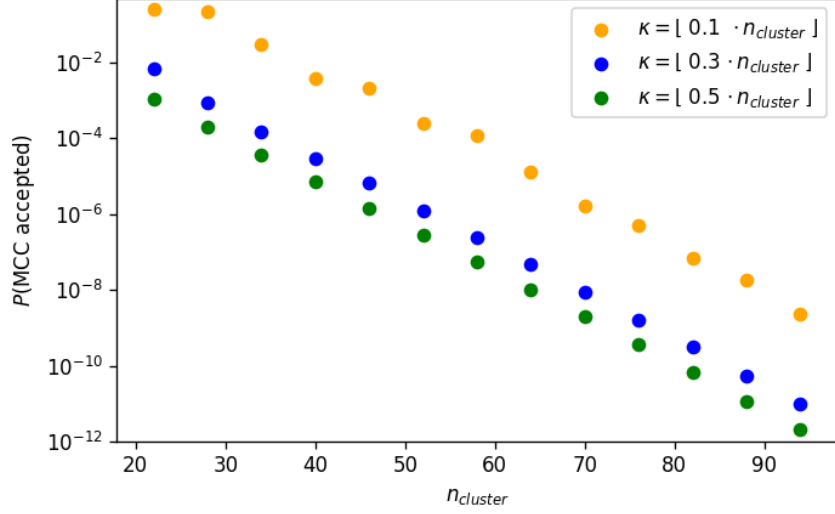
Figure 7: Worst-case probability that a collection cannot be resolved. The graph shows numerical values for $P(\text{MCC accepted})$, equation (25), for $N = 1000$, $\widetilde{N} = 667$ and $\bar{N} = 333$.

As honest nodes independently contact guarantors, each honest node conducts a Bernoulli trial and issues a MCAs with probability $\mathcal{P}_{\kappa,n_{\mathrm{g}},\tilde{n}_{\mathrm{g}}}(0)$. Consequently, the probability that r honest nodes issue MCAs follows a Binomial distribution

$$\mathcal{B}(r) = \binom{\widetilde{N}}{r} \left[\mathcal{P}_{\kappa,n_{\mathrm{g}},\tilde{n}_{\mathrm{g}}}(0)\right]^{r} \left[1 - \mathcal{P}_{\kappa,n_{\mathrm{g}},\tilde{n}_{\mathrm{g}}}(0)\right]^{\widetilde{N}-r}. \tag{23}$$

Given the number $\bar{n}_{\mathrm{cluster}}$ of byzantine actors in the cluster, the worst-case probability that the MCC is accepted by the system is

$$P(\text{MCC accepted} \,|\, \bar{n}_{\mathrm{cluster}}) = \max_{\substack{\bar{n}_{\mathrm{g}} \in \\ \{0,1,\ldots,\min(n_{\mathrm{g}}, \bar{n}_{\mathrm{cluster}})\}}} \sum_{\substack{r= \\ \widetilde{N}-\bar{N}+\bar{n}_{\mathrm{g}}}}^{\widetilde{N}} \mathcal{B}(r). \tag{24}$$

The overall probability of an MCC being accepted is, therefore,

$$P(\text{MCC accepted}) = \sum_{\substack{\bar{n}_{\mathrm{cluster}} \in \\ \{0,\ldots,n_{\mathrm{cluster}}\}}} P(\text{MCC accepted} \,|\, \bar{n}_{\mathrm{cluster}}) \cdot \mathcal{P}_{n_{\mathrm{cluster}},N,\bar{N}}(\bar{n}_{\mathrm{cluster}}) \tag{25}$$

Figure 7 illustrates the worst-case probability of a chunk missing, i.e., that a MCC shows the probabilities according to equation (25).

$\square$

4.3 Placing errors in chunks checked by colluding Verifier Nodes

If an Execution Node and several Verifier Nodes are colluding, they have the ability to secretly determine which chunks would be checked by the colluding Verifiers before even publishing an Execution Receipt. However, the self-assignment scheme defined in section 3.4 is independent for

each Verifier and in addition non-predictable for anyone without the Verifier's private key. Therefore, honest Verifiers will still check each chunk with uniform probability, independently of the colluding Verifiers. Consequently, if a malicious Execution Node wants to introduce a computation error, there is no advantage in placing the error in chunks that are checked by colluding Verifiers. This insight is formalized as Corollary 4.

Corollary 4

Given a system with partially synchronous network conditions with message traversal time bounded by Δ_t. Let there be a malicious Execution Node that tries to introduce a computation error into one of the chunks of a block. The success probability cannot be increased by the chunk selection of Byzantine Verifiers.

Acknowledgments

References

[1] Alexander Hentschel, Dieter Shirley, Layne Lafrance, and Ross Nicoll. Flow: Separating consensus and compute, 2019.

[2] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 120–, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[4] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in ethereum. 2017.

[5] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *ASIACRYPT*, 2018.

[6] Jae Kyun Kwon. Tendermint : Consensus without mining. 2014. https://github.com/cosmos/cosmos/tree/master/tendermint.

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

[8] Jae Kwon and Ethan Buchman. Cosmos - A Network of Distributed Ledgers. 2016. https://cosmos.network/cosmos-whitepaper.pdf.

[9] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

[10] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.

[11] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Milliar-Smith, R. E. Shostak, and C. B. Weinstock. Tutorial: Hard real-time systems. chapter SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control, pages 560–575. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[12] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[13] L. Lamport. The Weak Byzantine Generals Problem. *J. ACM*, 30(3):668–676, July 1983.

[14] Heidi Howard. *Distributed consensus revised*. PhD thesis, 2018. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf.

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[17] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, 2019. ACM.

[18] Vlad Zamfir. A Template for Correct-By-Construction Consensus Protocols. 2017. https://github.com/ethereum/research/tree/master/papers/cbc-consensus.

[19] Vlad Zamfir. Casper the Friendly Ghost: A 'Correct By Construction' Blockchain Consensus Protocol. 2017. https://github.com/ethereum/research/blob/master/papers/CasperTFG.

[20] Vlad Zamfir, Nate Rush, Aditya Asgaonkar, and Georgios Piliouras. Introducing the "Minimal CBC Casper" Family of Consensus Protocols. 2018. https://github.com/cbc-casper/cbc-casper-paper.

[21] Sarah Azouvi, Patrick McCorry, and Sarah Meiklejohn. Betting on blockchain consensus with fantomette. *CoRR*, abs/1805.06786, 2018.

[22] 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.

[23] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/docs/proto`.

[24] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 223–235, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[25] Ethereum Foundation. Patricia tree. `https://github.com/ethereum/wiki/wiki/Patricia-Tree`.

[26] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[27] Ronald A. Fisher and Frank. Yates. *Statistical tables for biological, agricultural and medical research / by the late Sir Ronald A. Fisher and Frank Yates*. Longman [Harlow], 6th ed. revised and enlarged. edition, 1974.

[28] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[29] G. Barrie Wetherill. *Acceptance sampling: basic ideas*, pages 12–43. Springer US, Boston, MA, 1977.

[30] E.G. Schilling and Dean Neubauer. *Acceptance Sampling in Quality Control*. CRC Press, 2017.

[31] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[32] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Advances in Cryptology – CRYPTO 2018 - 38th Annual International Cryptology Conference, 2018, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 757–788. Springer-Verlag, 1 2018.