## Title

ByzID: Byzantine Fault Tolerance from Intrusion Detection

Authors
Duan, Sisi
Levitt, Karl
Meling, Hein
et al.

Publication Date
2014-10-06

Peer reviewed

# ByzID: Byzantine Fault Tolerance from Intrusion Detection

Sisi Duan
*UC Davis*
*sduan@ucdavis.edu*

Karl Levitt
*UC Davis*
*levitt@ucdavis.edu*

Hein Meling
*University of Stavanger, Norway*
*hein.meling@uis.no*

Sean Peisert
*UC Davis and LBNL*
*speisert@ucdavis.edu*

Haibin Zhang
*UC Davis*
*hbzhang@ucdavis.edu*

*Abstract*—**Building robust network services that can withstand a wide range of failure types is a fundamental problem in distributed systems. The most general approach, called Byzantine fault tolerance, can mask arbitrary failures. Yet it is often considered too costly to deploy in practice, and many solutions are not resilient to performance attacks. To address this concern we leverage two key technologies already widely deployed in cloud computing infrastructures: replicated state machines and intrusion detection systems.**

**First, we have designed a general framework for constructing Byzantine failure detectors based on an intrusion detection system. Based on such a failure detector, we have designed and built a practical Byzantine fault-tolerant protocol, which has costs comparable to crash-resilient protocols like Paxos. More importantly, our protocol is particularly robust against several key attacks such as flooding attacks, timing attacks, and fairness attacks, that are typically not handled well by Byzantine fault masking procedures.**

## I. Introduction

The availability and integrity of critical network services are often protected using two key technologies: a *replicated state machine* (RSM) and an *intrusion detection system* (IDS).

An RSM is used to increase the availability of a service through consistent replication of state and masking different types of failures. RSMs can be made to mask arbitrary failures, including compromises such as those introduced by malware. Such RSMs are referred to as Byzantine fault-tolerant (BFT). Despite significant progress in making BFT practical [6, 21], it has not been widely adopted, mainly because of the complexity of the techniques involved and high overheads. In addition, BFT is not a panacea, since there are a variety of attacks, such as various performance attacks that BFT does not handle well [2, 9]. Also, if too many servers are compromised then masking is not possible.

An IDS is a tool for (near) real-time monitoring of host and network devices to detect events that could indicate an ongoing attack. There are three types of intrusion detection: (*a*) *Anomaly-based* intrusion detection [15] looks for a statistical deviation from a known "safe" set of data. Most spam filters use anomaly detection. (*b*) *Misuse-based* intrusion detection [33] looks for a pre-defined set of signatures of known "bad" things. Most host and network-based intrusion detection systems and virus scanners are misuse detectors. (*c*) *Specification-based* intrusion detection systems [26] are the opposite of misuse detectors. They look for a pre-defined set of signatures of known "good" things.

In practice, BFT and IDSs are almost always used independently of each other. Additionally, the most commonly used fault-tolerance techniques typically only handle crash failures. For instance, Google uses Paxos-based RSMs in many core infrastructure services [5, 12]. As a result, only a handful of additional techniques are typically used to cope with other failures than crashes. However, those techniques are either ad hoc or are unable to handle attacks and arbitrary failures (e.g., software bugs). For attacks that are hard to mask (e.g., too many corrupted servers, simultaneous intrusions, and various performance attacks), IDSs are usually used. However, IDSs themselves suffer from deficiencies that limit their utility, including false positives that overly burden a human administrator who has to process intrusion alerts, and false negatives for when an ongoing attack is not detected. Also, IDSs themselves are not resilient to crashes.

In this paper, we propose a unified approach that leverages intrusion detection to improve RSM resilience, rather than using each technique independently. We describe the design and implementation of a BFT protocol—ByzID—in which we use a lightweight specification-based IDS as a failure detection component to build a Byzantine-resilient RSM. ByzID distinguishes itself from previous BFT protocols in two respects: (1) Its efficiency is comparable to its crash failure counterpart. (2) It is robust against a wide range of failures, providing consistent performance even under various attacks such as flooding, timing, and fairness attacks. We note that ByzID does not protect against all possible attacks, only those that the IDS can help with. Underlying ByzID are several new design ideas:

*Byzantine-resilient RSM.* ByzID is a primary-based RSM protocol, adapted for combining with an IDS. In this protocol, a primary receives client requests and issues ordering commands to the other replicas (backups). All replicas process requests and they all reply to the client. In the event of a replica failure, a new replica runs a reconfiguration protocol to replace the failed one. The primary reconfiguration runs *in-band*, where other replicas wait until reconfiguration completes. Reconfiguration for other replicas runs *out-of-band*, where replicas continue to run the protocol without waiting for the reconfiguration.

*Monitoring instead of Ordering.* Our protocol relies on a *trusted* specification-based IDS [26], to detect and suppress primary equivocation, enforce fairness, detect various other replica failures, and trigger replica reconfiguration. Our IDS is provided with a *specification* of our ByzID protocol, allowing the IDS to monitor the behavior of the replica. Note that, the way our protocol uses the IDS is so simple that the IDS could be implemented as a trivially small, timed state machine that can be embedded in a simple reference monitor, and can thus easily be built in hardware. However, for our proof of concept prototype we leverage the Bro IDS framework [38]. While some existing BFT protocols use *trusted components* [7, 24, 32, 40] to decide on the ordering client requests, our trusted IDS approach simply monitors and discards messages to enforce ordering.

*Independent Trusted Components.* In ByzID, each RSM replica is associated with a separate IDS component. However, even if an IDS experiences a crash, its replica can continue to process requests. Hence, both liveness and safety can be retained as long as the replicas themselves remain correct. For BFT protocols relying on trusted components, replicas typically fail together with their trusted components.

*Simple Rooted-Tree Structure.* When deploying ByzID in a local area network (LAN), we organize the replicas in a simple rooted-tree structure, where the primary is the root and the backups are its direct siblings (leafs). Furthermore, backups are not connected with one another. With such a structure and together with the aid of IDSs we can avoid using cryptography to protect the links between the primary and the backups. This is because the IDS can enforce non-equivocation, identify the source and destination of messages, and prevent message injection. Moreover, a backup only needs to send or receive messages from the primary, thus backups need not broadcast. Such a structure also helps to prevent flooding attacks from faulty replicas.

**Our contributions** can be summarized as follows:

(1) We have designed and implemented a general and efficient framework for constructing Byzantine failure detectors from a specification-based IDS.
(2) Relying on such failure detectors, our ByzID protocol uses only $2f+1$ replicas to mask $f$ failures. ByzID uses only *three* message delays from a client's request to receiving a reply, just one more than non-replicated client/server.
(3) We have conducted a performance evaluation of ByzID for both local and wide area network environments. For LANs, ByzID has comparable performance to Paxos [29] in terms of throughput, latency, and scalability. We also compare ByzID's performance with existing BFT protocols.
(4) We prove the correctness of ByzID under Byzantine failures, and discuss how ByzID withstands a variety of attacks. We also provide a performance analysis for a number of BFT protocols experiencing a failure.
(5) Finally, we use ByzID to implement an NFS service, and show that its performance overhead, with and without failure, is low, both compared to non-replicated NFS and other BFT implementations.

## II. System Model

We consider an asynchronous distributed system where faulty replicas can behave arbitrarily. The system can only be compromised if an adversary can gain control over enough "faulty" replicas. Let $n$ denote the total number of replicas in the system, while $f$ denotes the maximum number of faulty replicas that the system can tolerate. Let $\Pi = \{p_0, p_1, \cdots, p_{n-1}\}$ be the set of replicas. The replicas can be in one or more administrative domains, perhaps geographically separated.

Replicas may be connected in a complete graph or an incomplete graph network. We assume *fair-loss links*, where if a message is sent infinitely often by a correct sender to a correct recipient, it is received infinitely often. Furthermore, links do not produce spurious messages and do not repeatedly perform more transmissions than the sender. Note that one can use fair-loss links to build *reliable links*, but only when both the sender and receiver are correct. However, our protocol needs to build reliable links from fair-loss links even when the sender is potentially (Byzantine) faulty. We therefore assume the fair-loss link abstraction. We further assume that adversaries are unable to inject messages on the links between the replicas. This is reasonable when all replicas are monitored by IDSs and they reside in the same administrative domain. We assume that IDSs are trusted components, that is, they cannot be compromised, but they may experience benign failures, such as crashes.

The safety of our system holds in any asynchronous environment, where messages may be delayed, dropped, altered, or delivered out of order. Liveness is ensured under *partial synchrony* [18]. That is, synchrony holds only after some unknown global stabilization time, but the bounds on communication and processing delays may be unknown.

Let $\langle X \rangle_{i,j}$ denote an authentication certificate for $X$, sent from $i$ to $j$. Such certificates can be implemented using MACs or signatures. We use MACs for authentication unless otherwise stated. Let $[Z]$ denote an unauthenticated message for $Z$, where no MACs or signatures are appended.

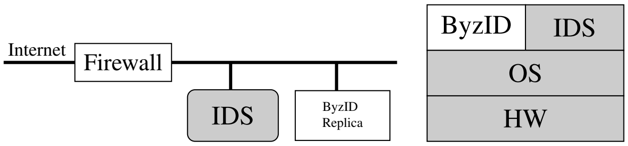
## III. Byzantine Failure Detector from Specification-Based Intrusion Detection

Specification-based intrusion detection is a technique used to describe the *desirable* behavior of a system. Therefore, by definition, any sequence of operations outside of the specifications is considered to be a violation.

In order to detect such violations, we use an IDS to monitor the behavior of the ByzID replication protocol, executed by a replica. This is illustrated in Fig. 1(a), where each replica has an associated IDS component, which monitors the replica's incoming and outgoing messages. Thus, the IDS cannot modify any messages, only detect misbehavior.

Instead, the IDS only captures the network packets of the protocol and analyze them according to the specification. Thus, the IDS acts as a distributed oracle and triggers alerts if the replica does not follow the specifications of the prescribed ByzID protocol. In case of an alert, the detected replica should be recovered, or removed through a reconfiguration procedure. Meanwhile, the messages sent by the faulty replica should be blocked. This is accomplished by the IDS inserting a packet filter into the local firewall.

The trusted IDS and the ByzID protocol can be separated in various ways [7], e.g., as shown in Fig. 1(a) or by using separate virtual machines. The IDS can also be implemented in trusted hardware or directly on top of a security microkernel [31]. In our prototype however, the IDS and ByzID replica simply execute as separate processes under the same OS, as shown in Fig. 1(b).



(a) The IDS interface at a replica. (b) IDS implementation.

Figure 1. The IDS/ByzID architecture. (Components shown on gray background are considered to be trusted.)

### A. Byzantine Failure Detector Specifications

The primary orders client requests by maintaining a queue, as shown in Fig. 2. To ensure that the primary orders messages *correctly*, we define a set of IDS *specifications* for Byzantine failure detectors. Such detectors can be used together with most existing primary-based BFT protocols. Below we summarize the specifications for our Byzantine failure detector.

- *Consistency.* The primary sends consistent messages to the other replicas.
- *No Gap.* The primary sends totally ordered requests to the replicas.
- *Fairness.* The primary orders requests in FIFO order.
- *Timely Action.* The primary orders client requests in a timely manner.

(1) The *consistency* rule prevents the primary from sending "inconsistent" *order* messages to the other replicas without being detected. The *order* message is the message sent by the primary to initialize a round of agreement protocol, such as the *pre-prepare* message in PBFT [6]. More specifically, the primary must send the *same* order message to the remaining $n-1$ replicas. To this end, the IDS can monitor the number of matching messages with the same sequence number. In case of inconsistencies, an alert is raised and the inconsistent messages are blocked.

(2) The *no gap* rule prevents the primary from introducing gaps in the message ordering. The sequence number in the order messages sent by the primary must be incremented by exactly one. Namely, the primary sends an order message with sequence number $N$ only after it has sent an order message for $N-1$. In the event that the primary sends out an "out-of-order" message, an alert is raised by the IDS.

(3) We argue that the conventional fairness definition is insufficient for many fairness-critical applications, such as registration systems for popular events, e.g. concerts or developer conferences with limited capacity. Thus, we define *perfect fairness* such that the RSMs must execute the client requests in FIFO order. As shown in Fig. 2, the IDS monitors client requests received by the primary and the order messages sent by the primary. With this, the IDS can verify that the primary follows the correct client ordering observed by the IDS. This is typically hard to achieve for common BFT protocols.

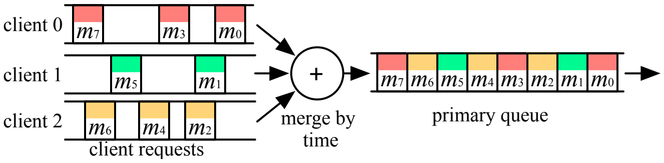


Figure 2. Queue of client requests.

(4) The *timely action* rule detects crash-stop and a "slow" primary. The IDS simply starts a timer for the first request in the queue. If the primary sends a valid order message before the timer expires, the IDS cancels the timer. Otherwise, the IDS raises an alert. The timer can be a fixed value or adjusted adaptively, e.g. based on input from an anomaly-based IDS.

Traditionally, BFT protocols used arbitrarily-chosen timeouts for detecting faulty actors with excessive latencies. But those timeouts may not reflect reality. As such, *anomaly detection* is another intrusion detection technique that can help address this issue. Since anomaly detection is typically based on a statistical deviation from normal behavior, we use anomaly detection to baseline the latencies between replicas at the beginning and then look for deviations from the baseline outside a particular bound. The baseline is updated over time to take benign changes in system and network performance into account. This is typically done by weighting recent baselines less than older baselines so that an adversary cannot "game" the system as easily.

### B. The IDS Algorithm

Our IDS specifications are detailed in Algorithm 1. The IDS maintains the following values: a queue of client requests $\mathcal{Q}$, current [ORDER] message $M$, current sequence number $N$, a boolean array $\mathcal{C}[n]$ used to ensure that an [ORDER] message is sent to all replicas, and a timer $\Delta$ for the timely action.

As depicted in Fig. 2, the primary stores the client requests in a the order of receiving them. The IDS also keeps the same queue of requests and monitors the [ORDER] messages sent by the primary. As shown in Algorithm 1, when the IDS observes a new [Order] message, it verifies the correctness of *no gap*, *consistency*, and *fairness*. The no gap rule is violated, if the sequence number in the [Order] message is different from $N + 1$. Consistency is violated if the primary does not send to the other $n - 1$ replicas. Fairness is violated, if the request in the [Order] message is not equal to the first request in the IDS's queue.

**Algorithm 1** The IDS Specifications

```
1:  **Initialization:**
2:  $n$                                        {Number of replicas}
3:  $\Pi = \{p_0, p_1, \cdots, p_{n-1}\}$      {Replica set; $p_0$ is the primary}
4:  $\mathcal{Q}$                              {Queue of client requests}
5:  $M$                                        {Current [Order] msg being tracked}
6:  $N \leftarrow 0$                           {Current sequence number}
7:  $\mathcal{C} \leftarrow \emptyset$         {Array: $\mathcal{C}[i] = 1$ if seen [Order] msgs to $p_i$}
8:  $\Delta$                                   {Timer; initialized by anomaly-based IDS}
9:  **on event** $m = \langle \text{Request}, o, T, c \rangle_{c,p_0}$ **do**
10:     **if** ($|\mathcal{Q}| = 0$) **then** starttimer($\Delta$)     {For timely action}
11:     $\mathcal{Q}$.add($m$)                 {Add client $c$'s msg to $\mathcal{Q}$}
12: **on event** $M' = [\text{Order}, N', m, v, c]_{p_0,p_i}$ **do**
13:     **if** $N' = N + 1 \wedge |\mathcal{C}| = 0 \wedge m = \mathcal{Q}$.front() **then**
14:         $N \leftarrow N'$                  {New current sequence number}
15:         $M \leftarrow M'$                  {New current [Order] msg}
16:         $\mathcal{C}[i] \leftarrow 1$      {Have seen [Order] msg to $p_i$}
17:     **else if** $|\mathcal{C}| > 0 \wedge \mathcal{C}[i] = 0 \wedge M = M'$ **then**
18:         $\mathcal{C}[i] \leftarrow 1$      {Have seen [Order] msg to $p_i$}
19:         **if** $|\mathcal{C}| = n - 1$ **then**     {Seen enough [Order] msgs?}
20:             $\mathcal{C} \leftarrow \emptyset$     {Reset array}
21:             $\mathcal{Q}$.remove()         {Remove msg from $\mathcal{Q}$}
22:             canceltimer($\Delta$)
23:             **if** ($|\mathcal{Q}| > 0$) **then** starttimer($\Delta$)
24:         **else alert**                     {Violation of first three specifications}
25: **on event** timeout($\Delta$) **do alert**
```

To monitor the *timely action*, the IDS starts a timer in two cases: a) The queue is empty and the IDS observes a new client request, as shown in Lines 10; b) The primary has already sent an [Order] message to the other replicas and the queue is not empty, as shown in Lines 23. Finally, an alert is also raised if the primary does not send the [Order] message to the other replicas before the timer expires.

## IV. The ByzID Protocol

ByzID has three subprotocols: *ordering*, *checkpointing*, and *replica reconfiguration*. The ordering protocol is used during normal case operation to order client requests. The checkpoint protocol bounds the growth of message logs and reduces the cost of reconfiguration. The reconfiguration protocol reconfigures the replica when its associated IDS generates an alert.

Following *h*BFT [17], we distinguish between *normal* and *fault-free* cases as follows: we define the normal case as the primary being correct, while the other replicas might be faulty. Note that, the normal case definition is less restrictive than the fault-free case, where all replicas must be correct.

BFT protocols that rely on trusted components, e.g. [7, 24, 32], can use $2f + 1$ replicas to tolerate $f$ failures and use one less round of communication than PBFT. While these other protocols use trusted hardware directly to order clients requests, we achieve the same goal using IDSs that conduct monitoring and filtering. This feature makes it possible for the system to achieve safety even if all IDSs are faulty. We use the Byzantine failure detector for the primary to ensure that the requests are delivered consistently, in a total order, and in a timely and fair manner. With the aid of the IDS, it is possible to reduce communication rounds further for the normal case. Ideally, we seek a protocol comparable to the fault-free protocol of Zyzzyva [27] (and minZyzzyva [40]).

To this end, we follow a primary-backup scheme [1, 4], where in each configuration, one replica is designated as the primary and the rest are backups. The correct primary sends order messages to the backups, and *all* correct replicas execute the requests and send replies to clients.

However, two technical problems remain. First, since our protocol lacks the regular commit round, we need the primary to reliably send messages through fair-loss links between the potentially faulty primary and the backups. Second, the Byzantine failure detector does not enforce authentication between the primary and the backups.

To address the first problem, we require backups to send [Ack] messages to the primary. And with the aid of the IDSs, we also provide a mechanism to handle message retransmissions. For the second problem, we distinguish between the core ByzID protocol for LANs, and ByzID-W for wide area networks (WANs). ByzID exploits the non-equivocation property provided by the IDS, and its ability to track the source and destination of messages. This allows ByzID to operate without cryptography on the links connecting the replicas.

To cope with the possibility of message injections in WANs, the ByzID-W primary instead uses authenticated order messages. These must be verified by both the backup replicas and the IDS. See §IV-B for further details.

### A. The ByzID Protocol

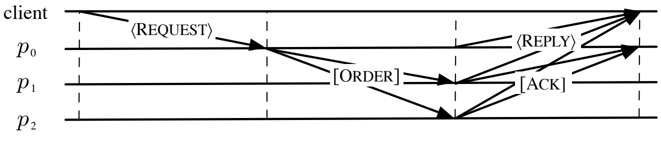


Figure 3. The ByzID protocol message flow.

**The ordering protocol.** Fig. 3 and Fig. 4 depict normal case operation. Below we describe the steps involved in the ordering protocol.

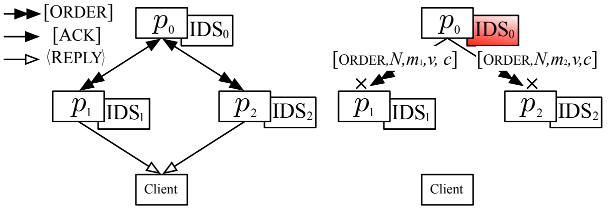


Figure 4. ByzID equipped with IDSs. The primary assigns sequence number to the request and sends [Order] message to the replicas. If the messages to different replicas are not consistent, the messages are blocked by the IDS equipped at the primary.

*Step 1: Client sends a request to the primary.* Client $c$ sends the primary $p_0$ a request message $\langle \text{Request}, o, T, c\rangle_{c,p_0}$, where $o$ is the requested operation, and $T$ is the timestamp.

*Step 2: Primary assigns a sequence number and sends an* [Order] *message to the backups.* When the primary receives a request from the client, it assigns a sequence number $N$ to the request and sends an $[\text{Order}, N, m, v, c]$ message to the backups, where $m$ is the request from the client, $v$ is the configuration number, and $c$ is the identity of the client.

*IDS details (at primary):* The IDS verifies the specifications mentioned in §III. Each time the specifications are violated, the IDS blocks the corresponding messages and generates an alert such that the primary will be reconfigured.

*Step 3: Replica receives an* [Order] *message, replies with an* [Ack] *message to the primary, executes the request, and sends a* ⟨Reply⟩ *to the client.* When replica $p_i$ receives an $[\text{Order}, N, m, v, c]$ message, it sends the primary an $[\text{Ack}, N, D(m), v, c]$ message with the same $N$, $m$, $v$, and $c$ as in the [Order] message. A backup $p_i$ accepts the [Order] message if the request $m$ is valid, its current configuration is $v$, and $N = N' + 1$, where $N'$ is the sequence number of its last accepted request. If the replica $p_i$ accepts the [Order] message, it executes operation $o$ in $m$ and sends the client a message $\langle \text{Reply}, c, r, T\rangle_{p_i,c}$, where $r$ is the execution result of operation $o$ and $T$ is the timestamp of request $m$. If $p_i$ receives an [Order] message with sequence number $N > N' + 1$, it stores the message in its log and waits for messages with sequence numbers between $N$ and $N'$. It executes the request with $N$ after it executes requests with sequence numbers between $N'$ and $N$.

*IDS details (at backups):* The IDS at a backup $p_i$ starts a timer when it observes an [Order] message. If $p_i$ does not send an [Ack] message in time, the IDS generates an alert.

*Step 4: Primary receives* [Ack] *messages from all backups and completes the request. Otherwise, it retransmits the* [Order] *message.* When the primary receives an $[\text{Ack}, N, D(m), v, c]$ message, it accepts the message if the fields $N$, $m$, $v$, and $c$ match those in the corresponding [Order] message. If the primary collects [Ack] messages from all the backups, it completes the request.

Our protocol is also compatible with common optimizations such as batching and pipelining. For pipelining, the primary can simply order a new request before the previous one is completed. However, to prevent the primary from sending [Order] messages too rapidly, we limit the number of outstanding [Order] messages to a threshold $\tau$. The primary sends an [Order] message with sequence number $N$ only if it completes requests with sequence numbers smaller than $N - \tau$.

The primary keeps track of the sequence number of the last completed request, $N_1$, and the sequence number of its most recently sent [Order] message, $N_2$. Obviously, we have that $N_2 \geq N_1$. When the primary sends an [Order] message for sequence number $N_1$, it starts a timer $\Delta_1$. If the primary does not receive [Ack] messages from all the backups before the timer expires, it retransmits the [Order] message to the backups from which [Ack] messages are missing. Otherwise, the primary cancels the timer and starts a new timer for the next request, if any.

An example is illustrated in Fig. 5, where the primary sends [Order] messages for requests with sequence numbers from $N_1$ to $N_2$. At $t_1$, the primary sends an [Order] message for $N_1$, and starts a timer $\Delta_1$. At $t_3$, it has collected [Ack] messages from all backups and cancels the timer. Since the primary has already completed the request with sequence number $N_1 + 1$ at $t_2$, it just starts a new timer for a request with $N_1 + 2$ at $t_3$.

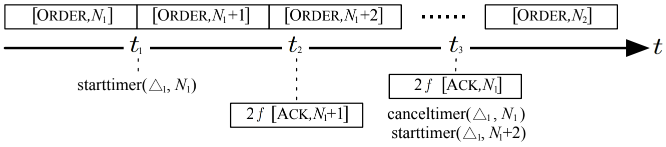


Figure 5. An example for Step 4.

*IDS details (at primary):* An alert is raised if the primary: (1) does not retransmit the [Order] message in time, or (2) it "retransmits" an inconsistent [Order] message. To accomplish these detections, the IDS starts a timer corresponding to the primary's $\Delta_1$ timer. If the primary receives enough [Ack] messages before $\Delta_1$ expires, the IDS cancels the timer. However, if the primary does not receive [Ack] messages from all backups before $\Delta_1$ expires, the IDS starts another timer, $\Delta_2$. If this timer expires, before the IDS observes a retransmitted [Order] message, an alert is raised. Meanwhile, the IDS keeps track of the sequence number of the last [Order] message sent by the primary, $N_3$. Each time the primary sends an [Order] message with sequence number smaller than $N_3$, the IDS checks if this retransmitted [Order] message matches an [Order] message in its log. If there is no match, an alert is raised.

*Step 5: Client collects* $f + 1$ *matching* ⟨Reply⟩ *messages to complete the request.* The client completes a request when it receives $f + 1$ matching reply messages.

**Checkpointing.** ByzID replicas store messages in their logs, which are truncated by the checkpoint protocol. Each replica maintains a stable checkpoint that captures both the protocol state and application level state. In addition, a replica also keeps some tentative checkpoints. A tentative checkpoint at a replica is proven stable only if all its previous checkpoints are stable and it collects certain message(s) in the checkpoint protocol to prove that the current state is correct.

We now briefly describe the ByzID checkpoint protocol. Every replica constructs a tentative checkpoint at regular intervals, e.g., every 128 requests. A backup replica $p_i$ sends a $[\textsc{Checkpoint}, N, d, i]$ message to the primary, where $N$ is the sequence number of last request whose execution is reflected in the checkpoint and $d$ is the digest of the state. The primary considers a checkpoint to be stable when it has collected $f$ matching $[\textsc{Checkpoint}]$ messages from different backups, and then sends a $[\textsc{StableCheckpoint}, N, d]$ message to the backups. The primary and $f$ backups prove that the checkpoint is stable. When a backup receives a $[\textsc{StableCheckpoint}]$, it considers the checkpoint stable. A replica can truncate its log by discarding messages with sequence numbers lower than $N$.

*IDS details:* The IDS audits the $[\textsc{Checkpoint}]$ messages from the backups. When it collects $f+1$ matching messages from the backups, it starts a timer. If the primary does not send the corresponding $[\textsc{StableCheckpoint}]$ message to all the backups before the timer expires, an alert is raised. IDS can also run a checkpoint protocol to prevent its own log from growing without bound. However, it delays discarding its stable checkpoints to help replica reconfiguration, as detailed in the following.

**Replica reconfiguration.** Reconfiguration is a technique for stopping the current RSM and restarting it with a new set of replicas [30]. We now describe ByzID's reconfiguration scheme. Recall that when any specifications of a replica are violated, the IDS generates an alert and triggers reconfiguration. If the IDS at the primary generates an alert, all the replicas are notified and stop accepting messages. The primary reconfiguration procedure operates *in-band* where all backups wait until the procedure completes. The backup reconfiguration procedure operates *out-of-band*. Namely, only the primary is notified with a backup replica IDS alert; the remaining replicas continue to run the protocol without having to wait for the procedure to complete. Assume in a configuration $v$ the set of replicas is $\Pi = \{p_0, p_1, \cdots, p_{n-1}\}$. We assume that after a reconfiguration, $p_i \in \Pi$ is replaced by $p_j \notin \Pi$. If $p_i$ is the primary, the configuration number becomes $v+1$ after reconfiguration. Clearly, replica $p_j$ is also equipped with an IDS component.

*Primary reconfiguration.* To initialize primary reconfiguration, a new primary $p_j$ sends a $[\textsc{ReconRequest}]$ message to *all* replicas in $\Pi$.[1] To respond, each replica $p_k$ sends $p_j$ a signed $\langle\textsc{Reconfigure}, v+1, N, \mathcal{C}, \mathcal{S}\rangle_{p_k}$ message, where $N$ is the sequence number of the last stable checkpoint, $\mathcal{C}$ is the last stable checkpoint, and $\mathcal{S}$ is a set of valid $[\textsc{Order}]$ messages accepted by $p_k$ with sequence numbers greater than $N$. When $p_j$ collects at least $f+1$ matching authenticated $\langle\textsc{Reconfigure}\rangle$ messages, it updates its state using the state snapshot in $\mathcal{C}$ and sends a $[\textsc{NewConfig}, v+1, \mathcal{V}, \mathcal{O}]$ to $\Pi \setminus p_i$, where $\mathcal{V}$ is a set of $f+1$ $\langle\textsc{Reconfigure}\rangle$ messages and $\mathcal{O}$ is a set of $[\textsc{Order}]$ messages computed as follows: first, the primary $p_j$ obtains the sequence number $min$ of the last stable checkpoint in $\mathcal{C}$ and the largest sequence number $max$ of the $[\textsc{Order}]$ message that has been accepted by at least one replica, which is obtained from $\mathcal{S}$.

The primary then creates an $[\textsc{Order}]$ message for each sequence number $N$ between $min$ and $max$. There are two cases: (1) If there is at least one request in the $\mathcal{S}$ field with sequence number $N$, $p_j$ uses the $[\textsc{Order}]$ message; (2) If there is no such request in $\mathcal{S}$, $p_j$ creates an $[\textsc{Order}]$ message with a NULL request. A backup accepts a $[\textsc{NewConfig}]$ message if the set of $\langle\textsc{Reconfigure}\rangle$ messages in $\mathcal{V}$ are valid and $\mathcal{O}$ is correct. The correctness of $\mathcal{O}$ can be verified through a similar computation as the one used by the primary to create $\mathcal{O}$. It then enters configuration $v+1$.

*Backup reconfiguration.* A new backup replica $p_j$ sends a message $[\textsc{ReconRequest}]$ to the primary. The primary then responds a message $[\textsc{Reconfigure}, v+1, N, \mathcal{C}, \mathcal{S}]$ to $p_j$, where $N$ is the sequence number of the primary's last stable checkpoint, $\mathcal{C}$ is its last stable checkpoint, and $\mathcal{S}$ is a set of valid $[\textsc{Order}]$ messages sent by the primary with sequence number greater than its last stable checkpoint. When $p_j$ receives the $[\textsc{Reconfigure}]$ message, it updates its state by the state snapshot in $\mathcal{C}$, and then processes the $[\textsc{Order}]$ messages in $\mathcal{S}$.

*IDS details:* The IDS coupled with $p_j$ obtains its own state from the IDS of replica $p_i$.

During primary reconfiguration, the IDS at new primary $p_j$ monitors all the $\langle\textsc{Reconfigure}\rangle$ messages from all the replicas in $\Pi$ and checks if they match its own IDS log. If the checkpoint is not valid or the $[\textsc{Order}]$ messages in $\mathcal{S}$ are not the same as the messages sent by $p_i$, the IDS blocks the $\langle\textsc{Reconfigure}\rangle$ message. Clearly it is with the aid of IDS that primary reconfiguration becomes simpler.

During backup reconfiguration, the IDS at the primary checks if the primary sends the backup a $[\textsc{Reconfigure}]$ message with the matching $\mathcal{C}$ and $\mathcal{S}$ as in its IDS log. This ensures that replicas receive consistent state.

**Correctness.** We now prove ByzID is both safe and live.

**Theorem 1 (Safety).** If no more than $f$ replicas are faulty, non-faulty replicas agree on a total order on client requests.

[1] Note that $p_j$ should also send the message to the current primary, because it might still be correct.

**Proof:** We first show that ByzID is safe within a configuration and then show that the ordering and replica reconfiguration protocols together ensure safety across configurations.

*Within a configuration.* We prove that if a request $m$ commits at a correct replica $p_i$ and a request $m'$ commits at a correct replica $p_j$ with the same sequence number $N$ within a configuration, it holds that $m$ equals $m'$. We distinguish three cases: (1) either $p_i$ or $p_j$ is the primary; (2) neither $p_i$ nor $p_j$ is the primary, and neither has been reconfigured; (3) neither $p_i$ nor $p_j$ is the primary, and at least one of the two replicas has been reconfigured. We briefly prove the (most involved) case (3). During a backup reconfiguration, its state can be recovered by communicating with the primary with the aid of the IDS. Thereafter, the new reconfigured replica is indistinguishable from the correct replica without having been reconfigured. If $m$ with sequence number $N$ commits at a correct replica $p_i$, it holds that $p_i$ receives an [Order] message with $m$ and $N$ from the primary (either due to the ordering or backup reconfiguration protocols), since we assume there are no channel injections. Similarly, $p_j$ receives an [Order] message with $m'$ and $N$ from the primary. Therefore, it must be that $m = m'$, since otherwise it violates the consistency specification enforced by the IDS. The total order thus follows from the fact that that the requests commit at the replicas in sequence-number order.

*Across configurations.* We prove that if $m$ with sequence number $N$ is executed by a correct replica $p_i$ in configuration $v$ and $m'$ with sequence number $N$ is executed by a correct replica $p_j$ in configuration $v'$, it holds that $m$ equals $m'$. We assume w.l.o.g. that $v < v'$. Recall that if a backup is reconfigured, the state of the new replica is consistent with other backups. Thus, we do not bother differentiating reconfigured replicas from correct ones and focus on the case where $p_i$ and $p_j$ are both backups.

The proof proceeds as follows. If $m$ with sequence number $N$ is executed by $p_i$ in configuration $v$, the primary must have sent consistent [Order] messages for $m$ to all the backups. Alternatively, if $m'$ with $N$ is executed by $p_j$ in configuration $v'$, the primary in $v'$ sends consistent [Order] messages for $m'$ to all the backups. This implies that the primary in $v'$ receives ⟨Reconfigure⟩ messages from at least $f + 1$ replicas with $m'$ and $N$, at least one of which is correct. Inductively, we can prove that there must exist an intermediate configuration $v_1$ where the corresponding primary sent an [Order] message with $m$ and $N$ and an [Order] message with $m'$ and $N$. Due to the consistency specification enforced by the IDS, it holds that $m$ equals $m'$. The total order of client requests thus follows from the fact that requests are executed in sequence-number order. ■

**Theorem 2 (Liveness).** If no more than $f$ replicas are faulty and a non-faulty replica receives a request from a correct client, the request will eventually be executed by all non-faulty replicas. Therefore clients eventually receive replies to their requests.

**Proof:** We begin by showing that if a correct replica accepts an [Order] message with request $m$ and $N$, all the correct replicas eventually accept the same [Order] message.

There are two types of timers used for IDSs: (1) the timers to monitor the timely actions for the replicas' local operations, and (2) the timer in the primary IDS to wait for the [Ack] message. The first type of timers are initialized and tuned by the anomaly-based IDS. For the [Ack] timer, the IDS at the primary can double the timeouts when less than $f + 1$ replicas send the [Ack] messages on time. Alternatively, the primary retransmits the [Order] message but starts a timer with the same value. If the retransmission occurs too frequently, the timer can be doubled.

We now show that if a correct replica $p_i$ accepts an [Order] message with request $m$ and $N$, all the correct replicas accept the same [Order] message. According to the protocol and the consistency rule, if $p_i$ receives an [Order] message with $m$ and $N$, the primary sends the same [Order] message to all backups. The primary completes the request when it collects $n - 1$ matching [Ack] messages. If a faulty backup does not send the [Ack] message, the IDS raises an alert and the faulty replica is reconfigured. The [Order] message may be dropped by the fair-loss channel, in which case the primary will not receive the [Ack] message on time. The primary retransmits the [Order] messages until the backups receive it. If the primary does not do so, it will be detected by the IDS and be reconfigured. Then the new primary will send (and probably need to retransmit) the [Order] messages until the backups receive it. Therefore, all correct replicas will receive the [Order] message eventually. The "no gap" specification is also vital to achieve liveness. If the specification is not enforced, then according to our protocol, backups will have to wait for the [Order] messages with incremental sequence numbers to execute. Since there are at least $f+1$ correct replicas, the client always receives a majority of $f$ matching replies from the replicas, as long as the correct replicas reach an agreement. If it does not receive enough replies on time, it simply retransmits the request and doubles its own timer. ■

### *B. The ByzID-W Protocol*

When deploying ByzID in a WAN environment, several adjustments to the core protocol are needed. First, there must be a complete graph network between the replicas. Second, since the IDS cannot be relied upon to prevent message injection on the WAN links, we use authenticated links between the replicas. That is, [Order] messages are authenticated using *deterministic* signatures, allowing the IDS to efficiently support retransmissions of previously signed order messages.

## V. ByzID Implementation with Bro

As a proof of concept, we have implemented our Byzantine failure detector for ByzID using the Bro [38] IDS.

Bro detects intrusions by hooking into the kernel using `libpcap` [36], parsing network traffic to extract semantics, and then executing event analyzers. To support ByzID, we have adapted Bro as shown in Fig. 6. First, we have built a new ByzID parser to process messages and generate ByzID-specific events. These events are then delivered to their event handler, based on their type. The IDS specifications for ByzID is implemented as scripts written in the Bro language. The policy interpreter executes the scripts to produce real-time notification of analysis results, including alerts describing violation of BFT protocol specifications.

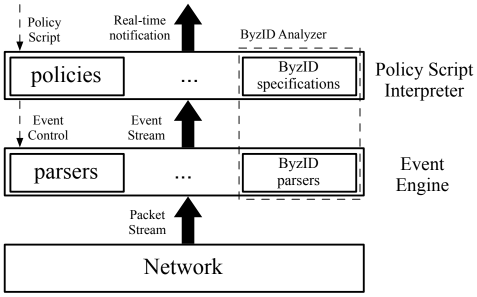


Figure 6. ByzID analyzer based on Bro.

**ByzID parser.** The network packet parser decodes byte streams into meaningful data fields. We use *binpac* [37], a high-level language for describing protocol parsers to automatically translate the packets into a C++ representation, which can be used by both Bro and ByzID. We represent the syntax of ByzID messages by *binpac* scripts. During parsing, the parser first extracts the message tag, sequence number, and configuration number. The messages unrelated to the specifications are passed during parsing; other messages are delivered to their corresponding event handler.

**Event handler.** Event handlers analyze network events generated by the ByzID parser. The event handler provides an interface between the ByzID parser and the policy script interpreter. Each message type is associated with a separate event handler, and only messages with the appropriate tags are delivered to that handler. The events are then passed to the policy script interpreter to validate that the events do not violate the specifications.

**ByzID specifications.** The policy script contains the specifications of the ByzID protocol. Once event streams are generated by the event handler, it performs the inter-packet validation. The policy script interpreter maintains state from the parsed network packets, from which the incoming packets are further correlated and analyzed. Messages that violate the specifications are blocked and an alert is raised.

## VI. Performance Evaluation

In this section we evaluate the performance of ByzID by comparing it with three well-known BFT protocols—PBFT [6], Zyzzyva [27], Aliph [21], and an implementation of the crash fault tolerant protocol—Paxos [29]. All of the protocol implementations are based on Castro et al.'s implementation of PBFT. The main conclusion that we can draw from our evaluation is that ByzID's performance is slightly worse that Paxos due to the overheads of the IDS and cryptographic operations. Considering the similarity in message flow between ByzID and Paxos, this is unsurprising. However, ByzID's performance is generally better than the other BFT protocols in our comparison.

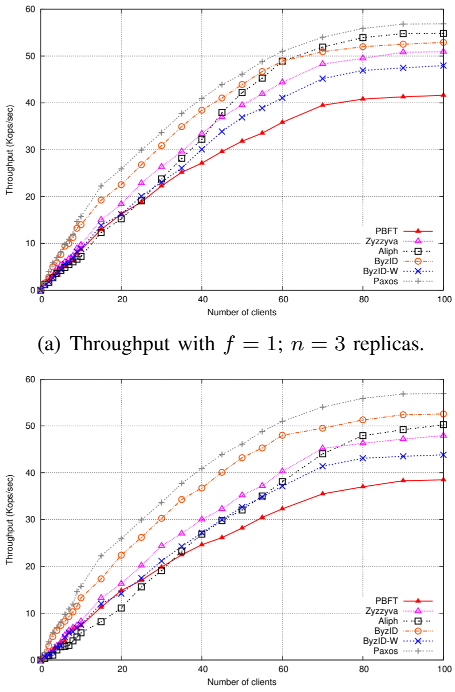
We do not compare ByzID with other BFT protocols that depend on trusted hardware, such as A2M [7], TrInc [32], and MinBFT [40], since we do not have access to the relevant hardware platforms. However, based on published performance data for these protocols, they generally do not offer higher throughput and lower latency than Aliph [24, 40].[2] We note that, the IDS component of ByzID could be implemented efficiently in trusted hardware as well.

We evaluated throughput, latency, and scalability using the $x/y$ micro-benchmarks by Castro and Liskov [6]. In these benchmarks, clients send $x$ kB requests and receive $y$ kB replies. Clients issue requests in a *closed-loop*, i.e., a client issues a new request only after having received the reply to its previous request. All protocols in our comparison implement batching of concurrent requests to reduce cryptographic and communication overheads. All experiments were carried out on Deterlab, utilizing a cluster of up to 56 identical machines. Each machine is equipped with a 3 GHz Xeon processor and 2 GB of RAM. They run Linux 2.6.12 and are connected through a 100 Mbps switched LAN.

**Throughput.** We first examined the throughput of both ByzID and ByzID-W under contention and compared them with PBFT, Zyzzyva, Aliph, and Paxos. Fig. 7 shows the throughput for the $0/0$ benchmark when $f = 1$ and $f = 3$, as the number of clients varies. Our results show that ByzID outperforms other BFT protocols in most cases and is only marginally slower than Paxos. As observed in Fig. 7(a), ByzID consistently outperforms Zyzzyva, which achieves better performance than ByzID-W and PBFT. Since ByzID-W uses signatures, it achieves lower throughput than Zyzzyva. The reason ByzID-W has better performance than PBFT is due to the reduction of communication rounds. Aliph outperforms Zyzzyva and ByzID when the number of clients is big enough, mainly because it exploits the pipelined execution of client requests. But as shown in Fig. 7(b), ByzID consistently outperforms other BFT protocols when $f = 3$. For both $f = 1$ and $f = 3$, ByzID achieves an average throughput degradation of $5\%$ with respect to Paxos. This overhead is mainly due to the cryptographic operations and IDS analysis. Similar results are observed in other benchmarks.

**Latency.** We have also compared the latency of the protocols without contention where a *single* client issues requests in a

[2] A2M and TrInc must use signatures due to the impossibility result of [8].



(a) Throughput with $f = 1$; $n = 3$ replicas.

(b) Throughput with $f = 3$; $n = 7$ replicas.

Figure 7. Throughput for the 0/0 benchmark as the number of clients varies. This and subsequent graphs are best viewed in color.

close-loop. The results for the 0/0, 0/4, 4/0, and 4/4 benchmarks with $f = 1$ are depicted in Fig. 8. We observe that ByzID outperforms other protocols except Paxos. However, the difference between ByzID and Paxos is less than 0.1 ms. The reason ByzID generally has low latency is that it only has three one-way message latencies in the fault-free case.

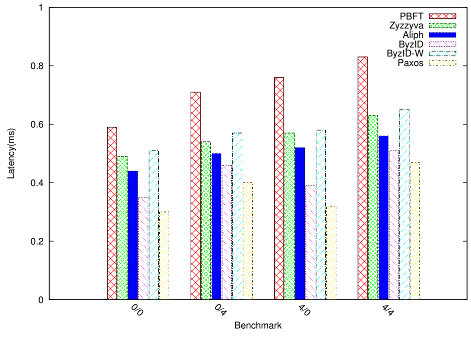


Figure 8. Latency for the 0/0, 0/4, 4/0, and 4/4 benchmarks.

Table I
THROUGHPUT IMPROVEMENT OF BYZID OVER OTHER BFT PROTOCOLS. VALUES IN (RED) REPRESENT NEGATIVE IMPROVEMENT.

| Clients | Protocol | $f = 1$ | $f = 2$ | $f = 3$ | $f = 4$ | $f = 5$ |
|---|---|---|---|---|---|---|
| 25 | PBFT | 42.37% | 45.71% | 46.80% | 49.14% | 51.37% |
| 25 | Zyzzyva | 17.19% | 19.49% | 25.49% | 26.07% | 27.72% |
| 25 | Aliph | 40.42% | 47.84% | 67.56% | 73.46% | 76.98% |
| peak | PBFT | 27.15% | 32.57% | 36.59% | 41.82% | 43.90% |
| peak | Zyzzyva | 3.92% | 8.43% | 9.68% | 12.25% | 11.08% |
| peak | Aliph | (3.48%) | (1.24%) | 4.57% | 7.71% | 8.92% |

Table II
THROUGHPUT DEGRADATION WHEN $f$ INCREASES.

| Clients | Protocol | $f = 2$ | $f = 3$ | $f = 4$ | $f = 5$ |
|---|---|---|---|---|---|
| 25 | PBFT | 3.82% | 9.40% | 10.20% | 15.04% |
| 25 | Zyzzyva | 3.45% | 8.66% | 12.50% | 16.80% |
| 25 | Aliph | 6.50% | 18.30% | 28.00% | 35.60% |
| 25 | ByzID | 1.56% | 2.20% | 5.93% | 9.67% |
| peak | PBFT | 4.25% | 7.54% | 13.88% | 17.85% |
| peak | Zyzzyva | 4.32% | 5.89% | 11.07% | 13.02% |
| peak | Aliph | 4.84% | 8.33% | 13.93% | 17.61% |
| peak | ByzID | 1.70% | 2.80% | 3.94% | 7.02% |

**Scalability.** To understand the scalability properties of ByzID, we increase $f$ for all protocols and compare their throughput. All experiments are carried out using the 0/0 benchmark. Table I compares the throughput of ByzID with three other BFT protocols, and Table II shows the throughput degradation for all four BFT protocols as $f$ increases. We observe in Table I that the throughput improvement for ByzID over the other BFT protocols consistently increases as $f$ grows. Table II shows that ByzID's own throughput has the lowest degradation rate among all four BFT protocols. For instance, ByzID's peak throughput is only reduced by 7.02% as $f$ increases to 5 (i.e., when $n = 11$). These results clearly show that ByzID has much better scaling properties than the other BFT protocols.

## VII. FAILURES, ATTACKS, AND DEFENSES

The fact that a BFT protocol is live does not mean that the protocol is efficient. It is therefore important to analyze the performance and resilience of the protocol in face of replica failures and malicious attacks. In this section, we discuss how well ByzID withstands a variety of Byzantine failures, and also demonstrate some key design principles underlying our design. We distinguish the replica failures due to system crashes, software bugs, and hardware failures from those attacks induced by dedicated adversaries that aim to subvert the system or deliberately reduce the system performance. Note that such a distinction is neither strict nor accurate. However, one can view the two types of evaluation as different perspectives to analyze the performance of ByzID.

### A. Performance During Failures

We study the performance of the different BFT protocols for $f = 1$ under high concurrency, and in the presence of one backup failure.[3] To avoid clutter in the plot, PBFT, Zyzzyva, and ByzID experience a failure at $t = 1.5$ s, while for Aliph at $t = 2.0$ s. In case of failures, we require Aliph to switch

[3] The situation falls into our generalized definition of a *normal* case.

between Chain and a backup abstract (e.g., PBFT) since its Quorum abstract does not work under contention. We set the configuration parameter $k$ as $2^i$, i.e., Aliph switches to Chain after executing $k = 2^i$ requests using its backup abstract.[4]

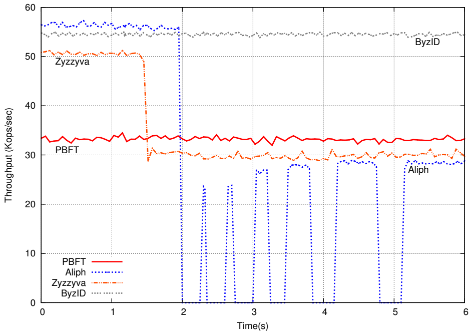


Figure 9. Throughput after failure at 1.5 s (2.0 s for Aliph).

As shown in Fig. 9, neither PBFT or ByzID experience any throughput degradation after a failure injection. This is mainly due to their broadcast nature. However, the performance of Zyzzyva after a failure is reduced by about 40% because it switches to its slower backup protocol. Though Aliph has a slightly higher throughput than ByzID prior to the failure, its throughput reduces sharply upon failure, dropping below that of the PBFT baseline. Aliph periodically switches between Chain and PBFT after the failure, which explains the throughput gaps in Aliph. Since $k$ increases exponentially for every protocol switch, it stays in the backup protocol for an increasing period of time.

### B. Performance under Active Attacks

**Too-Many-Server Compromises.** Like other BFT protocols relying on trusted components, ByzID can mask at most $f$ failures using $2f + 1$ replicas. With passage of time however, the number of faulty replicas might exceed $f$. This can happen if a dedicated attacker is able to compromise replicas one by one, and only asks them to manifest faulty behavior when a sufficient number of replicas have been compromised. If these compromises can go undetected by the IDSs, ByzID cannot defend against such an attack. However, ByzID uses a proactive approach to prevent too many servers from being corrupted simultaneously. For other attacks, it is clear that our approach provides robustness.

**Fairness Attacks.** Fairness usually refers to the ability of every component to take a step infinitely often. This is inappropriate for time-critical applications such as in real-time transactional databases. For instance, in a stock system, a faulty primary might collude with a client to help the latter gain unjust advantages. Our IDS aided ByzID can achieve perfect fairness—ensuring that requests are executed in a "first come, first served" manner. Aardvark [9] can achieve a certain level of fairness, but does not achieve perfect fairness and is not suitable for time-critical applications. In contrast, ByzID achieves perfect fairness by leveraging IDSs, and has a significant performance advantage over Aardvark.

**Flooding Attacks.** We describe a flooding attack as one in which faulty replicas might continuously send "meaningful but repeating" or "meaningless" messages to other replicas. The goal of such attacks is to occupy the computational resources that are supposed to execute the pre-determined operations. This type of attacks is particularly harmful, as verifying the correctness of the cryptographic operations is relatively expensive. Such attacks can largely impact the performance of all the traditional BFT protocols. We take a number of countermeasures to defend against such attacks. First, we do not adopt the traditional pairwise channels between every replica pair. Instead, the primary forms the root of a tree, with backup replicas as leafs directly connected to the root. In particular, backups do not communicate with each other to prevent backups from flooding one another. Second, we use the IDSs to prevent the primary from flooding messages other than the [ORDER] messages to backups, and prevent the backups from flooding messages other than [ACK] messages to the primary. Finally, we also use IDSs at backups to determine if received messages are from clients or the primary. A backup IDS simply filters all the incoming messages from the clients.

**Timing Attacks ("Slow" Replica Attacks).** We define *timing failures*, as the situation when replicas produce correct results but deliver them outside of a specified time window. One or more compromised replicas might delay several operations to degrade the performance of the system. For example, the primary can deliberately delay the sending of ordering messages in response to client requests. It is usually hard to distinguish such faulty replicas from slow replicas. It is also hard to distinguish if the failures are due to faulty replicas or channel failures. We use IDSs to monitor such kind of attacks. In particular, the timers can be setup by the anomaly-based intrusion detection. IDSs only monitor the node processing delays, not channel failures. Therefore, the monitoring can be accurate. Once the timer exceeds the prescribed value, an IDS will trigger an alert.

### C. IDS Crashes

The IDSs themselves are not resilient to crashes. So what if the IDSs crash? One distinguishing advantage of ByzID is that it can still achieve safety (and liveness) even if all the IDSs crash. Indeed, ByzID has the following two properties that other BFT protocols relying on trusted components do not have: (1) Even if all IDSs crash, as long as the primary is correct, safety is never compromised. (2) Even if all IDSs crash, as long as all the replicas are correct, both safety and liveness are still achieved. Clearly, ByzID cannot provide the same resilience against attacks without the IDSs.

[4] Another option is to set $k$ as a constant [21], but in our experience its performance during failure is inferior to using $k = 2^i$.

## VIII. NFS Use Case

This section describes our evaluation of a BFT-NFS service implemented using PBFT [6], Zyzzyva [27], and ByzID, respectively. The BFT-NFS service exports a file system, which can then be mounted on a client machine. The replication library and the NFS daemon are called when the replicas receive client requests. After replicas process the client requests, replies are sent to the clients. The NFS daemon is implemented using a fixed-size memory-mapped file. We use the Bonnie++ benchmark [10] to compare the three implementations with NFS-std, an unreplicated NFS V3 implementation, using an I/O intensive workload. We evaluate the Bonnie++ benchmark with sequential input (including per-character and block file reading), sequential output (including per-character and block file writing), and the following directory operations (DirOps): (1) create files in numeric order; (2) stat() files in the same order; (3) delete them in the same order; (4) create files in an order that will appear random to the file system; (5) stat() random files; (6) delete the files in random order. We measure the average latency when a single client runs the benchmark, as shown in Fig. 10. The bar chart includes both the fault-free case and the normal case where a backup failure occurs at time zero. We observe that in both cases, ByzID implementation outperforms both PBFT and Zyzzyva, and is only marginally slower than NFS-std.

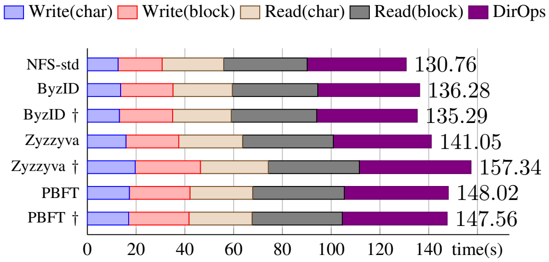


Figure 10. NFS evaluation with the Bonnie++ benchmark. The † symbol marks experiments with failure.

## IX. Related Work

Equivocation refers to the behavior of an adversarial component that lies to other components in different ways. It was shown that the problem (and any consensus problem) cannot be solved if more than one third of its processes are faulty. Fitzi and Maurer [20] showed that with the existence of a "two-cast channel" (i.e., broadcast channel among three players), Byzantine agreement is achievable if and only if the number of faulty processes is less than a half. The result was later extended [11] for general multicast channels.

Beginning with Correia *et al.* [13], a number of BFT approaches relying on (small) trusted components to prevent equivocation and circumvent the one-third bound have been developed, including A2M [7], TrInc [32], MinBFT and MinZyzzyva [40], and CheapBFT [24]. All of these require only $2f+1$ replicas to tolerate $f$ failures, and they have to rely on signatures [8]. van Renesse, Ho, and Schiper [39] also proposed Shuttle, that can tolerate $f$ failures among $2f+1$ replicas. The protocol relies on a trusted and Byzantine-fault resilient server to achieve liveness. ByzID also falls into the category of using trusted components, but we deploy an IDS that is not only more powerful but also simpler. Our approach achieves better efficiency than the prior BFT protocols (with or without trusted components) both during failures and in the absence of failures.

Another approach in BFT research has been the study of how to improve the resilience under active attacks, such as Aardvark [9] and Prime [2]. The methods they use are different from ours in that they do not rely on trusted components. However, since they are both based on PBFT, ByzID outperforms them in terms of performance, and also better handles fairness and flooding attacks. BP Fast [35] relies on $3f+1$ non-Byzantine servers, but can protect itself against denial of service attacks waged by clients. *h*BFT [17] moves jobs to the clients while tolerating faulty clients.

Chandra and Toueg [14] introduced the notion of unreliable failure detectors, which could be used to solve consensus in the presence of crash failures. In their design, the failure detector outputs the identity of processes suspected to have crashed. In contrast to crash failures, Byzantine failures are not context-free, and thus it is impossible to define a general failure detector in a Byzantine environment, independently of the algorithm. Some previous work [3, 16, 25, 34] has extended the failure detector notion to cover a wider range of failures, For example, the muteness failure detector interacts with the algorithm of a remote process to detect if the remote process has turned mute. Our IDS, together with the primary, also serves as a Byzantine failure detector specifically designed for our ByzID protocol. However, these existing solutions focus on solving the consensus problem, whereas we provide state machine replication.

PeerReview [22] builds a system that replicas review and report the failure of other replicas. It ensures that faulty behavior is detected and no correct node is observed to be faulty through the use of secure logging and auditing techniques. Reputation systems such as EigenTrust [23] can also be used to detect a family of Byzantine faults but they typically detect only repeated misbehavior.

## X. Conclusion

We have shown a viable method to establish an efficient and robust BFT protocol by leveraging specification-based intrusion detection. Our protocol leverages the key assumption of a trusted reference monitor, but the approach we use is different from other BFT approaches relying on trusted components in that we apply a simple IDS monitoring and filtering technique. The reasons we use intrusion detection techniques can be summarized as follows: (1) The IDS for our BFT protocol is very simple in both code size and applicability—no heavy operations or cryptographic operations involved, and therefore relatively easy to implement as a reference monitor. (2) Although IDSs themselves are not resilient to crashes, we can still achieve a form of safety even if all IDSs fail. (3) Equipped with IDSs, our BFT protocol is more robust against a number of important attacks. (4) Our IDS-aided ByzID protocol is also more efficient than other BFT protocols. Indeed, our experimental evaluation shows that ByzID is only marginally slower than Paxos.

## XI. Acknowledgements

This research is based on work supported in part by the National Science Foundation under Grants Number CCF-1018871, CNS-0904380, and CNS-1228828. Sisi Duan and Haibin Zhang's work was also supported in part by a Leiv Eiriksson Mobility Grant from RCN. Hein Meling's contributions were supported by the Tidal News project under grant no. 201406 from RCN. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the employers or sponsors of this work.

## References

[1] P. Alsberg, and J. Day. A principle for resilient sharing of distributed resources. *ICSE*, 1976.

[2] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dep. Sec. Comp.*, 8(4):564–577, 2011.

[3] R. Baldoni, J. Helary, and M. Raynal. From crash fault-tolerance to arbitrary-fault tolerance: towards a modular approach. *DSN*, 2000.

[4] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. S. Mullender (ed.) *Distributed systems, 2nd ed*, 1993.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI*, 2006.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. *OSDI*, 1999.

[7] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. *SOSP*, 2007.

[8] A. Clement, F. Junqueira, A. Kate, R. Rodrigues. On the (limited) power of non-equivocation. *PODC*, 2012.

[9] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. *NSDI*, 2009.

[10] R. Coker. www.coker.com.au/bonnie++.

[11] J. Considine, M. Fitzi, M. Franklin, L. Levin, U. Maurer, and D. Metcalf. Byzantine agreement given partial broadcast. *J. Cryptology*, 18:191–217, 2005.

[12] J. Corbett *et al*. Spanner: Google's globally-distributed database. *OSDI*, 2012.

[13] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. *SRDS*, 2004.

[14] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, 1996.

[15] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, 1987.

[16] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. *EDCC*, 1999.

[17] S. Duan, S. Peisert, and K. Levitt. hBFT: speculative Byzantine fault tolerance with minimum cost. *IEEE Trans. Dep. Sec. Comp.*, 2014.

[18] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM* 35(2):288–323, 1988.

[19] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM* 32(2):374–382, 1985.

[20] M. Fitzi and U. Maurer. From partial consistency to global broadcast. *STOC*, 2000.

[21] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The next 700 BFT protocols. *EuroSys*, 2010.

[22] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *SOSP*, 2007.

[23] S. D. Kanvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in p2p networks. *WWW*, 2003.

[24] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. *EuroSys*, 2012.

[25] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. *Comput. J.* 46(1):16–35, 2003.

[26] C. Ko, M. Ruschitzka, and K. N. Levitt. Execution monitoring of security-critical programs in distributed systems: a specification-based approach. *S&P*, 1997.

[27] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *SOSP*, 2007.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[29] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[30] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News* 41(1):63–73, 2010.

[31] T. E. Levin, C. E. Irvine, and T. D. Nguyen. Least Privilege in Separation Kernels. *SECRYPT*, 2006.

[32] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda.. TrInc: Small trusted hardware for large distributed systems. *NSDI*, 2009.

[33] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. *S&P*, 1988.

[34] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. *CSFW*, 1997.

[35] H. Meling, K. Marzullo and A. Mei. When You Don't Trust Clients: Byzantine Proposer Fast Paxos. *ICDCS*, 2012.

[36] L. MartinGarcia. http://www.tcpdump.org

[37] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. *IMC*, 2006.

[38] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435-2463, 1999.

[39] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. *OPODIS*, 2012.

[40] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault tolerance. *IEEE Trans. Comput.*, 62(1):16–30, 2013.