Flow: Separating Consensus and Compute
– Block Formation and Execution –

Dr. Alexander Hentschel

alex.hentschel@dapperlabs.com

Dr. Yahya Hassanzadeh-Nazarabadi

yahya@dapperlabs.com

Ramtin Seraj

ramtin.seraj@dapperlabs.com

Dieter Shirley

dete@dapperlabs.com

Layne Lafrance

layne@dapperlabs.com

Abstract

Most current blockchains are built as a homogeneous system, comprised of *full nodes*, which are responsible for all tasks: collecting the transactions, block formation, consensus, and transaction execution. Requiring all full nodes to execute all tasks limits the throughput of existing blockchains, which are well documented and among the most significant hurdles for the widespread adoption of decentralized technology [1, 2, 3].

This paper is a follow-up on our previous proof-of-concept, Flow [4], a pipelined blockchain architecture, which separates the process of consensus on the transaction order from transaction computation. As we experimentally showed in our previous white paper, this provides a significant throughput improvement while preserving the security of the system. Flow exploits the heterogeneity offered by the nodes, in terms of bandwidth, storage, and computational capacity, and defines the roles for the nodes based on their tasks in the pipeline, i.e., Collector, Consensus, Execution, and Verification. While transaction collection from the user agents is completed through the bandwidth-optimized Collector Nodes, the execution of them is done by the compute-optimized Execution Nodes. Checking the execution result is then distributed among a more extensive set of Verification Nodes, which confirm the result is correct in a distributed and parallel manner. In contrast to more traditional blockchain architectures, Flow's Consensus Nodes do not execute the transaction. Instead, Verification Nodes report observed faulty executions to the Consensus Nodes, which adjudicate the received challenges and slash malicious actors.

In this paper, we detail the lifecycle of the transactions from the submission to the system until they are getting executed. The paper covers the Collector, Consensus, and Execution role. We provide a protocol specification of collecting the transactions, forming a block, and executing the resulting block. Moreover, we elaborate on the safety and liveness of the system concerning these processes.

Contents

1 Introduction

1.1 Architecture Overview

In most conventional blockchains, all the operational tasks are executed by *full nodes*, i.e., each full node is expected to collect and choose the transactions to be included in the next block, execute the block, come to a consensus over the output of the block with other full nodes, and finally sign the block, and append it to the chain. The structure resembles the single-cycle processing architecture of the early computers, where one instruction (i.e., block) is executed per step. On the other hand, the pipelined structure offered by Flow resembles the pipelined architecture of modern CPU design. While some unit of Flow is executing the current block, the pipeline architecture allows another unit to generate the next block, which results in an improvement over the throughput.

In contrast to the majority of existing blockchains, which operate on the assumption that nodes are homogeneous, Flow considers resources heterogeneous in their network bandwidth, computation power, and storage capabilities. The Flow blockchain is built on a diverse set of heterogeneous nodes, each performing a specific set of tasks associated with their role(s). Specifically, there are four different roles: *Collector Nodes, Consensus Nodes, Execution Nodes, and Verification Nodes*, which handle the collection of transactions, generating blocks, executing blocks, and verifying the execution results, respectively. Flow allows a high level of participation in Consensus and Verification by requiring only a high-end consumer internet connection while leveraging large-scale data centers to operate as Execution Nodes. All types of nodes in Flow receive compensation through crypto-economic incentives. Prior research indicates that exploiting the heterogeneity of the nodes and taking the pipelined approach results in a throughput improvement by a multiplicative factor of 56 [4] compared to homogeneous architectures using only full nodes. We also showed that the decentralization and safety of the network is preserved by the Flow architecture [5].

It is worth noting that the separation of consensus from execution is done based on the separation of the *subjective* from *objective* operations. The *Subjective* operations are those that do not have a deterministic result. Instead, they require a *consensus* to be reached over the result. An example of a *Subjective* act is the set of transactions to be included in the next block. On the other hand, the *Objective* operations are the ones with a deterministic result and do not require the consensus over their result. An example of an *Objective* act is the execution of a transaction that transfers some tokens between two accounts. In Flow, the separation of consensus and execution is done by assigning the *Subjective* tasks to the Consensus Nodes, and the *Objective* tasks to the Execution Nodes. Hence, even a few data centers may run the Execution Nodes. Nevertheless, their computation results are deterministic, verifiable, attributable, punishable, and recoverable, and do not compromise the decentralization of the system. We iterate over these terminologies in the rest of this section.

1.2 Design Principles

For Flow to be secure against Byzantine failures and attacks, we identify the following core architectural principles. In this paper, by an honest actor, we mean a node that exactly follows protocol associated with its role and never deviates from the protocol.

- *Detectable*: Any honest actor in the network can detect deterministic faults and prove the fault to all other honest nodes. The proof only requires asking other honest nodes to redo the faulty parts.

4

- *Attributable*: For performance reasons, many tasks in Flow are assigned randomly to a group of nodes. While the assignments are random for security reasons, it is based on *Verifiable Random Functions (VRFs)* [6]. Hence, any fault upon detection is also attributable to the responsible node(s) of the associated task.

- *Punishable*: Every node participating in the Flow network must put up a stake, which is slashed when a detected fault is attributed to the node. 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 act as countermeasures against the *attempts* of the malicious nodes to inject faults into the system. The probability of successfully injecting such faults is negligible.

1.2.1 Safety and Liveness

As a distributed multi-process system, the correctness of Flow protocol sets are evaluated against their *safety* and *liveness* [7]. A distributed multi-process system is safe against a set of features, if it provides guarantees that those features never happen in the system, unless with a negligible probability in some specific system parameter. Similarly, the system shows liveness respect to a set of features, if it provides guarantees that those features always persist, unless with a negligible probability in some specific system's parameter. In this paper, we formally identify the liveness and safety features of the introduced protocols. We prove Flow is safe and live in the presence of a limited fraction of Byzantine actors in each role.

In the design of Flow, we *prioritize safety over liveness* in case of a network split. A network split happens when at least a non-empty subset of nodes in the system is not accessible by rest of the nodes. In the extremely unlikely circumstance of a large-scale network split, where no connected subset of the network includes enough nodes to make forward progress, we allow the network to halt. This preserves the safety of the network even in the face of intentional network disruption, such as with an Eclipse attack [8].

1.3 Assumptions

The Flow architecture makes the following set of assumptions:

- *Consensus and Authentication*

  - All nodes participating in the system are known to each other.
  - Each node is authenticated through its unforgeable digital signature.
  - To reduce the computational cost and preserve the sustainability of the system, the consensus protocol of Flow is based on Proof of Stake (PoS).

- *Participation in network*

---

[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 our previous white paper the procedure of distribuing and parallelizing the verification of the computation of the blocks [5]. Hence, each Verification Node only performs a small fraction of the overall block computation.

- The evolution of the Flow blockchain is comprised of fixed intervals[2], called epochs.
- To participate in the network, a node must put up the minimum required stake for that role in a specific epoch
- A Flow node may participate over multiple epochs.

- *Source of randomness*

  - Flow requires a reliable source of randomness for seeding its pseudo-random number generators.
  - The source of randomness enables each seed to be unpredictable by any individual node until the seed itself is generated and published in a fully decentralized manner.
  - In Flow, we use the Distributed Random Beacon (DRB) protocol [9] to generate a fully decentralised, reliable source of randomness.

- *Cryptography Primitives*

  - Flow requires an aggregatable and non-interactive signature scheme, such as BLS [10].

- *Network Model*

  - Flow operates on a partially synchronous network with message traverse time bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t.
  - Flow uses a Byzantine Fault Tolerant message routing system, that guarantees message delivery with a high probability.

- *Rewarding and Slashing*

  - Flow requires adequate compensation and slashing mechanics that incentivize nodes to comply with the protocol.

- *Honest Stake Fraction*

  - Flow requires *more* than $\frac{2}{3}$ of stake from Collection Nodes, Consensus Nodes, and Verification Nodes to be controlled by honest actors (for each node role separately). We will refer to a group of nodes with *more* than $\frac{2}{3}$ of stake as a super-majority. A super-majorities of honest nodes probabilistically guarantees the safety of the Flow protocol.

1.4 Flow's Roles

In Flow, roles are the services that the nodes provide to the system. One may assume each role as a specific set of protocols that a node executes. Accordingly, in Flow, we have four different roles (i.e., set of protocols), which are: *Collector Role*, *Consensus Role*, *Execution Role*, and *Verification Role*. We refer to the network nodes that execute the set of respective protocols as *Collector Node*, *Consensus Node*, *Execution Node*, and *Verification Node*. For each role, a minimum stake deposit is required from each of the participating nodes. Hence, single hardware system may host multiple roles in the network by staking for each of them individually. However, Flow treats individual

---

[2]We envision that the length of an epoch will be measured in the number of blocks.

roles as if they are independent entities. In other words, each role is staked, unstaked, and slashed independently. The staked nodes associated with the roles are compensated through a combination of block rewards and transaction fees. Any combination of a public key and a role should be unique, and each peer has an independent staked value for each of its roles. We also recommend that multiple roles of a single peer do not share their staking keys for security reasons.

We present a detailed description of each node role below. A high-level illustration of the role interactions is shown in Figure 1. Figure 2 shows an overview of the transaction's lifecycle over considering the different roles of the nodes.

1.4.1 Collector Role

For the sake of load-balancing, redundancy, and Byzantine resilience, the Collector Nodes are staked equally and randomly partitioned into clusters of roughly identical size. At the beginning of an epoch, each Collection Node is randomly assigned to exactly one cluster. Each cluster of Collector Nodes acts as a gateway of Flow with the external world. In the mature Flow, we envision that a cluster will contain somewhere between 20 and 80 Collector Nodes.

An external client submits their transaction to a Collector Node. Upon receiving a submitted, well-formed [3] transaction, a Collector Node introduces it to the rest of its cluster. The Collector Nodes of one cluster batch the received transactions into so-called collections. Only a hash reference to a collection is submitted to the Consensus Nodes for inclusion in a block.

Each cluster of the collector nodes generates their collections one at a time. Before a new collection is started, the current one is closed and sent to the Consensus Nodes for inclusion in a block. The collections of a cluster are built collaboratively by each Collector Node sharing the transactions submitted to it with the rest of its cluster, and participating in a (light) consensus protocol with the other Collector Nodes of its cluster. The nodes come to consensus on both the end

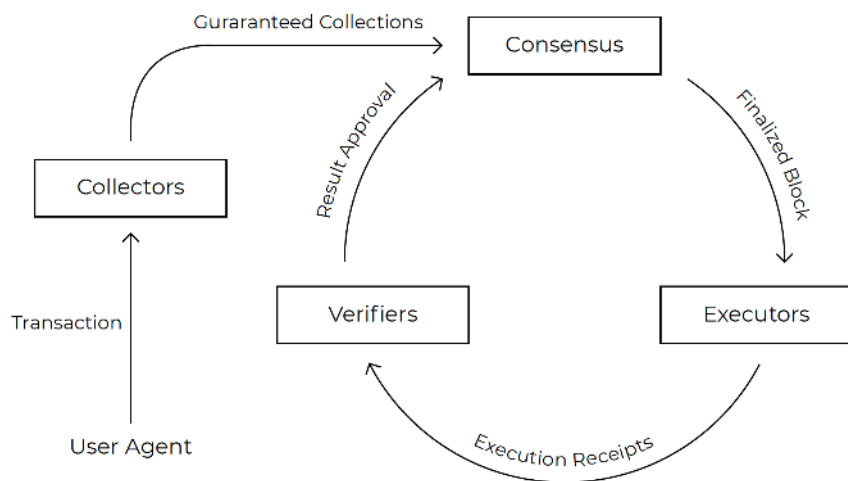Figure 1: An overview of the different roles in Flow as well as their interaction. For simplicity, only the message exchange during normal operation is shown. Messages for raising or adjudicating slashing challenges are omitted.

---

[3]As we detail later in this paper, a well-formed transaction has all required fields filled properly and contains valid signature(s) from staked account(s) of Flow

of the current collection, to start a new one, and on the transactions included in the new collection. The collection generated as the result of a consensus among the Collector Nodes of a cluster is called a guaranteed collection.

1.4.2 Consensus Role

In Flow, the Consensus Nodes maintain the chain of blocks and are responsible for the chain extension by appending new blocks. They receive hash references to the guaranteed collections that were generated by the Collector Nodes. Furthermore, Consensus Nodes run a Byzantine Fault Tolerant (BFT) consensus algorithm to reach an agreement over the set of collections to be included
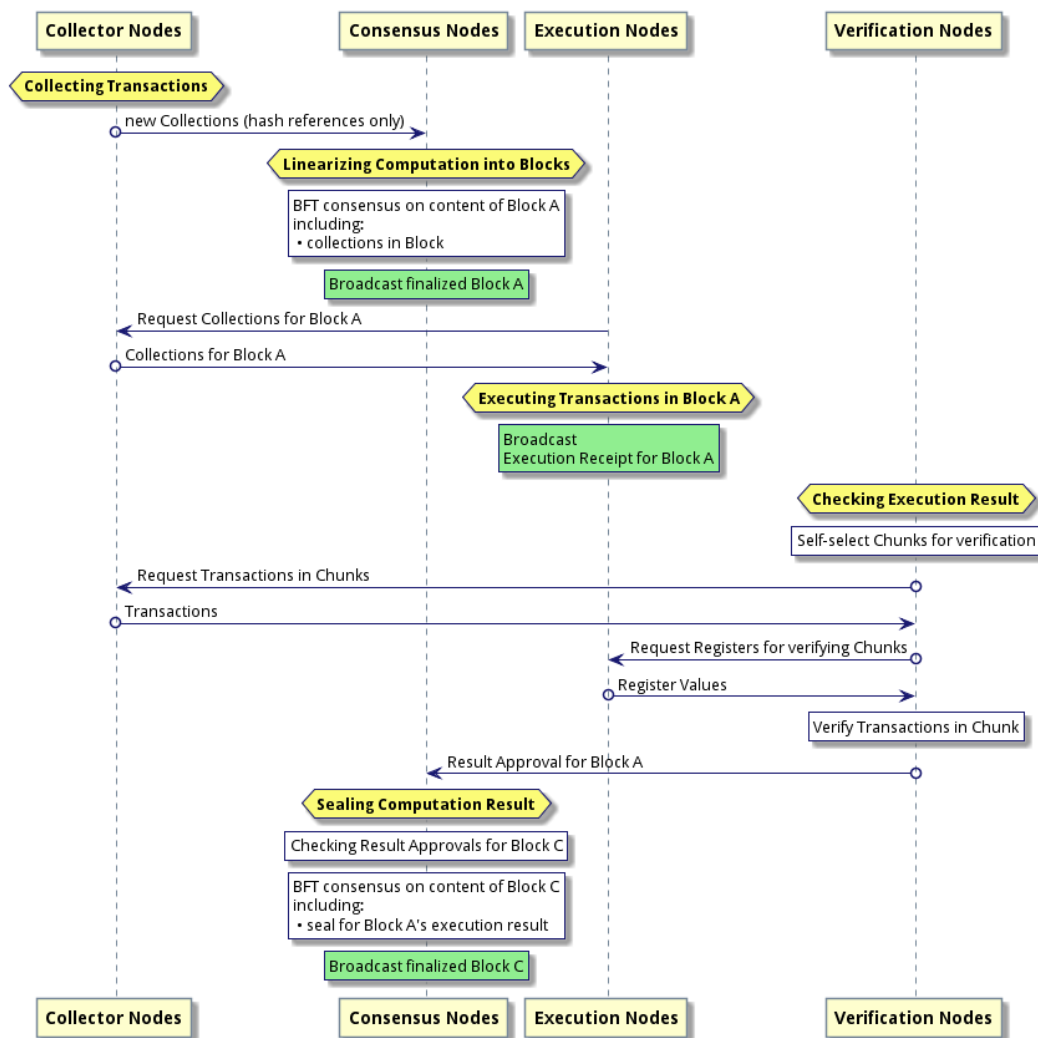
Figure 2: The lifecycle of a transaction in Flow. The yellow hexagons indicate the start of the individual stages in the Flow pipeline, e.g., the block sealing (i.e., Sealing Computation Results). The arrows show the inter-role message exchange between the nodes. Green rectangles correspond to the broadcasting events, in which a message is disseminated to all of the staked nodes. White rectangles represent the execution of the operations of the nodes. For the sake of simplicity, we represent the normal flow of the transaction lifecycle and ignored the message exchanges related to adjudicating slashing challenges.

8

in the next block. The block of the ordered collection that has undergone the complete BFT consensus algorithm is called finalized blocks. In Flow, a block specifies[4] the included transactions as well as the other inputs (e.g., the random seed), which are required to execute the computation. It is worth noting that a block in Flow does not include the resulting execution state of the block execution.

Consensus Nodes are also responsible for sealing a block. A Block Seal is a commitment to the execution result of a block after it is executed and verified (see [5] for more details on Block Sealing). Moreover, Consensus Nodes are responsible for maintaining a part of the state of the system related to the stakes of the nodes, receiving and adjudicating the slashing challenges, and slashing faulty nodes. We elaborate on the notion of protocol state on Flow in Section 1.5.

1.4.3   Execution Role

The Execution Nodes are powerful computational resources that are primarily responsible for scaling the computational power of Flow. Execution Nodes execute the finalized blocks generated by the Consensus Nodes. They publish the resulting execution state as an Execution Receipt. The Execution Nodes also need to provide the required information to the Verification Nodes so they can check the execution result. For this purpose, Execution Nodes break the computations of a block into chunks. Each Execution Node publishes additional information about each chunk in its Execution Receipt for the block executed. We detail the block formation process and chunking in Section 4 of this paper.

1.4.4   Verification Role

Verification Nodes are responsible for collectively verifying the correctness of the Execution Nodes' published results. With the chunking approach of Flow, each Verification Node only checks a small fraction of chunks. A Verification Node requests the information it needs for re-computing the chunks it is checking from the Execution Nodes. A Verification Node approves the result of a chunk by publishing a Result Approval for that chunk, which means that the Verification Node has verified and agrees with the result of the execution of that chunk.

Breaking the verification work into small chunks enables Verification Nodes to check the execution of chunks independently and in parallel. However, as we formally have shown in [5], all Verification Nodes together will check all chunks of the executed blocks with an overwhelming probability.

1.5   States, Staking, and Slashing

State: There are two sorts of states in Flow blockchain known as the execution state and protocol state. Each type of these states is maintained and updated independently. Both states are represented as key-value stores. While the execution state is maintained and updated by the Execution Nodes, the protocol state is maintained and updated by the Consensus Nodes.

The execution state contains the register values, which are modified by transaction execution. Although the updates on the execution state of the system are done by the Execution Nodes, the integrity of the updates is governed by Flow's verification process [5].

---

[4]A block *implicitly* specifies its transactions by referencing collections of transactions.

Protocol state, on the other hand, keeps track of the system-related features including, all the staked nodes, their roles, public keys, network addresses, and staking amounts. The protocol state is updated when nodes are slashed, join the system via staking, or leave the system via un-staking. Consensus Nodes publish updates on the protocol state directly as part of the blocks they generate. The integrity of the protocol state is guaranteed by the consensus protocol. We elaborate on updates to the protocol state, affected by the consensus protocol, in Section 4 of this paper.

Staking: A node in Flow is required to deposit some stake in order to run a role. This requires the node to submit a staking transaction. The staking transactions for the next epoch take place before a specific deadline in the current epoch. Once the staking transaction is processed by the Execution Nodes, the stake is withdrawn from the node's account balance and is explicitly recorded in the Execution Receipt. Upon Consensus Nodes sealing the block that contains this staking transaction they update the protocol state affected by this staking transaction, and publish the corresponding staking update in the block that holds the seal. Staked nodes are compensated through both block rewards and transaction fees and all roles require a minimum stake to formally participate in that role.

Slashing: Any staked node of Flow can detect and attribute misbehavior to another staked node who committed it. Upon detecting and attributing misbehavior, the node issues a slashing challenge against the faulty node. Slashing challenges are submitted to the Consensus Nodes. The slashing challenge is a request for slashing a staked node due to misbehavior and deviation from the protocol. As the sole entity of the system responsible for updating the protocol state, Consensus Nodes adjudicate slashing challenges and adjust the protocol state (i.e., staking balances) of the faulty nodes accordingly. Based on the result of adjudication, the protocol state (i.e., the stake) of a node may be slashed within an epoch. The slashing update is announced in the subsequent blocks and is effective for all honest nodes as soon as they process the block containing the respective stake update.

2 Preliminaries

2.1 Adversarial Model

We denote the *honest* nodes (i.e., non-Byzantine nodes) as the ones that follow the description of the Flow protocols. We call a node *Byzantine* if it deviates from any of Flow's protocols at any arbitrary point. We also presume the non-responsiveness of a staked node, to messages and queries, to be a Byzantine act.

Definition 2.1 *Effective Votes*
*During the consensus among a group of nodes, we consider the **effective votes** of the nodes as the overall staked fraction of the nodes who vote positively in favor of the consensus proposal. The fraction is taken over the entire stakes of the nodes in that group.*

For the Collector Role, Consensus Role, and Verification Role, we assume that *more than* $\frac{2}{3}$ of the accumulated stake of each role belong to the honest nodes, and the rest is owned by the Byzantine nodes. For example, more than $\frac{2}{3}$ of the stakes of the Collector Nodes belong to the honest ones and less than $\frac{1}{3}$ of their overall stakes are owned by the Byzantine ones.

2.2 HotStuff

HotStuff [11, 12] is a distributed Byzantine Fault Tolerant (BFT) consensus algorithm. In this subsection, we present a summary of the basic HotStuff protocol. However, we scope out the optimization improvements which are presented in the HotStuff proposal for sake of brevity, and refer the interested readers to [12].

HotStuff assumes the nodes as state machines that hold the same shared replicated data (e.g., the blockchain ledger), and aim to reach consensus over the transition of the next state of the replicated data (e.g., the set of the transactions to be included into the next block). HotStuff is live under partially-synchronous network conditions. It is safe if less than one third of the consensus nodes' total stake is controlled by Byzantine actors.

The protocol progresses in rounds, i.e., each node increments a local counter when reaching consensus, or a timeout. For sake of liveness, the nodes double their timeout interval each time they move to a new round as the result of a time out. In each round, a unique consensus node assumes the role of the leader. Nodes can consistently identify the leader of each round by invoking a deterministic function locally on the round number. The leader advances the consensus via a three-phase commit protocol of prepare, pre-commit, and commit. The consensus at each round starts with a block proposal by the leader, and ends with reaching a consensus over the proposal or a timeout. In Flow, we assume that moving from one phase to another phase of HotStuff requires a minimum effective vote of $\frac{2}{3}$ in favor of progressing with the leader's proposal. The leader collects and aggregates the votes and broadcasts the aggregated signature to the entire system as a proof of moving to the next phase. Hence, each node can track and verify the correctness of the consensus progress in each phase by confirming that there are at least $\frac{2}{3}$ effective votes.

The correctness of the HotStuff protocol is formally proven via the safety and liveness theorems [12]. The safety implies that all honest nodes will eventually commit the same state transitions (i.e., blocks) in the same order. The liveness property of HotStuff guarantees that, after a Global Synchronization Time has passed, there is a bounded interval that enables an honest leader to advance the round towards reaching a consensus. Furthermore, HotStuff provides deterministic finality,

i.e., the protocol guarantees that there will be no forks in the chain of committed state transitions. This essentially means that the canonical chain of replicated state transitions (e.g., the ledger of blocks) solely acts as an append-only chain. Hence, in contrast to many existing consensus protocols (including Bitcoin's Proof-of-Work (PoW)), chain reorganizations do not occur in HotStuff[5].

The advantages of using HotStuff compared to the existing BFT leader-based counterparts include

- Having n nodes participating in the consensus, the communication complexity of HotStuff is $O(n)$, which makes switching of faulty or non-responsive leaders relatively easy.

- The proposal of a failed or timeout leader has the potential to be resumed by the leader of the next round.

- Preferring safety over the liveness, i.e., the safety over HotStuff is guaranteed regardless of the underlying synchrony model. However, for the protocol to guarantee its liveness and the progress, the Global Synchronization Time should be passed.

- Ability to operate under partially-synchronous network conditions.

- It has the so-called *responsiveness* property, i.e., an honest leader advances the protocol to the consensus in a time that is bounded by the message transmission delay, i.e., Δ_t.

- It has deterministic finality.

2.3 Distributed Random Beacon

In Flow, we utilize the Distributed Random Beacon (DRB) protocol [9] among a subset of nodes to generate an unbiased and verifiable source of randomness in a fully decentralized manner. By unbiased, we mean that no adversarial party can manipulate the source of randomness towards its desired distribution. By verifiability, we mean that once the source of randomness is determined, it is verifiable against the correct execution of the protocol. As a decentralized protocol, DRB is executed collaboratively among a subset of nodes as an interactive multi-party protocol.

In Flow, we consider DRB as a 2-phase protocol consisting of three polynomial time protocols: `Setup`, `RandomGenerator`, and `Verify`. The first phase of DRB is the setup phase that happens only once among the participants. The second phase of DRB is the random generation phase, which is a repetitive operation, i.e., it may happen many times within the lifetime of the system. Each time the participants enter the random generation phase, they generate a random string that is denoted as the *source of randomness*. In this paper, we call every single execution of the random generation phase of DRB a *DRB round*.

- $(V_G, \text{sk}_i) \leftarrow \text{Setup}(P_i, 1^\lambda)$: `Setup` is a probabilistic decentralized *protocol* where each participating party i receives its private key, i.e., sk_i, as well as a public verification vector, i.e., V_G. We elaborate on these in the rest of this section.

- $r \leftarrow \text{RandomGenerator}(\text{sk}_i, \text{proto}(b))$: `RandomGenerator` is a deterministic decentralized *protocol* where parties collaboratively generate a fresh source of randomness. For each participating party in this protocol, the inputs are the party's private key (i.e., sk_i) and a *proto*

---

[5]Provided that the stake controlled by Byzantine actors is strictly less than $\frac{1}{3}$.

block (i.e., $proto(b)$). The output of the protocol is the fresh source of randomness, i.e., r. We elaborate on the proto block notion in Section 4 of this paper.

- `Verify`$(r, proto(b), V_G) =$ `True/False`: `Verify` is a deterministic protocol that is executed locally by any external or internal party to DRB. Knowing a generated source of randomness (i.e., r), its associated proto block (i.e., $proto(b)$) and the public verification vector of the DRB (i.e., V_G), one can deterministically verify the correctness of the protocol execution.

In addition to being unbiased, DRB provides the following properties:

- **Unpredictability:** Denoting the security parameter of the system by λ, before the execution of a new round of DRB, for each probabilistic polynomial-time predictor A, there exists a negligible function $negl(\lambda)$, such that:

$$Pr[A(r_1, r_2, r_3, ..., r_{x-1}, proto(b_1), proto(b_2), proto(b_3), ..., proto(b_{x-1}), proto(b_x)) = r_x]$$
$$\leq negl(\lambda)$$

  where r_i is the output of the ith execution of DRB, and $proto(b_i)$ is the associated proto block to the r_i. In other words, given all the generated entropies as well as their associated proto blocks to the predictor A, it is not able to predict the output of the next round before its execution unless with a negligible probability in the security parameter of the system.

- **Verifiability:** Given the generated source of randomness of a round, its corresponding proto block, as well as some public metadata of the protocol, any external party to the protocol can verify the generated source of randomness.

2.3.1 DRB Random Generation Phase (Threshold Signatures)

The main idea of DRB is based on the existence of a deterministic and unique non-interactive *threshold signature* protocol. BLS signature primitive [10] provides all these required properties. A threshold signature is a tuple of four polynomial-time algorithms known as (`Gen, Sign, Recover, Verify`), which is defined over the set G of n parties (i.e., $G = \{P_1, P_2, ..., P_n\}$) and is identified by two parameters: t and n. An (t, n)-threshold signature enables the set of n parties to collaboratively generate an *aggregated signature* over a message m using a distributed secret key in a decentralized manner. The threshold signature enables any entity to efficiently verify a valid signature over m against the distributed secret key. The verification is solely done by verifying the aggregated signature, without requiring the individual party's signatures.

- $(V_G, \text{sk}_1, \text{sk}_2, ..., \text{sk}_n) \leftarrow$ `Gen`(1^λ): `Gen` is a probabilistic polynomial-time algorithm that on receiving the security parameter, it implicitly generates a secret key sk_G and distributes individual key shares over the participating parties i.e., sk_i for the ith party. sk_G remains secret and not explicitly computed although it is jointly generated by all parties. Moreover, a public verification vector V_G is generated. V_G enables parties to recover each others' public key shares. By the public key of the ith party, we mean the public key pk_i associated with sk_i. V_G also contains the public key of the group of parties, i.e., pk_G associated with sk_G.

13

- $\sigma_i \leftarrow$ Sign(sk_i, m): Sign is a deterministic polynomial-time algorithm that is executed by each party individually. On receiving a message m and the private key of the ith party (i.e., sk_i), Sign(sk_i, m) generates a signature σ_i on the message m that is attributable to the ith party. σ_i is also commonly called the **threshold signature share** of the ith party. Sign is the same signature function defined in the BLS signature scheme.

- $\sigma \leftarrow$ Recover$(P_{i_1}, P_{i_2}, ..., P_{i_{t+1}}, \sigma_{i_1}, ..., \sigma_{i_{t+1}})$: Recover is a deterministic polynomial-time algorithm that is executed by any entity either internal or external to the group of parties. On receiving the distinct threshold signature shares σ_{i_k} and the list of corresponding distinct parties, Recover combines all the signature shares and recovers the group signature, i.e., σ. Having any combination of more than t signature shares of the parties over the same message m, Recover generates a group signature σ that is attributable to the group secret key, i.e., sk_G.

- Verify$(\sigma, \text{pk}_G, m) =$ True/False: Verify is a deterministic polynomial-time algorithm that is executed by any entity either internal or external to the group of parties. On receiving a recovered threshold signature (i.e., σ) on a message m, it returns True if the signature σ was generated using the group secret key (i.e., sk_G), and returns False otherwise. Verify is the same signature function defined in the BLS signature scheme.

While Sign, Recover and Verify are non-interactive and relatively inexpensive operations, Gen requires to run an interactive and relatively slow protocol by all parties. Keeping the performance factor in mind, Gen is separated from the repetitive DRB random generation phase in Flow. Gen serves as the Setup phase while the tuple (Sign, Recover, Verify) serves as the random generation phase. This is possible because the random generation allows generating repetitive signatures using sk_G without explicitly reconstructing it, and therefore protecting its confidentiality. The DRB random generation phase is started every time the nodes need to generate a new source of randomness. Protocol 1 shows a DRB random generation round for a specific proto block, i.e., $proto(b)$. To generate a new source of randomness, each node i computes and broadcasts a threshold signature share over $proto(b)$. Upon receiving at least $t + 1$ distinct threshold signature shares $\{\sigma_{b,1}, \sigma_{b,2}, \sigma_{b,3}, ..., \sigma_{b,t+1}\}$ over the proto block, any party can aggregate the signatures into a threshold signature σ_b, using the function Recover. If the signature σ_b is correctly formed, it is verifiable against the public key of the group G, i.e., Verify$(\sigma_r, proto(b), \text{pk}_G) =$ True. The source of randomness of this DRB random generation round is σ_b. The deterministic digest of σ_b is used to seed the pseudo-random generators during the current round.

Protocol 1 DRB Random Generator

Inputs: For each $i \in [1, n]$, party P_i holds the inputs of $proto(b)$ and $\mathtt{sk}_i$. H is a cryptographic hash function.

Goal: Parties jointly compute a fresh source of randomness, r, based on $proto(b)$.

The protocol:

1. Each party P_i generates $\sigma_{b,i} \leftarrow \mathtt{Sign}(proto(b), \mathtt{sk}_i)$, and broadcasts $\sigma_{b,i}$ to all other parties.

2. Any internal or external party then captures the broadcasted $\sigma_{b,j}$

3. Upon having at least $t + 1$-many $\sigma_{b,.}$ from $t + 1$ distinct parties, each party recovers the aggregated threshold signature $r = \sigma_b \leftarrow \mathtt{Recover}(P_{i_1}, ..., P_{i_{t+1}}, \sigma_{b,i_1}, ..., \sigma_{b,i_{t+1}})$.

4. Having $r = \sigma_b$, any party extracts a fresh seed as $H(r)$

In Flow, we use [13] as the underlying threshold signature scheme for the DRB, which is non-interactive and provides unique signatures. By non-interactive, we mean that the protocol is modeled by a single round of communication. Uniqueness is an inherited property from the BLS scheme, and it means that there exists a unique aggregated threshold signature for each pair of $(\mathtt{pk}_G, m)$. Hence, the aggregated threshold signature is deterministic and is always the same regardless of the subset of collected threshold signature shares over the same message. The non-interactiveness feature is for sake of improvement on the communication complexity of the DRB protocol over several instances (i.e., rounds) of execution. The uniqueness is vital to guarantee a consistent source of randomness independent of the subset of threshold signature shares, i.e., every valid subset of $t + 1$-many threshold signature shares results in the same aggregated threshold signature.

2.3.2 DRB Setup Phase (Distributed Key Generation)

Distributed Random Beacon (DRB) setup phase is to generate the keys for the threshold signature protocol. As shown by Protocol 2, the setup phase of DRB in Flow is mainly the $\mathtt{Gen}$ function of the threshold signature. During this phase, the parties of the group G collaboratively execute a *Distributed Key Generation* protocol (DKG) [14] as the decentralized implementation of $\mathtt{Gen}$ function. The DKG implicitly generates a group secret key $\mathtt{sk}_G$. As the result, each party P_i receives the public verification vector V_G as well as its individual private key $\mathtt{sk}_i$. The secret key $\mathtt{sk}_G$ is generated using entropy provided by all participating parties in an unbiased manner. It remains unknown to all participants although it can be reconstructed by any party receiving more than t shares of the secret key.

The DKG implemented in Flow is a variant of *Pedersen*'s protocol called *Joint-Feldman* [14] protocol. It is a Discrete Logarithm-based protocol and mainly constitutes of n parallel executions of the *Feldman verifiable secret sharing* (VSS) protocol with each party acting as a leader in exactly a single instance. Although it has been proven that Joint-Feldman protocol does not guarantee a uniform distribution of the secret key sk_G, the security of the DKG in Flow is based on the hardness of the Discrete Logarithm Problem, which is not weakened by the distribution bias.

Protocol 2 DRB Setup

Inputs: For each $i \in [1, n]$, party P_i holds its index in the set as well as the system's security parameter, λ.

Goal: Parties jointly compute a public verification vector V_G as well as their threshold signature private keys, i.e., $\mathtt{sk}_i$ for P_i.

The protocol:

1. Each party P_i invokes an instance of DKG protocol as specified in [14], and obtains:

   $(V_G, \mathtt{sk}_i) \leftarrow \mathtt{DKG}(P_i, 1^\lambda)$

2.4 Distributed Random Beacon Setup in Flow

We assume the setup phase of DRB happens once every epoch to prepare the DRB protocol keys. During the setup phase, a protocol-selected subset of n_s Consensus Nodes jointly execute the DRB setup protocol. Through executing the setup protocol, each node i generates its *DRB private key* $\mathtt{sk}_i$ for the threshold signature as well as a public verification vector V_G.

As explained in Section 2.3, (n, t) are the parameters of the threshold signature scheme. An adversary who corrupts up to t parties is not able to forge a valid threshold signature. It is also required that at least $t + 1$ parties act honestly to guarantee the liveness of the protocol. In order to satisfy both properties with the high probability, i.e unforgeability and liveness, we compute t as follows.

$$t = \lfloor \frac{n_s - 1}{2} \rfloor \tag{1}$$

In Flow, the size of the DRB committee n_s is tuned to ensure that unforgeability and liveness are guaranteed with a high probability in the security parameter of the system (i.e., λ). However, n_s also is chosen small enough to preserve the efficiency of the DRB execution with respect to the operational complexities.

3 Collection Formation

3.1 Overview

In Flow, collection formation denotes the process which starts with a user agent submitting a transaction to the Collector Nodes and ends when a guaranteed collection is submitted to the Consensus Nodes. The Collector Nodes are partitioned into clusters. Each transaction is assigned to a certain cluster based on its transaction hash. A user agent submits a transaction to some Collector Nodes in the responsible cluster. On receiving a transaction, the Collector Nodes of the cluster broadcast the transaction among each other. Collector Nodes of a cluster continuously form consensus over the set of transactions contained in the collection under construction. As a result of the consensus, a collection grows over time. Once one of two conditions is met: either the collection size reaches a threshold or a pre-defined time span has passed, the collection is closed and submitted to the Consensus Nodes for inclusion in a block.

3.2 Cluster Formation

In Flow architecture, the stake is a measure of the nodes' accountability. In specific for the Collector Nodes, the workload accountability of processing the submitted transactions and hence their compensation is a monotonically increasing function of their stake *deposit per processed transaction*. To make all the Collector Nodes equally accountable, we desire a setup where the stake deposit per processed transaction is similar for all the Collector Nodes. This setup with similarly staked Collector Nodes results in a similar workload distribution among them on processing the submitted transactions. This setup stands against the non-uniform stake distribution of the Collector Nodes, which results in a non-uniform workload of processing the submitted transactions among them.

Despite our equally staked Collector Nodes principle, the user agents submitting their transactions to an arbitrary Collector Node of their choice results in a biased distribution of the workload and its associated rewards. It is trivial that over the time the user agents may establish a relationship with Collector Nodes, e.g., running the client software provided by the Collector Nodes, submitting to the Collector Node that provides a better quality of service, etc. This, however, leads to a heterogeneous load distribution on the Collector Nodes and degrades the decentralization of the system. For example, a Collector Node may provide significantly higher utilization than the others, attract more rewards and essentially starve other Collector Nodes of income.

To design a system where the stake deposit per processed transaction is comparable for all collectors, Flow introduces the notion of *clusters*. Several Collector Nodes are grouped into the same cluster to collect and process the same set of transactions. In other words, instead of having a pool of transactions where each Collector Node arbitrarily selects transactions to process, there is a one-way deterministic assignment between each transaction and a cluster of the Collector Nodes using the source of randomness. Clustering of Collector Nodes is done at the beginning of each epoch, e.g., once every week. The number of clusters in Flow is a protocol parameter denoted by c. As the result of clustering, the Collector Nodes are randomly partitioned into c clusters. Clustering is done in a way that the size of each two different clusters varies by at most a single node. The assignment is nevertheless verifiable, i.e., each node is able to run the clustering algorithm offline and reach the same cluster assignment for the Collector Nodes as every other node. As we detail in the rest of this section, this enables the user agents to efficiently map their signed transactions to the cluster of the Collector Nodes that is responsible for processing them.

The clustering is done by each Collector Node running the cluster assignment algorithm as illustrated by Algorithm 3.1. The inputs to this algorithm are the list of all Collector Nodes' public keys (i.e., N_c), the number of clusters (i.e., c), and the source of randomness (i.e., r), which is generated by the Distributed Random Beacon protocol [9] (see Section 2.3). We denote the cardinality of N_c as n_c, which corresponds to the number of Collector Nodes in the system. The output of the algorithm is Cls, which is a map from the public keys of the Collector Nodes to their assigned clusters, i.e., for every $\texttt{pk} \in N_c$, element $\texttt{Cls}[\texttt{pk}]$ is the Collector Node's assigned cluster id.

The clustering is done by deriving a seed s from the source of randomness r solely for the clustering of Collector Nodes (Algorithm 3.1, Line 1). The seed is derived in a deterministic manner and has the same entropy as r. The derived seed is then being utilized by the Fisher-Yates shuffling algorithm [15]. On receiving the seed s and the list of Collector Nodes' public keys N_c, the Fisher-Yates algorithm turns N_c into a pseudo-random permutation of Collector Nodes' public keys, denoted by π_c. Note that as π_c is a permutation of N_c, it is of the same cardinality as N_c, i.e., $|N_c| = |\pi_c| = n_c$ (Algorithm 3.1, Line 2).

Once the permutation π_c is determined, the clustering algorithm partitions the Collector Nodes into c clusters. It does that by clustering the first k Collector Nodes into the cluster number 0, the second k Collector Nodes into the cluster number 1, and similarly the ith sequence of the Collector Nodes of size k into the cluster number $i - 1$. k is the size of each cluster and is determined as $k := \lfloor \frac{n_c}{c} \rfloor$ (Algorithm 3.1, Lines 3-10).

In case the number of Collector Nodes is a multiple of the number of clusters, i.e., $\lfloor \frac{n_c}{c} \rfloor = \frac{n_c}{c} = \lceil \frac{n_c}{c} \rceil = k$, clustering algorithm stratifies the Collector Nodes based on their position in the permutation π_c into c clusters of size k. Otherwise, if the number of Collector Nodes is *not* a multiple of the number of clusters, clustering based on Lines 3-10, results in some Collector Nodes to be leftover (Specifically, the number of leftover Collector Nodes will be less than c). In that situation, the clustering algorithm does not create an extra cluster to accommodate the leftovers. It rather distributes the leftovers among the existing clusters by adding the ith Collector Node of the leftovers to the ith cluster. Size of the leftovers in this case is $n_c \bmod c$, i.e., the remainder of n_c divided by c. Hence, clustering in this manner results $n_c \bmod c$ clusters of $k + 1$ Collector Nodes, and $c - n_c \bmod c$ clusters of k Collector Nodes (Algorithm 3.1, Lines 11-15). Having n_c Collector Nodes in the system, Algorithm 3.1 has both an asymptotic time and memory complexity of $O(n_c)$, and does not impose a communication overhead to the system.

Algorithm 3.1: ClusterAssignment

Input:
  N_c: List; element $N_c[i]$ is public key of ith Collector Node
  c: System-wide constant unsigned integer; number of the clusters
  r: byte array; Source of randomness
Output:
  Cls: Map; element Cls[pk] is the cluster id of Collector Node with public key of pk

  // Deriving a random seed from r for clustering Collector Nodes
1 $s := H(`collector'||`cluster'||r)$;
  // Shuffling the collectors list
2 $\pi_c := \texttt{FisherYates}(s, N_c)$;
  // k defines the size of clusters
3 $k := \lfloor \frac{n_c}{c} \rfloor$;
  // i keeps current cluster's id
4 $i := 0$;
  // j keeps current Collector Node's index in π_c
5 $j := 0$;
  // stratifying Collector Nodes into c clusters of size k
6 **while** $j < c \times k$ **do**
  | // adding j^{th} Collector Node to cluster i
7 | $\texttt{Cls}[\pi_c[j]] := i$;
  | // moving to the next Collector Node
8 | $j + +$;
9 | **if** $j \bmod k = 0$ **then**
  | | // current cluster has reached size k
  | | // moving to the next cluster
10 | | $i + +$;
11 $i := 0$;
12 **while** $j < n_c$ **do**
  | // adding j^{th} Collector Node to cluster i
13 | $\texttt{Cls}[\pi_c[j]] := i$;
  | // moving to the next cluster
14 | $i + +$;
  | // moving to the next Collector Node
15 | $j + +$;

3.3 Transaction submission

Listing 1 represents the structure of the Flow's transactions that the user agents submit to the Collector Nodes. In this listing, Script corresponds to the content of the transaction. The transaction's content manipulates the execution state of Flow, and requires a transaction fee to be processed. PayerSignature corresponds to the signature created by the account that is paying the transaction fee (e.g., the gas fee). The ScriptSignature of the transaction corresponds to the signatures of the execution state owners, which grant the transaction permission to manipulate their execution state. A transaction may manipulate several parts of the execution state and hence may require the signatures of several execution state owners. ReferenceBlockHash points to the hash of a previous block, which is used to provide a deadline for the transaction to be processed. Each submitted transaction must be processed within a limited window of blocks, e.g., the height

difference between the block that the transaction's `ReferenceBlockHash` points to and the block that the transaction appears in should be less than a predefined parameter. For example, assume that the window size for the transactions is 10 blocks and a transaction is pointing to a block at height 1000 in its `ReferenceBlockHash`. This means that the transaction can only be processed between blocks of height 1001 and 1010. Otherwise, the transaction is invalid.

Let h denote hash of the `SignedTransaction`, interpreted as an *unsigned integer*. To submit its transaction, the user agent determines the cluster of the Collector Nodes that are responsible for collecting its transaction. The collection determination is done as shown by Equation (2), where c corresponds to the number of clusters in the system. The index of the cluster, to which the transaction with hash h should be submitted to is

$$c_h = h \bmod c. \tag{2}$$

Here, `mod` is the modulus operation that determines the remainder of the integer-division of the left-side operand over the right-side operand.

In Flow, the assignment of Collector Nodes to clusters is easily computed in a deterministic fashion by executing Algorithm 3.1. Hence, a user agent can efficiently map its signed transaction to a cluster of the Collector Nodes using Algorithm 3.1 and Equation (2). The transaction submission is done by the user agent sending its signed transaction to the cluster with index c_h that is driven by Equation (2). By sending a transaction to a cluster, we mean sending to at least one of the Collector Nodes belonging to that cluster. However, for sake of fault tolerance in the presence of Byzantine Collector Nodes as well as (network) failures, a user agent may send the same transaction to several Collector Nodes of the assigned cluster. In Flow, we leave the number of Collector Nodes that a user agent needs to submit its transaction as a user-dependent decision. A user with a strong set of utilities may prefer to send all or only its important transactions to the entire cluster, while another user may prefer to submit it only to a single node of the assigned cluster. Upon receiving a transaction from a user agent, the Collector Node checks whether the transaction is submitted to the correct cluster according to Equation (2) as well as whether the transaction is well-formed. A transaction is well-formed if it contains all the fields as shown by Listing 1 as well as valid `ScriptSignature`(s) and `PayerSignature` by the registered accounts of Flow. If the transaction is well-formed as well as submitted to the right cluster, the Collector Node broadcasts the signed transaction to the entire cluster.

Listing 1: User agent broadcasts the transaction to multiple Collector Nodes of the assigned cluster.

```
message SignedTransaction {
  bytes Script;
  bytes PayerSignature;
  repeated Signature ScriptSignature;
  bytes ReferenceBlockHash;
}
```

3.4 Collection Formation

In Flow, the term *collection* refers to an ordered list of one or more **hashes** of signed transactions. Collector Nodes form collections by coming to consensus on the ordered list of transaction hashes. As discussed in Section 2.2, Flow uses HotStuff [11, 12] as the consensus protocol among the

Collection Node clusters, for them to form collections. However, any BFT consensus protocol with deterministic finality would apply to the architecture. In each round of consensus, the selected leader either resumes an uncompleted or time-outed proposal of the previous round or makes a new proposal. A proposal is either on appending a list of the signed transactions' hashes to the current collection or on closing the current collection and starting a new empty collection. For each phase of the consensus to proceed a minimum effective vote of $\frac{2}{3}$ on the leader's message is required. For a Collector Node to vote in favor of an append proposal, the following conditions must all be satisfied:

- The Collector Node has received all the signed transactions that their hashes represent are reflected in the append proposal.

- Each of the transaction hashes represented in the append proposal should be well-formed and signed by at least one of the Collector Nodes of the same cluster.

- Appending the transactions from the proposal to the collection should not result in duplicate transactions.

- There should not be common transactions between the current collection under construction and any other collection that the cluster of the Collector Node has already guaranteed.

Once a message has the minimum effective vote of $\frac{2}{3}$, the leader aggregates the signatures associated with the individual votes which advances the consensus process to the next phase. In this way, every individual Collector Node can verify the correctness of the consensus process in a fully decentralized manner.

Initially, a new empty collection is created as a shared replicated state in every Collector Node. Upon reaching a consensus over an append operation, the signed transactions' hashes are appended to the collection. Once the collection size reaches a protocol's predefined threshold, a non-Byzantine leader of the subsequent rounds propose closing the collection. Consensus over the closing a collection is done by following the HotStuff protocol. The structure of a Guaranteed Collection is presented in Listing 2, where `CollectionHash` is the hash of the closed collection. `ClusterIndex` is the id of the cluster (see Algorithm 3.1) that has generated the collection, reached a consensus over and closed it. `AggregatedCollectorSigs` holds the aggregated signatures of the Collector Nodes that have guaranteed the collection. For the `GuaranteedCollection` to be considered as valid, `AggregatedCollectorSigs` should contain at least $\frac{2}{3}$ effective votes of the Collector cluster. We call each of the Collection Nodes that participate in making a guaranteed collection, a **guarantor** of that collection. By singing a collection, a guarantor attests to the followings:

- All the transactions in the collection are well-formed.

- They will store the entire collection including the full script of all transactions.

In Flow, we consider a guaranteed collection to be an immutable data structure. The guaranteed collection is broadcasted by the guarantors to all Consensus Nodes for inclusion in a block.

Listing 2: Each Collector Node broadcasts the guaranteed collection reference to all the Consensus Nodes.

```
1 message GuaranteedCollection {
2     bytes   CollectionHash;
3     uint32  ClusterIndex;
4     Signature AggregatedCollectorSigs;
5 }
```

3.5   Liveness

We introduce Lemma 3.1 before presenting the safety theorem of the collection formation process, and directly utilize it as part of the proof for the safety of the collection formation process.

Lemma 3.1 *Given a guaranteed collection that is generated by a set of guarantors in a cluster with the following conditions:*

- ***Network Model:** partially synchronous network with message traverse time-bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t*

- ***Message Delivery:** a fault-tolerant message routing functionality that guarantees the message delivery with high probability*

- ***Stake Distribution:** all the Collector Nodes are equally staked*

- ***Honest Behavior:** the honest actors follow the protocols as specified, and maintain their availability and responsiveness in the system over time*

- ***Honest Lower-bound:** the guaranteed collection has at least one honest guarantor*

The guaranteed collection is always available and recoverable in the system.

Proof of Lemma 3.1: As detailed in Section 3.4, the guarantors of a guaranteed collection maintain the *original copy* of it, and only broadcast the *collection reference* that is presented as a *GuaranteedCollection* to the Consensus Nodes. In other words, a guaranteed collection is supposed to be replicated on all of its guarantors, i.e., the Collector Nodes that signed it. By contradiction, assume that the original content of a guaranteed collection is not recoverable. This would imply the occurrence of at least one of the following:

- None of the collection's guarantors are available or responsive.

- The message delivery to and from the collection's guarantors is compromised by an attack, e.g., Routing Attack.

- There is no time-bound on the message delivery to and from the collection's guarantors.

- There is no time-bound on the processing clock of the collection's guarantors.

As following the Honest Lower-bound assumption, there exists at least one honest Collector Node among the guarantors, having none of the collection's guarantors responsive or available contradicts the Honest Behavior assumption of the lemma stating the necessity on the honest

actors to maintain their availability and responsiveness. Having the message delivery concerning the guarantors under attack contradicts the Message Delivery assumption of this lemma on the guaranteed message delivery to and from the guarantors. Having no bound on the processing clock of guarantors or the message delivery to and from them contradicts the Network Model assumption of the lemma. Hence, we conclude that as long as there exists at least one honest guarantor for a guaranteed collection under the assumptions of Lemma 3.1, the collection remains available, accessible, and recoverable.

$\square$

Theorem 3.2 (Collection Text Availability)
Given a system with the following properties:

- **Network Model:** *a partially synchronous network with message traverse time bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t. Each node is assumed to have a proper defence and protection against network attacks (e.g. Denial-of-service attacks).*

- **Message Delivery:** *a fault-tolerant message routing functionality that guarantees the message delivery with high probability*

- **Stake Distribution:** *equally staked Collector Nodes*

- **Byzantine Fraction:** *more than $\frac{2}{3}$ of Collector Nodes' stakes are controlled by honest actors*

- **Honest Behavior:** *honest actors follow the protocols as specified, and maintain their availability and responsiveness in the system over time*

There exists a configuration of Flow where each guaranteed collection is available with a high probability.

Proof of Theorem 3.2:

In our previous white paper [5], we have shown that by having $n_c = 1040$, 1/3 of these nodes being Byzantine, and our typical cluster size of $[50, 80]$ nodes, the probability of having a Byzantine cluster is very unlikely.

Despite Flow's proactive architectural setup for maintaining the availability of the collections, Flow also introduces a reactive mechanism (i.e., Missing Collection Challenge[6]) to confirm and slash the Byzantine actors attributed to a missing collection [5].

Theorem 3.3 (Liveness of Collection Formation)
Given a system with the following properties:

- **Network Model:** *a partially synchronous network with message traverse time-bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t*

---

[6]The analysis of the Missing Collection Challenge in [5] (specifically Proof of Theorem 3 therein) is more extensive than the proof presented here. Here, we assume that an honest actor will diligently but unsuccessfully query all guarantors for the desired collection before raising a Missing Collection Challenge. The analysis in [5] also covers the case of a lazy or trolling requester, which might query a subset of the guarantors and raise a challenge if the queried subset does not respond.

- ***Message Delivery:*** *a fault-tolerant message routing functionality that guarantees the message delivery with high probability*

- ***Stake distribution:*** *equally staked Collectors Nodes*

- ***Byzantine fraction:*** *more than $\frac{2}{3}$ of Collector Nodes' stakes are controlled by honest actors*

- ***Honest Behavior:*** *the honest actors follow the protocols as specified, and maintain their availability and responsiveness in the system over time*

the collection formation always progresses.

Proof of Theorem 3.3: By contradiction, assume that there is an unknown point of time t_{stop} for which the collection formation process as described in this section stops working for all points of time $t > t_{stop}$. By collection formation being stopped, we mean that there is absolutely no single collection formed by any of the clusters of Collector Nodes once t_{stop} elapses. This implies at least one of the following situations applies to the system by elapsing t_{stop}, which results in none of the clusters being able to reach a consensus over closing their collection:

- Less than $\frac{2}{3}$ of the Collector Nodes of **every** cluster are honest.

- The message delivery to a fraction of at least $\frac{2}{3}$ of Collector Nodes of each cluster is compromised by an attack, e.g., Routing Attack.

- There is no time-bound on the message delivery to and from a subset of at least $\frac{2}{3}$ of the Collector Nodes in each cluster.

- There is no time-bound on the processing clock of at least $\frac{2}{3}$ of Collector Nodes of each cluster.

Following the Honest behavior and Stake distribution assumptions of the theorem, having less than $\frac{2}{3}$ honest Collector Nodes contradicts the Byzantine fraction assumption of the theorem. Likewise, compromising the message delivery to a fraction of at least $\frac{2}{3}$ of the Collector Nodes on each cluster contradicts the Message Delivery assumption of the theorem. Finally, having no bound on either the message delivery time to or the processing time of at least $\frac{2}{3}$ of Collector Nodes in each cluster contradicts the Network Model assumption of the theorem. Hence, we conclude that as long as the assumptions of the theorem hold, the collection formation progresses.

It is worth mentioning that, as discussed earlier, we envision it is rare for a Byzantine cluster to exist. Such a cluster may aim at compromising the liveness of the system by ignoring all the submitted transactions to that cluster. Nevertheless, upon detecting an attack on liveness by a user agent, the agent may simply try to switch the corresponding cluster of its transaction by changing some metadata of its transaction, e.g., changing the transaction's `ReferenceBlockHash` (see Listing 1). This minor change results in the transaction's cluster assignment changing and so, it can be routed to a (non-Byzantine) cluster (see Equation (2)) and hence processed towards inclusion in a collection.

4 Block Formation

4.1 Overview

Block formation is a continuous process executed by consensus nodes to form new blocks. The block formation process serves multiple purposes:

- Including the newly submitted guaranteed collections and reaching agreement over the order of them.

- Providing a measure of elapsed time by continuously publishing blocks and determining the length of an epoch.

- Publishing result seals for previous blocks of the chain. A block's result is ready to be sealed, once it is finalized by the Consensus Nodes, executed by the Execution Nodes, and the execution result is approved by the Verification Nodes (see [5] for more details).

- Publishing slashing challenges and respective adjudication results.

- Publishing protocol state updates, i.e., slashing, staking, and unstaking. Whenever a node's stake changes, the Consensus Nodes include this update in their next block.

- Providing a source of randomness (see Section 2.3 for more details).

In an unlikely event of no new guaranteed collection, consensus nodes continue block formation with an empty set of collections. In other terms, block formation process never gets blocked.

The block-formation process utilizes the underlying round-base consensus protocol we described in Section 2.2. In Flow, we distinguish between `ProtoBlock`s and (proper) `Block`s, which are formally defined in section 4.2 below. The BFT consensus algorithm generates and finalizes `ProtoBlock`s. The only difference between a `ProtoBlock` and a (proper) `Block` is that a `ProtoBlock` does not contain any source of randomness. The source of randomness in a `Block` is required for various processes in Flow, including deterministically generating pseudo-random numbers during transaction execution, assigning collectors to clusters, etc. Hence, `ProtoBlock`s cannot be processed by nodes other than consensus nodes due to the lack of source of randomness.

After a `ProtoBlock` is generated by the BFT consensus protocol, the Distributed Random Beacon (DRB) nodes sign the `ProtoBlock`. Their threshold signature over the `ProtoBlock` is the `Block`'s source of randomness. The `ProtoBlock` plus the DRB's signature together form the (proper) `Block`. While the Random Beacon is run by a set of consensus nodes, generating the randomness is *not* part of the consensus process. Instead, the source of randomness is added in a subsequent step to generate a proper `Block`.

4.2 Data Structures

ProtoBlocks

The BFT consensus algorithm works on the level of `ProtoBlock`. In each round, a consensus node is selected as primary, whose main task is to collect signatures for the last `ProtoBlock` and propose the next `ProtoBlock`. Once a `ProtoBlock` is proposed, the Consensus Nodes vote for its inclusion in the finalized chain. The structure of a well-formed `ProtoBlock` is specified in Listing 3. We provide definitions for the `ProtoBlock`'s individual fields below.

Listing 3: Flow's proto block structure

```
1  message ProtoBlock {
2    bytes   previousBlockHash;
3    uint64  height;
4    repeated GuaranteedCollection guaranteedCollections;
5    repeated BlockSeal blockSeals;
6    SlashingChallenges slashingChallenges;
7    ProtocolStateUpdates protocolStateUpdates;
8    }
```

- **previousBlockHash** points to the hash of the immediate predecessor of this block on the chain. By pointing to the previous block, blocks form a blockchain.
- **height** corresponds to the current index of the block in the chain. Formally, the block chain forms a tree with the blocks as vertices. The height of a block is the depth of the respective vertex in the tree.
- **guaranteedCollections** is an ordered list of new guaranteed collections that are introduced by this block. As explained in Section 3, a guaranteed collection includes the hash value of the transactions in the collection, but not the transactions themselves. The transaction texts are stored by the Collector Nodes, who guaranteed the collection, and provided by them upon request.
- **blockSeals** is a list of hash commitments to the verified execution results of the previous blocks. Sealing a block happens after the submission of a minimum effective vote of $\frac{2}{3}$ of the block's result approvals from the Verification Nodes to the Consensus Nodes, with no challenges are pending. Check out our previous paper [5] for more details on execution and verification.
- **slashingChallenges** is the list of new slashing challenges that have been submitted to the Consensus Nodes for adjudication. Any staked node can submit slashing challenges against another node in the Flow network. For example, Verification Nodes may submit slashing challenges against the computation result of the Execution Nodes.
- **protocolStateUpdates** contains a cryptograpic commitment to the updated protocol state of the system as a result of executing this block. In Flow, all the staked nodes follow recently finalized blocks to update their view of the system and progress with their roles. Moreover, publishing this information in the blocks also acts as a public bulletin board of the changes in the system state.
- **aggregatedConsensusSigs** contains the aggregated signatures of the Consensus Nodes that voted in favor of this block.

Blocks

The structure of a well-formed `Block` is presented by Listing 4. It contains commitments to all necessary information for progressing with transaction computation and evolving the protocol state in a deterministic manner. However, from a data perspective, a `Block` is *not* self-contained. For example, while cryptographic hashes of the execution state are included, the state data itself is not. This implies that anyone can verify the integrity of any subset of the execution state using the

```
1 message Block {
2    ProtoBlock protoBlock;
3    Signature sourceOfRandomness;
4 }
```

hashes in the `Block` (and merkle proofs which must be provided alongside the actual data).

Essentially, a `Block` is simply a wrapper around a `ProtoBlock`, which adds the `sourceOfRandomness` to the block. Formally, the source of randomness is a reconstructed threshold signature from the DRB Nodes (see Section 2.3 for details.)

4.3 Block formation process

Flow requires that all Consensus Nodes follow the same protocol when responding to events. We describe the block formation process steps from the perspective of a Consensus Node which we denote by `this`. A Consensus Node can hold two separate secret keys:

- sk_{this}^{stake} is used for staking.

- If the Consensus Node is a member of the DRB, it has a second secret key sk_{this}^{DRB} which is used for generating the node's threshold signature shares.

For the node `this` in the Flow network, its role, public staking key pk_{this}^{stake}, current stake amount, and the node's network IP address ip_addr_{this} are known to all other nodes. Furthermore, if the node is a member of the DRB committee, also its public key pk_{this}^{DRB} is publically known.

4.3.1 Block Proposal

Following Flow's underlying consensus protocol, Consensus Nodes are selected as primaries with a probability proportional to their stake using a probabilistic but verifiable protocol. The Consensus Node selected as the primary of the current round has to propose the next `ProtoBlock` pb. It includes the guaranteed collections, block seals, slashing challenges, protocol state updates, and a commitment to the protocol state after the updates. Subsequently, it broadcasts the proposed `ProtoBlock` pb to all other Consensus Nodes. As a security measure, the underlying consensus protocol has a timer to detect the primary's failure. If the primary does not generate a block within this time, the Consensus Nodes progress to the next round and primary.

As mentioned earlier, if there are no guaranteed collections available, the primary should produce a `ProtoBlock` pb with an empty list of guaranteed collections (i.e., $b.\texttt{guaranteedCollections} = \emptyset$), but still include the potential block seals, protocol state updates, challenges, adjudications, etc. It is worth noting that the Consensus Nodes only work with and decide upon the order of the collection hashes but not the full collection texts. Hence, they do not need to inspect the collections' transactions unless an execution result is being challenged [5].

4.3.2 Block Proposal Evaluation and Voting

After the `primary` broadcasts its `ProtoBlock`, the rest of Consensus Nodes evaluate the proposal and vote. Each Consensus Node follows the consensus steps for this round (see Section 2.2), if it approves the proposed `ProtoBlock`. Each Consensus Node runs Protocol 3 and only votes in favor of a `ProtoBlock` if all the listed conditions are satisfied. In Protocol 3, we denote the current Consensus Node as 'this'.

Protocol 3 Flow Consensus Protocol

Input

A proposed `ProtoBlock` pb

Procedure

Check these conditions:

1. pb's proposer is the selected `primary` of this round and pb is signed by this node.
2. pb extends the known blockchain and is a descendent of the genesis block (without missing blocks).
3. pb is safe to vote on according to the rules of the consensus protocol.
4. pb.`guaranteedCollections` contains a set of new guaranteed collections or is empty.
5. `this` has received all guaranteed collections in pb.`guaranteedCollections`.
6. All the guaranteed collections in pb.`guaranteedCollections` are authenticated (as defined in section 3.4).
7. `this` has received all block seals in pb.`blockSeals`.
8. All the block seals in pb.`blockSeals` are valid according to the conditions in [5].
9. `this` has received and verified all slashing challenges in pb.`slashingChallenges`.
10. Apply the protocol state updates listed in pb to the final protocol state of the parent block (referenced by pb.`previousBlockHash`). Verify that the resulting protocol state matches the commitment in the block pb.

If all of these conditions are satisfied, vote in favour of pb.

4.3.3 Finalizing a `ProtoBlock`

Consensus Nodes sign a `ProtoBlock` to indicate their approval. In the current architecture of Flow, we implement HotStuff, which is a BFT consensus algorithm with deterministic finality. However, without loss of generality, any other consensus algorithm that is BFT and has deterministic finality is suitable. In HotStuff, a `ProtoBlock` is finalized when there are three `ProtoBlock`s built on top of it. In order to build on top of a `ProtoBlock`, HotStuff requires more than $\frac{2}{3}$ effective votes from Consensus Nodes for it. The safety of the HotStuff protocol guarantees that never two or more competing blocks are finalized.

4.3.4 Source of randomness Attachment

In Flow, a reliable and verifiable randomness is essential for the system's Byzantine fault tolerance. The `sourceOfRandomness` field of a `Block` is used by Flow nodes to seed multiple pseudo-random-number generators. To provide a reliable source of randomness, Flow's DRB closely follows Dfinity's proposal [9]. Protocol 4 specifies the steps for generating the source of randomness for a `ProtoBlock`.

Participating Consensus Nodes follow a two-step procedure to compute `sourceOfRandomness` for a given block. First, they sign the hash of the `ProtoBlock` and share their signatures with the other DRB nodes. Second, a DRB node waits for the signature shares from other DRB members. Upon having more than t distinct threshold signature shares over the hash of the `ProtoBlock`, each DRB node can recover the full threshold signatures over the hash for the `ProtoBlock`. The threshold signature is the source of randomness of the block and its digest is used to seed the pseudo-random generators. Once the source of randomness is ready, a DRB node broadcasts the (proper) `Block` to the entire network. In Protocol 4, we denote the current Consensus Node as '`this`'.

Protocol 4 Flow Distributed Random Beacon steps

Input

A `ProtoBlock` pb

Procedure

1. **Step 1: Sign the proto block**

    (a) $\sigma_{\texttt{this}} \leftarrow \texttt{Sign}(\texttt{Hash}(pb), \texttt{sk}^{\texttt{DRB}}_{\texttt{this}})$

    (b) Broadcast $\sigma_{\texttt{this}}$ to all DRB nodes.

2. **Step 2: Wait for $(t+1)$-many distinct threshold signature shares $\sigma_{i_1}, \sigma_{i_2}, ..., \sigma_{i_{t+1}}$**

    (a) $\sigma \leftarrow \texttt{Recover}(P_{i_1}, P_{i_2}, ..., P_{i_{t+1}}, \ \sigma_{i_1}, \sigma_{i_2}, ..., \sigma_{i_{t+1}})$ (see Section 2.3)

    (b) create `Block` b with $b.\texttt{sourceOfRandomness} = \sigma$ and $b.\texttt{protoBlock} = pb$

    (c) broadcast b to the entire network

4.4 Protocol State Updates

For each block, one may view the protocol state as a key-value dictionary. A key in the protocol state corresponds to the public staking key $\texttt{pk}^{\texttt{stake}}$ of a node. The corresponding dictionary value stores all of the node's properties. Relevant properties for a staked node include the node's role, its staking amount, and its DRB key (if applicable). The way Flow maintains its protocol state is akin to the way Ethereum handles its state:

- The protocol state is directly maintained by the Consensus Nodes (which compare to miners in Ethereum).
- Each block contains a commitment to the resulting protocol state *after* all protocol state updates in the block have been applied.

Changes in the protocol state of a block propagate to the children of this block. Hence, the protocol state is a relative concept that is evaluated with respect to a base block. For example, having two consecutive blocks A and B where B is the immediate child of A, one must evaluate the correctness of the protocol state updates as a transition from block A to B. If there are forks in the unfinalized part of the chain, the protocol state might be different in the forks.

The protocol state of a node may change due to slashing, staking, and unstaking. Any staked node can submit a slashing challenge against another node. The slashing challenges are adjudicated by the Consensus Nodes, and the protocol state of a node may change accordingly. For each epoch,

there is a certain block height (i.e., protocol-determined parameter) before which all new staking requests for the next epoch must be submitted as transactions. We envision that in mature Flow this block height is set such that the staking period ends about one day (approximately 80 000 blocks) before the new epoch.

To stake, an actor submits a staking transaction which includes its public staking key. Once the staking transactions are included in a block and executed by the Execution Nodes, a notification is embedded into the corresponding Execution Receipt. When sealing the execution result, the Consensus Nodes will update the protocol state of the staking nodes accordingly.

For unstaking, a node submits a transaction signed by its staking key. Once an unstaking transaction is included in a block during an epoch, it discharges the associated node's protocol state as of the following epoch. The discharged stake of an unstaked node is effectively maintained on hold, i.e., it can be slashed but it is not returned to the unstaked node's account. The stake is returned to the unstaked node after a waiting period of at least one epoch. The reason for doing so is two-fold. First, detecting and adjudicating protocol violations might require some time. Hence, some delay is required to ensure that there is enough time to slash a misbehaving node before its stake is refunded. Second, to prevent a long-range attack wherein a node unstakes, and then retroactively misbehaves, e.g., a Consensus Node signing an alternative blockchain to fork the protocol.

4.5 Adjudicating Slashing Challenges

Consensus Nodes adjudicate slashing challenges and update the protocol state accordingly. On receiving a slashing challenge from a node (the 'challenger') against a challenged node, the Consensus Nodes include it into a block. This is done to embed an official record of the slashing challenges in the blockchain. In case the challenge contains a full proof, the Consensus Nodes adjudicate it right away. Otherwise, they wait for the challenged node to reply accordingly. If the challenged node does not reply within a timeout, it is slashed right away. Otherwise, if the challenged party responds with data required to adjudicate the challenge, the Consensus Nodes process the data and slash whomever (i.e., challenger or challenged node) that is at fault. In both cases, the Consensus Nodes publish the adjudication result in the next block together with the protocol state updates. The stakes of the nodes are affected as the result of the updates.

4.6 Correctness Proof

HotStuff [11, 12] is at the heart of Flow's Consensus Protocol 3. Flow only imposes some additional requirements on the validity of blocks, which the HotStuff protocol is agnostic to. Therefore, the safety of Flow's consensus protocol follows from HotStuff's safety. We formalize this insight in the following Corollary 4.0.1.

Corollary 4.0.1 (Safety) *Given a system with:*

- ***Network Model:*** *partially synchronous network with message traverse time-bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t*

- ***Message Delivery:*** *a fault-tolerant message routing functionality that guarantees the message delivery with probability close to 1.*

- **Byzantine Fraction:** *more than $\frac{2}{3}$ of Consensus Nodes' stakes are controlled by honest actors*

- **Honest Behavior:** *the honest actors follow the protocols as specified, and maintain their availability and responsiveness in the system over time*

- **Consensus Protocol:** *the Consensus Nodes follow HotStuff, which is a BFT consensus protocol with deterministic finality*

Under these prerequisites, Protocol 3 is safe against arbitrary behaviour of Byzantine nodes. Specifically, Byzantine actors cannot introduce errors by deliberately publishing or voting for faulty blocks. Furthermore, the consensus Protocol 3 has deterministic finality.

HotStuff [11, 12] is a general-purpose BFT consensus protocol which is agnostic to the payload contained in the block. In other words, the detailed structure of the block payload as well as the corresponding validity requirements do not affect liveness of HotStuff as long as an honest Primary has the ability to always propose a block. In Flow's Consensus Protocol 3, there are no hard requirements on the amount of content a Primary has to put in the block. An honest Primary will only include valid content, or propose an empty block. Therefore, liveness of Flow's Consensus Protocol 3 is also immediately follows from HotStuff, as stated in the following Corollary 4.0.2.

Corollary 4.0.2 (Liveness) *Given a system with:*

- **Network Model:** *partially synchronous network with message traverse time-bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t*

- **Message Delivery:** *a fault-tolerant message routing functionality that guarantees the message delivery with probability close to 1*

- **Byzantine Fraction:** *more than $\frac{2}{3}$ of Consensus Nodes' stakes are controlled by honest actors*

- **Honest Behavior:** *the honest actors follow the protocols as specified, and maintain their availability and responsiveness in the system over time*

- **Consensus Protocol:** *the Consensus Nodes only vote on* `ProtoBlocks` *if they are safe according to the rules of the HotStuff consensus protocol*

Under these prerequisites, Protocol 3 is live, i.e., the protocol will continue to finalize valid blocks.

5 Execution

Execution Nodes follow the new blocks as they are published by the Consensus Nodes. A conservative[7] Execution Node only processed finalized blocks to avoid wasting resources on unfinalized blocks, which might be abandoned later. While only Consensus Nodes actively participate in consensus, all other nodes still verify for themselves that a block is finalized according to the criteria of the BFT consensus protocol.

5.1 Overview

As illustrated by Figure 3, in Flow, the *execution* process is triggered when the Consensus Nodes broadcast evidence that the next block is finalized. It completes when the Execution Nodes broadcast their Execution Receipts. The Execution Nodes are responsible for receiving a finalized block, executing its transactions, and updating the execution state of the system. During this process each Execution Node takes the following steps:

- Retrieves each collection of transactions that appeared in the finalized block from the relevant clusters.

- Executes the transactions according to their canonical order in the finalized block and updates the execution state of the system.
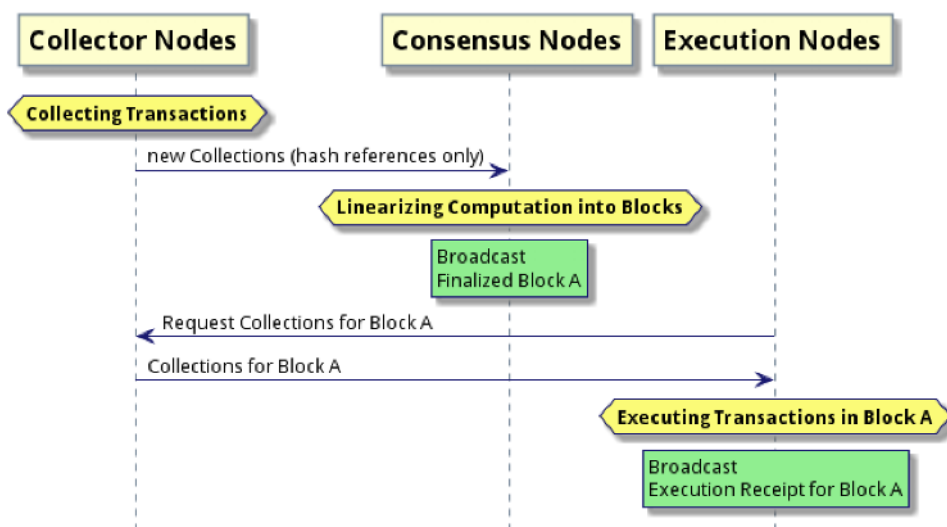

Figure 3: Lifecycle of a transaction from collection until the execution. The yellow hexagons indicate the start of the individual stages in the Flow pipeline, e.g., the block sealing (i.e., Transactions Collection). The arrows show the inter-role message exchange between the nodes. Green rectangles correspond to the broadcasting events, in which a message is disseminated to all of the staked nodes. For the sake of simplicity, we represent the normal flow of the transaction lifecycle and ignored the message exchanges related to adjudicating slashing challenges.

---

[7]The Flow protocol allows Execution Nodes to compute unfinalized blocks. This might allow the Execution Nodes to gain more revenue in an incentive model that specifically rewards speed of execution. However, such an optimistic Execution Node risks investing resources for computing blocks which are abandoned later. Furthermore, in contrast to finalized blocks, the BFT consensus algorithm does not guarantee the validity of unfinalized block proposals. Hence, an optimistic Execution Node must protect itself against processing invalid blocks.

- Builds an Execution Receipt for the block, i.e., a cryptographic commitment on the execution result of the block

- Broadcasts the Execution Receipt to Verification and Consensus Nodes.

We elaborate on each of these steps in the rest of this section. For the sake of simplicity, we describe the process from the perspective of a *single* Execution Node. In the live system, the Execution Nodes operate completely independently from each other and execute the protocol in parallel.

5.2 Collection Retrieval

A guaranteed collection, as listed in the block (compare Listing 2), only contains a hash of the set of transactions that it represents but not their full texts. As explained in Section 3, the Collector Nodes that sign a guaranteed collection are also responsible sending its full text to other nodes upon request.

Once an Execution Node confirms for itself that a block is finalized (see Section 4), it asks each cluster for their respective collection text so it can begin processing the transactions in order. If the responsible guarantors do not provide a guaranteed collection, a node can issues a Missing Collection Challenge against the guarantors. The challenge is submitted directly to the Consensus Nodes. Missing Collection Challenges are discussed in detail in the third Flow Technical Paper [5].

When an Execution Node retrieves the transactions for a guaranteed collection, it reconstructs the collection's hash in order to verify that the transaction data is consistent with the guaranteed collection. Provided verification succeeds, a collection is considered successfully recovered.

5.3 Block Execution

An Execution Node computes a finalized block b, when the following conditions are satisfied:

- The execution result of the previous block to b (the one referened by b.previousBlockHash) is available from either the Execution Node itself or from another Execution Node.

- All guaranteed collections in b.guaranteedCollections have been successfully recovered. Alternatively, the Execution Node has a Missing Collection Attestation, which allows it to skip a lost collection (details are discussed in detail in the third Flow Technical Paper [5]).

Once the Execution Node has computed block b, it broadcasts an Execution Receipt to Verification and Consensus Nodes. Listing 5 shows the Execution Receipt's message format. In addition to the Execution Node's signature executorSig, an Execution Receipt has two types of data structures: an Execution Result (field executionResult), and a Specialized Proof of Confidential Knowledge (SPoCK) (field Zs). We detail each of these in the following sections 5.3.1 and 5.3.2.

Listing 5: Structure of an Execution Receipt in Flow. It consists of two primary data structures: an `ExecutionResult` and one or more `SPoCK`(s).

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

5.3.1 Execution Result

An Execution Result for a block b represents a commitment from an Execution Node to the interim and final states of computing the block b. As shown by Listing 6, an Execution Result for a block contains the block hash (`blockHash`), hash reference to the previous Execution Result (`previousExecutionResultHash`), one or more chunks (`chunks`), and a commitment to the final execution state of the system after executing this block (`finalState`).

The `previousExecutionResultHash` acts as a commitment to the starting execution state for executing the block. It allows tracing an execution error back to the actor that originally introduced it. Figure 4 shows an example of this case. It is worth noting that this linked sequence of Execution Results could be called *"a chain"*, but we avoid the word in order to avoid confusion with the primary *"block chain"* maintained by the Consensus Nodes. Also, as shown by Listing 6 each Execution Result has a hash reference to the block that it is associated with. However, we skip the inclusion of the block references in Figure 4 for the sake of simplicity. Once a Verification Node attributes a computation fault to an Execution Node, it issues a Faulty Computation Challenge (FCC) against the faulty Execution Node. Verification of Execution Results and adjudicating the FCCs are covered in detail in our other white paper [5].
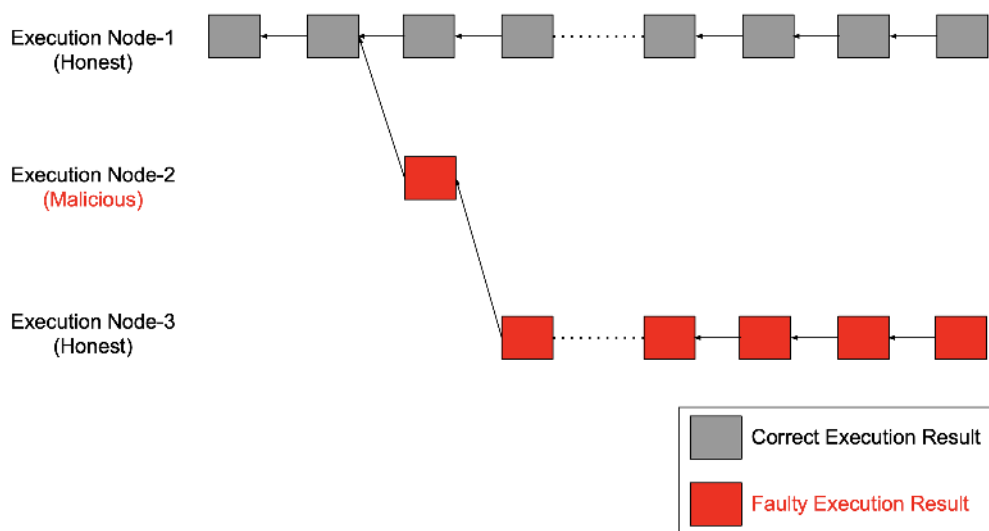

Figure 4: The `previousExecutionResultHash` field of the Execution Results enables identifying and slashing the malicious Execution Nodes that originally injected an error into the system, instead of punishing the honest ones that only propagate the error. In this figure, the gray rectangles represent the correct Execution Results, and the red ones represent a faulty Execution Result. The arrows correspond to `previousExecutionResultHash` references, i.e., the Execution Result on the right side of an arrow takes its initial execution state from the result of the left one. The faults in the Execution Results that are generated by the honest Execution Node-3 are attributable to the malicious Execution Node-2 that initially injected them into its own Execution Result. In other words, `previousExecutionResultHash` of the Execution Results provides a chain of attributability that enables the Verification Nodes to trace back the errors, and identify and slash the malicious Execution Node-2 that originally introduced the error instead of punishing honest Execution Node-3 that only propagated the error.

Execution Results consist of one or more chunks. Chunking is the process for dividing a block's computation so such that the resulting chunks can be verified in a distributed and parallelized manner by many Verification Nodes. The execution of the transactions of a block is broken into a

set of chucks. The chunking is done based on the computation consumption of the transactions, and in a way that the overall computation consumption of the transactions of chunks be similar to each other and do not exceed the system-wide threshold of Γ_{chunk}. Homogeneity of the chunks computation consumption is a security countermeasure against the injected faults in the computation-heavy chunks. In other words, if the computation consumption of the chunks is heterogeneous, the Verifier Nodes of the computationally heavier chunks may not be able to finish the verification before the sealing of the actual block associated with those chunks [5]. This enables the malicious Execution Nodes to attack the integrity of the system by injecting a computational fault into the computationally heavy chunks, which are likely to be left out of verification.

The structure of a chunk is represented by Listing 6, where `startStateCommitment` denotes a hash commitment to the execution state of the system *before* the execution of the chunk. `startingTransactionCC` and `startingTransactionIndex` are the computation consumption and the index of the chunk's first transaction. `computationConsumption` denotes the overall computation consumption of the chunk. The chunking convention is: `computationConsumption` $\leq \Gamma_{\text{chunk}}$.

Listing 6: Structure of an Execution Result and Chunk in Flow

```
1 message ExecutionResult {
2     bytes blockHash;
3     bytes previousExecutionResultHash;
4     repeated Chunk chunks;
5     StateCommitment  finalState;
6 }
7 message Chunk {
8     StateCommitment  startStateCommitment;
9     float   startingTransactionCC;
10    uint32  startingTransactionIndex;
11    float   computationConsumption;
12 }
```

5.3.2 Specialized Proof of Confidential Knowledge (SPoCK)

The SPoCK is Flow's countermeasure against Execution Nodes copying Execution Results from each other instead of computing the block on their own. Also, SPoCKs are used to prevent Verification Nodes from blindly approving execution results without doing the verification work. SPoCKs are detailed in [5], but it is sufficient for the purposes of this paper to understand that SPoCKs are cryptographic commitments, which are generated as part of the verification process to indicate that the Verification Node has done its job. Essentially, a SPoCK is a commitment to the execution trace for a single chunk. The field `ExecutionRecipt.Zs` is a list of SPoCKs, where `ExecutionRecipt.Zs[i]` is the SPoCK for the ith chunk. It is worth emphasizing that `ExecutionRecipt.Zs[i]` can be generated by only executing the ith chunk.

5.3.3 Execution Algorithm

Algorithm 5.1 represents the `BlockExecution` algorithm that an Execution Node invokes individually for executing a block. The inputs to this algorithm are a finalized block b, which is ready for the execution, the hash of the previous Execution Receipt $h_{\text{er}_{prev}}$, and the execution state of the system before the execution of the block b, i.e., Λ. The output of the algorithm is an Execution

Receipt for block b, which is denoted by $\mathtt{er}_b$. The algorithm is invoked as follows:

$$\mathtt{er}_b = \mathtt{BlockExecution}(b, h_{\mathtt{er}_{prev}}, \Lambda)$$

The Execution Node processes the transactions referenced by block b in their canonical order. As shown by Figure 5, for the proto block associated with block b, the canonical order of the transactions is achieved by first sorting its guaranteed collections in the same order as they appear in the proto block, and then for each collection, sorting its transactions in the same order as the collection references. The first transaction of the first collection, and the last transaction of the last collection are then the first and last transactions of block b based following the canonical ordering, respectively. To resolve the canonical order of the transactions associated with block b, the Execution Node invokes the function $\mathtt{canonical(b)}$. The execution of each transaction t_i is done by invoking the $\mathtt{execute}$ function, where t_i is the ith transaction of the block b following its canonical order. On receiving the current execution state of the system Λ and the transaction text t_i, the $\mathtt{execute}$ function returns the resulting execution state, the computation consumption of t_i,
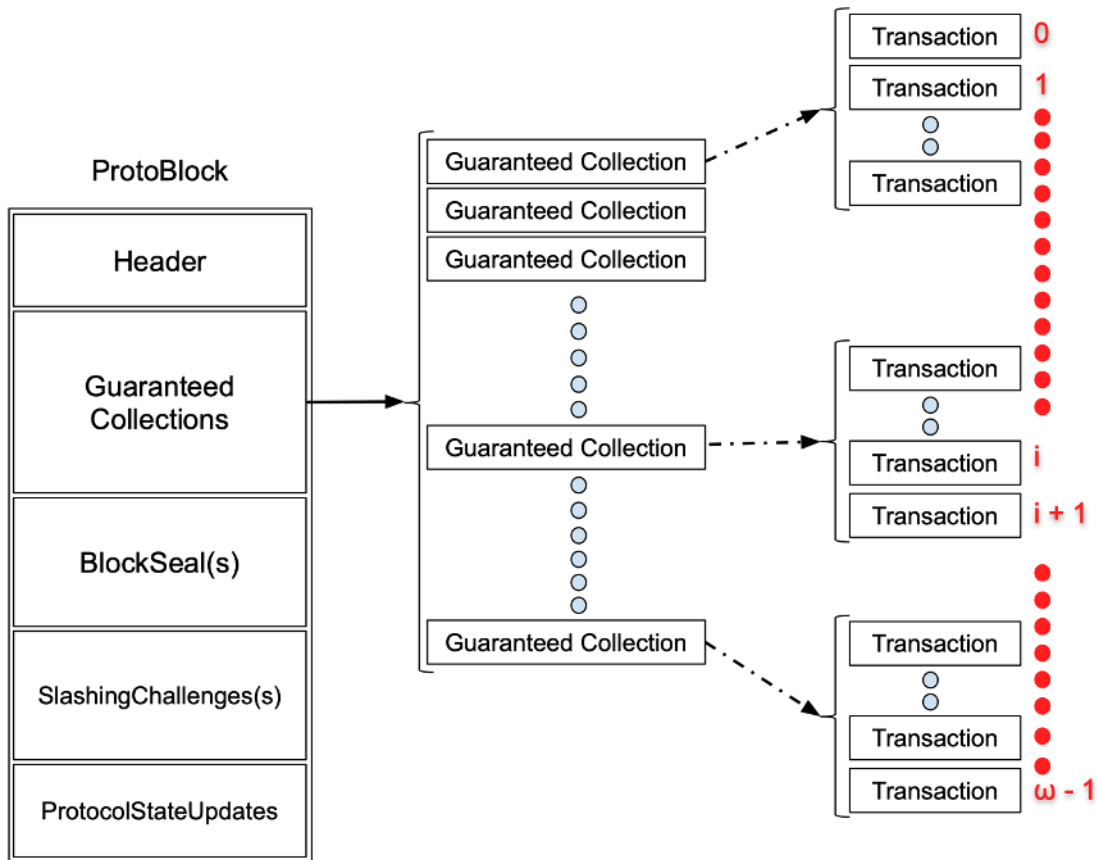

Figure 5: The canonical order of the transactions in a proto block with ω transactions is shown in red. The transactions with the canonical orders of 1 and ω are the first and last transactions in the sequential execution order of the proto block, respectively. The tick arrow corresponds to the inclusion relationship, i.e., a proto block includes several guaranteed collections. The dashed arrows correspond to the referential relationship, i.e., each guaranteed collection holds a reference to a collection of several transactions.

i.e., τ, and an SPoCK for the execution of t_i, i.e., ζ (Algorithm 5.1, Line 9). `BlockExecution` keeps track of the computation consumption of the current chunk by aggregating the overall computation consumption of the executed transactions since the start of the current chunk (Algorithm 5.1, Line 22).

For each transaction t_i, the Execution Node checks whether the current chunk's total computation consumption would exceed the threshold Γ_{chunk} when transaction t_i is added. If the threshold is not met, the transaction is considered as part of the current chunk, and the SPoCK of the current chunk (i.e., $\widetilde{\zeta}$) is updated by the SPoCK trace of t_i's execution (Algorithm 5.1, Line 23).

Otherwise, if the threshold Γ_{chunk} is reached for the inclusion of the transaction t_i in the current chunk, the Execution Node closes the current chunk without including t_i. Accordingly, the transaction t_i is taken as the first transaction of the next chunk. To exclude transactions without the need for reverting, the `BlockExecution` algorithm keeps track of the execution state by $\widetilde{\Lambda}$. Closing the current chunk is done by generating a commitment for the starting state of the chunk, which is tracked by Λ_{start}, and casting the chunk attributes into a `Chunk` message structure.

The commitment for the start state is generated by Flow's Authenticated State Proof system, which provides three deterministic polynomial-time functions:

- `StateProofGen`
- `ValueProofGen`
- `ValueProofVrfy`

`StateProofGen`(Λ_{start}) returns a verifiable hash commitment `startStateCommitment` to a given state snapshot Λ_{start}. Formally, Flow's execution state Λ is composed of key-value pairs. The key r can be thought of as the memory address and the value v the memory content. As only a small fraction of state values (i.e., key-value pairs) are touched in a chunk, it would be inefficient to transmit the full state to the Verification Node. Instead, Execution Nodes transmit only the key-value pairs that are needed for checking their computation. In addition, an Execution Node also provides a proof to allow the Verification Node to check that the received key-value pairs are consistent with the state commitment in the chunk. The Execution Node generates these proofs by running $\gamma_x := $ `ValueProofGen`$(\Lambda_{\text{start}}, r_x)$. Here, r_x denotes the key for an element of the state and v_x the corresponding value. The Execution node sends the (r_x, v_x, γ_x) to the Verification Node. The recipient can then verify that the values are indeed from the desired state by executing `ValueProofVrfy`$(r_x, v_x, \gamma_x, $ `startStateCommitment`$)$. We leave the details of the Authenticated State Proof scheme as well as its construction to our future publications. Once the commitment to the starting state of the current chunk is ready, `BlockExecution` creates a new `Chunk` message (see Listing 6). The generated chunk is appended to the list $\mathcal{C}$, which holds the chunks of the block under execution. The SPoCK $\widetilde{\zeta}$ for the generated chunk is stored in the list `Zs`, which holds the SPoCKs for the chunks of the block. Once the current chunk is closed, the algorithm moves to the next chunk by re-initializing the variables that keep the attributes of the current chunk (Algorithm 5.1, Lines 12-21). τ_0 keeps track of the computation consumption of the first transaction of the current chunk.

Once all transactions of the block b are executed in their canonical order and the corresponding chunks were created, `BlockExecution` creates an Execution Receipt (er_b) out of the hash of the block b (h_b), the hash of the previous Execution Receipt ($h_{\text{er}_{prev}}$), the list of all the chunks of block b ($\mathcal{C}$), the execution state of the system *after* executing block b (Λ), and the respective SPoCKs (`Zs`). The execution of block b ends by each Execution Node individually broadcasting er_b to the entire system.

Algorithm 5.1: BlockExecution

Input:

    b: finalized block

    $h_{\mathtt{er}_{prev}}$: hash of the previous Execution Receipt

    Λ: execution state of system prior to execution of b

Output:

    $\mathtt{er}_b$: Execution Receipt of block b

    // initializing the SPoCK list of the Execution Receipt

1 $\mathtt{Zs} := [\,]$;

    // initializing the list of chunks for block b

2 $\mathcal{C} := [\,]$;

    // initializing start state of the current chunk

3 $\Lambda_{\mathrm{start}} := \Lambda$;

    // initializing starting transaction index of current chunk

4 $\mathtt{startTransactionIndex} := 0$;

    // initializing computation consumption of the current chunk

5 $c := 0$;

    // initializing the SPoCK of the current chunk

6 $\widetilde{\zeta} := \emptyset$;

7 **for** $t_i \in \mathtt{canonical}(b)$ **do**

        // caching the state prior to execution of t_i for chunk bookkeeping

8      $\widetilde{\Lambda} := \Lambda$;

9      $\Lambda, \tau, \zeta := \mathtt{execute}(\Lambda, t_i)$;

10   **if** $i = 0$ **then**

11 | $\tau_0 := \tau$

12   **if** $c + \tau > \Gamma_{\mathrm{chunk}}$ **then**

          // closing the current chunk

13        $\mathtt{startStateCommitment} := \mathtt{StateProofGen}(\Lambda_{\mathrm{start}})$;

14        $chunk := \mathbf{new}\ \mathtt{Chunk}(\mathtt{startStateCommitment}, \tau_0, \mathtt{startTransactionIndex}, c)$;

15        $\mathcal{C}.\mathtt{append}(chunk)$;

16        $\mathtt{Zs}.\mathtt{append}(\widetilde{\zeta})$;

          // initializing the variables for the next chunk

17        $\Lambda_{\mathrm{start}} := \widetilde{\Lambda}$;

18        $\mathtt{startTransactionIndex} := i$;

19        $\tau_0 := \tau$;

20        $\widetilde{\zeta} := \emptyset$;

21        $c := 0$;

22     $c := c + \tau$;

        // Updating the execution trace of the current chunk

23     $\widetilde{\zeta} := \mathtt{TraceUpdate}(\widetilde{\zeta}, \zeta)$;

    // closing the last chunk

24 $\mathtt{startStateCommitment} := \mathtt{StateProofGen}(\Lambda_{\mathrm{start}})$;

25 $chunk := \mathbf{new}\ \mathtt{Chunk}(\mathtt{startStateCommitment}, \tau_0, \mathtt{startTransactionIndex}, c)$;

26 $\mathcal{C}.\mathtt{append}(chunk)$;

27 $\mathtt{Zs}.\mathtt{append}(\widetilde{\zeta})$;

28 $\mathtt{er}_b := \mathbf{new}\ \mathtt{ExecutionReceipt}(h_b, h_{\mathtt{er}_{prev}}, \mathcal{C}, \Lambda, \mathtt{Zs})$

5.4 Correctness Proof

For completeness, we include the following safety Theorem 5.1. For further details and the formal proof, the reader is referred to [5].

Theorem 5.1 (Safety) *Given a system with:*

- ***Network Model:*** *a partially synchronous network with message traverse time-bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t*

- ***Message Delivery:*** *a fault-tolerant message routing functionality that guarantees the message delivery with high probability*

- ***Honest Behavior:*** *honest actors which follow the protocols as specified and maintain their availability and responsiveness in the system over time*

- ***Byzantine Fraction:*** *at least one honest Execution Node*

Even if all but one Execution Nodes are Byzantine and publishing the same faulty result, the faulty result is not sealed with a probability greater than $1 - \epsilon$ with $\epsilon \ll 1$. Depending on its system parameters, Flow can be tuned such that $\epsilon \leq 10^{-12}$. The system parameters are: the number of chunks in a block and the fraction of chunks of a block that each Verifier Node checks.

Assuming that there is at least one honest Execution Node in the system, a valid execution receipt will be published for each block. As HotStuff guarantees liveness of block formation, liveness of execution receipt generation is also guaranteed. We formalize this statement in the following Corollary 5.1.1.

Corollary 5.1.1 (Liveness) *Given a system with:*

- ***Network Model:*** *a partially synchronous network with message traverse time-bound by Δ_t and the relative processing clock of the nodes bound by ϕ_t*

- ***Message Delivery:*** *a fault-tolerant message routing functionality that guarantees the message delivery with high probability*

- ***Honest Behavior:*** *honest actors follow the protocols as specifiedand maintain their availability and responsiveness in the system over time*

- ***Byzantine Fraction:*** *at least one honest Execution Node*

The liveness block execution is guaranteed as the honest Execution Node(s) follow the specified algorithms and eventually, generate an Execution Receipt for the input block.

6 Mitigating Attack Vectors

6.1 Byzantine Clusters

In mature Flow we envision clusters will be comprised of 20-80 randomly selected Collector nodes. While the probability for sampling a cluster with too many Byzantine nodes is small, it is unfortunately not negligible. A Byzantine cluster may attack the integrity of the system by injecting faulty transactions in their guaranteed collections. It also may attack the liveness of the system by withholding the text of a guaranteed collection, to prevent the Execution Nodes from computing a block. The former attack on the integrity of the system is mitigated following the detectability and attributability design principle of Flow, i.e., the guarantors of a collection containing malformed transactions are slashed. The latter attack on the availability of collections is mitigated by introducing the Missing Collection Challenges (see [5] for details). This challenge is a slashing request against the Collector Nodes that originally guaranteed the availability of the missing collection. The challenge is directly submitted to the Consensus Nodes for adjudication.

6.2 Faulty Computation Result

A malicious Execution Node may publish faulty *Execution Receipt*s. However, we formally prove in [5] that faulty execution results are detected and challenged by the Verification Nodes with overwhelming probability. Upon detection of such faulty execution results, a Verification Node submits a slashing challenge against the faulty Execution Node to the Consensus Nodes (see [5] for more details).

Acknowledgments

References

[1] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[2] BBC News. Cryptokitties craze slows down transactions on ethereum. 2017. https://www.bbc.com/news/technology-42237162.

[3] Richard Dennis and Jules Pagna Diss. *An Analysis into the Scalability of Bitcoin and Ethereum*, pages 619–627. 01 2019.

[4] Shirley Dieter Hentschel, Alexander and Layne Lafrance. Flow: Separating Consensus and Compute. 2019.

[5] Alexander Hentschel, Dieter Shirley, Layne Lafrance, and Maor Zamski. Flow: Separating Consensus and Compute – Execution Verification. *arXiv preprint arXiv:1909.05832*, 2019.

[6] 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.

[7] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[8] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoins peer-to-peer network. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 129–144, 2015.

[9] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[10] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Advances in Cryptology-ASIACRYPT 2001*, pages 514–532, 2001.

[11] 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.

[12] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. *arXiv preprint 1803.05069v6*, 2019. paper version 6.

[13] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[14] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

[15] Ronald A Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd Ltd, London, 1943.