OsirisBFT: Say No to Task Replication for Scalable Byzantine Fault Tolerant Analytics

Kasra Jamshidi  
School of Computing Science  
Simon Fraser University  
British Columbia, Canada  
kJamshid@cs.sfu.ca

Keval Vora  
School of Computing Science  
Simon Fraser University  
British Columbia, Canada  
keval@cs.sfu.ca

Abstract
We present a verification-based Byzantine Fault Tolerant processing system, called OsirisBFT, for distributed task-parallel applications. OsirisBFT treats computation tasks differently from state update tasks, allowing the application to scale independently from number of expected failures. OsirisBFT captures application-specific verification semantics via generic verification operators and employs lightweight verification strategies with little coordination during graceful execution. Evaluation across multiple applications and workloads shows that OsirisBFT delivers high processing throughput and scalability compared to replicated processing. Importantly, the scalable nature of OsirisBFT enables it to reduce the performance gap compared to baseline with no fault tolerance by simply scaling out.

CCS Concepts: • Computer systems organization → Dependable and fault-tolerant systems and networks: Distributed architectures; • Computing methodologies → Distributed computing methodologies.

Keywords: Distributed Computing, Byzantine Fault Tolerance, Data Processing Systems, Resilient Systems.

1 Introduction
This paper presents OsirisBFT, a verification-based Byzantine fault tolerant processing architecture for distributed task-parallel applications.

Task-parallel applications decompose the workload into independent tasks performed concurrently by different processes. Although task-parallel applications are common in various settings, Byzantine failures where faulty machines behave arbitrarily are seldom considered despite occurring frequently in practice [47, 55, 60]. For instance, task-parallel applications in settings like cybersecurity [42], business intelligence [71], and fraud detection [73] are open to threats where adversaries are incentivized to cause failures in order to gain access to user data, create outages, or commit unchecked fraud. Similarly, accidental Byzantine failures are also pernicious. Physical failures like memory corruption can create subtle flaws in the output of a computation even despite the presence of Error-Correcting Codes [53, 55, 67], with significant humanitarian [56] and legal [22] consequences.

Use Case: Anomaly Detection. Consider the problem of detecting anomalies in fast-changing networks. The application maintains an up-to-date version of the network graph using a continuous stream of link updates (insertion/deletion of network links), and performs pattern matching on the updated portion of the network to identify anomalous substructures [26]. As shown in Figure 1, the updates are applied to a multiversioned data store and multiple tasks perform pattern matching in parallel using the appropriate versions of the network. The pattern matching computation (detectAnomaly() in Figure 1) is orders of magnitude more expensive than performing link updates in the data store (updateNetwork() in Figure 1).

Here, Byzantine failures can affect the network graph in the data store as well as the pattern matching computation. In the first situation, faulty workers can apply incorrect/malicious link updates resulting in inconsistent views of the network graph, hence generating unreliable results computed from inconsistent data. To safeguard against such failures, various Byzantine fault tolerant protocols for managing state have been developed, for example Kauri [59], Basil [70] and others [1, 5, 11, 19, 29, 40, 49, 50, 80]. These protocols target applications composed of read/write transactions where ordering requests via BFT consensus is the major bottleneck.

In the second situation, even if the data store is maintained correctly, faulty workers performing pattern matching on correct data can result in incorrect output. For this, solutions like Medusa [28] and others [27, 57, 63, 69] enable BFT for applications dominated by computation instead of consensus. Like these works, our paper targets computation scalability.
At the heart of these BFT solutions are replicated state machine protocols (RSM) that replicate both application state and task execution [65]. Specifically, workers are divided into independent subsets of replicas that all maintain the same application state and execute the same tasks, such that different subsets maintain distinct partitions of application state and execute different tasks in parallel. In such an approach, safety and liveness are guaranteed if all subsets contain $O(f)$ workers and at most $f$ workers in each subset are faulty, as a majority of replicas will compute the correct result.

**Limits on Scalability.** Replicating application tasks in such RSM-based systems significantly limits scalability and processing throughput. Specifically, a cluster of $n$ workers can execute at most $\left\lfloor n/(2f + 1) \right\rfloor$ tasks in parallel using RSM. This means even with minimum fault tolerance $f = 1$, processing with RSM requires $3 \times$ the computation resources as a regular execution. Figure 2a shows the number of tasks that can be executed in parallel with RSM as a function of cluster size, and we corroborate this analysis by measuring the processing throughput in terms of number of result records generated per second for Anomaly Detection in Figure 2b. As seen, RSM-based processing on 32 nodes with $f = 1$ achieves similar throughput to only 8 nodes without fault tolerance.

**Observation.** The computation in task-parallel applications often involves multiple steps that are performed iteratively (e.g., matching the pattern step-by-step, computing until error converges, etc.). Hence, computations are often orders of magnitude more expensive than state updates since the latter only involve agreement on the ordering of updates and modifying the underlying data structures.

Although computation in task-parallel applications is time-consuming, verifying results is often much less expensive. This is because verification simply involves checking whether the results satisfy application semantics (e.g., whether the reported anomalies indeed match the pattern) which is much simpler than computing the solution itself. Hence, with verification being much faster than the original computation, BFT executions can be guaranteed without replicating the computation by judiciously verifying results.

---

**Figure 1.** Anomaly Detection. Update tasks modify the network graph in data store and computation tasks perform pattern matching on the modified graph.

**Figure 2.** Scaling of RSM-based processing for Anomaly Detection (i.e., with detectAnomaly() replicated).

**Our Approach.** In this paper, we separate state management from expensive computation and explore the possibility of BFT without replicating expensive computation. We develop OsirisBFT, a distributed BFT architecture for task-parallel applications backed by two components: a BFT data store for managing global state, and verification-based processing for application computation. Prior solutions [3, 14, 49, 80] already provide efficient BFT state management as discussed above. We incorporate an existing RSM-based BFT design [3] for our data store, and primarily focus on developing an efficient verification-based processing architecture.

**OsirisBFT.** OsirisBFT decouples task computation from fault tolerance by offloading the responsibility of detecting faults to a special subset of workers called verifiers. Regular workers execute computation tasks, and their results are analyzed by verifiers to protect against failures. Hence, workers executing computation tasks need not be replicated to ensure safety, enabling scalable task-parallel execution. Furthermore, verifiers check the generated results independently, and only perform consensus to linearize input tasks. Hence, OsirisBFT can execute $n - O(f)$ parallel tasks in a cluster with $n$ workers, as opposed to $n/O(f)$ in RSM.

While such verification-based BFT processing seems promising, realizing it in practice poses several challenges.

The first challenge is how to distinguish Byzantine executions from graceful executions that are not impacted by Byzantine failures? Byzantine failures can impact the execution and violate the application semantics in various ways (e.g., partially executing tasks, repeatedly executing the same task, simply outputting results that appear valid but do not satisfy the task requirements, etc.). Developing custom verification protocols to identify these different behaviors can easily become intractable, especially since several of these issues require understanding the application semantics to distinguish a Byzantine behavior from a correct one.

To address this, we develop an output failure model that captures how Byzantine failures impact the application results. Our model groups all possible application failures into three classes of output failures. We then formalize properties required to detect each class of output failures, and develop generic verification operators that allow our processing architecture to capture the required application semantics so that verifiers can safeguard against all classes of output failures.

---

$^{1} n/(2f + 1)$ using non-equivocation from modern RDMA networks [3] or trusted hardware [52]; otherwise the bound degrades to $n/(3f + 1)$.
The second challenge is how to perform verification robustly and efficiently? While verification operators capture faults that are observable from application results, verifiers themselves can be faulty, which can in turn cause complex failures even if workers correctly execute their tasks.

For verification that is both efficient and resilient, we develop robust and lightweight protocols. Verifiers certify the application results using the verification operators, with zero coordination among the verifiers during graceful executions. To achieve robustness in our verification pipeline, we rely on redundancy in communication between verifiers and other actors in OsirisBFT. Our careful use of cryptography, timeouts, and limited use of heavy communication primitives like non-equivocating multicast capture complex failure cases while retaining efficiency when processes are well-behaved.

The final challenge is how to maintain resource utilization and processing throughput as processing workload varies over time? As processing workload changes over time, tasks demanding high computation can keep workers busy even though the verification workload remains low. On the other hand, failed workers leaving the system can result in throughput drops that can persist in the remaining execution. We design a dynamic role-switching strategy to improve resource utilization and processing throughput across different processing conditions.

Results. To the best of our knowledge, this paper provides the first treatment of enabling Byzantine fault tolerance for task-parallel applications without replicating application computation. OsirisBFT is backed by safety and liveness proofs to ensure correctness under all circumstances and progress even in presence of Byzantine failures. We evaluated OsirisBFT with three distributed task-parallel applications and across different processing workloads. Our results show that OsirisBFT delivers high processing throughput and better scalability compared to replicated processing, and it scales comparably to a baseline without any fault tolerance. Importantly, OsirisBFT overcomes the performance gap from ensuring fault tolerance by simply scaling out.

2 Overview of OsirisBFT

The system is modeled as a pipeline with three steps: (i) input processes IP generate or ingest tasks and distribute them downstream; (ii) worker processes WP execute the tasks and output a sequence of records; and, (iii) output processes OP receive the results. IP and OP can overlap. Tasks can involve state updates (e.g., updateNetwork() in Figure 1), computation (e.g., detectAnomaly() in Figure 1), or both.

Figure 3 shows the verifiable processing architecture. WP is divided into two sub-clusters: the execution cluster EP and the verifier clusters VP. The execution processes (or simply, executors) execute computation tasks and output records, whereas the verifier processes (or simply, verifiers) deal with verification of the generated records. A computation task is executed on each input exactly once by an executor in EP (i.e., no task replication). VP is partitioned further into k independent Byzantine fault tolerant sub-clusters VPi...VPk−1 with each |VPi| ≥ 2f + 1 (0 ≤ i < k). One of the verifier sub-clusters is arbitrarily chosen to be responsible for performing consensus to linearize tasks and coordinating the remaining processes throughout the entire execution; we refer to this sub-cluster as the coordinator VPco.

State Management. The state management layer resembles the learner architecture [39]. For simplicity and maximal use of hardware resources, the application state is colocated with WP. As we discuss later, we make no assumptions about failures in EP. To safely perform concurrent state updates, the coordinator sub-cluster VPco linearizes tasks to enforce a global order on state updates, and keeps the rest of WP appraised so correct processes can maintain fresh, globally consistent copies of their state. This design avoids inflating the cost of queries with read requests to a disaggregated storage system or cross-shard transactions in a sharded solution by ensuring all processes maintain a local copy of the state, since analytics queries frequently perform reads.

Verifiable Processing. OsirisBFT enables scalability by placing all responsibility for Byzantine fault tolerance on VP, freeing EP to execute tasks without overheads. Tasks flow from IP to the coordinator VPco, VPco linearizes the tasks and broadcasts state updates to WP while distributing computation tasks among EP. State updates mutate local application state, and computation tasks operate on the local state to produce output records. While every state update is sent to all of WP, each computation task is assigned to a single executor at a time and reassigned only if a failure is suspected. Then, the results of computation tasks flow from EP to VP to OP. Each output record is sent to 2f + 1 verifiers in a Byzantine fault tolerant sub-cluster VPi for verification to ensure output processes only observe correct records.

Since only verifiers interact with the downstream and upstream processes, correct processes in IP and OP never observe failures in EP, even though computation tasks are never replicated.

Computation-Communication Tradeoff. Table 1 shows the computation redundancy, the communication redundancy, the fault tolerance, and the computation scalability provided by OsirisBFT, compared with the RSM-based replicated computation strategy (RCP) where WP is divided into
sub-clusters $W_{P_{i}}$ of $2f + 1$ processes each, and computation is replicated in all processes in a sub-cluster. OsirisBFT optimizes for application computations. It favors replicating communication rather than computation when possible, leveraging ample bandwidth in high performance networks to maximize utilization of cluster resources. Each output record is replicated over the network to $2f + 1$ verifiers in $V_{P_{i}}$, and in exchange, computation tasks are not replicated in graceful executions.

Hence, $O(f)$ processes verify records while $|WP| - O(f)$ processes execute tasks. The number of verifier sub-clusters can be kept small relative to $|WP|$, hence achieving higher performance than RCP by not replicating the expensive application computation, and only replicating the lightweight verification. Moreover, OsirisBFT tolerates faults more freely, since no executor is assumed correct. Each $V_{P_{i}}$ tolerates $f$ failures (similar to $W_{P_{i}}$ in RCP); in addition, OsirisBFT tolerates complete failure of $EP$. Hence, executors, and the application, can scale independently of $f$.

### 3 System Model

**Service Guarantees.** We adopt the Byzantine failure model, where processes can behave arbitrarily, including crashes, adversarial failures, and coordination between malicious processes. We define safety and liveness to limit the impact of Byzantine faults in $WP$. For all $i$, if at most $f$ processes in $V_{P_{i}}$ fail, and $V_{P_{i}}$ contains $2f + 1$ processes that can verify output records from other workers, OsirisBFT is linearizable (safety): all correct $OP$ observe records corresponding to a legal sequential execution of correct tasks submitted by $IP$. Furthermore, all correct $OP$ eventually observe results for every task submitted by $IP$ (liveness). Note that safety is not compromised even if all processes in $EP$ are faulty. However, the system is also bound by assumptions made by its state management layer. As mentioned above, we assume the state is managed by the Byzantine fault tolerant $VP$ processes, and $EP$ learn of state updates from $VP$. If state must be safely stored on $EP$ using a different approach then additional assumptions about failures in $EP$ may be necessary. We make no assumptions about the number of failures in $IP$ or $OP$.

We assume that adversaries have finite resources proportionate to correct processes, and cannot overwhelm correct processes with network traffic or break cryptographic primitives like digital signatures. Hence by authenticating all communication, correct processes cannot be impersonated.

Safety can be guaranteed if the system is asynchronous, and we make the standard assumptions from previous work [19, 59, 70, 80] regarding partial synchrony for liveness: there is some known $\Delta$ and unknown global synchronization time (GST) such that after GST, all messages between correct processes arrive with maximum latency $\Delta$ [33].

**Communication Primitives.** To achieve fault tolerance with $2f + 1$ processes in a sub-cluster instead of the well-known lower bound of $3f + 1$ processes [16], our techniques rely on a multicast primitive that guarantees non-equivocation of certain messages (e.g., Reliable Broadcast using RDMA [3] or trusted hardware [52]). In conjunction with digital signatures, non-equivocating multicast enables atomic delivery of a message to $2f + 1$ processes where $f$ are faulty [23]. Such primitives are relatively heavyweight, and hence they are used sparingly. For situations where non-equivocating multicast is not available, OsirisBFT can operate with $3f + 1$ processes in each sub-cluster. All other messages use reliable links that guarantee messages are not dropped or reordered (e.g., using RDMA RC protocol [46]).

### 4 Identifying Application Faults

OsirisBFT detects violations due to Byzantine failures by verifying the output records produced by executors. In this section, we model the impact of Byzantine failures on application results and develop verification operators to efficiently validate the records returned by executors.

#### 4.1 Modeling Task-Parallel Applications

Task-parallel applications consume a stream of tasks as input and produce a stream of records as output. Formally, applications operate on global states drawn from a set $S$, possible output records from a set $R$, and tasks from a set $T$, using a pair of functions $\langle U, A \rangle$. Here, $U(s,t)$ updates a global state $s \in S$ based on task $t \in T$ and returns a new state; and $A(s,t)$ executes an application-specific computation on a global state $s \in S$ and task $t \in T$ and returns a sequence of records $R = [r_0, r_1, ...]$ such that $\forall r_i \in R$.

Records in $R$ are application-defined, while tasks in $T$ consist of an opcode describing whether to execute $U, A$, or both, along with the application-defined data passed as the input to $U/A$. For example in the Anomaly Detection use case (Figure 1), each version of the network graph is a global state $s \in S$, records in $R$ represent subgraphs, $T$ consists of link updates to be applied to the network, $stateUpdate()$ is $U$, and $computeMatch()$ is $A$.

This model captures common use cases such as: (i) event-driven analytics where computation occurs in response to an update (i.e., tasks call for both a state update $U$ and a computation $A$); (ii) time-based analytics where computation and updates are decoupled (i.e., some tasks define only $U$ and appear whenever updates arrive, others define only $A$ and appear periodically to compute analytics logic); (iii) batch processing where the state is static (i.e., tasks never define
An output record \( r \) corresponding to task \( t \) is a **mismatch** if it does not satisfy the problem statement of \( t \) (i.e., \( r \notin R \) or \( r \notin \mathcal{A}(s, t) \)). A faulty process in WP can invalidate downstream computations by generating correct records for the wrong task, or simply random records.

**Duplication** A faulty process in WP can perform a replay attack by outputting a record \( r \) multiple times. An output record \( r \) corresponding to task \( t \) is a **duplication** if it has been output previously in the result stream for \( t \). Duplication can skew the output distribution and hence break applications.

**Omission** A faulty process in WP can omit portions of the output (i.e., produce a strict subset of \( \mathcal{A}(s, t) \)). For example, a malicious process can hide suspicious records from downstream analysis in a cybersecurity application.

The output failure model is **complete** with respect to Byzantine failures from the perspective of application output: if a certain sequence of records is expected, incorrect results can only arise due to mismatch, duplication, or omission.

**Lemma 4.1.** All invalid records produced by an executor correspond to an output failure.

**Proof.** We proceed by contradiction. If an executor neglects to produce any records, it will be classified as omission. So suppose that an executor produces an invalid record \( r \) which does not qualify as an output failure. To avoid mismatch, \( r \in R \) and \( r \in \mathcal{A}(s, t) \), for some valid task \( t \in T \) and corresponding state \( s, t \in S \). But then either either \( r \) repeats in \( \mathcal{A}(s, t) \) (classified as duplication), or \( r \) is valid since it follows all application semantics. \( \square \)

**Lemma 4.2.** Correct processes executing \( \mathcal{A} \) do not generate output failures. Faithfully executing \( \mathcal{A} \) with correct tasks does not generate output failures.

**Proof.** Given a valid task \( t \in T \) and corresponding state \( s, t \in S \), \( \mathcal{A}(s, t) \) does not result in an output failure by definition. Therefore the only way for a correct process to produce an output failure is if \( \mathcal{A} \) is executed with an invalid task \( t \notin T \) or a state \( s \) such that \( s \notin S \) or \( s \) does not correspond to \( t \). But this is impossible by Lemma 6.1, which proves that correct processes share the same view of the global application state, corresponding to a consistently ordered sequence of valid tasks. Therefore, no correct process will observe an incorrect state or invalid task while executing \( \mathcal{A} \). \( \square \)

### 4.3 Properties for Verification

An application is **verifiable** if it satisfies the following four properties:

- **[TASK-VALIDITY]** For an arbitrary object \( t \), it is possible to determine whether \( t \in T \) (i.e., whether \( \mathcal{A}(s, t) \) is defined for arbitrary \( s \in S \)).
- **[TASK-SCOPE]** For an arbitrary record \( r \), it is possible to determine whether \( r \in R \) (i.e., whether \( \mathcal{A} \) can produce \( r \)).
- **[TASK-ORDERED]** For every task \( t \in T \) and \( s \in S \), \( \mathcal{A}(s, t) \) is totally ordered.
- **[TASK-BOUNDED]** For every task \( t \in T \) and \( s \in S \), \( \mathcal{A}(s, t) \) is finite.

The Task-Validity property prevents mismatch failures where Byzantine input processes submit invalid tasks to be executed. A worker executing a task it was not assigned implies either mismatch (i.e., no input process generated the task) or duplication (i.e., a different worker was assigned the task). The Task-Scope property distinguishes valid and invalid records, so that mismatch failures involving incorrect or nonsensical records can be identified. The Task-Ordered property represents the process-local program order of the executing worker. A worker executing a task in a Task-Ordered application produces records in a specific order, and hence out-of-order output would imply duplication of mismatch. Finally, the Task-Bounded property requires that applications guarantee termination. Without this property, it is impossible to detect omission because observed output from a worker process cannot necessarily be compared with the expected output of \( \mathcal{A} \) (i.e., they can both be infinite), which makes it impossible to identify whether a record is missing.

### 4.4 Output Verification Model

Depending on the nature of the failures, they can be detected by: (a) employing generic protocols in the underlying system; or, (b) verifying output records against application semantics. Generic verification will be discussed in Section 5.2.2. Here we enable application-specific verification.

**Verification Operators.** We model application-specific verification operators that analyze output records. Verifiable applications implement these operators, which are invoked by verifiers (explained later in Section 5.2.1).

Algorithm 1 shows the three verification operators. isValid() checks whether a record \( r \) is valid (i.e., \( r \in R \) and is generated by the given task \( t \)). happensBefore() captures the process-local program order of the executing worker by checking whether a record \( a \) is ordered before record \( b \). Finally, outputSize() returns the number of output records for a task \( t \). The verification operators are also complete, i.e., they combine to detect all types of output failures (see proof in Section 6.1). Mismatch is detected by isValid() and outputSize() that together ensure the output records are the ones expected from the tasks. Duplication is detected...
We present the verifiable processing architecture. We first overview the system during normal execution, we develop optimistic protocols that optimize for low replication and fast communication. These decisions lead to graceful executions similar to a system without fault tolerance, but create a larger threat surface.

Task Batches & Record Chunks. To reduce communication overheads, tasks are streamed in batches. Likewise, the sequence of records generated by a single computation task is split into disjoint subsequences called chunks. Executors output a stream of chunks, allowing verifiers to proceed in parallel instead of waiting for the entire sequence of records.

5 Verifiable Processing with OsirisBFT

We present the verifiable processing architecture. We first summarize how tasks, records, and state are managed, and then describe the normal execution followed by verification protocols and strategies for workload management.

State Management. Guaranteeing the linearizability of computations on concurrently updating state requires efficient mechanisms for isolating state snapshots. Modern data analytics systems [18, 54, 81] employ multiversioning in their mechanisms for isolating state snapshots. Modern data analytics systems [18, 54, 81] employ multiversioning in their mechanisms for isolating state snapshots. Specifically, both the state and updates to the state are associated with a logical timestamp, and computations are restricted to specific states based on time intervals or windows. We replicate the timestamped state across all WP processes so that updates to the state are associated with a logical timestamps (line 4 in Algorithm 3). Tasks with only computations are given the timestamp of the most recent state update. Since ids are unique throughout the execution, faulty executors computing incorrect tasks can be identified. In the same consensus, VPCO assigns computations to executors. The tasks are then distributed among the cluster [P2]: computations are sent to assigned executors and state updates are broadcast to WP.

Coordination-Free Task Assignment. Each task has: (a) an assigned executor which computes the task and generates records; and, (b) an assigned verifier sub-cluster to verify those records. While task messages are smaller than the record chunks produced by those tasks, communicating these two assignments separately creates a race condition; the executor may send record chunks to its assigned verifiers before the coordinator can inform them of the assignment, causing them to falsely believe they are faulty.

To avoid this, tasks are assigned using a coordination-free scheme (lines 8-10 in Algorithm 3). VPCO sends signed task assignment messages to both executors and verifiers of the form (t, E, i), where t is the task to be executed by E in EP and verified by VP. The executor E receives f + 1 signed assignments for task t from different verifiers in the coordinator before executing t. As record chunks are generated for

---

**Algorithm 1** API for verification operators.

```java
def isValid(Record r, Task t):
    return isSubgraph(Network, r) && isMatch(Pattern, r) // PatternMatcher contains the matching logic.

def happensBefore(Record a, Record b):
    for i in range(len(a)): if a[i] < b[i] return true; if a[i] > b[i] return false; // if(a[i] == b[i]) continue;
    return false;

def outputSize(Task t):
    return PatternMatcher.count(Network, Pattern, t);  
```

using happensBefore() and outputSize() that identify repeated records arriving from the correct task. Omission is detected using outputSize().

**Example.** Algorithm 2 illustrates the verification operators for our Anomaly Detection use case from Section 1, where the computation primarily involves pattern matching on the network graph. isValid() ensures that each subgraph record is indeed a subgraph of the network graph, matches the pattern, and contains the updated link that resulted in the task to compute that record. happensBefore() determines the order between two subgraph records based on their prefix-ordering (prefix-ordering is guaranteed by most pattern matching systems [44, 68]). Finally, outputSize() simply returns the true number of subgraphs using efficient and exact counting optimizations (e.g., inclusion-exclusion [68] or subgraph morphing [45]) which are orders of magnitude faster than matching each individual subgraph.

![Figure 4. Overview of verification-based processing.](image-url)
Algorithm 3 Task Flow protocol.

1 // [P1] Coordinator receives task from input process
2 Void onRecvTask(Task t) {
3   if (!validTask(t)) return; // t \notin \mathcal{T}
4  t.timestamp = consensus(t, getTimestamp()); // Linearize
5  // [P2] Broadcast state updates and assign computations
6  if (hasStateUpdate(t)) broadcast(t);
7  if (hasComputation(t)) {
8      <e, vpi> = getNextExecutorAndVPi();
9      send(e, <t, e, vpi>);
10     multicast(vpi, <t, e, vpi>);
11     startReassignmentTimeout(t);
12  } else {
13     // [P2] All other workers receive f+1 copies from \mathcal{VP}_{CO}
14     Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
15        allChunks[t].append(chunk);
16    }
17  // [P3] Verifier receives f+1 copies from \mathcal{VP}_{CO}
18  Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
19      vpi = vpi.map(key -> key.vpi);
20      allChunks[t].append(chunk);
21      outputSize = allChunks[t].size();
22  }
23  // [P2] Executor receives f+1 copies from \mathcal{VP}_{CO}
24  Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
25      if (!validAssignment(<t, e, vpi>, sender))
26          return;
27      if (hasComputation(t)) {
28          if (isValid(r, t)) sendDownStream(t, allChunks[t].append(chunk));
29          else {
30              // [P3] Send output to assigned verifiers
31              for (chunk in compute(t)) {
32                  multicast(vpi, chunk);
33                  nonEquivocatingMulticast(vpi, digest(chunk));
34              }
35          }
36      } else {
37          cancelReassignmentTimeout(t);
38          cancelReassignmentTimeout(t);
39          resetReassignmentTimeout(t);
40          return true;
41      }
42  }
43  return true;
44  }
45  }
46  }
47  }
48  }
49  }
50  }
51  }
52  }
53  }
54  }
55  }
56  }
57  }
58  }
59  }
60  }
61  }
62  }
63  }
64  }
65  }

Algorithm 4 Verifier Output Flow protocol.

1 // [P3] Verifier receives from executor
2 Void onRecvAssignment(RecordMessage msg, String digest) {
3   TaskAssignment <t, e, vpi> = msg.getAssignment();
4   Executor sender = msg.getSender();
5   RecordList chunk = msg.getChunks();
6   if (!validAssignment(<t, e, vpi>, sender))
7       return;
8   digest = computeDigest(chunk);
9   // verify digest
10   if (!digest.hasComputeDigest(chunk))
11       return;
12   // verify task
13   if (!isValid(r) || !happensBefore(r, next(r)))
14       return;
15   if (chunk.taskFinished())
16       return;
17   allChunks[t].append(chunk);
18   allChunks[t].append(chunk);
19   return true;
20   

5.2 Detecting Failures

Failures manifest where messages flow between fault tolerant verifiers and processes without fault tolerance.

5.2.1 Application-Specific Failures. Theorem 3 and Algorithm 4 show the verification protocols run by the verifiers in the Task Flow and Output Flow, respectively.

Task Verification. The task assignment messages (signed originally by verifiers in \mathcal{VP}_{CO}) are prepended to each chunk and sent to \mathcal{VP}_{P}. Likewise, verifiers in \mathcal{VP}_{P} begin computing outputSize(t) upon receiving f+1 assignment messages, in order to overlap verification and execution. Verifiers in \mathcal{VP}_{P} ensure the assignment messages were signed by \mathcal{VP}_{CO} processes.

5.1.2 Output Flow: \mathcal{EP} \mapsto \mathcal{VP} \mapsto \mathcal{OP}. As an executor computes a task, it sends each record chunk \mathcal{C} to the assigned verifiers \mathcal{VP}_{P}, alongside a digest \sigma(C) [P3] using non-equivocating multicast (lines 27-30 in Algorithm 3). The final chunk for a task is tagged to signal its completion.

Verifiers independently check that they received a valid digest for \mathcal{C} and verify the records in \mathcal{C} are correct. Chunks are buffered until verification is complete before forwarding to downstream processes. To reduce message sizes, the leader verifier sends \mathcal{C} to \mathcal{VP}_{P} while each other verifier sends only \sigma(C). An output process accepts \mathcal{C} if it receives f+1 matching digests (including the one that accompanied \mathcal{C}) from the same verifier sub-cluster [P4].

5.2.2 Generic Protocol Failures. Generic failures range from impersonating processes to sophisticated attacks by different Byzantine processes cooperating across multiple phases in order to prevent output and compromise liveness.
Speculative Task Reassignment. Byzantine executors can cause omission faults and compromise liveness by responding to most messages but neglecting to send a final chunk, making them indistinguishable from a correct executor working on a difficult task. We address this issue using a speculative reassignment scheme. In the case where the final chunk is not marked or no output is received at all, when sufficient time passes after $\Lambda$, the task times out (line 11 in Algorithm 3) and $\text{VP}_{CO}$ assigns the task to another executor. Verifiers accept results from whichever executor finishes first. To avoid tying up all of $\text{EP}$ on one large task, the timeout duration for a given task is increased using exponential backoff.

Faulty Verifiers & Output Processes. A faulty verifier can compromise liveness by never forwarding chunks to $\text{OP}$ when it serves as leader of its sub-cluster. If an output process receives $f + 1$ digests $\sigma(C)$ from $\text{VP}_i$ but does not receive a matching chunk $C$ in some time after $\Lambda$ has passed, it multicasts messages to $\text{VP}_1$ to report a negligent leader. When verifiers receive a negligent leader report, they initiate an election for a new leader, and the new leader sends $C$ instead.

However, a negligent leader report is not sufficient to conclude a verifier is faulty due to the possibility of faulty output processes. Since there can only be $f$ failures in $\text{VP}_i$, verifiers track which leaders have been reported and assume an output process is Byzantine if it reports $f + 1$ different leaders in the same sub-cluster. Finally, to avoid spurious reports due to innocent network delays in communicating chunks, correct output processes apply exponential backoff to their timeout duration after each negligent leader report.

Limited Equivocation. Equivocation occurs if a faulty process sends different messages to different verifiers in a sub-cluster when it was expected to send identical messages. This is expected in three situations: (1) In [P1] when an input process sends tasks to $\text{VP}_{CO}$; (2) In [P3] when an executor sends record chunks to assigned verifiers of a task; and, (3) In [P4] when an output process sends negligent leader reports to all verifiers in the sub-cluster that sent chunk digests.

In [P1], equivocation by an input process does not affect the system because $\text{VP}_{CO}$ performs a Byzantine agreement protocol to linearize tasks, and conflicting task messages will simply not be agreed upon. Similarly in [P4], equivocation by an output process has no effect since $f + 1$ verifiers must initiate a leader election.

Finally, equivocation in [P3] is avoided by requiring executors to send chunk digests using non-equivocating multicast, and having correct output processes that receive at least one but fewer than $f + 1$ digests $\sigma(C)$ send a report containing $\sigma(C)$ to the verifiers, similar to negligent leader reports. Upon receiving the report, correct verifiers which have chunk $C$ broadcast it to the rest of the sub-cluster. The verifier that had not previously received $C$ but had received $\sigma(C)$ now processes $C$ as if it were sent from the original executor, eventually forwarding a digest to the output process.

5.3 Dynamic Role-Switching

The task execution workload and the verification workload can remain incongruous across various scenarios, impacting processing throughput. For example, tasks producing few results can leave verifiers idle despite executors being busy. Moreover, executors failing and leaving the system can drop processing throughput until new executors join the cluster.

To maintain throughput in such situations, verifiers can switch roles. When verifier resource utilization is low and there are many outstanding computation tasks, $\text{VP}_{CO}$ assigns tasks to verifiers from an underutilized sub-cluster $\text{VP}_j$ as if they were executors, and their output is routed through another sub-cluster $\text{VP}_i$. Verifiers in $\text{VP}_i$ finish their verification work and then execute assigned tasks. In the meantime, $\text{VP}_{CO}$ avoids assigning $\text{VP}_j$ as verifiers of tasks.

6 Safety and Liveness

This section proves correctness guarantees of OsirisBFT.

6.1 Safety

OsirisBFT satisfies safety; every correct output process observes records corresponding to a legal sequential execution of correct tasks submitted by input processes.

Lemma 6.1. The Task Flow results in a globally consistent ordering of tasks and task assignments to executors.

Proof. In [P1], input processes act as clients to $\text{VP}_{CO}$ in a Byzantine agreement protocol (correct by [3]), hence the tasks are safely linearized and correct verifiers agree on which executor is assigned the task in [P2]. A correct executor only accepts task assignments accompanied by $f + 1$ signatures, hence it can never be fooled into performing incorrect tasks. Correct executors and verifiers have a consistent view of task ordering and assignment, because network messages cannot be reordered and reassignment does not occur until after $\Lambda$ has passed, so initial task assignment messages are received strictly before reassignment messages. Furthermore, correct processes in $\text{WP}$ have a consistent view of the state. Monotonic timestamps mean that if a correct process receives $f + 1$ copies of a task with timestamp $k$ before receiving sufficient copies of a task with timestamp $k - 1$, the process simply waits to receive tasks in order before executing. A correct process receiving $f + 1$ correctly timestamped task assignments before the corresponding state update simply applies the state update before performing the computation. □

Lemma 6.2. Let $t_1, t_2, \ldots$ be the global (linearized) ordering of tasks submitted by $\text{IP}$, where $t_i, t_j \in T$. Let $s_i \in S$ be the state obtained by applying all state updates from tasks $t_1, \ldots, t_i$ to the initial application state in order.

Correct verifiers send $\text{OP}$ a sequence of records $R$ corresponding to a task $t$ if and only if $R = A(s_i, t)$.

Proof. By Lemma 6.1, all correct processes have access to $s_i$ during execution of $t$. Write $R$ as a concatenation of $k$
chunks, \( R = R_1|R_2|\ldots|R_k \), with chunk \( R_i \) consisting of \( l \) records \( r_{i1}|\ldots|r_{il} \). By Algorithm 4, verifiers forward \( R \) to \( OP \) whenever the following hold:

\[
\begin{align*}
&\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{l} \text{isValid}(r_{ij}) \quad (1) \\
&\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{l-j-1} \text{happensBefore}(r_{ij}, r_{i(j+1)}) \quad (2) \\
&\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{l-i-1} \text{happensBefore}(r_{il}, r_{i(i+1)}) \quad (3) \\
&\sum_{i=1}^{k} |R_i| = \text{outputSize}(t) \quad (4)
\end{align*}
\]

By (1), for all \( r \in R, r \in \mathcal{A}(s_i, t) \). Hence we can write \( \{ r : r \in R \} \subseteq \{ r : r \in \mathcal{A}(s_i, t) \} \). By (2) and (3), \( R \) is totally ordered according to \( \prec \), so every element of \( R \) is unique and we can write \( |\{ r : r \in R \}| = |R| \). Finally, by (4), \( |R| = |\mathcal{A}(s_i, t)| \), and we get \( \{ r : r \in R \} = \{ r \in \mathcal{A}(s_i, t) \} \). Since both \( R \) and \( \mathcal{A}(s_i, t) \) are totally ordered according to \( \prec \), we have \( R = \mathcal{A}(s_i, t) \). \( \square \)

**Corollary 6.1.** Correct output processes only observe correct records.

**Proof.** We prove this by contradiction. Suppose a correct output process \( O \) observes an incorrect sequence of records. As every sequence is made up of chunks, \( O \) must observe an incorrect chunk \( R_i \).

To accept \( R_i, O \) receives \( R_i \) and \( f \) digests \( \sigma(R_i) \) from \( f + 1 \) verifiers in the same sub-cluster. This implies either a correct verifier forwarded an incorrect chunk, contradicting Lemma 6.2, or there are more than \( f \) failures in the same sub-cluster. \( \square \)

**Theorem 6.3.** Every correct output process observes records corresponding to a legal sequential execution of tasks submitted by correct input processes.

**Proof.** By Corollary 6.1, there is a sequence of tasks \( T \) with corresponding states \( S \) such that correct output processes only receive records corresponding to \( \mathcal{A}(s_i, t) \) for \( t \in T \). By Lemma 6.1, all correct tasks are consistently ordered and successfully distributed to executors. Correct verifiers reject records corresponding to unassigned tasks and hence \( T \) contains only correct tasks submitted by an input process. Furthermore, correct verifiers have a consistent view of the ordering of tasks when verifying \( \mathcal{A}(s_i, t) \). Therefore, \( \forall t \in T, \mathcal{A}(s_i, t) \) follows a legal sequential execution of \( T \). \( \square \)

### 6.2 Liveness

Reliable links alongside partial synchrony guarantee that sent messages are always delivered without reordering. This constrains potential liveness issues to Byzantine behavior from processes, namely full or partial unresponsiveness leading to omission failures. We begin by proving that such failures cannot occur.

**Lemma 6.4.** If there is a non-faulty executor in \( EP \), every correct task is executed.

**Proof.** Let \( t \) be a correct task and suppose for contradiction that \( t \) is never executed. By Lemma 6.1, \( t \) is correctly distributed to an executor \( E \). If \( E \) is correct it will execute \( t \), so \( E \) must be faulty. But then \( f + 1 \) correct verifiers in \( VP_{CO} \) will eventually reassign \( t \) to a different executor, succeeding once again due to Lemma 6.1. If any other executor is correct, \( t \) will be executed after enough reassignments. Hence, \( t \) would remain unexecuted only when \( VP_{CO} \) cannot find a correct executor to reassign \( t \). This means all executors must be faulty, which is a contradiction. \( \square \)

**Theorem 6.5.** All correct output processes receive output records for every correct task submitted by input processes.

**Proof.** The underlying network is partially synchronous and messages are delivered reliably, thus executors can successfully forward output records to \( f + 1 \) verifiers. Additionally by Lemma 6.2, verifiers will successfully forward output records to the output processes. Finally, by Lemma 6.4, every correct task is executed. Therefore, all output records will be received by \( f + 1 \) verifiers whether they are generated by a correct executor or by the verifiers themselves. \( \square \)

### 7 Evaluation

We seek to understand how OsirisBFT affects performance and fault tolerance in realistic processing scenarios.

**System Details.** All experiments were conducted using a 40-node cluster with each node containing 8 logical cores and 6GB RAM, implemented as Docker containers like in [59]. Nodes are distributed among a testbed of machines connected by a Mellanox 100Gbps Infiniband network (0.075ms TCP ping latency), each with a 2-socket Intel Xeon Gold 6242R CPU. All experiments have a single node acting as both \( IP \) and \( OP \), and the remaining nodes allocated to \( WP \).

**OsirisBFT Implementation.** OsirisBFT is implemented in approximately 3500 lines of C++20 code. Regular communications use RDMA RC [46] via the ibverbs library, the non-equivocating multicast implementation follows open-source code for Mu [4], and the Fast & Robust algorithm [3] is
used for consensus. Processes use one CPU core for network operations, and the rest for cryptography and executing application tasks (executors) or verifying results (verifiers).

**Baselines.** We compare OsirisBFT performance against a baseline with zero fault tolerance (ZFT), as well as a replicated computing processing architecture (RCP) based on the RSM philosophy of replicating computation tasks. In ZFT, IP sends tasks to a coordinator worker in WP, which distributes the tasks to other workers who execute A and simply forward the results. BFT processing systems like Medusa [28] and others [27, 63, 69] target narrow application models such as map-reduce or lack open-source code, and state-of-the-art RSM systems like Kauri [59] focus on consensus and are inappropriate for heavyweight computations. Therefore, we implement RCP using the same network and consensus algorithms as OsirisBFT to capture the essence of the replicated processing design while ensuring prior works are represented fairly. Every worker is replicated to create sub-clusters of size $2f + 1$, with a designated coordinator sub-cluster WP$_{CO}$ that linearizes tasks from IP and distributes them among the other sub-clusters to be executed. The worker sub-clusters and OP only accept messages that are sent from $f + 1$ processes in a sub-cluster.

ZFT, RCP, and OsirisBFT all use a fully replicated data store since execution is bottlenecked by computations and not state updates. To confirm this, we ran write-only workloads on state-of-the-art BFT state management solutions Kauri [59] and Basil [70], as well as OsirisBFT. Figure 5a shows the results for different cluster sizes. The data store in OsirisBFT (and therefore the baselines) performs better as it does not incur overheads from transactional safety (Basil) or hashing blocks (Kauri), while also leveraging RDMA.

**Applications.** We consider three applications to evaluate performance under diverse conditions.

**Anomaly Detection:** Anomaly Detection computes instances of anomalous network structures that emerge as a result of state updates [26]. We built the application on top of OsirisBFT by integrating components from state-of-the-art pattern matching systems [10, 68] with verification operators implemented in only 100 lines of code.

**Motion Planning:** Motion Planning solves Mixed Integer Programs (MIP) to determine routes for e.g., airplanes [62] and robots [66], where output failures can lead to human harm. This is a batch-processing workload with no underlying state; tasks are drawn from a set of 107 standard MIP instances [25]. Executors use the state-of-the-art SCIP suite [15] to solve MIP instances. In OsirisBFT experiments, SCIP is configured to append a proof of optimality or infeasibility to each record [21]. The verification operators use built-in SCIP methods for validating the proof.

**Video Analysis:** This application operates on frequently updating video feed and periodically computes pixel clusters useful for image segmentation and motion detection [12, 17, 78] for, e.g., security cameras, where Byzantine fault tolerance is desirable. It uses clustering [43] where executors return the centroids of each cluster, and verifiers check the optimality of centroids.

**Methodology.** Experiments are run 5 times and their results averaged to account for variance. Throughput experiments measure average throughput (output records per second) over 5 minutes, with an initial 30 second warm-up period. IP submits tasks to WP in batches, and results are streamed continuously to OP. Except where specified otherwise, experiments are run with $f = 1$ and 1MB record chunks. In Anomaly Detection, IP streams 1K tasks per second, and EP finds 6-cliques missing 2 edges in the Orkut graph [77], common inputs in previous work [44, 68, 72]. In Video Analysis, IP streams 1K state updates per second and 5 computation tasks per second. In Motion Planning, IP streams 1K tasks per second. Dynamic role-switching is enabled in most experiments, and executions begin with $|WP|/(2f + 1)$ verifier sub-clusters. OsirisBFT converges to a stable number of sub-clusters during the warm-up period. Timeout values are calibrated empirically between 500 milliseconds and 5 seconds for each workload, necessary due to the complexity of the queries (tasks can take hundreds of seconds).

### 7.1 Graceful Execution Performance
We measure how output record throughput scales in OsirisBFT by varying the size of $n = |WP|$ between 1 and 32 nodes for each of the three applications. Figure 5 shows the results. OsirisBFT scales nearly as well as ZFT, with $1.2–4\times$ lower throughput. The performance gap between ZFT and OsirisBFT decreases as $n$ grows, with ZFT having $4\times$ higher throughput at $n = 4$ but only $1.4\times$ at $n = 32$ (Video Analysis). The other applications exhibit similar behaviour: in Motion Planning, ZFT initially has $2.3\times$ higher throughput at $n = 4$ but $1.4\times$ at $n = 32$, whereas the difference is $3.1\times$ to $1.6\times$ for Anomaly Detection. This aligns with our theoretical analysis indicating OsirisBFT scales in $O(n - f)$ instead of $O(n/f)$, as the relative cost of the $O(f)$ overhead reduces as $n$ grows.

Finally, OsirisBFT outperforms RCP in all workloads, achieving $1.9–2.3\times$ higher throughput at $n = 32$. The performance difference can be attributed to lower parallelism in RCP; at $n = 32$ RCP has 10 parallel worker sub-clusters
while OsirisBFT varies between 13 and 25 parallel executors based on how many verifiers switch roles.

OsirisBFT scales comparably to ZFT and scales better than RCP. OsirisBFT can reduce the performance penalty of fault tolerance relative to ZFT by scaling out.

7.2 Bottleneck Analysis

We performed detailed experiments to study performance across workloads. Results for Anomaly Detection are summarized below. By choosing appropriate queries from the literature, we emphasize stress on the CPU or the network, obtaining three workloads:

Medium CPU & Medium Output (MM): Listing instances of a dense size-6 pattern in the Orkut graph [77], a fairly expensive query with fairly large output.

Low CPU & High Output (LH): Listing 3-hop paths in Amazon Products [41], a computationally cheap query that creates massive result sets, to identify network bottlenecks.

High CPU & Low Output (HL): Listing 6-cliques in the Orkut graph [77], a computationally expensive query with relatively few results, to identify CPU bottlenecks.

Figure 6 shows the scalability on these workloads. As before, OsirisBFT scales nearly as well as ZFT, achieving 1.4–3.7× lower throughput, with the gap closing as n grows. Drilling down, we notice that MM and LH lead to worse scaling than the low output workload HL. By profiling network and CPU usage of the workloads in ZFT and OsirisBFT at n = 32, we discover that bandwidth usage on the link between OP and WP is similar during the high output workloads. In OsirisBFT WP sends messages to OP at a rate of 2.2GB/s in LH, 2.0GB/s in MM, but 1.8GB/s in HL, and in ZFT the rates are 3.4GB/s in both LH and MM, and 2.7GB/s in HL. Meanwhile average CPU usage of executors in OsirisBFT and ZFT is 93–95% during HL but 79–84% in LH and MM.

Finally, comparing OsirisBFT to RCP shows that with different workloads, OsirisBFT achieves 1.5–4× higher throughput at n = 32, due to better parallelism. We observe that in network-bound LH, RCP has 2.1×/1.5× lower throughput than ZFT/OsirisBFT, since parallelism is least important, but 6.5×/4× lower than ZFT/OsirisBFT in CPU-bound HL, where parallelism is most important. This follows our performance analysis in Section 2, as OsirisBFT is CPU-efficient.

Locating the Network Bottleneck. Since output rates during LH and MM are nearly identical and higher than HL, and CPU utilization is low, we confirm these workloads are bottlenecked by record communication in both OsirisBFT and ZFT. Importantly, this bottleneck only occurs at the link to OP, where records converge. The replicated communication between executors and verifier sub-clusters is parallelized over multiple links, and avoids this bottleneck. To further support this claim, we fix n = 32 and vary system load by controlling the rate of task submission between 100 per second and 100K per second, measuring task execution latency and output record throughput. Figure 6e shows the results.

In LH and MM, heavy task loads severely impact latency as network bandwidth to OP saturates. Increasing from 10K to 100K tasks per second leads to slim increases in throughput compared to the increase in latency. However, in the CPU-bound HL workload OsirisBFT continues to achieve higher throughput as load increases. Mean latency was not affected from 10K to 100K tasks/sec since tasks in HL are expensive, and the cluster has sufficient parallelism and bandwidth.

7.3 Dynamic Role-Switching

We investigate whether role-switching balances verification and execution by comparing the throughput with executions where verifier sub-clusters are kept static (i.e., verifiers cannot switch roles). Figure 6d shows the average throughput of the static executions, and plots the throughput over 2 minutes of execution with dynamic role-switching. The best static configuration is 4 sub-clusters, with 5 sub-clusters leaving verifiers idle and fewer than 4 sub-clusters causing a verification bottleneck. The role-switching execution began with 5 sub-clusters but settled at 4 during its warm-up phase. Two other role-switches occur at near 45 and 95 seconds to transition from 4 sub-clusters to 3 when the verification workload dips due to a few consecutive batches of cheap tasks, then back to 4 sub-clusters when output records become too many to handle. Overall, dynamic role-switching results in 11% higher average throughput and 31% higher peak throughput than the best static configuration.

7.4 Performance Under Failures

Executor Failures. OsirisBFT theoretically tolerates the failure of all executor processes. We investigate the behaviour of OsirisBFT when executors fail by injecting output failures in every process from EP. Figure 7a shows the throughput and bandwidth observed at OP during an execution of MM with f = 1, |VP| = 15 and |EP| = 16. At 45 seconds, each executor corrupts the final record in the next chunk it outputs to cause a mismatch. The failures are detected quickly, and throughput does not drop to 0, because 3 verifiers had previously switched roles to act as executors. OsirisBFT automatically recovers to half its previous throughput by 61 seconds, as 6 more verifiers switch roles to make up for faulty executors. We repeated this experiment with other failure types and observed that OsirisBFT always recovers to approximately half its previous throughput seconds after fault detection.

Verifier Failures. Faulty verifiers mainly affect performance when sub-cluster leaders do not forward chunks as expected and require OP to report them. We repeat the previous experiment but instead of faulty executors, verifier sub-cluster leaders do not send chunks to OP. We observe that throughput is only affected until a new leader is elected,
and OsirisBFT recovers to the same level since the executors are still correct.

**Fault Scalability.** We evaluate how OsirisBFT copes as more possibly faulty verifiers must be tolerated. Figure 7b compares executions of MM by OsirisBFT and RCP with \( n = 32 \) and varying fault tolerance levels \( f \). OsirisBFT with role-switching ran with up to 2 verifier sub-clusters and 9–20 executors. We observe OsirisBFT execution with \( f = 6 \) achieves 2.7x higher throughput than RCP with \( f = 2 \).

**8 Related Work**

Byzantine fault tolerance is a well-studied research area. We summarize the proposed solutions below.

**Byzantine Fault Tolerant Data Processing.** [27, 28, 57, 63, 69] enable Byzantine fault tolerance in data processing systems. ClusterBFT [69] replicates data-flow nodes in dataflow systems \( 2f + 1 \) times. [27] replicates mapper and reducer tasks in MapReduce so that an answer is correct when a quorum of 2 survivors achieve the same result. Medusa [28] extends this fault tolerance to MapReduce clusters in multi-cloud environments, where failures can affect entire data centers. [57] also replicates MapReduce tasks and chooses results that occur most frequently. Finally, Greta [63] is a BFT graph processing system that replicates vertex functions, relying on a trusted master process to detect if values differ.

These works rely on replication of core computation, limiting their scalability, while OsirisBFT enables BFT processing without replicating application tasks.

**Byzantine Fault Tolerance Protocols.** Research regarding BFT consensus spans decades, with seminal works like PBFT [19] inspiring many works that improve usability, resiliency, and performance [1, 5, 6, 13, 14, 20, 24, 29, 50, 51, 64, 75, 79]. [2] proposed the message-and-memory model used by [3] to achieve BFT consensus with \( 2f + 1 \) replicas. [79] divides workers into an agreement sub-cluster and execution sub-clusters; however, both the sub-clusters replicate tasks.

More recently, the popularity of permissioned blockchains has caused a resurgence in BFT research. HotStuff [80], Kauri [59], Fabric [11], Narwhal and Tusk [30], Damusys [32], and others [7, 31, 48, 49] develop efficient consensus strategies using optimized communication and transaction scheduling techniques as well as trusted components.

There has also been ample work on Byzantine fault tolerance for databases, like [8, 9, 40, 61, 70, 74, 76] that focus on serializable concurrent execution of transactions, and [34, 35, 58] that marry blockchain and database features.

All these works focus on consensus in the client-server model, where agreeing on an ordering of client requests is the only consistency requirement. As such, they relate only to the Task Flow of OsirisBFT, where tasks from IP are linearized. However, we target task-parallel processing and focus on computation and not state management, and our solution ensures the computation is not replicated.

**Byzantine Fault Detection.** PeerReview [38] and others [36, 37] propose failure detectors for Byzantine faults. These are modules in each node which only eventually detect simple deviations from a protocol, whereas a faulty executor can communicate correctly with other nodes while outputting incorrect records. OsirisBFT does not have such limitations since all communication with downstream nodes occurs through verifiers which can detect all output failures.

**9 Conclusion**

We presented OsirisBFT, a verification-based Byzantine fault tolerant processing architecture for distributed task-parallel applications that does not replicate computation tasks. We formalized the application failures and developed generic verification protocols that capture Byzantine failures with little coordination. OsirisBFT does not replicate computation tasks, hence delivering high processing throughput and scalability, for the first time allowing the performance gap between BFT and unreliable systems to close through horizontal scaling.
References


