NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering

Guangda Sun
National University of Singapore

Mingliang Jiang
National University of Singapore

Xin Zhe Khooi
National University of Singapore

Yunfan Li
National University of Singapore

Jialin Li
National University of Singapore

ABSTRACT

Mission critical systems deployed in data centers today are facing more sophisticated failures. Byzantine fault-tolerant (BFT) protocols are capable of masking these types of failures, but are rarely deployed due to their performance cost and complexity. In this work, we propose a new approach to designing high performance BFT protocols in data centers. By re-examining the ordering responsibility between the network and the BFT protocol, we advocate a new abstraction offered by the data center network infrastructure. Concretely, we design a new authenticated ordered multicast primitive (AOM) that provides transferable authentication and non-equivocation guarantees. Feasibility of the design is demonstrated by two hardware implementations of AOM– one using HMAC and the other using public key cryptography for authentication – on new-generation programmable switches. We then co-design a new BFT protocol, NeoBFT, that leverages the guarantees of AOM to eliminate cross-replica coordination and authentication in the common case. Evaluation results show that NeoBFT outperforms state-of-the-art protocols on both latency and throughput metrics by a wide margin, demonstrating the benefit of our new network ordering abstraction for BFT systems.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; • **Networks** → **In-network processing**; • **Security and privacy** → **Distributed systems security**.

KEYWORDS

State Machine Replication, Byzantine-Fault Tolerance, In-Network Ordering, Programmable Networks

ACM Reference Format:
Guangda Sun, Mingliang Jiang, Xin Zhe Khooi, Yunfan Li, and Jialin Li. 2023. NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3603269.3604874

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604874

1 INTRODUCTION

Online services today are commonly deployed in large data centers and rely on fault-tolerance protocols to provide high availability in the presence of failures. An important class of fault-tolerance protocols is state machine replication (SMR). SMR protocols [32, 35, 40, 44, 54] have long been deployed in production systems [15, 17, 21, 29] to ensure a set of distributed nodes behaves like a single, always available state machine, despite failures of individual machines. These protocols, however, can only tolerate node crash failures. In reality, systems running in data centers are facing more sophisticated failures. This is particularly relevant today with the growing adoption of permissioned blockchain systems [2, 46, 47] in data centers for applications such as trading [3, 51]. Major cloud providers have also introduced infrastructure support for blockchain-based platforms [5, 6, 26], highlighting their increasing demand. These systems require tolerance to adversarial nodes and attacks while maintaining low latency and high transaction throughput. Recent work [46] has demonstrated that fault tolerance protocols are becoming their main performance bottleneck.

Numerous Byzantine fault-tolerant (BFT) protocols [16, 20, 34, 43, 55, 58, 60] have been proposed to handle arbitrary node failures. Their strong failure models, however, come with significant performance implications. BFT protocols typically incur rounds of replica communication coupled with expensive cryptographic operations, resulting in low system throughput and high request latency. To obtain higher throughput, many BFT protocols resort to heavy request batching, which leads to long end-to-end decision latency – often in the range of tens of milliseconds. Unfortunately, such latency overheads are prohibitive for modern data center applications with strict service-level objectives (SLOs). Speculative BFT protocols, such as Zyzzyva [34], offer improved commitment latency. However, even a single faulty replica would negate the performance benefit of these latency-optimized protocols.

In this paper, we introduce a new approach to building high-performance BFT protocols in data centers. We observe that traditional BFT protocols are designed with minimum assumptions about the underlying network, assuming only best-effort message delivery. As a result of this weak network model, application-level protocols are responsible for enforcing all correctness properties, such as total ordering, durability, and authentication. Our key insight is that by strengthening the network model to provide *ordered message delivery*, the complexity and performance overhead of BFT protocols can be reduced. Prior research [38, 39] has demonstrated the promises of in-network sequencing for crash fault-tolerant systems. However, these approaches fall short in the presence of

Byzantine failures. For instance, faulty nodes can disseminate conflicting message orders, and the network sequencer may equivocate by assigning different sequence numbers to each replica.

In this work, we propose a new network-level primitive, authenticated ordered multicast (AOM), that addresses the above challenges. AOM ensures that correct receivers always deliver multicast messages in the same order, even in the presence of Byzantine participants. A key property offered by AOM is *transferable in-network authentication*: Receivers can verify that a multicast message is properly delivered by AOM, and they can prove the authenticity of the message to other receivers in the system. We additionally propose a mixed failure model [41, 52] where possible faulty behaviors of the network infrastructure are considered separately. For deployments that *trust* the network infrastructure, AOM assumes a crash failure model for the network. This slightly weaker model allows AOM to provide ordering guarantees with minimum network-level overhead. For systems that require tolerance of Byzantine network devices, AOM employs a simple cross-receiver communication round to handle equivocating sequencers.

We demonstrate the feasibility of AOM by implementing it on commercially available programmable switches [30]. The switch data plane performs both *sequencing* and *authentication* for AOM messages. While implementing packet sequencing is relatively simple, generating secure authentication codes poses major challenges given the switch's limited resources and computational constraints. We propose two designs of in-switch message authentication, each with its own set of trade-offs between switch resource utilization, performance, and scalability. The first variant implements SipHash-based [4] message authentication code (HMAC) vectors directly on the switch ASICs. The second variant generates signatures using public-key cryptography. Given the hardware constraints, a direct implementation of cryptographic algorithms such as RSA [49] and ECDSA [31] remains infeasible on these switches. To overcome this limitation, we introduce a novel heterogeneous switch architecture that couples FPGA-based cryptographic coprocessors with the switch pipelines. This design enables efficient in-network processing and signing of AOM messages, scales to larger AOM groups, and minimizes the hardware resource requirements of the switch data plane.

Leveraging the strong properties of AOM, we co-design a new BFT protocol, NeoBFT. In the common case, NeoBFT replicas rely on the ordering guarantee of AOM to commit client requests in a *single round trip*, eliminating all cross-replica communication and authentication. Furthermore, even in the presence of (up to f) faulty replicas, NeoBFT stays in this fast path protocol while meeting the theoretical minimum replication factor ($3f + 1$). In the event of network failures, we design efficient protocols to handle message drops and faulty switch sequencers while preserving the protocol's correctness. By evaluating against state-of-the-art BFT protocols, we show that NeoBFT can improve both protocol throughput by up to 4.1× and end-to-end latency by 42×. Additionally, NeoBFT maintains its high performance in the presence of Byzantine participants, scales to 100 replicas, and is robust to network anomalies and sequencer failures.

2 BACKGROUND

In this section, we give an overview of state-of-the-arts BFT protocols. We then review recent proposals that use in-network ordering to accelerate SMR systems. Lastly, we specify the targeted deployment model of our work.

2.1 State-of-the-Art BFT Protocols

There has been a long line of work on BFT SMR protocols. We present a summary of the state-of-the-art BFT protocols and their key properties in Table 1. PBFT [16] is the first practical BFT protocol that tolerates up to f Byzantine nodes using $3f + 1$ replicas, which has been shown to be the theoretical lower bound [14]. In PBFT, client requests are committed in *five* message delays. First, the client sends a request to a primary replica, who then sequences and forwards the request to the backup replicas. Next, the backup replicas authenticate the requests and broadcast their acceptance. Once a replica receives a quorum of acceptance, it broadcasts a commit decision. Finally, replicas execute the request and reply to the client after collecting quorum commit decisions. As replicas exchange messages in an all-to-all manner, each replica processes $O(N)$ messages, which results in an authenticator complexity of $O(N^2)$.

Zyzzyva [34] employs speculative execution of client requests to reduce communication overhead. The protocol offers two execution paths: a fast path that completes in three message delays when clients receiving matching replies from *all* replicas, and a slow path that requires at least five message delays. The primary replica in Zyzzyva still sends signed messages to all backup replicas ($O(N)$). But with all-to-all communication eliminated, the authenticator complexity is reduced to $O(N)$.

Rather than relying on the clients to collect authenticators, SBFT [27] uses a round-robin message collector among all replicas to eliminate all-to-all communication. As a result, authenticator complexity is similarly reduced to $O(N)$. Additionally, SBFT leverages threshold signatures to reduce message size and to decrease the number of client replies to one per decision.

Several BFT protocols ([16, 27, 34]) use an expensive view change protocol to handle leader failure. For instance, the standard view change protocol in PBFT requires $O(N^3)$ message authenticators, limiting its scalability. HotStuff [58] introduces an additional phase during normal operation to address this issue. This modification reduces the authenticator complexity of the leader failure protocol to $O(N)$, matching that of the normal case protocol. However, it incurs an extra one-way network latency to the request commit delay.

BFT with trusted components. To reduce protocol complexity, recent research [20, 22, 37, 55, 61] proposes to use *trusted components* on each replica. These components can be implemented in a Trusted Platform Module (TPM) [53] or run in a trusted hypervisor, and are assumed to always function correctly, even when residing on Byzantine nodes.

A2M-PBFT-EA [20] utilizes an *attested append-only memory* (A2M) to securely store operations as entries in a log. Each A2M log entry is associated with a monotonically increasing, gap-less sequence number. Once a log entry is appended, it becomes immutable and its content can be attested by any node in the system.

	PBFT [16]	Zyzzyva [34]	SBFT [27]	HotStuff [58]	A2M [20]	MinBFT [55]	NeoBFT
Replication Factor	$3f+1$	$3f+1$	$3f+1$	$3f+1$	$2f+1$	$2f+1$	$3f+1$
Bottleneck Complexity	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$
Authenticator Complexity	$O(N^2)$	$O(N)$	$O(N)$	$O(N)$	$O(N^2)$	$O(N^2)$	$O(N)$
Message Delay	5	3	6	4	5	4	2

Table 1: Comparison of NeoBFT to state-of-the-art BFT protocols. Here, *bottleneck complexity* denotes the number of messages the bottleneck replica needs to process; *authenticator complexity* shows the total number of signatures processed by all replicas.

With A2M, replicas can eliminate equivocation, thereby reducing the replication factor to $2f+1$. However, the protocol still suffers from the same bottleneck complexity, authenticator complexity, and message latency as PBFT. To address this issue, TrInc [37] reduces the trusted component to a single counter and leaves the log in untrusted memory.

MinBFT [55] introduces a message-based trusted primitive called Unique Sequential Identifier Generator (USIG). USIG generates a unique identifier for each input message that is monotonic, sequential, and verifiable. By authenticating USIG identifiers, MinBFT replicas can validate that all other replicas have received the same messages in the same order. This property enables MinBFT to reduce the message delay to four. Unfortunately, MinBFT's authenticator complexity remains at $O(N^2)$.

2.2 In-Network Ordering for CFT Protocols

Another recent line of work [38, 39, 45] proposes a new approach to designing crash fault-tolerant (CFT) SMR protocols. These systems move the responsibility of request ordering to the data center network. By doing so, application-level protocols only ensure durability of client operations. This network co-design approach improves SMR protocol performance by reducing coordination overhead among servers needed to commit an operation. For instance, NOPaxos [39] dedicates a programmable switch in the network as a sequencer, which stamps sequence numbers to each request. This ensures that all replicas execute requests in the same sequence number order. However, these solutions only target CFT protocols and are unable to handle Byzantine faults, such as when a Byzantine node impersonates the sequencer and broadcasts conflicting message orders.

2.3 Deployment Model

Our work targets the permissioned [2, 46] BFT setting, where access to the system is controlled and there is no mutual trust among the participants. We further target blockchain applications with strict latency requirement such as trading [3, 51]. For performance considerations, these systems are commonly deployed within a single data center. Our solution can be easily extended to geo-distributed settings, but in this work we focus on the single data center use case.

3 AUTHENTICATED IN-NETWORK ORDERING

The core of a BFT SMR protocol is to establish a consistent order of requests even in the presence of failures. Traditionally, this task is accomplished by explicit communication among the replicas, typically coordinated by a leader. In this work, we propose a new approach to improving the efficiency of BFT protocols. Our approach shifts the responsibility of request ordering to the network infrastructure.

3.1 The Case for an Authenticated Ordering Service in Data Center Networks

To guarantee linearizability [28], BFT SMR protocols require that all non-faulty replicas execute client requests in the same order. However, due to the best-effort network assumptions, an application-level protocol is fully responsible for establishing a total order of requests among the replicas. For example, in PBFT [16], the primary replica assigns an order to client requests before broadcasting to backup replicas. All replicas then use two rounds of communication to agree on this ordering while tolerating faulty participants. As discussed in §2, adding trusted components to each replica does not alleviate the coordination and authentication overhead in BFT protocols. Replicas still require remote attestations to verify the received messages.

What if the underlying network can provide stronger guarantees? Prior work [38, 39, 45] has already demonstrated that in-network ordering, realized through network programmability [13, 30], can offer compelling performance benefits to crash fault-tolerant SMR protocols. In this work, we argue that BFT protocols can similarly benefit from shifting the ordering responsibility to the network. By offloading this task to the network, BFT replicas can avoid explicit communication to establish an execution order, thereby reducing cross-replica coordination and authentication overhead. This network ordering approach improves both protocol throughput and latency, as less work is performed on each replica, and fewer message delays are needed to commit a request.

Why authenticated ordering in the network? In previous network ordering systems, the responsibility of ordering requests is entirely delegated to the network primitive, such as the Ordered Unreliable Multicast in NOPaxos [39] and the multi-sequenced groupcast in Eris [38]. In non-Byzantine contexts, this network-level ordering is the only request order observed by any replica. However, in a BFT deployment, a faulty node can easily impersonate the network primitive and assign a conflicting message order, violating the ordering guarantee of the network layer. To prevent this, we augment the network primitive to provide *authentication*: Non-faulty replicas can independently verify that the received message order is indeed established by the network and not by any faulty node. In §4, we explain how such authentication can be efficiently implemented using commodity switch hardware.

Hybrid fault model and Byzantine network. If the network itself exhibits Byzantine faults, it can equivocate by assigning different message orders to different replicas, thereby violating the ordering guarantee. In this work, we argue for a *dual fault model*. The model always assumes a Byzantine failure model for end-hosts. The network infrastructure, on the other hand, can either be crash-faulty or Byzantine-faulty. Our argument is inspired by prior work proposing hybrid fault model [52] and work [41] that separates machine faults from network faults. Our approach provides deployment flexibility — users can choose either a hybrid failure model or the traditional Byzantine model — with an explicit trade-off between fault tolerance and performance. For deployments that *trust* the network to only exhibit crash and omission faults, i.e., a hybrid fault model, our solution offers the optimal performance; if the network infrastructure can behave arbitrarily, our solution can tolerate Byzantine faults in the network, albeit taking a small performance penalty.

We contend that a hybrid fault model, which assumes the network is crash-faulty, is a practical choice for many systems deployed in data centers. Networking hardware presents a smaller attack surface and is less vulnerable to bugs compared to software-based components. They are single application ASICs without sophisticated system software, and formal verification of their hardware designs is a common practice. Furthermore, systems deployed in data centers inherently place some level of trust in the hardware infrastructure. Data center operators also have a strong economic incentive to maintain the trust of their customers by providing reliable services. We, however, admit that this model is weaker than those assumed by traditional BFT protocols. Under the hybrid fault model, our system no longer guarantees safety or liveness if the network becomes adversarial.

Our fault model resembles existing deployment options in the public cloud, where only deployments that do not trust the cloud infrastructure run their virtual machines on instances with a Trusted Execution Environment (TEE) such as Intel SGX. Most use cases, however, place trust on cloud hardware and hypervisors. In return, they attain higher performance compared to their TEE counterpart.

3.2 Authenticated Ordered Multicast

So far, we have argued for an authenticated ordering service in the network for BFT protocols. To that end, we propose a new Authenticated Ordered Multicast (AOM) primitive as a concrete instance of such model. Similar to other multicast primitives like IP multicast, an AOM deployment consists of one or multiple AOM groups, each identified by a unique group address. AOM receivers can join and leave an AOM group by contacting a configuration service. A sender sends an AOM message to an AOM group address, which the network is responsible for routing to all group receivers. Notably, senders do not have knowledge of the identity or the address of individual receivers. Instead, they only specify the group address as the destination.

Unlike traditional best-effort IP multicast, AOM provides a set of stronger guarantees, which we formally define here:

- **Asynchrony.** There is no bound on the delivery latency of AOM messages.
- **Unreliability.** There is no guarantee that an AOM message will be received by any receiver in the destination group.

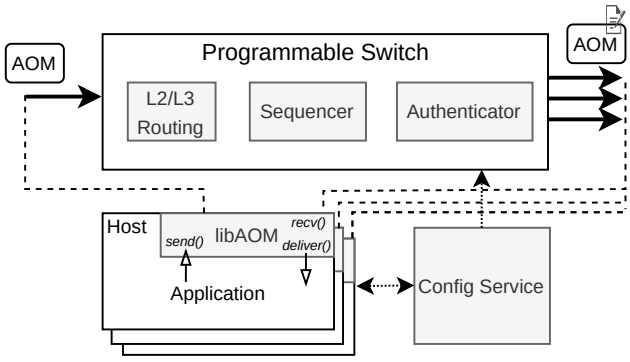

Figure 1: System architecture of AOM. A user-space library is loaded into each AOM sender and receiver. All AOM messages are routed to a designated programmable switch. The switch generates sequence numbers and authentication codes for each AOM message. A separate service handles AOM group membership and switch configurations.

- **Authentication.** A receiver can verify the authenticity of an AOM message, i.e., the message is correctly processed by the AOM network primitive. A correct receiver only delivers authentic AOM messages.
- **Transferable Authentication.** If a receiver r_1 forwards an AOM message to another receiver r_2, r_2 can independently verify the authenticity of the message.
- **Ordering.** For any two authentic AOM messages m_1 and m_2 that destined to the same AOM group G, all correct receivers in G that receive both m_1 and m_2 deliver them in the same order.
- **Drop Detection.** Receivers can detect AOM message drops in the network and deliver DROP-NOTIFICATION. Formally, for any authentic AOM message m, either 1) all correct receivers in the destination group G delivers m or a DROP-NOTIFICATION for m before delivering the next AOM message, or 2) none of the correct receivers in G delivers m or a DROP-NOTIFICATION for m.

One of the key distinguishing properties of AOM is the ability for receivers to verify the *authenticity* of AOM messages independently. In this context, authenticity refers not to the identity of the sender, which still requires end-to-end cryptography. Instead, it assures that a message has been correctly processed by the AOM primitive, and that its ordering has not been tampered by other participant in the system. The authentication capability is also transferable: an AOM message can be relayed to any other receiver in the group, who can independently verify its authenticity.

4 AOM DESIGN

In this section, we present our design of the proposed AOM network primitive on programmable switches. The overall system architecture is shown in Figure 1.

4.1 Design Overview

Our AOM primitive design consists of three major components: a network-wide configuration service, a programmable network data plane, and an application-level library running on AOM senders and receivers. Analogous to IP multicast, receivers create and join

an AOM group by contacting the configuration service via secure TLS channels. The configuration service then designates one programmable switch in the network as the *sequencer* for the group and directs it to broadcast routing advertisement for the group address using a protocol such as BGP. Upon successful propagation of the advertisement, AOM messages destined for the group address will be forwarded to the designated sequencer switch.

The sender-side library generates a custom packet header that follows the UDP header. This custom header includes the *group ID*, a *sequence number*, an *epoch number*, a message *digest*, and an *authenticator*. The digest is generated using a collision-resistant hash function [48]. The sequencer switch is responsible for filling in all fields in the custom header, excluding the group ID and the message digest.

The switch features a sequencing module (§4.2), which stamps sequence numbers onto AOM packets. The pipeline then feeds the stamped sequencer number, concatenated with the message digest, into an authentication module. This module generates an authenticator, which can be a vector of HMAC (§4.3) or a single public key signature (§4.4). The switch then incorporates the generated authenticator into the header and multicasts the packet to all group receivers.

The receiver-side library verifies the authenticator and delivers AOM messages in sequence number order. For any gap in the number sequence, the receiver delivers a DROP-NOTIFICATION. For deployments that operate in a Byzantine-faulty network, the receiver-side library additionally exchanges CONFIRM messages with other receivers within the group to tolerate sequencer equivocation (§4.2).

4.2 Message Ordering and Failure Handling

To establish a consistent ordering of AOM messages, we leverage programmable switches to stamp monotonically increasing sequence numbers to each AOM packet. The sequencer switch employs a register array to maintain a counter for each AOM group. A separate match table maps AOM group IDs to indices into this array. During AOM packet processing, the switch locates the counter register using the group ID within the header, increments the counter, and inserts the counter value into the header. After the switch generates an authenticator, it uses its replication engine to multicast the stamped AOM message to all AOM receivers within the group. As detailed in §4.1, receivers deliver authenticated AOM messages in sequence number order. If a gap in the number sequence is observed, the receiver delivers a DROP-NOTIFICATION.

Tolerating Byzantine-faulty network. If the network infrastructure is trusted to be non-Byzantine (§3.1), a receiver can directly deliver authenticated AOM messages in sequence number order. This delivery rule conforms our *ordering* property, as any two receivers are guaranteed to receive identical messages for each sequence number. Combining with the *transferable authentication* property, a single AOM message suffices as a publicly verifiable *ordering certificate* for itself, which we exploit in our NeoBFT protocol (§5).

However, in a Byzantine-faulty network, the sequencer may equivocate by sending different message ordering to each receiver. To tolerate network-level equivocation, upon receiving an AOM message, a receiver r broadcasts a signed confirmation $\langle \text{CONFIRM}, s, h \rangle_{\sigma_r}$ to the group, where s is the message sequence number and h is the hash of the message. r ignores subsequent AOM messages with the same sequence number, and only delivers an AOM message after it collects enough matching CONFIRMs (at least $2f + 1$ where f is the number of faulty receivers). This strengthened delivery rules ensures ordering, as *quorum intersection* guarantees that no two non-faulty replicas can deliver distinct AOM messages for the same sequence number. The entire message set, including the AOM message and the matching CONFIRMs, is delivered to the application and serves as an ordering certificate.

Sequencer switch failover. Due to network partitions, faulty sequencer switches, or other network anomalies, AOM receivers may fail to receive authenticated AOM messages indefinitely. In such cases, receivers can request the configuration service to fail over to a different sequencer for the group. However, the set of delivered messages at each receiver may differ when the new switch takes over. Furthermore, the old sequencer may only suffer a transient fault. To properly handle a switch failover, the application-level protocol is responsible for reaching consensus on the set of messages delivered by the failed sequencer (§5.5). Once an agreement is reached, the receivers ask the configuration service to select a new sequencer switch and exchange the necessary authentication keys. Subsequently, they can start delivering *authenticated* AOM messages from the new sequencer switch and ignore messages from the old one.

4.3 HMAC-Based Authentication

Generating *secure* and *transferable* authentication tokens in network hardware is more challenging than message sequencing due to switch resource constraints. Our first design uses HMAC vector as an AOM authentication token [16]. Upon joining an AOM group, a receiver uses a key exchange protocol [42] to share a secret key with the sequencer switch, facilitated by the configuration service. The switch control plane installs the secret key of each receiver in the data plane. To authenticate an AOM message, the switch generates a vector of HMACs, one entry for each receiver. Each code is computed by inputting the concatenated message digest and the sequence number (§4.1), and the receiver's secret key to a keyed cryptographic hash function. The switch then writes the entire HMAC vector into the message header. A receiver authenticates an AOM message by comparing a locally computed HMAC to the corresponding entry in the received vector. By including the entire HMAC vector, AOM authentication is *transferable* (§3.2).

In-switch HMAC implementation. Implementing an unforgeable HMAC requires access to collision resistant cryptographic hash functions. Recent advancements, such as HalfSipHash [59] and P4-AES [19], demonstrate the feasibility of implementing high-throughput cryptographic hash functions on switches. In this work, we use HalfSipHash as a building block for our in-switch HMAC design.

A cryptographic hash function, however, only solves half of the equation; implementing *HMAC vector* in the switch data plane introduces several new technical challenges. Firstly, the hash function consumes significant switch hardware resources. For instance, the reference HalfSipHash implementation uses all 12 pipeline stages of a Tofino [30] switch. Naively replicating HMAC calculation to

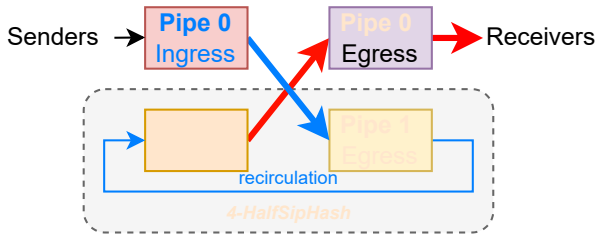

Figure 2: Folded pipeline design for generating HMAC vectors on a Tofino switch. Blue arrows denote AOM packets without HMACs. Red arrows represent authenticated AOM packets. Thick arrows refer to multicast.

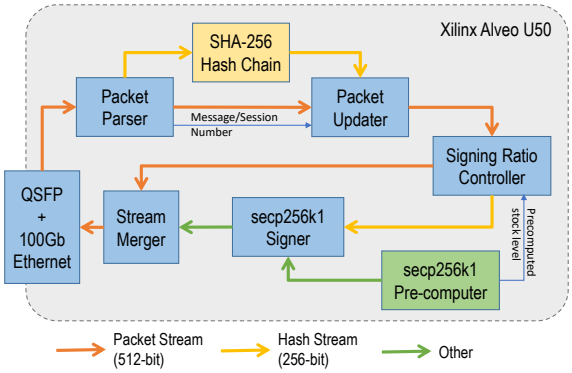

Figure 3: AOM FPGA-based public-key coprocessor design

generate HMAC vectors will easily exceed the switch resource constraints. Second, there exists data dependencies between HMAC computation and other switch logic (e.g., sequencing). A sequential combination of the components would result in a dependency chain that surpasses the hardware limit [33]. Lastly, the size of HMAC vectors grows *linearly* with the AOM group size. The switch data plane design needs to scale to handle large AOM instances given a fixed set of resources.

Our approach. In Figure 2, we illustrate our switch data plane design for HMAC vector generation. To overcome the challenges of limited resources and data dependency, we dedicate one switch pipeline (PIPE 1) solely for the computation of HMAC vectors. After ingress processing, packets requiring HMAC vector generation are forwarded to the loopback ports in the designated HMAC pipeline. Upon completion of the HMAC vector calculation, the HMAC module multicasts the resultant packets to the intended egress pipelines or returns them to the original ingress for further processing. Our pipeline-folding architecture extends the computation capacity beyond the available pipeline stages. Furthermore, our approach *decouples* HMAC vector computation from the other packet processing logic, leading to a more streamlined and modular design.

The original HalfSipHash design [59] requires 6 pipeline passes to produce one HMAC. Even for a small AOM group with 4 receivers, this design uses 24 pipelines passes to generate the entire HMAC vector. To improve the overall vector generation latency, we trade off the number of pipeline passes per HMAC for higher degree of parallelism. Specifically, our design unrolls the reference HalfSipHash implementation, which extends the number of pipeline passes to 12. In return, we reduce the hardware resources required for one HMAC instance by half. As a result, our design can fit four parallel instances of HalfSipHash in the HMAC module, thereby generating a 4-HMAC vector in 12 pipelines passes.

To scale beyond 4 receivers in an AOM group, we leverage the additional loopback ports in the dedicated HMAC pipeline. As our base design can produce 4 HMACs each time, we partition receivers into subgroups of 4. To request HMAC vector generation, the switch *multicasts* the packet to n loopback ports in the HMAC pipeline, where n is the number of subgroups. The switch then runs n independent instances of the based design, each generating 4 HMACs for a subgroup. The resulting n packets are all sent to the receivers, who assemble the complete HMAC vector. With 16 loopback ports [30], our design can scale up to 64 receivers. For smaller AOM groups, the switch load balances between the loopback ports to increase the vector generation capacity.

4.4 In-Network Public-Key Cryptography

Our switch HMAC design is optimized for small AOM groups. However, as the group size increases, performance of our HMAC authentication degrades. The switch requires additional pipeline passes to compute the larger HMAC vector, which reduces the effective HMAC vector generation rate. Moreover, each receiver processes more packets for each AOM message, as the number of HMACs that can fit in a single header is limited by the Packet Header Vector (PHV). To address this scalability issue, we propose an alternative design that implements public-key signature-based authentication.

Concretely, each sequencer switch generates a private-public key pair. All public keys are stored and distributed by the configuration service. To authenticate an AOM message, the switch uses its private key and the concatenated message digest and sequence number (§4.1) as input for a public key algorithm [31] that generates a digital signature. AOM receivers then use the switch public key to verify the sequencer signature in the message. Performance of our signature-based authentication design is group size agnostic, as the switch generates a *single* signature for each AOM message, regardless of the number of receivers.

In-network cryptography design. Implementing public-key cryptography in a network switch is a daunting task. The RSA [49] public-key algorithm requires modular exponentiation of large prime numbers. Even with aggressive optimizations, calculating an RSA signature still involves unbounded loops of multiplications and modulo operations. The ECDSA [31] algorithm involves similar complexity, and additionally requires random number generations and multiplicative inverses. Unfortunately, current generation programmable switches lack support for these computations, and future switch data planes are unlikely to accommodate them due to strict timing, power, and resource constraints.

To overcome the limitations of existing switches, we propose a new switch architecture that includes a specialized cryptographic coprocessor alongside the primary switching chip. This coprocessor is equipped with a simple processing element, dedicated fast memory, and cryptographic accelerators, and is connected to the

switching chip through high-speed network links. To offload cryptographic operations to the coprocessor, the switch constructs a remote procedure call (RPC) metadata that specifies the operation type, key identifier, input message, and offsets into the packet for operation outputs. After egress pipeline processing, the switch submits both the RPC metadata and the original packet to the coprocessor for processing. The coprocessor then writes the result into the packet and forwards it back to the switch. Our offloading design is best-effort. The coprocessor implements a tail-drop queue for submitted operations, and the switch does not maintain RPC state locally.

Cryptographic coprocessor implementation. We develop a coprocessor prototype for AOM signature signing on a Xilinx Alveo U50 FPGA card [1]. Figure 3 shows the high-level architecture of our hardware design. The card connects to one of the switch ports through a 100Gbps QSFP28 cable. A parser module parses the RPC metadata, and forwards the operation input to a hashing module to calculate an SHA-256 [24] hash. A signing module then generates a signature of the hash using the `secp256k1` elliptic curve [31]. Finally, a stream merger module stamps the signature into the packet and sends it back to the QSFP28 port. We developed all the hardware modules, except the Xilinx QSFP28 hard IP, in-house using a combination of RTL and HLS.

Even with a powerful FPGA chip, calculating an `secp256k1` signature is still a time-consuming process. We reduce signing latency by exploiting the underlying mathematical property of `secp256k1` — a significant portion of the curve computation is input-independent. Specifically, we design a pre-compute module that continuously calculates multiples of a generator point of the elliptic curve and stores them in a pre-computed table in fast block RAM. The signer module uses values in this table to speed up scalar point multiplication.

The rate of generating pre-computed table entries can limit the overall coprocessor signing throughput. To address this limitation, we propose a novel hash chaining technique. A packet updater module stamps into each AOM packet an additional SHA-256 hash of the *preceding packet* in the number sequence. A signing ratio controller monitors the stock level of the pre-computed table and instructs the signer module to skip generating signatures for packets once the stock level falls below a threshold. Consequently, while all AOM packets contain an SHA-256 hash of the previous packet in the stream, only a subset of them may include a signature. To authenticate signature-less packets, receivers wait until the next signed packet and verify the entire batch by validating the hash chain in the reverse order.

4.5 Which Authentication Variant to Use?

The two authentication variants have distinct set of trade-offs. The HMAC-based scheme is lighter weight and can be implemented on existing switches without special hardware support. However, it suffers from poor scalability, requires more complex credential setup, and has weaker security guarantees due to the hash function limitation. The scheme is therefore a better option for smaller deployments with stricter performance requirements. The public-key signature variant scales to large receiver groups and is more secure. It, however, requires switches that are not yet commercially available or additional FPGA hardware. Verifying public-key signatures also incurs a higher overhead on the receivers.

5 THE NEOBFT PROTOCOL

Leveraging the authenticated ordering guarantee provided by AOM, we co-designed a new BFT protocol, NeoBFT, that commits client operations in a *single RTT*, even in the presence of Byzantine replicas.

5.1 System Model

We assume a Byzantine failure model for both clients and replicas. The fault model of the AOM primitive can be either hybrid or Byzantine, as discussed in §3.1. We make standard failure assumptions [16] about the configuration service. The service ensures that no more than f faulty replicas are present in a replication group, and a correct[1] sequencer switch is *eventually* installed for each group. Note that the eventual correct switch assumption is only for protocol liveness, not safety.

We make standard cryptography assumptions: Nodes do not possess enough computational resources to subvert the cryptographic hash functions, message authentication codes, and public-key crypto algorithms we use in the protocol. We also assume a strong adversary model: Byzantine nodes can collude, but cannot delay correct nodes indefinitely.

NeoBFT is a state machine replication [50] protocol. We assume all operations executed by the protocol are deterministic. With less than $\lfloor \frac{n-1}{3} \rfloor$ (Byzantine) faulty replicas in the system (where n is the total number of replicas), NeoBFT guarantees linearizability [28] of client operations. Due to the impossibility of asynchronous consensus [25], NeoBFT only ensures liveness in periods of synchrony.

5.2 Protocol Overview

NeoBFT relies on the guarantees provided by the AOM network primitive to achieve single RTT commitment in the *common case*. Specifically, clients multicast requests to NeoBFT replicas using AOM. In the absence of network-level anomalies (e.g., message drops and switch failures), all correct replicas delivers AOM messages in the exact same order. Crucially, such guarantee implies that replicas require no explicit communication to agree on the order of messages. NeoBFT thus avoids the expensive cross-replica coordination and server signature signing/verification required by other BFT protocols. Moreover, adversaries can not temper with the order of messages nor their content, as correct replicas can independently verify the authenticity and integrity of each AOM message. Once an AOM message is delivered, replicas can immediately execute the request and respond to the client, resulting in a single phase fast path protocol. As discussed in §4.2, when delivering messages, the network primitive provides an *ordering certificate*. Similar to previous speculative protocols [34, 39, 45], NeoBFT relies on clients to confirm operation durability. As a result, replicas execute speculatively, before the final commitment. However, since all correct replicas have already established a total order of operations, NeoBFT does not require extra protocols to handle faulty replicas (Zyzzyva [34])

[1] A correct sequencer switch is reachable from all non-faulty replicas and properly follows the AOM design. If the network is trusted, the second condition always holds.

or state divergence due to out-of-order executions (Speculative Paxos [45]). Only in the exceptional case where a speculatively executed operation is later agreed to be skipped (due to the gap commit or the view change protocol), a NeoBFT replica is required to roll back application state.

In the rare case where AOM messages are dropped in the network, the AOM primitive delivers DROP-NOTIFICATIONS to non-faulty replicas. To handle DROP-NOTIFICATIONS, replicas only need to agree on whether to process or to skip the message, not the order of messages. NeoBFT uses a BFT binary consensus protocol, driven by a leader replica, to reach this agreement. In this protocol, the leader uses a single ordering certificate (received by any replica) to commit the corresponding message. To permanently skip the message, the leader replica collects evidences from a quorum of replicas to form a *drop certificate*, which non-leader replicas would verify before committing the message as a NO-OP. We use drop certificates to prevent Byzantine replicas from delaying the agreement indefinitely.

A faulty AOM sequencer may stop multicasting messages or deliberately equivocating or dropping messages. To ensure progress, replicas request the configuration service to replace the faulty sequencer. Installment of a new sequencer indicates the start of a new epoch. Correctness of the protocol requires replicas to agree on the set of messages processed in the last epoch before entering the new epoch. To that end, each NeoBFT instance goes through a sequence of *views*; each view is identified by a view number represented as a ⟨*epoch-num*, *leader-num*⟩ 2-tuple. When the current leader replica has failed (or suspected to be failed) or an old epoch has ended, replicas advance the respective field in the view number, and use a view change protocol [16, 39, 40, 45] to reach agreement on the set of messages in the last view.

NeoBFT also includes a protocol to periodically synchronize replica states and finalize speculatively executed requests. Details of the protocol can be found in §B.2.

5.3 Normal Operation

We first consider the common case protocol in which AOM messages, instead of DROP-NOTIFICATIONS, are delivered to NeoBFT replicas in a stable epoch. A client c requests execution of an operation *op* by sending a signed message $\langle \text{REQUEST}, op, request\text{-}id \rangle_{\sigma_c}$ using the AOM primitive, where *request-id* is a client-generated identification to match replica replies. The message is processed by the network primitive (§3.2), and an ordering certificate (OC) for the message is delivered to all replicas. If the client does not receive replies in a timely manner, it uses regular unicast to send the request to all replicas (while keeps resending the request using AOM). Our view change protocol (§5.5) ensures that the request is eventually committed even if the AOM sequencer is faulty.

Replica i verifies the OC by authenticating the AOM authenticator and checking the $2f + 1$ matching CONFIRMS (only for Byzantine-faulty network). It then adds the OC to its log, speculatively executes *op*, signs and replies $\langle \text{REPLY}, view\text{-}id, i, log\text{-}slot\text{-}num, log\text{-}hash, request\text{-}id, result \rangle_{\sigma_i}$ to the client, where *view-id* is the current view number, *log-slot-num* is the log index the request occupies, *log-hash* is a hash of the log up to the index, and *result* is the execution result. We use hash chaining for $O(1)$ hash calculation [45].

Client c waits for $2f + 1$ replies from different replicas with valid signatures and matching *view-id*, *log-slot-num*, *log-hash*, and *result*. It then accepts the result in the reply.

5.4 Handling Dropped Messages

When a non-leader replica i receives a DROP-NOTIFICATION, it attempts to recover the missing message from the leader. To do so, it sends a ⟨QUERY, *view-id*, *log-slot-num*⟩ to the leader. QUERY messages require no signatures since they do not alter the state of a correct replica. If the leader has the corresponding OC, it responds with a ⟨QUERY-REPLY, *view-id*, *log-slot-num*, OC⟩. Replica i verifies the OC and ensures the enclosed AOM message is the missing message by checking the internal sequence number. It then resumes normal operation. Because OC can be independently verified by any replica, QUERY-REPLYS also require no signatures. Note that replica i *blocks* on waiting for the leader's response or a committed NO-OP before processing subsequent client requests, resending QUERY messages if necessary.

If the leader l itself receives a DROP-NOTIFICATION, it broadcasts a $\langle \text{GAP-FIND-MESSAGE}, view\text{-}id, log\text{-}slot\text{-}num \rangle_{\sigma_l}$ to all replicas. When replica i receives a GAP-FIND-MESSAGE, it replies to the leader with either a ⟨GAP-RECV-MESSAGE, *view-id*, *log-slot-num*, OC⟩ if it has received the ordering certificate, or a $\langle \text{GAP-DROP-MESSAGE}, view\text{-}id, i, log\text{-}slot\text{-}num \rangle_{\sigma_i}$ if it has also received a DROP-NOTIFICATION for the message. If a replica replies GAP-DROP-MESSAGE to a GAP-FIND-MESSAGE, it blocks until it receives the gap agreement decision (ignoring QUERY-REPLYS for the message).

Once the leader receives one GAP-RECV-MESSAGE or $2f + 1$ GAP-DROP-MESSAGE (including from itself), whichever happens first, it uses a *binary* Byzantine agreement protocol, similar to PBFT [16], to commit the decision. Specifically, the leader broadcasts a $\langle \text{GAP-DECISION}, view\text{-}id, log\text{-}slot\text{-}num, decision \rangle_{\sigma_l}$, where *decision* is either a single GAP-RECV-MESSAGE or $2f + 1$ GAP-DROP-MESSAGES. If a GAP-RECV-MESSAGE is received, the leader first verifies the enclosed OC following the same procedure as above.

When replica i receives a GAP-DECISION, it verifies the enclosed OC if the decision contains a GAP-RECV-MESSAGE. If the decision contains $2f+1$ GAP-DROP-MESSAGE, the replica verifies that all $2f+1$ messages are from distinct replicas, and their *log-slot-num* matches the one in the GAP-DECISION. It then broadcasts a $\langle \text{GAP-PREPARE}, view\text{-}id, i, log\text{-}slot\text{-}num, recv\text{-}or\text{-}drop \rangle_{\sigma_i}$, where *recv-or-drop* is a binary value indicating the decision, to all replicas.

Once replica i receives $2f$ GAP-PREPARES from distinct replicas (possibly including itself) and it has received a validated GAP-DECISION with a matching decision from the leader, it broadcasts a $\langle \text{GAP-COMMIT}, view\text{-}id, log\text{-}slot\text{-}num, recv\text{-}or\text{-}drop \rangle_{\sigma_i}$ to all replicas. The replica also stores the GAP-DECISION and $2f$ GAP-PREPARE in its log for the view change protocol.

When replica i receives $2f + 1$ GAP-COMMITS from different replicas (possibly including itself), it stores either the OC (if it hasn't done so) or a NO-OP to the log slot based on the decision, and resuming normal operation if it is blocking on a QUERY-REPLY or a gap agreement decision. It also stores all $2f + 1$ GAP-COMMITS in its log. This quorum of GAP-COMMIT will serve as a *gap certificate* for the state synchronization and view change protocols. In the rare case where the replica has already speculatively executed the

request and the decision is a drop, it rolls back the application state to right before *log-slot-num*, and re-executes subsequent requests in the log.

5.5 View Changes

We use a view change protocol, inspired by PBFT, to handle both leader failures and faulty AOM sequencers. The protocol guarantees that all committed operations (including NO-OPs) will carry over to the new view. View changes can be initiated when a non-leader replica fails to make progress in a gap agreement or state synchronization protocol, or when a replica receives a request message directly from the client (§5.3) but the request is not delivered by AOM after a timeout.

For view changes that involve switching epochs, the protocol requires log consistency before entering the new epoch. To that end, we introduce an *epoch certificate* consisting of $2f + 1$ valid EPOCH-START messages from distinct replicas. An epoch certificate is a proof of the agreed starting log position of the epoch. We then define *validity* of a replica log as the following: a replica log is valid if and only if (i) the starting log position of all epochs are supported by a valid *epoch-cert*, and (ii) within each epoch e, all log positions are filled with either a valid OC or a NO-OP supported by a gap certificate.

Our view change protocol is similar to that of PBFT. The main differences are the epoch certificates and the definition of log validity. Details of the protocol can be found in §B.1.

5.6 Correctness

Here, we sketch a correctness proof for the NeoBFT protocol. Complete safety and liveness proofs can be found in §C.

The safety property we are proving is linearizability [28]. A key definition we use in our proof is *committed operations*: an operation is committed in a log slot if it is executed by $2f + 1$ replicas with matching *view-id*s and *log-hash*es.

First, we show that within an epoch, if a request r is committed at log slot l, no other request r' ($r' \neq r$) or NO-OP can be committed at l. Due to the guarantees of AOM, no correct replicas will execute r' at log slot l given that some correct replica has already executed r, so r' can never be committed at l. To show that NO-OP cannot be committed, we prove by contradiction. Assume a NO-OP is committed at l, some replica would have received a GAP-DECISION with $2f+1$ valid GAP-DROP-MESSAGE from the leader, and $2f$ GAP-PREPARE containing a *drop* decision from different replicas. By quorum intersection, no replica can receive $2f$ distinct GAP-PREPAREs with a *recv* at l, making *drop* the only possible decision. Our view change protocol also ensures that the decision will persist in all subsequent views. Moreover, $2f + 1$ replicas have sent a GAP-DROP-MESSAGE, and at least $f + 1$ of those replicas are non-faulty. Since they block until they receive GAP-COMMITs and the only possible outcome is *drop*, they will not execute r. By quorum intersection, r cannot be committed, leading to a contradiction.

Next, we show that within an epoch, if a request is committed at log slot l, all log slots before l will also be committed. A committed request at l implies that at least $f+1$ correct replicas have matching logs up to l. For each log slot before l, since the same request has been processed by $f + 1$ correct replicas, we can apply the same reasoning as above to show that no other request or a NO-OP can ever be committed at that slot.

Lastly, we show that our view change protocol guarantees that correct replicas agree on all committed requests and NO-OPs across views, and that they start each epoch in a consistent log state. The first point is easy to show given that our view change protocol merges $2f + 1$ logs and using the quorum intersection principle. To prove the second point, we only need to show that for each epoch e, all correct replicas end e at the same log slot before starting $e+1$. To enter epoch $e+1$, a correct replica needs a valid *epoch-cert* for epoch $e + 1$: $2f + 1$ distinct EPOCH-STARTs with matching *log-slot-num*. By quorum intersection, no other *epoch-cert* can exist for $e + 1$ with a different *log-slot-num*. And since correct replicas verify *epoch-cert* for every epoch during view changes, by induction, their logs will be in consistent state.

Module	Stages	Action Data	Hash Bit	Hash Unit	VLIW
Pipe 0	7	0.8%	2.0%	0%	3.4%
Pipe 1	12	12.8%	21.2%	77.8%	12.0%

Table 2: Switch resource usage of the AOM HMAC vector switch prototype

Module	LUT	Register	BRAM	DSP
Pipeline	0.91%	0.70%	2.12%	0.57%
Signer	21.0%	19.4%	10.71%	28.52%
Total	34.69%	29.22%	28.76%	29.16%
Available	870K	1740K	1.34K	5.94K

Table 3: FPGA resource usage of the AOM public-key cryptographic coprocessor

6 EVALUATION

We implement NeoBFT and the AOM library in ~1600 lines of Rust code. The HMAC version of the AOM sequencer is implemented in ~1900 lines of P4 [13] code and compiled using the Intel P4 Studio version 9.7.0. The FPGA-based cryptographic accelerator is written in ~1500 lines of HLS C++/Verilog code, synthesized using the Xilinx Vivado Design Suite 2020.2 [56].

We compare NeoBFT to PBFT [16], HotStuff [58], Zyzzyva [34], and MinBFT [55]. To ensure a fair comparison, all protocols are implemented in the same Rust-based framework. For MinBFT, we run the trusted USIG service in Intel SGX [51]. We also add batching support to all protocols, following the batching techniques proposed in their original work. NeoBFT does not use any batching on the protocol level. Only when using the public key variant of AOM, NeoBFT replicas buffer packets until receiving a signed message from the network. We run all protocols on four replicas, thereby tolerating one Byzantine failure, unless specified otherwise.

Testbed. Our testbed consists of nine servers and a Xilinx Alveo U50 FPGA, all connected to an Intel Tofino-based [30] switch. Replicas are deployed on machines with dual 2.90GHz Intel Xeon Gold 6226R processors (32 physical cores), 256 GB RAM, and Mellanox CX-5 EN 100 Gbps NICs. Clients run on machines with a 2.10GHz

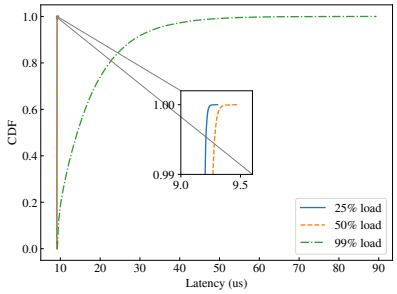

Figure 4: Latency distribution of the HMAC variant of AOM (AOM-HM)

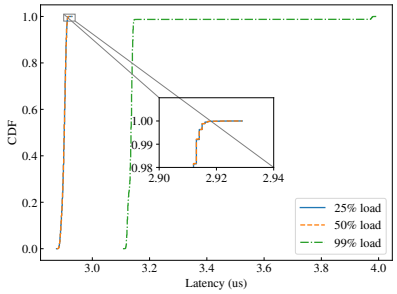

Figure 5: Latency distribution of the public-key variant of AOM (AOM-PK)

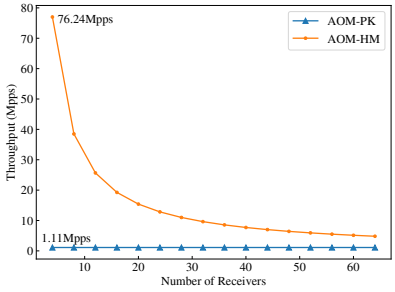

Figure 6: Comparing the maximum throughput of AOM-HM and AOM-PK with increasing group size

Intel Xeon Gold 6230 processor (20 physical cores), 96 GB RAM, and Mellanox CX-5 EN 100 Gbps NICs. All machines run Ubuntu Server 20.04 LTS.

6.1 Micro-benchmarks

We first conduct micro-benchmarks to evaluate the performance of our AOM network primitive. Both the HMAC-based switch design (AOM-HM) and the FPGA-accelerated public-key variant (AOM-PK) are evaluated. Specifically, we generate 64-byte AOM packets at line-rate using the Tofino built-in packet generator. To accurately measure the latency of our design, we take an ingress and an egress switch timestamp for each AOM packet. Table 2 shows the switch resource utilization of our AOM in-network HMAC vector design. Table 3 summarizes the FPGA resource usage of our public-key cryptography coprocessor design.

Figure 4 and Figure 5 shows the latency CDF of the two designs at different load levels. The AOM group size is fixed at 4. Before fully loaded, AOM-HM attains a median latency of ~9μs, while AOM-PK achieves a median latency of ~3μs. The longer latency of the HMAC design is due to the additional pipeline passes (12 in total) to generate a secure hash. Latencies of both hardware designs are highly consistent. The 99.9% latency increases by only 0.7% compared to the median for AOM-HM, and 0.6% for AOM-PK. At close-to-saturation load, AOM-HM shows longer latency tail due to significant queuing delays in the switch pipelines.

Figure 6 shows the maximum throughput attained by each design with varying AOM group size. With a group size of 4, AOM-HM achieves a maximum HMAC vector throughput of 77Mpps, which is around 300 million hashes per second. Its throughput, however, starts to drop when adding more receivers to the group. With 64 receivers, throughput of AOM-HM drops to 5.7Mpps which is only 8% that of the 4-receiver setting. AOM-PK, on the other hand, attains a constant signing throughput of 1.1Mpps regardless of the group size. We note that throughput of AOM under a single sequencer is at least an order of magnitude higher that of our NeoBFT protocol (§6.2). Our network ordering design, therefore, will not become the performance bottleneck of the system.

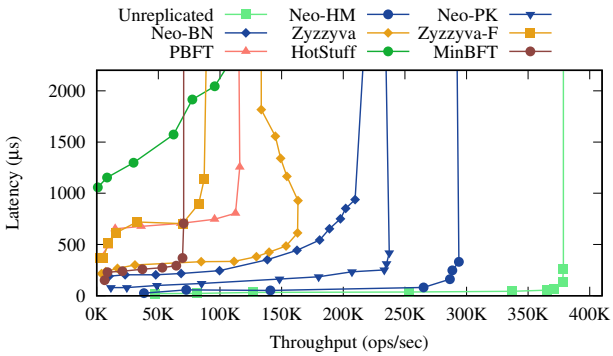

Figure 7: Comparing latency and throughput of NeoBFT and other BFT protocols. Neo-HM and Neo-PK are the HMAC and public-key version of NeoBFT. Neo-BN uses the AOM variant that tolerates Byzantine network. Zyzzyva-F is Zyzzyva with a non-responding Byzantine replica.

6.2 Latency vs. Throughput

We next evaluate the latency and throughput of NeoBFT and compare them to other BFT protocols. Our focus here is protocol-level performance, so we run an echo-RPC application with randomly generated strings as requests. We use an increasing number of closed-loop clients and measure the end-to-end latency and throughput observed by the clients.

As shown in Figure 7, HMAC-based NeoBFT achieves higher maximum throughput than PBFT (2.5×), HotStuff (3.4×), and MinBFT (4.1×). More aggressive batching can further increase HotStuff's throughput to a level comparable to NeoBFT; however, its latency also increases to more than 10ms. To commit a client operation, these protocols require explicit coordination among the replicas, with each message requiring expensive cryptographic operations. MinBFT utilizes the trusted component to reduce the replication factor to $2f + 1$, but does not improve the authenticator complexity. NeoBFT, on the other hand, leverages guarantees of AOM to eliminate coordination and cross-replica authentication overhead in the common case. Comparing to Zyzzyva, NeoBFT still achieves 1.8× higher throughput. Moreover, when one of the replicas becomes faulty, throughput of Zyzzyva (Zyzzyva-F) drops by more than

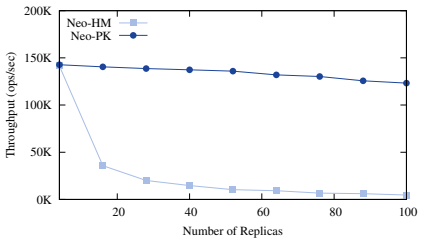

Figure 8: Throughput of NeoBFT with increasing number of replicas

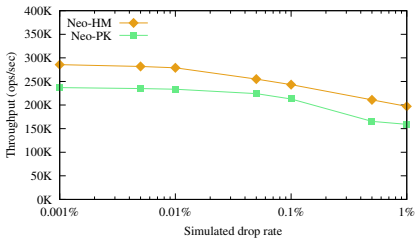

Figure 9: Throughput of NeoBFT with simulated packet drops

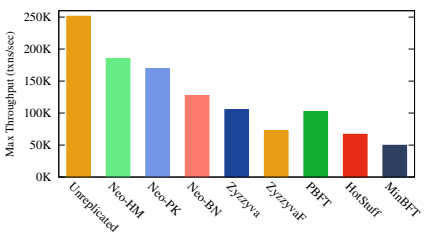

Figure 10: Performance of replicated key-value store using YCSB benchmark

54%, while throughput of NeoBFT is unaffected. When using the public-key variant of AOM, NeoBFT only suffers a 60K throughput decrease, despite requiring more expensive cryptographic operations, demonstrating the efficiency of our in-network crypto design.

Figure 7 also shows the bigger benefit of NeoBFT— latency. HMAC-based NeoBFT outperforms PBFT in latency by 14.68×, HotStuff by 42.28×, Zyzzyva by 8.56×, and MinBFT by 6.08×. NeoBFT commits client operations in two message delays, while the other four protocols require at least three message delays with additional authentication penalties. Using the public-key variant of AOM adds 55μs to the latency of NeoBFT. However, this version of NeoBFT still outperforms all the other protocols in latency by at least 2.7×.

Tolerating Byzantine network. As discussed in §4.2, to tolerate Byzantine sequencers, AOM receivers exchange and authenticate CONFIRM messages. This can lead to degraded throughput and latency compared to deployments that trust the network. Figure 7 shows the performance of NeoBFT when tolerating a Byzantine network. By batch processing CONFIRM messages, NeoBFT minimizes the impact of the additional message exchanges, and is able to sustain a high throughput at the expense of higher latency. As shown in the figure, this NeoBFT variant still outperforms the other comparison protocols in both throughput and latency.

6.3 Protocol Scalability

To evaluate the scalability of NeoBFT, we gradually increase the number of NeoBFT replicas, and measure the maximum sustainable throughput. Due to the limited capacity of our own cluster, we deploy AOM on Amazon EC2 in the AWS ap-east-1 region. We run up to 100 AOM replicas on m5.4xlarge instances with hyper-threading disabled, and clients on t3.micro instances. As EC2 does not offer programmable switches, we implement a software version of the AOM switch using Rust. Due to hardware differences, the maximum throughput number is lower compared to that of our cluster. As shown in Figure 8, NeoBFT using AOM-PK scales to 100 replicas with only a 13% throughput drop. NeoBFT replicas process a constant number of messages per client request, regardless of the replica count. This allows NeoBFT to scale its performance almost linearly with more replicas. Adding replicas, however, would increase the number of reply messages NeoBFT clients need to receive. NeoBFT effectively shift the collector load to the client, which can naturally scale. As discussed in §4.3, when using AOM-HM, replicas receive n messages for each client request, where n is the number of AOM-HM *subgroups*. When the group size increases, throughput of NeoBFT drops as each replica processes linearly more messages.

6.4 Resilience to Network Anomalies

When AOM messages are dropped in the network, NeoBFT replicas coordinate to agree on the fate of the message. To evaluate NeoBFT's resilience to network anomalies, we simulate packet drops in the network, and measure the maximum throughput of NeoBFT. As shown in Figure 9, throughput of NeoBFT is largely unaffected when moderate amount of packets are dropped, since DROP-NOTIFICATION allows non-faulty NeoBFT replicas to efficiently recover missing messages from each other, without the expensive agreement protocol. When a higher percentage of packets are dropped (1%), NeoBFT does suffer a more observable throughput drop.

Sequencer switch failover. When the sequencer switch becomes faulty, NeoBFT performs a view change and fails over to a different sequencer (§5.5). To understand the impact of a faulty sequencer, we measure the throughput of NeoBFT during a switch failover. We ran NeoBFT at maximum throughput for 10 seconds, and then simulated a sequencer failure by dropping AOM packets on the switch. Throughput of NeoBFT immediately dropped to zero. When the replicas detected the sequencer failure, they ran a view change protocol which finished in less than 200 μs. They then informed the configuration service to switch to a new sequencer, which we simulated by re-initializing the sequencer switch state through the control plane. After switch reconfiguration is done, throughput of NeoBFT quickly resumed to its peak. Overall, sequencer failover took less than 100ms, with the majority of the delay caused by network-level updates rather than the view change protocol.

6.5 BFT Storage System Performance

Lastly, we evaluated the performance of NeoBFT when running more complex real-world applications and compared against other protocols. We developed an in-memory, B-Tree-based key-value store, and ran YCSB workload A with 100K records and 128-bytes fields. Maximum YCSB throughput attained by each system is shown in Figure 10. NeoBFT achieved higher throughput than PBFT, HotStuff, Zyzzyva, and MinBFT when running a more complex application. This KV-store requires protocols to handle larger requests than previous experiments, leading to reduced batching efficiency for Zyzzyva, MinBFT, PBFT, and HotStuff. NeoBFT exploits its lower message complexity to attain higher performance.

7 RELATED WORK

BFT protocols. As discussed in §2.1, there has been a long line of work on designing practical BFT protocols [16, 27, 34, 58]. These protocols guarantee correctness in an asynchronous network, and ensure liveness during weak asynchrony. They all use a single leader node to coordinate ordering and agreement, and rely on view change (or similar) protocols to deal with faulty leaders. Byzantine Paxos [36] and DBFT [23] propose a leaderless BFT design, but require a synchronous protocol that commits in $O(f)$ or more rounds. HoneyBadger [43] attacks the weak synchrony assumption and provides optimal asymptotic efficiency. However, it introduces $O(N^3)$ message complexity and five message delays. NeoBFT leverages the guarantees of AOM to eliminate the leader and coordination overhead in the common case, leading to a bottleneck message complexity of $O(1)$ and two message delays to commit an operation.

BFT with trusted components. Recent work have proposed leveraging trusted components to improve BFT protocols [9, 20, 22, 37, 55]. Using a local trusted component on each replica enables these protocols to reduce the replication factor to $2f + 1$. However, since the trusted components are local to replicas, they still necessitate coordination among replicas to commit client operations. Moreover, many of these protocols implement trusted components in resource-constrained TPM hardware which significantly limit their performance. NeoBFT implements its ordering service in the data center network. Relying on authenticated network ordering, the protocol avoid all coordination in normal operation. And by implementing on fast networking hardware, the service does not become the performance bottleneck.

Network ordering. A classic line of work in distributed computing proposes stronger network models, such as atomic broadcast [12, 32] and virtual synchrony [10, 11], to simplify distributed system designs. These network primitives guarantee that a total order of messages are delivered to all broadcast receivers. However, atomic broadcast and virtual synchrony do not offer performance benefits to distributed systems – implementing them is equivalent to solving consensus [18]. NOPaxos [39] and Eris [38] pioneered a weaker network model in which messages are delivered in a consistent order, but reliable transmission is not guaranteed. This weaker model can be efficiently implemented using programmable switches. NOPaxos proposes an Ordered Unreliable Multicast primitive for state machine replication, while Eris designs a multi-sequenced groupcast primitive for distributed transactions. BIDL [46] uses sequencers to parallelize consensus and transaction execution in a permissioned blockchain system. However, BIDL still uses traditional BFT protocols for consensus and its sequencer design does not improve performance of the BFT protocol itself. The network sequencing approach has also been applied to other distributed system designs [7, 8, 57]. Our AOM primitive was inspired from these work, but additionally provides transferable authentication, a guarantee crucial for BFT protocols.

8 CONCLUSION

In this work, we propose a novel in-network authenticated ordering service, AOM. We demonstrated the feasibility of this design by implementing two variants, one using HMAC and another using public key cryptography for authentication. We then co-designed a new BFT SMR protocol, NeoBFT, that eliminates cross-replica coordination and authentication in the common case. NeoBFT outperforms state-of-the-art BFT protocols on both throughput and latency metrics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Andreas Haeberlen for their valuable feedback. Jialin Li is supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (T1 251RES2104) and Tier 2 (MOE-T2EP20222-0016).

Ethics statement. This work does not raise any ethical issues.

REFERENCES

[1] Product brief of Xilinx ALVEO U50 accelerator cards. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u50-product-brief-v2.pdf.
[2] 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, Porto, Portugal, 2018. Association for Computing Machinery.
[3] Australian securities exchange. https://www2.asx.com.au/.
[4] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: A fast short-input prf. In Steven Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012*, pages 489–508, 2012.
[5] Blockchain on AWS. https://aws.amazon.com/blockchain/.
[6] Oracle blockchain platform service. https://www.oracle.com/blockchain/cloud-platform/.
[7] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, San Jose, CA, USA, 2012. USENIX Association.
[8] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, Farminton, Pennsylvania, 2013. Association for Computing Machinery.
[9] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 222–237, New York, NY, USA, 2017. Association for Computing Machinery.
[10] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 123–138, New York, NY, USA, 1987. Association for Computing Machinery.
[11] Kenneth P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, page 79–86, New York, NY, USA, 1985. Association for Computing Machinery.
[12] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, jan 1987.
[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
[14] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.
[15] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association.
[16] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of Symposium on Operating Systems Design and Implementation*, volume 99, pages 173–186, 1999.
[17] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery.
[18] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Eleventh Annual ACM*

Symposium on Principles of Distributed Computing, PODC '92, page 147–158, New York, NY, USA, 1992. Association for Computing Machinery.

[19] Xiaoqi Chen. Implementing AES encryption on programmable switches via scrambled lookup tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, page 8–14, 2020.

[20] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of ACM SIGOPS Operating Systems Review (SOSP)*, volume 41, pages 189–204, 2007.

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 251–264, USA, 2012. USENIX Association.

[22] Miguel Correia, Nuno Ferreira Neves, and Paulo Verıssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 174–183, October 2004.

[23] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018.

[24] Quynh Dang. Secure hash standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Aug 2015.

[25] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[26] Google blockchain node engine. https://cloud.google.com/blog/products/infrastructure-modernization/introducing-blockchain-node-engine.

[27] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. arXiv 1804.01626, 2019.

[28] M. Herlihy and J M Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[30] Intel. Intel® Tofino™. https://bit.ly/3sY7beD, [Accessed: Sep 2020].

[31] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Secur.*, 1(1):36–63, aug 2001.

[32] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[33] Xin Zhe Khooi, Levente Csikor, Jialin Li, Min Suk Kang, and Dinil Mon Divakaran. Revisiting heavy-hitter detection on commodity programmable switches. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 79–87, 2021.

[34] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[35] L. LAMPORT. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[36] Leslie Lamport. Leaderless byzantine paxos. *Distributed Computing: 25th International Symposium: DISC 2011, David Peleg, editor.*, pages 141–142, December 2011.

[37] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[38] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, Shanghai, China, 2017. Association for Computing Machinery.

[39] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, November 2016.

[40] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical report, Computer Science and Artificial Intelligence Laboratory Technical Report, MIT, July 2012.

[41] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical Fault Tolerance beyond Crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, Savannah, GA, USA, 2016. USENIX Association.

[42] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, apr 1978.

[43] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, Vienna, Austria, 2016. Association for Computing Machinery.

[44] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of 2014 USENIX Annual Technical Conference*, JUNE 2014.

[45] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI '15, pages 43–57, Oakland, CA, 2015. USENIX Association. specpaxos.

[46] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, and Heming Cui. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 18–34, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Consensys quorum. https://consensys.net/quorum/.

[48] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC-1321, 1992.

[49] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[50] Fred Barry Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[51] Singapore exchange. https://www.sgx.com/.

[52] P. Thambidurai and You keun Park. Interactive consistency with multiple failure modes. In *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*, pages 93–100, 1988.

[53] Trusted platform module. https://en.wikipedia.org/wiki/Trusted_Platform_Module.

[54] R van Renesse and Deniz Altinbüken. paxos made moderately complex. *ACM Computing Surveys*, 47(3), February 2015.

[55] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 2013.

[56] Xilinx Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[57] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, Boston, MA, USA, 2017. USENIX Association.

[58] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[59] Sophia Yoo and Xiaoqi Chen. Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure*, page 16–22, 2021.

[60] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 633–649. USENIX Association, November 2020.

[61] Danyang Zhuo, Qiao Zhang, Dan R K Ports, Arvind Krishnamurthy, and Thomas Anderson. Machine fault tolerance for reliable datacenter systems. In *Proceedings of the 5th Asia-Pacific Workshop on Systems*, APSys'14, June 2014.

Appendices are supporting material that has not been peer-reviewed.

A ARTIFACT APPENDIX

A.1 Abstract

The NeoBFT artifact comprises two components: the source code of the main protocol, and paper-specific parts (such as artifact scripts and data) to replicate the results in this paper.

A.2 Scope

Our prototype serves as a demonstration of the following:

- The throughput of AOM, including both AOM-HM and AOM-PK variants, is compatible to support replication deployment without performance degradation.
- The co-designed replication protocol outperforms mainstream BFT protocols with lower latency and higher throughput.
- The replication protocol can support fault-tolerance deployment of realistic applications e.g. key-value store.

Result of all evaluations, i.e. Figure 4, 5, 6, 7, 8, 9, 10, can be reproduced.

A.3 Contents

The artifact includes the implementation of libNEO, the NeoBFT protocol, and all comparison BFT protocols: PBFT, Zyzzyva, HotStuff, and MinBFT. The artifact also includes P4 programs for implementing the AOM primitive, and an FPGA bitstream file that works as the AOM-PK accelerator.

A.4 Hosting

The source codes are hosted at https://github.com/nus-sys/neobft-artifact. The repository also includes instructions for running the system and reproducing evaluation results.

A.5 Requirement

Reproducing AOM micro-benchmarks requires a Tofino-1 switch and a Xilinx FPGA card. Experiments for reproducing Figure 7, 9 and 10 additionally require 4 servers running replicas. The scalability evaluation, i.e., Figure 8, requires up to 100 servers to host replicas and additional servers to simulate in-network sequencing and multicast. We conducted the evaluation on AWS EC2 instances.

B ADDITIONAL PROTOCOL DETAILS

B.1 View Change Protocol

In this section, we provide the detail of the view change protocol, which is omitted in the main paper.

When replica i initiates a view change, it broadcasts a $\langle$VIEW-CHANGE, *view-id*, v', *epoch-cert*, *log*$\rangle_{\sigma_i}$ to all other replicas, where *view-id* is its current view number, v' is the new view, and *epoch-cert* contains an epoch certificate for each epoch it has started.

When leader replica l of view v' receives $2f$ *valid* VIEW-CHANGE messages for view v' from different replicas, it merges the logs in the VIEW-CHANGE messages as follows:

(1) It finds the *view-id* with the largest epoch number e that is supported by a *epoch-cert*.
(2) If the replica has not started epoch e yet, it finds a valid log that has started epoch e. It then copies all requests and NO-OPs before the starting log position of epoch e to its own log.
(3) From all valid logs that have started epoch e, it locates the log with the largest *seq-num* in epoch e. It then copies all requests in epoch e from the log to its own log.
(4) From any valid log that have started epoch e, it copies all NO-OPs in epoch e from the log into its own log, possibly overwriting existing requests.

A $\langle$VIEW-START, v', *view-change-msgs*$\rangle_{\sigma_l}$ is then broadcasted by the leader replica l, where *view-change-msgs* contains the $2f$ VIEW-CHANGE messages it uses to merge the log and the VIEW-CHANGE message it would have sent for v'.

When replica i receives a VIEW-START message with v' higher than its current view, it checks that *view-change-msgs* are properly signed by $2f+1$ different replicas, they all contain the same next view number v', and their logs are valid. It then merges its log with logs in *view-change-msgs* using the same procedure we described above.

If the view change does not involve a epoch switch, replica i can immediately enter the new view. Otherwise, it broadcasts a $\langle$EPOCH-START, e', *log-slot-num*$\rangle_{\sigma_i}$ to all replicas, where e' is the new epoch number, and *log-slot-num* is the last log index after merging the logs during view change. Once a replica receives $2f+1$ EPOCH-START messages from different replicas with e' and *log-slot-num* matching its own, it can enter the new view. It also stores these EPOCH-START messages locally as an epoch certificate for future view changes.

B.2 State Synchronization

During normal operations, replicas execute client requests speculatively before they become durable. A speculatively executed request might be overwritten due to the gap agreement or view change protocols, and the replica has to roll back application state and re-executes all subsequent requests in the log. To further reduce the frequency of roll backs and the number of re-executions, we use a periodic synchronization protocol. The goal of the synchronization protocol is to produce a *sync-point*, where all log entries before and including the *sync-point* are *committed*. A committed log entry will never be overwritten or removed, and will be present in the log (at the same position) of all non-faulty replicas in all subsequent views.

After every N entries are added to the log (N is a configurable constant), a replica i broadcasts a $\langle$SYNC, *view-id*, *log-slot-num*, *drops*$\rangle_{\sigma_i}$ to all replicas, where *log-slot-num* is latest log index that is a multiple of N, and *drops* contains *gap certificates* for all log slots that have been committed as NO-OP in the current view. Once replica i receives $2f$ SYNC messages with the same *log-slot-num* from different replicas, for each entry in any of the *drops* that has a valid *gap certificate*, it writes a NO-OP (possibly overwriting existing request) to the corresponding log position and saves the gap certificate. It then updates its *sync-point* to *log-slot-num*.

C CORRECTNESS PROOF

This section contains complete safety and liveness proofs of the NeoBFT protocol.

C.1 Safety

The main safety property guaranteed by NeoBFT is linearizability. In this safety proof, we assume the network primitive AOM provides *transferable authentication*, *ordering*, and *drop detection*, as specified in the main paper.

THEOREM 1 (NEOBFT SAFETY). *NeoBFT guarantees linearizability of client operations and returned results.*

Before proving Theorem 1, we first define a few properties of NeoBFT replica logs and client REQUESTS.

Definition. A REQUEST is *committed* in a log slot if it is executed by $2f + 1$ distinct replicas in that slot with matching *view-id* and *log-hash*.

Definition. A REQUEST is *successful* if the client receives $2f+1$ valid REPLYS from different replicas with matching *view-id*, *log-slot-num*, *log-hash*, and *result*.

It is easy to see that a successful REQUEST implies that the REQUEST is committed.

Definition. A log is *stable* if it is a prefix of the log of every non-faulty replica in views higher than the current one.

LEMMA 1 (LOG STABILITY). *Every successful REQUEST was appended onto a stable log at some non-faulty replica, and the resulting log is also stable.*

To prove Lemma 1, we first prove the following set of lemmas.

LEMMA 2. *All non-faulty replicas that begin an epoch begin the epoch with the same log position.*

PROOF. Prove by induction. In the first epoch, all non-faulty replicas start with log position 0. This proves the base case. Now assume all non-faulty replicas start epoch e with the same log position. To enter the next epoch e', a non-faulty replica needs to receive $2f + 1$ EPOCH-START messages for e' from distinct replicas with *log-slot-num* matching its own. Define these $2f + 1$ EPOCH-STARTS as a epoch certificate. By quorum intersection, no two non-faulty replicas can have epoch certificates with different *log-slot-num*. Therefore, all non-faulty replicas enter epoch e' with the same log position. This proves the inductive step. □

LEMMA 3. *Within an epoch, if a REQUEST is committed in some log slot l, then no replica can include a NO-OP with a valid gap certificate in slot l in that epoch.*

PROOF. We prove by contradiction. Assume a replica inserts a NO-OP with a valid gap certificate in slot l. The replica then has received $2f + 1$ GAP-COMMITS with decision *drop* for slot l, implying some replica has received $2f$ distinct GAP-PREPARES containing a *drop* decision and a matching GAP-DECISION from the leader. Since a non-faulty replica only sends unique GAP-PREPARE for a log slot within a view, by quorum intersection, there cannot exist $2f$ distinct GAP-PREPARES containing a *recv* decision at slot l. Therefore, *drop* is the only possible commit decision for the gap agreement protocol. Moreover, a valid GAP-PREPARE containing a *drop* decision implies that $2f + 1$ replicas have sent a GAP-DROP-MESSAGE. Out of those $2f + 1$ replicas, at least $f + 1$ are non-faulty. Since non-faulty replicas block until they receive GAP-COMMIT and the only possible commit outcome is *drop*, they will not execute REQUEST. By quorum intersection, no $2f + 1$ replicas can execute REQUEST, so REQUEST cannot be committed. This leads to a contradiction. □

LEMMA 4. *For any two non-faulty replicas in the same epoch, no slot in their logs contains different REQUESTS.*

PROOF. Within the epoch, non-faulty replicas insert REQUESTS into their logs strictly in the order received from AOM. In the absence of DROP-NOTIFICATIONS, the *ordering* property of AOM ensures that all non-faulty replicas have identical sequence of REQUESTS in their logs. By Lemma 2, for any epoch, all non-faulty replicas start the epoch with the same log position. Consequently, for any two non-faulty replicas, no log slot within the epoch contains different REQUESTS. A non-faulty replica may also insert a REQUEST r into its log when handling a DROP-NOTIFICATION. To fill the gap caused by the DROP-NOTIFICATION, the replica requires the transfer of r with the corresponding ordering certificate OC (through QUERY-REPLY or GAP-COMMIT). The *transferable authentication* property of AOM ensures that r is identical to REQUESTS delivered by other non-faulty replicas at the same position in the AOM message sequence. The case is therefore equivalent to the case in which a REQUEST, not a DROP-NOTIFICATION, is received by the replica. A non-faulty replica may also insert a NO-OP into its log during the gap agreement protocol. Assume the replica inserts NO-OP at the $l + i$th log slot where l is the starting log position of the epoch. The replica ignores the corresponding ith REQUEST (by checking the sequence number) if it is later delivered by AOM. Consequently, if the replica receives the $i + 1$th REQUEST in the AOM message sequence, it can only insert the REQUEST at log position $l + i + 1$. By the above argument, the replica's log from $l + i + 1$ onward will not contain non-matching REQUESTS from other non-faulty replicas. By induction on i, no log slot within the epoch contains different REQUESTS at any two non-faulty replicas. □

LEMMA 5. *Any REQUEST or NO-OP that is committed at a log position in some view will be in the same log slot in all non-faulty replica's log in all subsequent views.*

PROOF. To enter a new view v' from the current view v, a non-faulty replica needs to receive a VIEW-START message which contains $2f + 1$ VIEW-CHANGE messages from distinct replicas. There are two cases. Case 1: If a request r is committed at log position i in view v, by definition, r is executed by $2f + 1$ replicas at the same log position in v. Therefore, at least $f + 1$ non-faulty replicas have inserted r into their log at position i. By quorum intersection, at least one of the VIEW-CHANGE messages contains r in the log. The log merging rule in the view change protocol ensures that the replica inserts r into its log at the same log position in view v'. And by Lemma 3, no NO-OP can be committed at the same log position, so the replica will not overwrite the slot with a NO-OP during log merging. Case 2: If a NO-OP is committed at log position i in view v, at least $2f + 1$ replicas have sent GAP-COMMIT with

decision DROP-NOTIFICATION for log slot i. Therefore, at least $f + 1$ non-faulty replicas have stored $2f$ GAP-PREPARE and the matching GAP-DECISION with decision DROP-NOTIFICATION. By quorum intersection, at least one of the VIEW-CHANGE messages contains the $2f$ GAP-PREPARE and the GAP-DECISION. The log merging procedure ensures that slot i is filled with a NO-OP in view v'. □

We are now ready to prove the main log stability lemma.

Proof of Log Stability (Lemma 1). A successful REQUEST implies that REQUEST is committed at log slot l. By definition of committed requests, at least $f + 1$ non-faulty replicas have *matching* logs up to l. And since non-faulty replicas insert log entries strictly in log order (blocking before the next log entry is resolved), all log entries before l are occupied. For any log slot $l' \leq l$, if a REQUEST r is stored in the matching logs, by Lemma 4, no other REQUEST can be inserted into the same slot at any non-faulty replica. And by quorum intersection and our gap agreement protocol, there cannot exist a valid gap certificate for slot l'. Therefore, only r can be committed at log slot l' in the view. Otherwise, if a NO-OP is stored in the matching logs, there must exist a valid gap certificate for slot l'. By definition, the NO-OP is committed at l'. Lemma 5 then guarantees that log entry at l' (either a REQUEST or a NO-OP) in the matching log will be in the same log slot in all non-faulty replica's log in all subsequent views. Consequently, the successful REQUEST was appended onto a stable log at least $f + 1$ non-faulty replica. Since the successful REQUEST is also committed at log slot l, by Lemma 5, REQUEST will be in slot l in all non-faulty replica's log in all subsequent views. The resulting log is thus also stable. □

With Lemma 1, we are ready to prove our main safety property.

Proof of NeoBFT Safety (Theorem 1). First, observe that, by definition, a stable log only grows monotonically. Combining this observation with Lemma 1, from the client's perspective, the behavior of NeoBFT is indistinguishable from the behavior of a single, correct machine that processes REQUEST sequentially. This implies that any execution of NeoBFT is equivalent to some serial execution of REQUESTs. Moreover, clients retry sending an operation until a REQUEST containing the operation is successful. NeoBFT applies standard at-most-once techniques to avoid executing duplicated REQUESTs. Therefore, a NeoBFT execution is equivalent to some serial execution of unique client operations.

The above argument proves serializability. When a client receives the necessary replies for a successful REQUEST r, by Lemma 1, r must have already been added to a stable log. For any successful REQUEST r' issued after this point in real-time, r' can only be inserted after r in the stable log. Since non-faulty replicas execute REQUESTs strictly in log order, the operations issued and results returned by NeoBFT are linearizable. □

C.2 Liveness

Due to the well-known FLP result, NeoBFT can not guarantee progress in a fully asynchronous network. We therefore only prove liveness given some weak synchrony assumptions.

Theorem 2 (Liveness). *REQUESTs sent by clients will eventually be successful if there is sufficient amount of time during which*

- *the network the replicas communicate over is fair-lossy,*
- *there is some bound on the relative processing speeds of replicas,*
- *the $2f + 1$ non-faulty replicas stay up,*
- *there is a non-faulty replica that stays up which no non-faulty replica suspects of having failed,*
- *there is a non-faulty AOM sequencer stays up which no non-faulty replica suspects of having failed,*
- *all non-faulty replicas correctly suspect faulty nodes and AOM sequencers,*
- *clients' REQUESTs are eventually delivered by AOM.*

Proof. Since there exist non-faulty replica and non-faulty AOM sequencer that stay up which no non-faulty replica suspects of having failed, there is a finite number of view changes during the synchrony period. Once the non-faulty replica that stays up has been elected as leader, and the non-faulty sequencer has been configured by the network, no view change with a higher view will start, as $2f + 1$ non-faulty replicas will not send the corresponding VIEW-CHANGE message.

Moreover, any view change that successfully starts will eventually finish. A non-faulty replica that initiates a view change keeps re-broadcasting its VIEW-CHANGE message until the new view starts, or until the view change is supplanted by one with a higher view. Since non-faulty replicas correctly suspect faulty nodes and AOM sequencers, eventually $2f + 1$ non-faulty replicas will initiate the view change. As all $2f + 1$ non-faulty replicas stay up, the leader for the new view, if it is non-faulty, will eventually receive the necessary $2f + 1$ VIEW-CHANGE messages to start the view. If the leader is faulty, non-faulty replicas will correctly suspect the fact, and start a higher view change which will supplant the current one.

Additionally, once a view starts, eventually all non-faulty replicas will adopt the new view and start processing REQUESTs in the view, as long as the view is not supplanted by a even higher view. If the leader is non-faulty, it will re-broadcast VIEW-START messages until it receives acknowledgement from all replicas. If the leader is faulty, non-faulty replicas will correctly suspect the fact, and start a higher view change which will supplant the current one.

The above arguments imply that eventually, there will be a view which stays active with a non-faulty leader and a non-faulty AOM sequencer. During that view, non-faulty replicas will eventually be able to resolve any DROP-NOTIFICATION from AOM: The replica receiving a DROP-NOTIFICATION will keep resending the QUERY message until receiving a QUERY-REPLY from the leader or enough GAP-COMMITs. If the leader does not have the REQUEST, it will continually broadcast GAP-FIND-MESSAGE to all replicas. Since $2f + 1$ non-faulty replicas stay up, eventually the leader will receive either one GAP-RECV-MESSAGE or $2f + 1$ GAP-DROP-MESSAGEs. Once the non-faulty leader starts the binary Byzantine agreement protocol with the decision, by applying the same line of reasoning, eventually non-faulty replicas blocking on the DROP-NOTIFICATION will receive the necessary GAP-COMMITs to resolve the DROP-NOTIFICATION.

Therefore, the system will eventually reach a point where a view stays active with a non-faulty leader and a non-faulty AOM sequencer, and non-faulty replicas only receive REQUESTs from AOM. After that point, every REQUEST delivered by AOM will eventually be successful. Because clients' REQUESTs eventually will be delivered by AOM and $2f + 1$ non-faulty replicas stay up, clients will eventually receive the necessary REPLYs for REQUESTs they have sent. □