# 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.

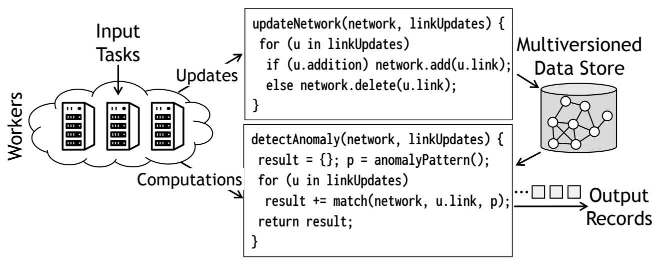
***Use Case: Anomaly Detection.*** Consider the problem of detecting anomalies in fast-changing networks. The application maintains an up-to-date version of the network graph using a continuous stream of link updates (insertion/deletion of network links), and performs pattern matching on the updated 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0435-2/24/03
https://doi.org/10.1145/3627535.3638468



```
updateNetwork(network, linkUpdates) {
 for (u in linkUpdates)
  if (u.addition) network.add(u.link);
  else network.delete(u.link);
}
```

```
detectAnomaly(network, linkUpdates) {
 result = {}; p = anomalyPattern();
 for (u in linkUpdates)
  result += match(network, u.link, p);
 return result;
}
```

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

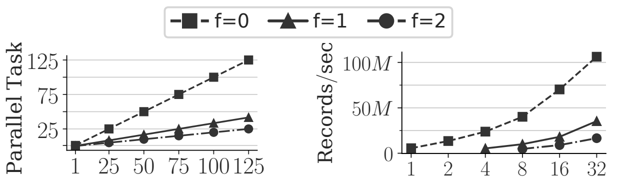
At the heart of these BFT solutions are replicated state machine protocols (RSM) that replicate both application state and task execution [65]. Specifically, workers are divided into independent subsets of replicas that all maintain the same application state and execute the same tasks, such that different 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 $\lfloor n/(2f+1) \rfloor$[1] tasks in parallel using RSM. This means even with minimum fault tolerance $f = 1$, processing with RSM requires 3× 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.*

[1] $\lfloor n/(2f+1) \rfloor$ using non-equivocation from modern RDMA networks [3] or trusted hardware [52]; otherwise the bound degrades to $\lfloor n/(3f+1) \rfloor$.



**(a)** Tasks **(b)** Throughput

**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.

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

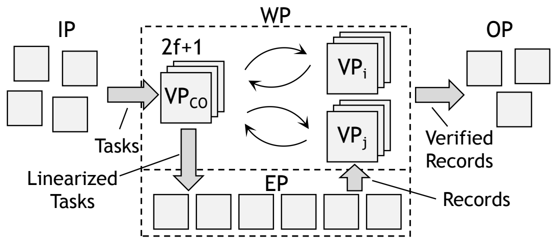


**Figure 3.** Verification-based processing architecture.

executor in $EP$ (*i.e.*, no task replication). $VP$ is partitioned further into $k$ independent Byzantine fault tolerant sub-clusters $VP_0 \dots VP_{k-1}$ with each $|VP_i| \geq 2f+1$ $(0 \leq i < k)$. 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*** $VP_{CO}$.

***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 $VP_{CO}$ 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 $VP_{CO}$. $VP_{CO}$ 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 $VP_i$ 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

| | Computation Replication | Computation Scalability | Communication Replication | Faults Tolerated |
|---|---|---|---|---|
| **ZFT** | 1 | $\lvert WP \rvert$ | 1 | 0 |
| **RCP** | $2f+1$ | $\lvert WP \rvert / O(f)$ | 1 | $\sum_{WP_i} f$ |
| **OsirisBFT** | 1 | $\lvert WP \rvert - O(f)$ | $2f+1$ | $\lvert EP \rvert + \sum_{VP_i} f$ |

**Table 1.** Performance and fault tolerance of OsirisBFT compared to replicated computation strategy (RCP) and a baseline with no fault tolerance (ZFT).

sub-clusters $WP_i$ 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 $VP_i$, 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 $VP_i$ tolerates $f$ failures (similar to $WP_i$ 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 $VP_i$ fail, and $VP_i$ 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 $\mathcal{S}$, possible output records from a set $\mathcal{R}$, and tasks from a set $\mathcal{T}$, using a pair of functions $\langle \mathcal{U}, \mathcal{A} \rangle$. Here, $\mathcal{U}(s,t)$ updates a global state $s \in \mathcal{S}$ based on task $t \in \mathcal{T}$ and returns a new state; and $\mathcal{A}(s,t)$ executes an application-specific computation on a global state $s \in \mathcal{S}$ and task $t \in \mathcal{T}$ and returns a sequence of records $R = [r_0, r_1, \ldots]$ such that $\forall_{r_i \in R}\, r_i \in \mathcal{R}$.

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

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 $\mathcal{U}$ and a computation $\mathcal{A}$); (ii) time-based analytics where computation and updates are decoupled (*i.e.*, some tasks define only $\mathcal{U}$ and appear whenever updates arrive, others define only $\mathcal{A}$ and appear periodically to compute analytics logic); (iii) batch processing where the state is static (*i.e.*, tasks never define

$\mathcal{U}$); as well as (iv) classic state management applications (*i.e.*, tasks never define $\mathcal{A}$).

## 4.2 Output Failure Model

A faulty worker can impact the output generated by task-parallel applications in various ways. We categorize the impact of arbitrary faults as three types of *output failures*.

**[Mismatch]** An output record $r$ corresponding to task $t$ is a *mismatch* if it does not satisfy the problem statement of $t$ (*i.e.*, $r \notin \mathcal{R}$ 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, $r \in \mathcal{R}$ and $r \in \mathcal{A}(s_t,t)$, for some valid task $t \in \mathcal{T}$ and corresponding state $s_t \in \mathcal{S}$. But then either $r$ repeats in $\mathcal{A}(s_t,t)$ (classified as Duplication), or $r$ is valid since it follows all application semantics. □

**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 $t \in \mathcal{T}$ and corresponding state $s_t \in \mathcal{S}$, $\mathcal{A}(s_t,t)$ 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 $t \notin \mathcal{T}$ or a state $s$ such that $s \notin \mathcal{S}$ 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 $\mathcal{A}$. □

## 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 $t \in \mathcal{T}$ (*i.e.*, whether $\mathcal{A}(s,t)$ is defined for arbitrary $s \in \mathcal{S}$).

**[Task-Scope]** For an arbitrary record $r$, it is possible to determine whether $r \in \mathcal{R}$ (*i.e.*, whether $\mathcal{A}$ can produce $r$).

**[Task-Ordered]** For every $t \in \mathcal{T}$ and $s \in \mathcal{S}$, $\mathcal{A}(s,t)$ is totally ordered.

**[Task-Bounded]** For every task $t \in \mathcal{T}$ and $s \in \mathcal{S}$, $\mathcal{A}(s,t)$ 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 or 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.*, $r \in \mathcal{R}$) 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

**Algorithm 1** API for verification operators.

```
Bool isValid(Record r, Task t);
Bool happensBefore(Record a, Record b);
Int outputSize(Task 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.

## 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 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 data stores to enable concurrent computations over well-defined deterministic 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 $WP$ to ensure consistency despite failures. Processes which incorrectly update state are caught because they output incorrect results (executor), or ignored as most processes in each sub-cluster operate correctly (verifier).

**Algorithm 2** Verification operators for Anomaly Detection.

```
// Network is network graph. Pattern is pattern to match.
// PatternMatcher contains the matching logic.
Bool isValid(Record r, Task t) {
  return isSubgraph(Network, r) && isMatch(Pattern, r)
    && r.links().contains(link(t));
}
Bool happensBefore(Record a, Record b) {
  for(int i=0; i<a.length(); ++i) {
    if(a[i] < b[i]) return true;
    if(a[i] > b[i]) return false;
    // if(a[i] == b[i]) continue;
  }
  return false;
}
Int outputSize(Task t) {
  PatternMatcher.count(Network, Pattern, t);
}
```

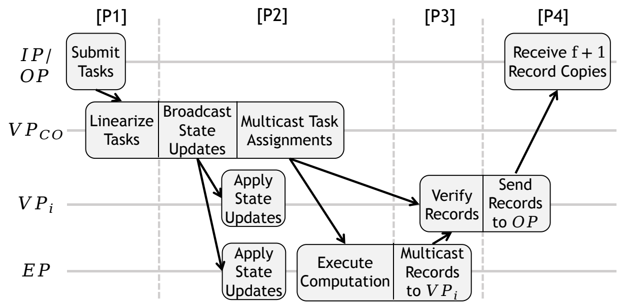


**Figure 4.** Overview of verification-based processing.

***Replication & Communication.*** To retain efficiency 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.

### 5.1 Normal Execution

Figure 4 shows the behavior of the system during graceful executions, divided across four phases (marked **[P1]**-**[P4]**).

**5.1.1 Task Flow: $IP \mapsto VP_{CO} \mapsto \{VP, EP\}$.** Algorithm 3 shows the protocols for this flow. The input processes send task batches to $VP_{CO}$ **[P1]**. $VP_{CO}$ performs consensus to linearize the tasks, assigning monotonically increasing ids to state updates, which serve as 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, $VP_{CO}$ 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). $VP_{CO}$ sends signed task assignment messages to both executors and verifiers of the form $\langle t, E, i \rangle$, where $t$ is the task to be executed by $E \in EP$ and verified by $VP_i$. 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 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> = getNextExecutorAndVP();
9      send(e, <t, e, vpi>);
10     multicast(vpi, <t, e, vpi>);
11     startReassignmentTimeout(t);
12   }
13 }
14 // [P2] All other workers receive f+1 copies from $VP_{CO}$
15 Void onRecvStateUpdate(Task t) { applyStateUpdate(t); }
16 //[P2] Verifier in $VP_i$ receives f+1 copies from $VP_{CO}$
17 Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
18   if (!validAssignment(<t,e,vpi>) || !hasComputation(t))
         return;
19   numRecords[t] = outputSize(t);
20   // report back for workload balancing
21   multicast(VPco, <t.id, numRecords[t]>);
22 }
23 // [P2] Executor receives f+1 copies from $VP_{CO}$
24 Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
25   if (!validAssignment(<t,e,vpi>) || !hasComputation(t))
         return;
26   // [P3] Send output to assigned verifiers
27   for (chunk in compute(t)) {
28     multicast(vpi, chunk);
29     nonEquivocatingMulticast(vpi, $\sigma$(chunk));
30   }
31 }
```

$t$, the task assignment messages (signed originally by verifiers in $VP_{CO}$) are prepended to each chunk and sent to $VP_i$. Likewise, verifiers in $VP_i$ begin computing `outputSize(`$t$`)` upon receiving $f+1$ assignment messages, in order to overlap verification and execution. Verifiers in $VP_i$ each ensure the assignment messages were signed by $VP_{CO}$ processes.

### 5.1.2 Output Flow: $EP \mapsto VP \mapsto OP$.

As an executor computes a task, it sends each record chunk $C$ to the assigned verifiers $VP_i$, 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 $C$ and verify the records in $C$ are correct. Chunks are buffered until verification is complete before forwarding to downstream processes. To reduce message sizes, the leader verifier sends $\langle C, \sigma(C)\rangle$ to the process in $OP$, while every other verifier sends only $\sigma(C)$. An output process accepts $C$ if it receives $f+1$ matching digests (including the one that accompanied $C$) from the same verifier sub-cluster **[P4]**.

## 5.2 Detecting Failures

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

### 5.2.1 Application-Specific Failures.

Algorithm 3 and Algorithm 4 show the verification protocols run by the verifiers in the Task Flow and Output Flow, respectively.

***Task Verification.*** MISMATCH caused by Byzantine $IP$ in **[P1]** is handled by validating input tasks before distributing them (`isValid()` on line 3 in Algorithm 3). Byzantine executors can also cause MISMATCH failures in **[P3]** by sending chunks that correspond to invalid tasks. Verifiers check that the task corresponding to every chunk has been assigned to that executor and verifier sub-cluster (line 37 in Algorithm 4).

**Algorithm 4** Verifier Output Flow protocol.

```
32 // [P3] Verifier receives from executor
33 Void onRecvRecords(RecordMessage msg, String digest) {
34   TaskAssignment <t, e, vpi> = msg.getAssignment();
35   Executor sender = msg.getSender();
36   RecordList chunk = msg.getChunk();
37   if (!validAssignment(<t, e, vpi>, sender)
38       || digest != computeDigest(chunk)
39       || !verify(chunk, t, e)) {
40     markByzantineExecutor(sender);
41     allChunks[t].clear();
42     reassignAllTasks(sender);
43   } else if (chunk.taskFinished()) {
44     cancelReassignmentTimeout(t);
45     sendDownStream(t, allChunks[t].append(chunk));
46   } else {
47     resetReassignmentTimeout(t);
48     seenRecords[t] += chunk.size();
49     allChunks[t].append(chunk);
50   }
51 }
52 Bool verify(RecordList chunk, Task t, Executor e) {
53   // ensure t is ongoing and chunks are sorted
54   RecordList prevChunk = allChunks[t][-1];
55   if (prevChunk != null && (prevChunk.taskFinished()
56         || !happensBefore(prevChunk[-1], chunk[0])))
57     return false;
58   for (r in chunk) // ensure all chunks are valid
59     if (!isValid(r, t) || !happensBefore(r, next(r)))
60       return false;
61   if (chunk.taskFinished()) // ensure nothing is missing
62     if (seenRecords[t] + chunk.size() != numRecords[t])
63       return false;
64   return true;
65 }
```

***Record Chunk Verification.*** Records in every chunk are verified against MISMATCH and DUPLICATION (lines 58-60 in Algorithm 4). Each record is checked for validity and whether it originates from the correct task. Finally, the records are verified to be in sorted order by applying `happensBefore()` to every adjacent pair of records.

***Inter-Chunk Ordering.*** A Byzantine executor can attempt to hide DUPLICATION across chunk boundaries, for example by sending a correct chunk twice. Verifiers protect against this by comparing the last record in the previous chunk with the first record of the newly received chunk, using the `happensBefore()` operator (lines 54-57 in Algorithm 4).

***Missing Records.*** Finally, OMISSION is detected by comparing the number of records sent by an executor with the true number of records corresponding to the task when its final chunk is received. The true count is available from `outputSize()` which runs asynchronously while the executor produces records (line 19 in Algorithm 3).

### 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 $\Delta$, the task times out (line 11 in Algorithm 3) and $VP_{CO}$ assigns the task to another executor. Verifiers accept results from whichever executor finishes first. To avoid tying up all of $EP$ 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 $OP$ when it serves as leader of its sub-cluster. If an output process receives $f + 1$ digests $\sigma(C)$ from $VP_i$ but does not receive a matching chunk $C$ in some time after $\Delta$ has passed, it multicasts messages to $VP_i$ 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 $VP_i$, 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 $VP_{CO}$; (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 $VP_{CO}$ 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, $VP_{CO}$ assigns tasks to verifiers from an underutilized sub-cluster $VP_i$ as if they were executors, and their output is routed through another sub-cluster $VP_j$. Verifiers in $VP_i$ finish their verification work and then execute assigned tasks. In the meantime, $VP_{CO}$ avoids assigning $VP_i$ 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 $VP_{CO}$ 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 $\Delta$ has passed, so initial task assignment messages are received strictly before reassignment messages. Furthermore, correct processes in $WP$ 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 IP, where $\forall i, t_i \in \mathcal{T}$. Let $s_t \in \mathcal{S}$ be the state obtained by applying all state updates from tasks $t_1, \ldots, t$ to the initial application state in order.*

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

*Proof.* By Lemma 6.1, all correct processes have access to $s_t$ during execution of $t$. Write $R$ as a concatenation of $k$

chunks, $R = R_1|R_2|\dots|R_k$, with chunk $R_i$ consisting of $l$ records $r_{i1}|\dots|r_{il}$. By Algorithm 4, verifiers forward $R$ to $OP$ whenever the following hold:

$$\bigwedge_{i=1}^{k}\bigwedge_{j=1}^{l}\texttt{isValid}(r_{ij}) \tag{1}$$

$$\bigwedge_{i=1}^{k}\bigwedge_{j=1}^{l-1}\texttt{happensBefore}(r_{ij}, r_{i(j+1)}) \tag{2}$$

$$\bigwedge_{i=1}^{k-1}\texttt{happensBefore}(r_{il}, r_{(i+1)1}) \tag{3}$$

$$\sum_{i}^{k}|R_i| = \texttt{outputSize}(t) \tag{4}$$

By (1), for all $r \in R$, $r \in \mathcal{A}(s_t, t)$. Hence we can write $\{r : r \in R\} \subseteq \{r : r \in \mathcal{A}(s_t, t)\}$. 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), $|R| = |\mathcal{A}(s_t, t)|$, and we get $\{r : r \in R\} = \{r \in \mathcal{A}(s_t, t)\}$. Since both $R$ and $\mathcal{A}(s_t, t)$ are totally ordered according to $\prec$, we have $R = \mathcal{A}(s_t, t)$. □

**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. □

**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 $\mathcal{A}(s_t, t)$ 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 $\mathcal{A}(s_t, t)$. Therefore, $\forall t \in \mathcal{T}, \mathcal{A}(s_t, t)$ follows a legal sequential execution of $T$. □

## 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. □

Lemma 6.4 relies on a non-faulty executor in *EP*. Without this assumption, it is impossible to tell whether all executors are faulty once $t$ has been assigned to every executor because the length of a task is not known *a priori*. To guarantee liveness in this worst case, after a final timeout $VP_{CO}$ can always reassign $t$ to a verifier sub-cluster, where at least $f + 1$ correct processes execute it and skip to **[P4]** in the Output Flow. In practice, however, executors can be assumed Byzantine after a sufficiently long timeout and failed over.

Using Lemma 6.4, Lemma 6.2, and our assumptions about the underlying network, we can now prove liveness.

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

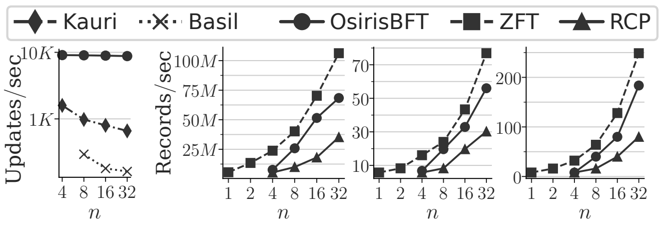
*Proof.* The underlying network is partially synchronous and messages are delivered reliably, thus executors can successfully forward output records to $f + 1$ verifiers. Additionally by Lemma 6.2, verifiers will successfully forward output records to the output processes. Finally, by Lemma 6.4, every 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. □

## 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



**(a)** State Updates. **(b)** Anomaly Detection. **(c)** Motion Planning. **(d)** Video Analysis.

**Figure 5.** Throughput scalability.

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 $\mathcal{A}$ 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× lower throughput. The performance gap between ZFT and OsirisBFT decreases as $n$ grows, with ZFT having 4× higher throughput at $n = 4$ but only 1.4× at $n = 32$ (Video Analysis). The other applications exhibit similar behaviour: in Motion Planning, ZFT initially has 2.3× higher throughput at $n = 4$ but 1.4× at $n = 32$, whereas the difference is 3.1× to 1.6× 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× 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.

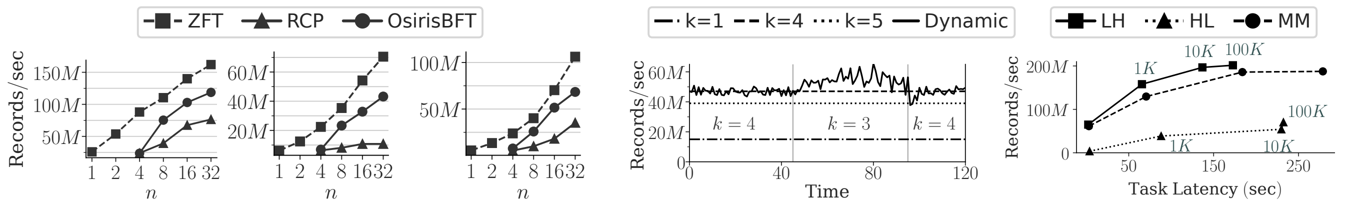
## 7.3 Dynamic Role-Switching

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

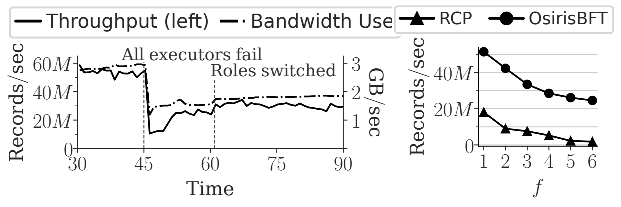
***Executor Failures.*** OsirisBFT theoretically tolerates the failure of all executor processes. We investigate the behaviour of OsirisBFT when executors fail by injecting output failures in every process from $EP$. Figure 7a shows the throughput and bandwidth observed at $OP$ during an execution of 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,



**(a)** Low CPU & High Output. **(b)** High CPU & Low Output. **(c)** Medium CPU & Output. **(d)** Effect of role-switching. $k$ is the number of active verifier sub-clusters. **(e)** Throughput-Latency for 100 to 100K tasks submitted per second.

**Figure 6.** Throughput performance across different Anomaly Detection workloads.



**(a)** Simultaneous failure of all executors in *EP* (*i.e.*, not the ones role-switched from verifiers). **(b)** Throughput for varying verifier fault tolerance level $f$.

**Figure 7.** Performance with Byzantine faults.

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 executing with $f = 6$ achieves 2.7× 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 data-flow systems $2f+1$ times. [27] replicates mapper and reducer tasks in MapReduce so that an answer is correct when a quorum of $2f+1$ tasks 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, Greft [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], Damysus [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 operators to capture the required application semantics for verification. OsirisBFT incorporates efficient 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

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '05, page 59–74, 2005.

[2] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing Messages While Sharing Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 51–60, 2018.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 409–418, 2019.

[4] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 599–616, November 2020.

[5] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. UBFT: Microsecond-Scale BFT Using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 862–877, 2023.

[6] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '05, page 45–58, 2005.

[7] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[8] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: A Cross-Application Permissioned Blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, July 2019.

[9] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 76–88, 2021.

[10] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, February 2018.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[12] Borislav Antić, Dragan Letić, Dubravko Ćulibrk, and Vladimir Crnojević. K-means based segmentation for real-time zenithal people counting. In *2009 16th IEEE International Conference on Image Processing*, ICIP '09, pages 2565–2568, 2009.

[13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant Byzantine Fault Tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 297–306, 2013.

[14] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, 2014.

[15] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021.

[16] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[17] Butler, Darren and Bove Jr, V. Michael and Sridha, Sridharan. Real-Time Adaptive Foreground/Background Segmentation. *EURASIP Journal on Advances in Signal Processing*, 2005, August 2005.

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[19] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, New Orleans, LA, February 1999.

[20] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th Conference on Symposium on Operating Systems Design & Implementation - Volume 4*, OSDI '00, USA, 2000.

[21] Kevin K. H. Cheung, Ambros Gleixner, and Daniel E. Steffy. Verifying Integer Programming Results. In *Integer Programming and Combinatorial Optimization*, pages 148–160. Springer International Publishing, 2017.

[22] Alexandra Chouldechova. Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data*, 5(2):153–163, June 2017.

[23] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, page 301–308, 2012.

[24] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 153–168, USA, 2009.

[25] William J. Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A Hybrid Branch-and-Bound Approach for Exact Rational Mixed-Integer Programming. *Mathematical Programming Computation*, 5(3):305–344, 2013.

[26] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. Communities of Interest. In *Proceedings of the 4th International Conference on Advances in Intelligent Data Analysis*, IDA '01, page 105–114, Berlin, Heidelberg, 2001. Springer-Verlag.

[27] Pedro Costa, Marcelo Pasin, Alysson N. Bessani, and Miguel Correia. Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 32–39, 2011.

[28] Pedro A. R. S. Costa, Xiao Bai, Fernando M. V. Ramos, and Miguel Correia. Medusa: An Efficient Cloud Fault-Tolerant MapReduce. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '16, pages 443–452, 2016.

[29] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 177–190, USA, 2006.

[30] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, 2022.

[31] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 123–140, 2019.

[32] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. DAMYSUS: Streamlined BFT Consensus Leveraging Trusted Components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 1–16, 2022.

[33] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[34] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A Shared Database on Blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, July 2019.

[35] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. Hybrid Blockchain Database Systems: Design and Performance. *Proceedings of the VLDB Endowment*, 15(5):1092–1104, May 2022.

[36] Fabiola Greve, Murilo Santos de Lima, Luciana Arantes, and Pierre Sens. A Time-Free Byzantine Failure Detector for Dynamic Networks. In *2012 Ninth European Dependable Computing Conference*, pages 191–202, 2012.

[37] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The Case for Byzantine Fault Detection. In *Proceedings of the 2nd Conference on Hot Topics in System Dependability - Volume 2*, HOTDEP '06, page 5, USA, 2006.

[38] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '07, page 175–188, 2007.

[39] Jelle Hellings and Mohammad Sadoghi. Coordination-Free Byzantine Replication with Minimal Communication Costs. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory*, volume 155 of *ICDT '20*, pages 17:1–17:20, Dagstuhl, Germany, 2020.

[40] Jelle Hellings and Mohammad Sadoghi. ByShard: Sharding in a Byzantine Environment. *Proceedings of the VLDB Endowment*, 14(11):2230–2243, October 2021.

[41] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR*, abs/2005.00687, 2020.

[42] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *Proceedings of the 20th USENIX Conference on Security*, SEC '11, page 29, USA, 2011.

[43] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., USA, 1988.

[44] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[45] Kasra Jamshidi, Harry Xu, and Keval Vora. Accelerating Graph Mining Systems with Subgraph Morphing. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 162–181, 2023.

[46] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference*, ATC '16, page 437–450, 2016.

[47] Miltiadis Kandias, Nikos Virvilis, and Dimitris Gritzalis. The Insider Threat in Cloud Computing. In Sandro Bologna, Bernhard Hämmerli, Dimitris Gritzalis, and Stephen Wolthusen, editors, *Critical Information Infrastructure Security*, pages 93–103, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[48] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.

[49] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy*, SP '18, pages 583–598, 2018.

[50] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems*, 27(4), January 2010.

[51] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, DSN '04, pages 575–584, 2004.

[52] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 1–14, USA, 2009.

[53] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. An Empirical Study of Memory Hardware Errors in a Server Farm. In *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, HotDep'07, page 13–es, 2007.

[54] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *CIDR*, 2013.

[55] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 415–426, 2015.

[56] Nema Milaninia. Biases in Machine Learning Models and Big Data Analytics: The International Criminal and Humanitarian Law Implications. *International Review of the Red Cross*, 102(913):199–234, 2020.

[57] Mircea Moca, Gheorghe Cosmin Silaghi, and Gilles Fedak. Distributed Results Checking for MapReduce in Volunteer Computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1847–1854, 2011.

[58] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proceedings of the VLDB Endowment*, 12(11):1539–1552, July 2019.

[59] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '21, page 35–48, 2021.

[60] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 343–356, 2011.

[61] Ricardo Padilha and Fernando Pedone. Augustus: Scalable and Robust Storage for Cloud Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 99–112, 2013.

[62] Lucia Pallottino, Eric M Feron, and Antonio Bicchi. Conflict resolution problems for air traffic management systems solved with mixed integer programming. *IEEE Transactions on Intelligent Transportation Systems*, 3(1):3–11, 2002.

[63] Daniel Presser, Lau Cheuk Lung, and Miguel Correia. Greft: Arbitrary Fault-Tolerant Distributed Graph Processing. In *2015 IEEE International Congress on Big Data*, pages 452–459, 2015.

[64] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *SIGOPS Operating Systems Review*, 35(5):15–28, October 2001.

[65] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Survey*, 22(4):299–319, December 1990.

[66] Tom Schouwenaars, Bart De Moor, Eric Feron, and Jonathan How. Mixed integer programming for multi-vehicle path planning. In *2001 European Control Conference*, ECC '01, pages 2603–2608. IEEE, 2001.

[67] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. *Communications of the ACM*, 54(2):100–107, February 2011.

[68] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, 2020.

[69] Julian James Stephen and Patrick Eugster. Assured Cloud-Based Data Analysis with ClusterBFT. In *14th International Middleware Conference*, volume LNCS-8275 of *Middleware '13*, pages 82–102, Beijing, China, December 2013. Springer. Part 1: Distributed Protocols.

[70] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (Transactions). In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '21, page 1–17, 2021.

[71] Michiaki Tatsubori and Shohei Hido. Opportunistic Adversaries: On Imminent Threats to Learning-Based Business Automation. In *Proceedings of the 2012 Annual SRII Global Conference*, SRII '12, page 120–129, USA, 2012.

[72] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, 2015.

[73] Tian Tian, Jun Zhu, Fen Xia, Xin Zhuang, and Tong Zhang. Crowd Fraud Detection in Internet Advertising. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1100–1110, 2015.

[74] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '07, page 59–72, 2007.

[75] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, page 135–144, USA, 2009.

[76] Hiroyuki Yamada and Jun Nemoto. Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems. *Proceedings of the VLDB Endowment*, 15(7):1324–1336, June 2022.

[77] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[78] Hong Yao, Qingling Duan, Daoliang Li, and Jianping Wang. An improved K-means clustering algorithm for fish image segmentation. *Mathematical and Computer Modelling*, 58(3):790–798, 2013.

[79] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '03, page 253–267, 2003.

[80] 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, page 347–356, 2019.

[81] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, 2013.