In Link We Trust: BFT at the Speed of CFT using Switches

Lior Zeno Naama Ben-David Mark Silberstein

Technion – Israel Institute of Technology

Abstract

We introduce *SwitchBFT*, a novel BFT consensus protocol for data centers that matches the performance and fault tolerance guarantees of the fastest Crash Fault Tolerance (CFT) protocols. We take advantage of several unique properties of the trusted network that have emerged in modern data centers. SwitchBFT leverages *packet source authentication* to eliminate the overheads of cryptographic signatures, thus speeding up the fault-free scenario, and utilizes *network switch programmability* to enforce agreement decisions and to verify that safety is not violated, thereby offering robust performance even when some replicas are faulty. Designing a practical BFT that makes the most of these properties requires solving several challenges, such as packet losses and switch crash faults, all within the tight switch resource budget. We show that SwitchBFT outperforms state-of-the-art BFTs in scalability and performance, attaining the speed of NOPaxos, an in-switch CFT implementation.

1 Introduction

Byzantine Fault Tolerant (BFT) consensus protocols are a fundamental building block of modern data center systems. They are broadly used in traditional State Machine Replication [7, 10, 34, 35, 43], as well as in emerging blockchain applications [4, 42]. Moreover, resilience to Byzantine failures is an appealing approach to a system design, not only for defending against an active adversary but also for surviving complex failures.

Handling Byzantine failures is known to be inherently more costly than crash failures, however. In general, BFT consensus tolerating f failures can be solved with $3f+1$ replicas, i.e., about 50% more than $2f+1$ for the crash-only case, and imposes significant communication overheads. Furthermore, for many BFT protocols, even a single faulty replica might cause a dramatic performance drop compared to a normal (good-case) execution [13]. Thus, achieving BFT consensus at the cost of Crash Fault Tolerant (CFT) consensus is an unsolved challenge.

BFT protocols that assume the presence of *trusted components* [5, 9, 16, 19, 20, 35] make significant steps toward that goal. Such components can run simple code or maintain a limited state outside of the reach of an adversary, and can be either collocated with each replica (e.g., TrInc [16], A2M [9]), or centralized, e.g., in a network switch (NeoBFT [20]). A key advantage of trusted components is that they can prevent Byzantine parties from *equivocating*, i.e., sending conflicting messages, thereby allowing BFT solutions that, like CFT, only require $2f+1$ replicas.

However, many such protocols still suffer from low performance in the presence of faults, where a single faulty replica triggers a costly recovery mode (slow path), leading to drastic performance degradation. Most popular BFT protocols suffer from this phenomenon, and others give up fast-path performance to avoid it [13].

Additionally, all protocols with trusted components employ costly *cryptographic* primitives, such as signatures. These are used to provide the powerful *transferable authentication* property, allowing parties to verify that a message originates from a known source. This property together with a non-equivocation mechanism were proven to be *necessary* to reach CFT-level fault tolerance of $N \leq 3f$ replicas [12].

The signature verification, however, is a well-known obstacle to achieving high performance even in the fault-free fast path [3, 10, 35] due to its computational cost. We measured that verifying a 64-byte ECDSA signature is up to 3× longer than the network round-trip. These overheads are hard to avoid. Batching improves throughput but affects latency [10, 19, 43]. The recent uBFT [35] protocol removes verification from the critical path, but pays in latency for an extra communication round to commit. NeoBFT [20] gains speed via hardware acceleration but relaxes security guarantees.

In this paper, we introduce **SwitchBFT**, a practical BFT consensus protocol for data centers that achieves the performance of CFT: it obviates the need for cryptographic signatures, tolerates f failures among $2f+1$ replicas, and offers robust performance in the presence of failures. While SwitchBFT might seem to contradict the aforementioned im-

possibility result of Clement et al. [12], we claim that achieving these benefits is possible due to *unique properties of modern data center networks which can be harnessed to improve the performance of BFT protocols*.

First, we propose to use programmable switches [39] to *verify* BFT participants' behavior and *enforce* consensus decisions entirely in the data plane. This extends the *trusted switch model* from prior work beyond packet routing [35] or sequencers [20] to *full data plane programs*. A trusted switch that verifies and enforces participants' decisions in the data plane significantly simplifies the BFT protocol without introducing performance bottlenecks, as switch programs can operate at line rate. While a switch may fail due to hardware malfunctions, we believe it can be trusted to securely run data plane programs because they cannot be hijacked via the data plane, ensuring strong isolation from network adversaries such as data center tenants. Furthermore, switch programs can be further secured against control-plane modifications, ensuring execution integrity even in the presence of a privileged adversary (we explain our trust model in §2).

Second, we seek to leverage the *inexpensive source authentication* for every packet naturally provided in modern networks. This hinges on the fact that production networks prevent *spoofing*, that is, injecting packets with a forged source address.[1] Anti-spoofing has been shown to provide source authentication before [26], but we are the first to use it in the BFT context. Thus, a switch and replicas can verify the origin of a message by relying on the *network links* instead of cryptography.

Building a fast BFT using a trusted switch and in-network source authentication might seem trivial: A switch would store all the messages from the clients, and forward them to the replicas in order (i.e., as in [20]) for redundancy. The replicas then verify that the switch is the source, and recover any missed messages from the switch.

This solution is obviously impractical, however, as it would require a switch to have very large memory to store the messages. In reality, the switch memory is too small, and the switch program cannot access more than the first 160 bytes in a packet without costly recirculation [48]. Instead, SwitchBFT design is guided by the following principles:

Switch as a trusted leader. The trusted switch plays a central role in *coordinating* agreement. It executes the protocol entirely in the data plane to maintain trust.

Minimal in-switch computations. We restrict the protocol on the switch to perform only simple operations without costly cryptographic hashes [20] which otherwise require special hardware or offer weaker security guarantees.

Minimal in-switch state. We limit the state stored on the switch. In particular, the switch does not store the client messages, and the in-switch state does not grow indefinitely as the execution progresses.

[1] This applies to both physical and virtual networks, as discussed in §7.

Packet loss as a first-class concern. Packet losses are handled explicitly in the protocol. While in most BFT protocols packet loss is assumed to be rare, or handled by the transport protocol, switches do not run transport protocols and do not buffer packets for retransmission. Relying on a lossless link layer would constrain our deployment options.

Resiliency to switch failures. To ensure uninterrupted operation in the face of crash failures, the switch state must be replicated among multiple trusted switches. The replication protocol runs entirely in the data plane.

SwitchBFT upholds all of these design principles to yield a BFT protocol that is as fast as CFT. A switch maintains a global centralized view of the protocol decisions, preventing rollbacks of committed decisions. It sequences and relays messages between the participants, while maintaining a log of client-provided *collision-resistant hashes* [44] of every message. This allows replicas to retrieve missed messages from other replicas and verify their integrity. The log is periodically trimmed, keeping only a digest of the message history. Packet losses are handled in one additional round-trip, and only for a small subset of messages initiated by the switch. Clients and replicas use retransmissions as they can buffer the messages. With our design, a faulty replica *does not* affect the progress of non-faulty replicas. Last, we employ the ChainPaxos replication protocol [17] among multiple trusted switches entirely in the data plane [15, 27, 32] to withstand switch failures. A switch failure may invoke the 'slow path' in the BFT, but we show this to impose minimal overheads.

We formally prove that SwitchBFT is correct (§6), and implement it in P4 for Intel Tofino switches (1800 LOC), as well as the client and replica library in Rust. We run on up to four switches to evaluate crash fault tolerance among them.

We compare SwitchBFT with multiple state-of-the-art BFT protocols, as well as one of the fastest in-switch CFT consensus implementations, NOPaxos [31].

SwitchBFT matches the throughput and latency of NOPaxos for messages of up to 256 bytes and is within 14% of its throughput for larger messages, while supporting the same proportion of faulty replicas. Compared to NeoBFT [20], which also relies on a trusted switch, SwitchBFT achieves 6.6× lower tail latency (5.3× lower average) at the same throughput, and 10% higher overall throughput, despite NeoBFT using an FPGA-equipped switch. Compared to the less-secure HMAC-based version of NeoBFT, SwitchBFT achieves 3× lower tail latency (1.9× lower average) at the same throughput, with 10% higher overall throughput, all while supporting more faulty replicas. Under packet loss, SwitchBFT maintains the throughput of a loss-free execution for the realistic loss rate of up to 0.01%, and then gradually degrades by up to 33% and 27% at an extreme 1% loss rate with 2 and 3 non-faulty replicas, respectively. Finally, we analyze SwitchBFT's scalability, showing it can fully utilize the switch capacity, achieving 300Mop/sec request rate and scaling to replica groups as large as 8K.

2 Trust Model

SwitchBFT assumes that the switch may serve as a trusted component in the BFT protocol. While switches are not fail-proof, we believe they are a useful trust boundary. This is in line with several prior works: NeoBFT [20] explicitly assumes that switches and their data-plane programs are trusted. uBFT [35] implicitly trusts switches to not alter unsigned messages.

SwitchBFT further extends this assumption to trust data plane programs executed by the switch. Specifically, in a practical threat model that assumes adversarial data center tenants connected via network, data plane programs running in switches are harder to subvert than host software and offer protection against such an adversary, thus enabling the execution of trusted BFT logic.

"Air-gapped" control-plane network. Data-plane programs can only be installed via control plane operations. Notably, SwitchBFT participants are data center tenants that do not have access to the control plane. Moreover, the control plane network is isolated from the tenant network traffic, effectively creating an "air-gapped" in-switch execution environment to run trusted switch programs.

Firmware locking for trusted in-switch programs. To guarantee that the switch runs only authorized programs, data plane programs as well as switch firmware can be secured and locked by vendors. They require vendor authorization and/or physical access for updates.

Data plane vulnerabilities are difficult to exploit. We are unaware of attacks that engineer network traffic to adversarially modify in-switch data or cause control flow integrity violations in data plane programs. Data plane packets cannot modify in-switch tables or hijack the match-action pipeline executing data plane programs, as the code and data are segregated, and there are no potentially exploitable elements like stacks or branches. Thus, SwitchBFT data plane logic is protected from the protocol participants (clients and replicas).

We follow the common assumption that the trusted BFT components are implemented correctly, but we make no such assumption about the software running on replicas or clients. In addition, beyond the switches running SwitchBFT, we assume that ToR switches are trusted, as they are essential for source authentication.

3 Switch Programming Constraints

SwitchBFT uses protocol-independent switch architecture (PISA) switches. PISA switches are programmed using the P4 language [50], which provides access to switch memory through tables and stateful objects such as registers, meters, and counters. Registers are particularly powerful in P4 as they support both read and update operations (via RMW actions) in the data plane. But, each packet program is limited to *one access* to each register array. With *recirculation*, packets can traverse the pipeline multiple times, albeit with lower throughput reduced proportionally to the number of recirculations.

These constraints put significant restrictions on the programmer. First, the switch is incapable of executing complex cryptographic operations. For instance, even generating four low-security HalfSipHash HMACs [24] requires 12 pipeline passes [20]. Second, due to its restricted memory and inability to access the entire packet payload (Intel Tofino is limited to 160 bytes without recirculation [48]), storing complete messages in the data plane is not an option.

4 Design

We first discuss the main protocol assuming no switch failures, and then explain how to handle them in §4.8.

4.1 Overview

There are three main actors in SwitchBFT: clients, replicas, and a *trusted* switch. We assume that all communication goes through the switch and that up to $f < n/2$ replicas, and any number of clients, may be Byzantine. The actors and their communication links are shown in Figure 1. We assume a partially synchronous model with message losses. Clients initiate their operations by sending their message to the switch. The switch then assigns a sequence number to the message and distributes it to all replicas. Replicas store the received messages in a log, and then execute them in the order of their sequence numbers, eliminating the need for inter-replica communication. Once executed, the replica issues an acknowledgment to the client, including the response from the application.

However, the protocol is more subtle due to the possibility of packet loss. Sequence numbers allow replicas to recognize when they have missed a message, but do not help them to recover the lost message's contents. It is therefore possible that some replica has a *gap* in its message log, while another received all messages, leading to divergence in application state. Since we cannot prevent message loss, we must implement a way for replicas to recover lost messages, or agree on log gaps. We therefore allow replicas to execute *speculatively*, and, upon detecting inconsistencies, roll back their state, re-execute diverging operations, and send a new acknowledgment to the client. The client then considers an operation committed once it receives $f+1$ acknowledgments with an identical application response.

We present the protocol in two parts; in the first, we define a new primitive, we call *Verifiable Unreliable Authenticated Broadcast (VUAB)*, which ensures that replicas receive client messages in order, though possibly with some gaps, and allows them to check whether a message value was received by some replica in a given log slot. This helps prevent Byzantine actors from inventing fake client messages.

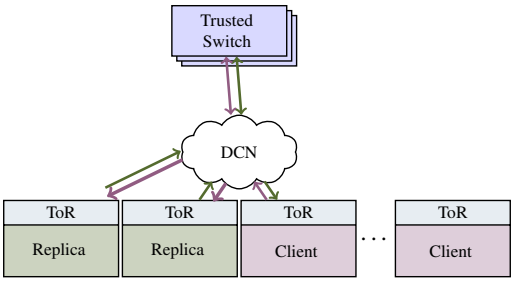

Figure 1: SwitchBFT deployment model: SwitchBFT is executed on any switch(es). ToRs provide packet source authentication. The switch processes all protocol messages. It broadcasts client requests to all replicas (purple), and unicasts acknowledgments to the requestor (green).

The second part of the protocol builds on VUAB and allows replicas that lost messages to recover message values (which they can then verify using VUAB), or agree that a certain log slot will be left blank in all replicas.

4.2 Verifiable Unreliable Atomic Broadcast

VUAB consists of three operations, broadcast(m), deliver(), and verify(m, k), intuitively allowing a client to broadcast a message to all replicas, a replica to deliver that message, and a replica to check whether the message m was the kth message to be delivered by other replicas. More specifically, the following properties define VUAB:

- **Validity:** In executions without message loss and message reordering, if a non-faulty client broadcasts a message m, then all non-faulty replicas deliver m.
- **Integrity:** For any broadcast message m, every non-faulty replica delivers m at most once.
- **Lossy Total Ordering:** For any two non-faulty replicas, if the k-th `deliver()` invocation returns a message, it is the same message.
- **Loss Detection:** If a message m is broadcast by a correct client then either (1) none of the correct replicas delivers m or $\perp$ or (2) every correct replica delivers m or $\perp$.
- **Verifiability:** `verify`(m, k) returns true if some non-faulty replica delivered m as its k-th delivery. Furthermore, if `verify`(m,k) returns true, then no non-faulty replica can ever deliver any non-$\perp$ message other than m in its kth delivery.

Unlike reliable broadcast, VUAB does not ensure the eventual delivery of messages from correct clients. Instead, VUAB delivers $\perp$ to signify message loss and guarantees that if at least one correct replica detects a message loss, all non-faulty replicas either receive the message or detect its loss.

The VUAB protocol. Just like the rest of our BFT protocol, the VUAB protocol relies on a *trusted* switch through which all messages are routed. Its pseudocode is presented in Appendix B. At a high level, the switch's job is to (a) assign a sequence number to each message that is broadcast, (b) store a *collision-resistant digest* [44] attached to each message to facilitate later verification, and (c) broadcast the message with its sequence number and digest to all replicas. Notably, the switch never computes the digest by itself, and never stores the entire broadcast message.

To broadcast a message, a client first calculates a digest, D of its message m, by using a hash function $h(m) = D$. It then sends a combined message $M = \langle m, D \rangle$, containing both the original message and its digest, to the switch.

Once the switch receives M, it stores the digest of the message and its assigned sequence number, S, in a local `digests` array, before sending $\langle M, S \rangle$ to all replicas. Each replica, upon receiving M from the switch, stores the message in a `pending` buffer, to be used for delivery. Before placing a message in the buffer, the replica compares its sequence number to the last sequence number it has seen, and appends $\perp$ messages to the buffer if it observes a gap. The replica also keeps track of the number of messages it has already delivered, and corrects the `pending` buffer if it received an out-of-order message and has not yet delivered a $\perp$ for that sequence number.

Before delivering, a replica first calculates the digest $D' = h(m)$, and compares it to the received digest D to ensure its validity. If the digests do not match, the replica discards the message and delivers $\perp$; otherwise, it delivers m.

To verify that a message m was the k-th message to be delivered by some replica, a replica retrieves the digest of the message with that sequence number from the switch, and checks if the digest of m matches the provided one.

4.3 Normal Operation Mode

Figure 2 shows SwitchBFT's fast path. The client uses VUAB to broadcast its messages, and replicas maintain a log with the messages they delivered through VUAB, in the order they delivered them. Once a replica delivers a client message, it executes the operation on the application, and sends back an acknowledgment to the client, along with the resulting application state. If a client receives $f + 1$ acknowledgments with the same application state, it can consider its operation to be committed. Note that this solves BFT SMR if packets cannot be lost. This highlights the power of the VUAB primitive, and in turn, the power of a trusted central switch.

To handle packet loss, our protocol also keeps track of the protocol execution even during normal operation. In particular, the switch tracks the number of acknowledgments from different replicas for each message, and replicas report the number of missing messages in their log when they send an acknowledgment. This information is used to ensure consistency when packets are lost, as we explain next.

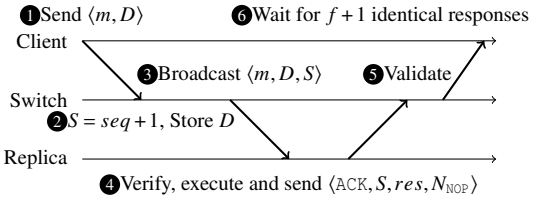

Figure 2: The flow of a client request (fast path).

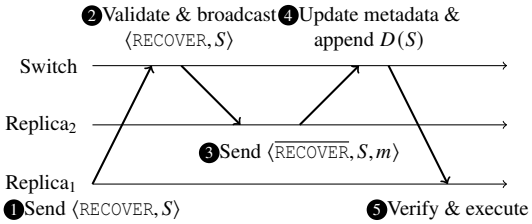

Figure 3: The flow for recovering a message #S.

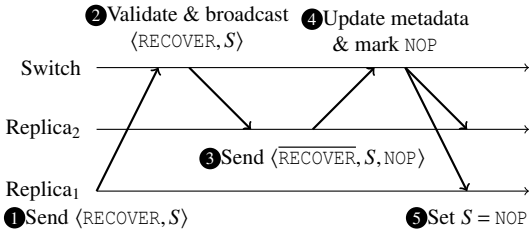

Figure 4: The flow for deciding a NOP.

4.4 Handling Packet Loss

When considering packet loss, we focus on one type of message: those forwarded by the switch from the client to the replicas. These messages pose a unique problem; since the switch has limited memory and *cannot* store the contents of such messages, it also cannot retransmit them in case of packet loss. We therefore implement a recovery protocol that allows replicas to agree on the content of log entries for which some (but maybe not all) replicas have not received the client message forwarded by the switch. We note that all other messages transmitted in our protocol *can* be stored and retransmitted, because their senders have a larger memory capacity.

If a replica delivers $\perp$ for some sequence number S, it triggers the *recovery* protocol (Figures 3 and 4) to try to retrieve the missing message. The replica buffers all other messages while executing the recovery protocol, deferring the processing of messages with larger sequence numbers until the missing message is resolved. Since the switch does not store any client messages, the recovery request, denoted $\langle \texttt{RECOVER}, S \rangle$, is sent to all replicas. Upon receiving a $\langle \texttt{RECOVER}, S \rangle$ message, a replica replies with a $\langle \overline{\texttt{RECOVER}}, S, \texttt{NOP}/m \rangle$ message, sending its own value for the requested sequence number: either the message m if it received it, or NOP otherwise. In principle, if the initiating replica receives at least one copy of a non-NOP message, then it can verify its authenticity using the verify operation of VUAB, and if valid, adopt that message. Otherwise, it waits to hear at least $f+1$ NOP replies, and then adopts a NOP value for that sequence number.

However, the protocol is in fact more complex, since we must ensure that all replicas agree on the same value (either NOP or the client's original message), despite asynchrony and potential equivocation. For this purpose, we rely on the switch, through which all messages are routed, to maintain transient state for the recovery protocol of each sequence number.

More specifically, the state maintained by the switch is used for two key purposes, outside of the state maintained for VUAB; first, it prevents equivocation by tracking which replicas have already replied to each request, and secondly, it stores the outcome of the recovery of each message and prevents any replica from deciding differently in the future.

Note that the equivocation is prevented in a *lazy* manner, i.e., the replica will succeed in sending an incorrect message, but it will not affect the protocol execution, making it more resilient in the presence of faulty replicas.

We now discuss each of these in more detail. The pseudocode of the protocol is found in Appendix B.

Mitigating equivocation and enforcing decisions. At a high level, to mitigate equivocation, the switch stores, for each replica r and message m, whether r has already replied to m, and if so, it drops any further replies to the same message m from the same replica r. More specifically, we consider three types of messages; *client requests*, *recovery requests*, and *commit requests* (which will be described in the next subsection). These are the messages for which the switch stores *non-equivocation metadata*.

In particular, for each client message m with sequence number S, the switch maintains a bit per replica, indicating which replicas sent an acknowledgment to the client for m. Once the switch sees at least $f+1$ acknowledgments from different replicas, it sets a flag indicating that the message with sequence number S has been *decided*, and that its decision is not NOP. The decision flag is stored in a History array, indexed by sequence number. Once such a decision is reached, if any replica initiates a recovery protocol for sequence number S, the switch drops any recovery-response messages with a NOP value; that is, the only possible outcome of the recovery is for the initiating replica to discover the value of the original message. This is guaranteed to eventually happen since there must have been at least one non-faulty replica that acknowledged the message, and that replica will eventually respond to the recovery request with the correct message.

Recovery messages are handled similarly, but require three states for each replica; no-reply, NOP-reply, or message-reply, with the latter two corresponding to whether the $\langle \overline{\texttt{RECOVER}}, S, * \rangle$ message from replica r contained a NOP or a non-NOP message, respectively. The state of all replicas starts as no-reply, and can only change once. Once $f+1$ replicas have sent a NOP response (are in the NOP-reply state) for

the recovery of sequence number S, the switch marks a NOP decision for that sequence number in the History array. Any further recovery request for that sequence number will receive a reply from the switch indicating the NOP decision, without forwarding the request to other replicas. It is possible that some replica r manages to recover the original message and completes its recovery operation, but in the meantime, the switch receives $f+1$ NOP responses and marks a NOP decision. This means that r will have an incorrect value in its log; however, this value will be corrected later, when r verifies its log's consistency, either when sending its next acknowledgment to the client, or when executing a log commit as described next.

NOP decision delays. Since packets can take time to arrive, initiating and replying to a recovery operation immediately after delivering a $\perp$ in the VUAB protocol may lead to many NOP decisions even without many packet drops. Thus, we add a timeout mechanism that prevents replicas from initiating or responding to a $\langle \texttt{RECOVER}, S \rangle$ operation before a certain amount of time has elapsed since they delivered sequence number S. Furthermore, replicas do not respond to $\langle \texttt{RECOVER}, S \rangle$ requests if they have not received either message S or message $S+1$. This is important for liveness for executions in which all messages arrive within a Δ delay.

4.5 Verifying Log Consistency

If a replica does not observe a dropped packet (never delivers a $\perp$ in the VUAB protocol), it never triggers the recovery protocol. It is thus possible that this replica has a message in slot S, whereas the other replicas agreed on a NOP in S (i.e., there were at least $f+1$ $\langle \overline{\texttt{RECOVER}}, S, \texttt{NOP} \rangle$ messages from different replicas). Such a case is not prevented because we rely on a simple majority to make decisions. The same problem may occur if the replica successfully recovered a message, but the final recovery decision was still NOP, as discussed above. As a result, the application state will diverge from the others, leading to incorrect responses to the client.

To prevent such outcomes, we implement a simple log consistency verification protocol. To verify its log's consistency, a replica sends a message to the switch indicating the number, $N_{\texttt{NOP}}$, of NOP decisions it is aware of so far in its log since the last trimming (discussed below). Note that the switch records *all* NOP decisions, and therefore the NOP decisions any replica is aware of is a subset of the NOP decisions on the switch. Thus, if the size of the set of NOP decisions is the same on the switch and the replica, the sets themselves are the same. Thus, if $N_{\texttt{NOP}}$ matches the number of NOP decisions recorded in the switch, then the replica's log is consistent. Otherwise, the switch replies with the list of slots for which there have been NOP decisions, allowing the replica to correct its log, roll back and reapply operations on its application.

Client acknowledgments with log verification. To ensure that the client receives correct responses from the replicas, and that once it receives $f+1$ acknowledgments with identical state, its operation is committed and the response will not change, we must ensure that non-faulty replicas only acknowledge client messages if their logs are consistent. We thus piggyback log verification on acknowledgment messages; when sending an acknowledgment to a client message, a replica r also sends the number, $N_{\texttt{NOP}}$, of NOP decisions it is aware of. The switch only forwards the acknowledgment to the client if r's log is consistent. Otherwise, if $N_{\texttt{NOP}}$ does not match the number of NOP decisions on the switch, the switch drops r's acknowledgment to the client, and instead sends a log-consistency-verification reply to r.

Furthermore, if replica r sends an acknowledgment for the message with sequence number k, the switch considers this to be an acknowledgment for *all* messages with smaller sequence numbers for which a NOP decision has not yet been reached. It updates all acknowledgment counts of messages with smaller sequence numbers accordingly, and, if any reach $f+1$ acknowledgments, updates the decision to non-NOP.

Together, the log-consistency-verification and acknowledgment updates ensure that if a client receives $f+1$ identical acknowledgments for some message with sequence number k, all messages with smaller sequence numbers have been decided, and therefore the log will not be rolled back.

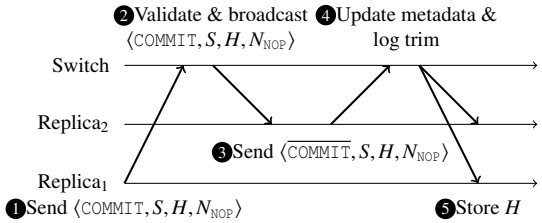

Figure 5: The message flow of the commitment protocol.

4.6 Commitment and Log Trimming

As described so far, the protocol requires the switch to maintain state for every client message in the whole execution history, which can quickly become unsustainable. We now describe a simple way to allow the switch to trim its log, called the *commitment protocol*, which replicas execute periodically. The flow of the commitment protocol is shown in Figure 5.

Once a replica has filled k log entries since its last commit (or since the beginning of the execution), reaching sequence number S, a replica r initiates the commit protocol. Intuitively, the replica does two things simultaneously: first, it verifies the consistency of its log, and secondly, it calculates a digest of its entire log to allow storing it more compactly. In more detail, when initiating the commit protocol, r calculates the digest of its history of messages up to S, H, and the number of NOPs, $N_{\texttt{NOP}}$, in the last k slots (since its last commit), and sends a $\langle \texttt{COMMIT}, S, H, N_{\texttt{NOP}} \rangle$ to the switch. The switch first verifies that $N_{\texttt{NOP}}$ matches the number of NOP decisions it has logged in its History array in the entries $S-k \ldots S$, and if

so, forwards the commit message to all replicas. Otherwise, it sends r the log-consistency-verification reply; it sends a message to r indicating that its commit is rejected, with the sequence numbers at which there is a NOP decision.

Upon receiving a commit message $\langle \texttt{COMMIT}, S, H, N_{\texttt{NOP}} \rangle$, each replica independently calculates the digest of its own history, and if in agreement (i.e., its own history digest matches H), sends a $\langle \overline{\texttt{COMMIT}}, S, H, N_{\texttt{NOP}} \rangle$ back to the switch. Note that if the history digests match, then necessarily the number of NOP entries matches as well. Upon receiving matching $\langle \overline{\texttt{COMMIT}}, S, H, N_{\texttt{NOP}} \rangle$ messages from $f+1$ different replicas, the switch replaces its History array for all entries up to S with the history digest H, and broadcasts this $\langle \overline{\texttt{COMMIT}}, S, H, N_{\texttt{NOP}} \rangle$ message to all replicas. Upon receiving a $\langle \overline{\texttt{COMMIT}}, S, H, N_{\texttt{NOP}} \rangle$ message from the switch, a replica checks whether H matches its log's digest up to S. If not, the replica queries the switch for the indices of the NOP decisions it missed, and executes a recovery protocol for any message it missed. For any recovery protocol triggered for some sequence number $S' \leq S$ at any point after the switch trims its log, the responding replicas send their entire log history,[2] and the switch sends the digest of the entire history H to verify the log's validity. Replicas can also query the switch to find the locations of NOP decisions that have been trimmed and the latest committed sequence number.

4.7 Optimizations

The protocol as presented so far solves BFT SMR using a central trusted switch despite packet drops. We now present a couple of optimizations that we implemented on top of it.

Overwrite messages. To minimize log divergences among the replicas, the switch can announce when a NOP decision is made for a certain sequence number. That is, it can send an '`overwrite(S)`' message to all replicas, letting them know that sequence number S should be overwritten to a NOP. Additionally, when a non-NOP $\langle \overline{\texttt{RECOVER}}, S, m \rangle$ is received by the switch, it can forward it, along with that message's digest, not only to the replica that initiated the recovery, but to all replicas. If the digest matches the message itself, this gives all replicas another chance to hear it. Furthermore, the replicas can then reply with an acknowledgment, speeding up the time until the message is committed, and reducing the chance that a NOP decision will be made.

Acknowledgment tracking. One of the main limitations of a P4 switch is that a P4 register can only be accessed once per packet. Therefore, accessing multiple indices of the same P4 register requires recirculation which leads to significant overhead, since they cannot be executed at line rate. In our protocol, all the calculations and updates that the switch executes only involve one update, and are therefore fast. The only exception is that, when processing an acknowledgment from a replica r to a client for sequence number S, the switch must update the state of all messages with sequence numbers smaller than S. However, this can be implemented in one memory update by keeping track of the *maximum* sequence number for which r sent an acknowledgment.

[2] There are various ways to reduce the size of such messages, including sending the history just starting at the sequence number being recovered, or sending the number of NOP decisions in the history.

4.8 Handling Switch Failures

As the switch is trusted, it is not susceptible to Byzantine failures; however, it may crash and lose its state. To address this, we replicate the protocol state in the switch using *chain replication* [52] across multiple trusted switches arranged in a chain and reconfigure the chain via Paxos upon switch failures similar to ChainPaxos [17]. The inter-switch replication and reconfiguration are implemented entirely in the data plane.

Packets that modify the state are directed to the head of the chain and subsequently propagated through the chain to the tail. Packets that read the state, access it from the tail switch. This approach ensures linearizability, i.e., any state update externalized (i.e., reaching clients or replicas) is fully replicated across all switches in the chain. While the switches in the chain may be in different states at a given time, this discrepancy does not violate safety because only the state of the tail switch is known to clients or replicas. Furthermore, all the switches in the chain possess the state of the tail switch plus some new updates that have not yet reached it.

SwitchBFT implements chain replication and Paxos entirely in the data plane, following the approach of NetChain [27], SwiSh [32], and P4xos [15], as follows. All the messages from the replicas and clients are sent to the head switch, they traverse through the chain and update the state in each switch, and then the tail switch sends them to their destination. This design allows for seamless failover to another switch replica in case of a head switch failure, without any loss of state, at the expense of slightly increased per-request latency as we show in the evaluation.

Chain reconfiguration is done via Paxos. Each switch monitors its successor in the chain through timeouts. When a failure is suspected, the monitoring switch sends a reconfiguration request to the head switch, which coordinates agreement among all switches on the new chain configuration. In the event of a head switch failure, the switches first run a leader election phase before reconfiguring the chain.

Note that reconfiguration can also be done via an external coordination service run by switches that are not part of the chain. When the chain self-reconfigures, $2f_s+1$ switches are needed to tolerate f_s switch crash failures, where f_s is the switch failure and is unrelated to the number f of faulty replicas participating in the BFT protocol.

Adding switches to the chain is also performed in data plane and done similarly to prior work [17, 32]. Switches can rejoin the chain by sending a reconfiguration request to the leader. Once the new configuration is decided, the switch is

added to the tail of the chain, where it requests the history from the previous switch. During synchronization time, the switch can participate and forward replication requests.

5 Analysis

Why source authentication is necessary. We rely on packet source authentication to establish mutual trust between the switch and replicas. Without this key primitive, Byzantine replicas may forge messages on behalf of other replicas or a switch. Thus, it allows replicas and the switch to authenticate the source of messages before they act on them. Note that anti-spoofing is not semantically different from any other authenticated BFT protocol, which is usually provided via digital signatures or MACs. In particular, it does not weaken or limit Byzantine behavior such as equivocation.

In-switch space and compute requirements. The switch only performs lightweight computations and never executes cryptographic primitives. Specifically, it only compares or performs simple arithmetic on integers.

The switch maintains two distinct types of state: the message digest history and the recovery/commit state. We give the breakdown of the size of the state, using the following variables:

- D – size of the digest (bits),
- R – number of replicas,
- H_L – history length before trimming (messages),
- S – sequence number or `NOP` count size.

The total history data is $(D+R+1)H_L+(D+S)$. The first factor is the number of bits required to store the digest and the bitset for each sequence number, plus a single bit for the `NOP` flag. The second factor is for the latest digest of the message history with the latest committed sequence number. We can reduce this further to: $(D+1)H_L+(D+S)+RS$ by replacing bitsets with per-replica message counters. For recovery and commit states, we require a tristate array of size R, the sequence number to commit/recover, the `NOP` count and the commit block digest. This sums up to: $3R+D+2S$.

The total space is expressed as: $(D+1)H_L+(D+S)+RS+3R+D+2S$ For the following values $H_L = 64K$, $D = 256$, $R = 8K$, $S = 32$, the total space requirement is: ~16Mbits. This fits within the switch data plane (§9.2).

The dominant factor here is the message digest history. We choose H_L such that it is big enough to hide the commitment latency, i.e., replicas always have requests to process during log trimming. This is because when replicas cannot finish the commit procedure and the switch runs out of space, it stops accepting requests as switches do not support dynamic allocations. In our experiments, we achieved full system throughput with a message digest history of size $H_L = 512$.

Scalability. A single Tofino switch pipeline can handle up to 1.2 billion packets per second [21], surpassing the throughput of a commodity CPU by several orders of magnitude. Therefore, SwitchBFT is designed to run on a single switch pipeline. The switch becomes a scalability bottleneck only with extremely large replica groups, where space or bandwidth constraints become the limiting factor. For instance, achieving such bottleneck conditions would necessitate a replica group of about 5K replicas if each processes packets at a rate of 220Kop/sec as we measure in our evaluation.

Replicas process a linear number of messages for commitments and recoveries, but handle only one packet in the fast path, regardless of the number of replicas. Further, the switch strives to reduce the message burden by aggregating responses during the recovery and commit phases, e.g., a replica only hears from the switch after $f+1$ `NOP` votes.

Read operations. SwitchBFT does not use cryptographic signatures, and therefore arbitrary reads need to be performed as quorum reads as outlined in §4.3. Reads may be optimized further by utilizing the switch, but we do not explore it in this work.

Slow path behavior. In SwitchBFT, the slow path can be seen as the need to recover lost packets, since that increases a replica's time to commitment by a round trip. However, only the replica that lost the packet is affected. The recovery takes a little longer if many non-faulty replicas all lost the *same* message, in which case the switch needs to wait to aggregate enough `NOP` votes before the recovery completes. Even in this scenario, this process is significantly faster than the multi-round view change protocols [10, 43], or even the message recovery protocol in other BFT algorithms [20].

Effect of Byzantine replicas. Byzantine replicas in SwitchBFT *cannot trigger the slow path*. Their only recourse is to attempt to diminish system throughput by consistently voting for `NOP` during recoveries. However, this tactic is equivalent to a Denial-of-Service attack, from which BFT protocols do not protect today. Fortunately, these irregularities may be readily identifiable by the switch, which can swiftly mitigate suspected malicious behavior from any replica.

Interplay between switch failures and BFT performance. When a non-head switch fails, replicas may face extra work post-recovery due to gaps in sequence numbers. These numbers continue to get increased by the chain head, but packets are not forwarded to the replicas during the failure. As a result, replicas would have to agree on many `NOP`s. To minimize this overhead, we propose two optimizations. First, reducing the message digest history length can limit the number of `NOP`s replicas must agree on. The minimal length of the message digest history is 512, which translates into about 33msec to recover the protocol in the worst case assuming 65μs per `NOP` decision. We measure the `NOP` decision latency in §9.2. In addition, the recovery protocol could be extended to support sequence number ranges, allowing replicas to batch recovery messages. This is left for future work.

6 Correctness Proof Sketch

In this section, we briefly sketch the correctness argument for our protocol. The full proof is in Appendix C.

Theorem 1 (Safety (Informal))**.** *Once a client receives $f+1$ identical acknowledgments, (1) the acknowledgment value represents the application state immediately after executing its most recent operation (2) the application will never be rolled back beyond this state.*

We begin by arguing that our VUAB protocol is correct.

Lemma 1. *The VUAB implementation is correct.*

Sketch. Since the switch is correct by assumption, it always forwards the message it receives from the broadcaster, with a unique sequence number and having stored the digest that the broadcaster sent. Note that non-faulty processes only deliver non-$\perp$ values that were received from the switch and whose digest matches their locally computed hash. Furthermore, a correct broadcaster always sends its message with a correctly calculated digest that corresponds to the message's hash. Therefore, validity and integrity are maintained. Due to the way the switch assigns sequence numbers, loss detection and lossy total ordering are maintained as well. Finally, due to digest verification before delivering each message, the verification property is also guaranteed. □

Note that a non-faulty replica r's acknowledgement for a message with sequence number S contains the result of applying the message in r's S'th log slot to the application. By the correctness of VUAB, the message that r applied to the application is the client's original message. Since the client waits for $f+1$ identical acknowledgments, at least one of them must be from a non-faulty replica. Therefore, part (1) of the safety property holds.

To see why part (2) holds, consider the state of each message with $\leq S$ in the switch, after a client receives $f+1$ identical acknowledgements for S. We define a message m with sequence number S to be *decided* if the switch has seen either (a) at least $f+1$ acknowledgments for m from different replicas (called *message-decision*) or (b) at least $f+1$ $\langle \overline{\texttt{RECOVER}}, S, \texttt{NOP} \rangle$ replies (*NOP-decision*). Note that once a message is decided, its decision cannot be reversed, and any replica that either tries to recover it or executes a log consistency verification will discover its decided value.

Recall that an acknowledgment for S is considered by the switch as an acknowledgment for every message before S. Thus, at the time at which the client receives $f+1$ identical acknowledgments, all messages with sequence number $\leq S$ have been decided (potentially a NOP-decision). Recall also that when acknowledging a message, replicas also execute a log-consistency verification, and the acknowledgment is dropped if the verification fails. Therefore, at least one of the $f+1$ acknowledgments received by a client must have arrived from a non-faulty replica with a consistent log, and therefore no message before S will be rolled back in that replica's log.

Theorem 2 (Liveness (Informal))**.** *If no packets are dropped and all replicas receive a client message m within a timeout Δ, then m will be committed.*

The liveness of the protocol stems from the way that the recovery protocol is implemented. In particular, if no messages are lost and all arrive within Δ timeout, no non-faulty replica will initiate a recovery operation for any message, and any non-faulty replica that responds to a recovery operation for any sequence number S will send the message value (non-NOP). Thus, at most f NOP responses can be collected for any recovery, and therefore no NOP decision will be reached.

7 Discussion

Deployment model. As depicted in Figure 1, SwitchBFT does not assume replicas are physically connected to the switch(es) running SwitchBFT. It only requires observing all protocol traffic among the replicas, and between them and the clients. This is enforced via source authentication, because replicas reject the protocol messages not originating at the SwitchBFT switch. Thus, network-wide anti-spoofing ensures that the switch address is unique and cannot be forged.

For the design of network-wide source authentication,[3] we take a similar approach to that proposed in previous work [47, 57]. Given that the physical topology within the datacenter changes infrequently [46], we maintain an accurate mapping between ingress ports and corresponding server IP addresses at each ToR switch. When a packet arrives with an IP address that does not match the mapped server, the switch discards it, effectively preventing packet spoofing across the network.

Since SwitchBFT does not rely on reliable broadcast, the VUAB mechanism can be realized entirely by the switch's multicast engine. VUAB is scalable due to its best-effort nature, with no packet delivery guarantees. From the control plane perspective, the switch can maintain large multicast groups and large number of such groups (up to 64K) [22]. From the data plane perspective, VUAB is unreliable and as discussed in §5, lost messages only stall a specific replica, while the slow path is triggered only if many replicas lose the *same* message.

Virtualized environments. SwitchBFT will function without modifications in virtualized environments but requires several minor additions to the system for correct operation.

To work properly, SwitchBFT poses the following additional requirements. First, the physical switch should implement anti-spoofing at the virtual network layer, as the hypervisor is untrusted. Second, the switch must expose a virtual IP to be accessible to the VMs. Last, each physical machine may

[3] Anti-spoofing mechanisms are part of modern L3 switches [11, 41].

Platform	Hardware capability		
	Deep parsing	Packet replication	Stateful operations
Intel Tofino [21]	160B	Yes	Yes
NVIDIA Spectrum [40]	512B	Yes	Yes (needs locking)
Juniper Trio [56]	200B	Yes	Yes (needs locking)
Xsight Labs X2 [54]	256B	Yes	Yes (needs locking)

Table 1: Comparison of switch platforms by support for SwitchBFT's key data plane features.

only run a single SwitchBFT replica, as a Byzantine hypervisor may compromise the network among the VMs, turning the replicas on the same host to Byzantine all at once. In fact, this requirement also applies to CFT, since co-locating replicas on the same physical resources does not improve fault tolerance.

Accelerating signatures on SmartNICs. Accelerating signature logic is challenging due the inherently sequential computations. While it is possible to trade power and area for additional throughput, lowering the latency of modern x86 CPUs running at high clock frequencies is hard [2, 18]. A recent effort, TNIC [18], proposes a trusted NIC architecture that supports HMAC offload and implements a BFT consensus protocol. However, the latency of HMAC computation on TNIC is 2× higher than an x86 implementation. Even with further optimization, the TNIC protocol still requires cross-replica communication to commit messages and remains vulnerable to Byzantine behavior due to its leader-based design.

Comparison to NOPaxos. The key design differences between SwitchBFT and NOPaxos stem from the failure model. In NOPaxos, replicas are assumed correct, so a lost message can be retrieved directly from a replica without additional verification. In contrast, because replicas in SwitchBFT may be Byzantine, their responses cannot be trusted without validation. To ensure correctness, we introduce the verifiable property to our VUAB protocol. Furthermore, since replicas may equivocate and we do not rely on signatures, the trusted switch orchestrates the recovery protocol. Finally, because the switch maintains digests of client messages, a log trimming mechanism is required. To this end, we introduce the commitment protocol, which is executed periodically by replicas and enables the switch to trim its log while preserving correctness.

SwitchBFT on other switch architectures. SwitchBFT's data plane requires three key hardware capabilities: (1) deep packet parsing and header modification, (2) stateful memory operations, (3) packet replication for multicast. To understand how these requirements map to other platforms, we analyze several switch architectures, including NVIDIA Spectrum [40], Juniper Trio [56], and Xsight Labs X2 [54]. The capabilities of each platform, as relevant to SwitchBFT, are summarized in Table 1.

All analyzed platforms support deep packet parsing and multicast replication. For example, the Xsight Labs X2 switch can parse up to 256B of headers and replicate a packet up to 8K times [53].

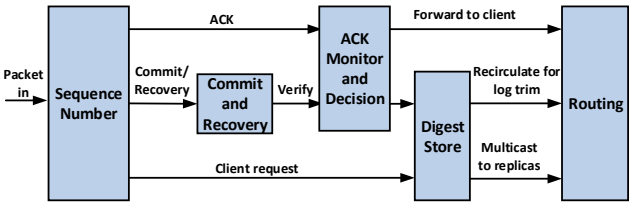

Figure 6: Logical pipeline view of the SwitchBFT P4 implementation.

Stateful operations are also supported on all switches, but performance may vary due to memory locking. In SwitchBFT, the most critical flow involves incrementing the sequence number, which requires a read-modify-write on the same memory location per client request. Yet, this operation runs at the rate that is determined by a single replica performance, which in our system is approximately 220Kop/sec. This rate is rather low for a switch increment operation even with locking, therefore unlikely to become a bottleneck. Updates to the digest store can be parallelized, since they involve independent writes to different memory locations distributed across different memory banks. The recovery and commit protocols execute at relatively low rates and do not impose significant load on the switch.

8 Implementation

We prototype SwitchBFT's switch, packet source authentication and all the optimizations from Appendix §4.7, on a Tofino programmable switch in $P4_{16}$ [50] (~1800 LOC). The client and replicas are implemented in Rust by extending NeoBFT's publicly available code [20]. We replicate the switch's state using chain replication and Paxos for reconfigurations, with no control plane involvement, as demonstrated by previous work [15, 27, 32]. Additionally, we implement NOPaxos [31] in P4, and the client and replica code in Rust.

The implementation presents challenges due to several factors, including the significant register footprint of storing 256-bit digests on the switch, which consumes a third of the available stages, and the extended data dependencies between protocol messages.

Our implementation avoids blocking during the `COMMIT` procedure. Specifically, when initiated by a replica, we opt not to block recoveries for sequence numbers within the committed digest block. Instead, we validate the initial state of the `COMMIT` procedure at decision time. For instance, if a `NOP` decision was made for a sequence number before commit completion, the commit fails upon receiving the $f+1$th $\overline{\texttt{COMMIT}}$ by the switch. This optimization not only facilitates implementation but also prevents Byzantine replicas from obstructing recovery procedures by issuing spurious `COMMIT`s.

P4 implementation details. As shown in Figure 6, the core data structure is the digest store, comprising of 8 register

CPU	2× Intel Xeon Silver 4216 CPU @ 2.10GHz (16 cores per socket)
NIC	2× Intel E810 100Gbps NIC
Switch	2× EdgeCore Wedge 100BF-32X (Intel Tofino 1)
OS/Kernel	Ubuntu Server 20.04 LTS / 5.4.0-131-generic

Table 2: Hardware and software details of the testbed.

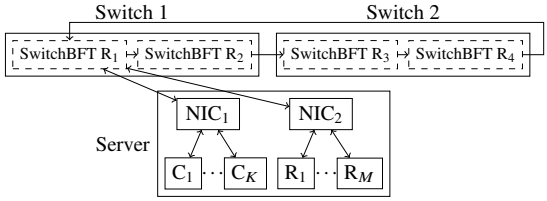

Figure 7: Testbed topology.

arrays, each with 32-bit elements. This width is the maximum that can be read and written in a single stateful ALU operation. The digest store can hold up to 64K entries, distributed across four pipeline stages—double the size of the commit block. We logically divide the register store into two windows: one stores the digests of incoming client requests, while the other is used for commitment by the replicas. If the digest store becomes full, the switch blocks new client requests. Upon completion of the commit protocol, the windows are swapped.

To address the dependency between commit and sequence number allocation, since each register can be accessed once by each packet, the final commit packet is recirculated to update the latest committed sequence number, which is stored at the beginning of the pipeline.

Each client request is assigned the next available sequence number, and its digest is stored in the digest store at the corresponding indexed entry. The request is then sent via multicast to all replicas using the switch's multicast engine. Finally, generated packets are updated with per-replica routing information in the egress pipeline.

Switch resource utilization. We report the switch resource utilization of our prototype on a Tofino switch in Appendix A.

Limitations. The current prototype supports up to 8K replicas, a digest store size of up to 64K, and digests of up to 256 bits. In addition, for simplicity, we assume FIFO ACK delivery from replicas, so we replace bitsets with a single counter per-replica to track unique ACKs from replicas.

9 Evaluation

Setup. Table 2 and Figure 7 summarize the hardware and network topology used in the experiments. We run all the replicas and clients on a single machine, each in its own network namespace, while forcing all communications through the switch.[4] Each replica is pinned to two physical cores on the server. There are two physical switches, each with two independent processing pipes, thus each acts as two switches. The switches are configured to form a replication chain. Since in the topology, the head switch is the only one connected to the clients and replicas, each packet traverses it twice, the first time running SwitchBFT logic, and the second only to forward to the destination.

Unless stated otherwise, we configure the system to tolerate the failure of a single switch. To achieve this, we use two pipes from the first switch and one pipe from the second switch as three independent switches. In total, the system operates with three switch replicas in a self-reconfiguring chain.

Baselines. We compare SwitchBFT against the following baselines: Unreplicated (a single replica), PBFT [10], Zyzzyva [43], Zyzzyva with one Byzantine replica (Zyzzyva-F), MinBFT [19] (with emulated SGX overhead, as done in NeoBFT [20]), HotStuff [34], and Neo-HM (with a low-security HMAC) and Neo-PK (that requires an FPGA in a switch) which run code in a switch [20]. Since we do not have the switch with an FPGA necessary to run Neo-PK, we use a fixed signature for every client request, let replicas verify it to emulate the associated CPU overhead, but always assume a success. Batching is supported by all baselines, except for Neo-HM and Unreplicated. Unless stated otherwise, we run all protocols using the minimum number of replicas to tolerate a single Byzantine failure.

SwitchBFT configuration. We set the commit block size to 32K. Thus, a single experiment at the maximum message rate encounters about 7 in-switch digest log trimming/sec. Trimming is performed in all the experiments.

9.1 End-to-end Performance

Protocol performance. We run a replicated `Echo` server where each replica sends back the message received from the client, thus highlighting the performance without application-related CPU load.

Figure 8 shows the throughput vs. 99th percentile latency. SwitchBFT reaches the same maximum throughput as NOPaxos. Its latency is within the measurement error (1%) compared to NOPaxos when approaching the maximum throughput at 213Kop/sec, and about 3% higher at low load. This is expected: it adds digest verification per message and the overhead of switch replication, both only a few μsec.

Compared to Neo-HM, SwitchBFT has 20% lower latency under light load, as Neo-HM requires packet recirculations for computing HMAC in a switch. This also translates to 10% higher throughput at the latency at which SwitchBFT reaches ~220Kreq/sec. We note that Neo-HM uses a low-security hash function and requires $3f+1$ replicas.

Neo-PK incurs much higher latency because it buffers multiple messages before processing them. Without batching, verifying every request would limit the maximum throughput to ~15Kop/sec given the 2 CPU cores we allocate for each replica. Zyzzyva and MinBFT also use batching to offset the

[4] Each namespace has a MACVLAN interface in VEPA mode [30].

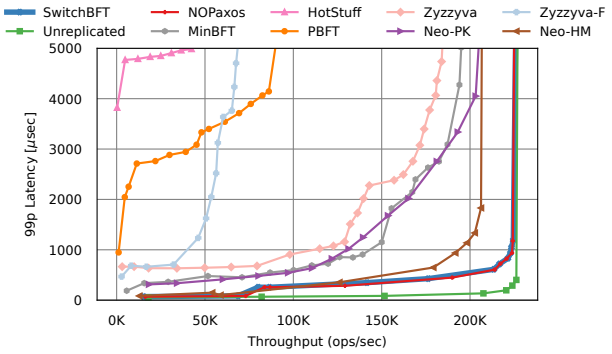

Figure 8: 99p Latency-throughput of all evaluated systems for the replicated `Echo` server. SwitchBFT outperforms all BFT baselines, and matches the performance of NOPaxos.

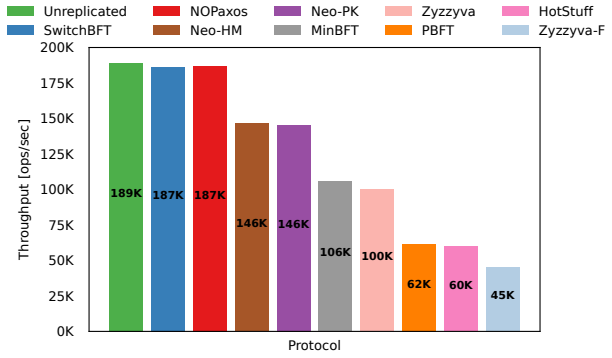

Figure 9: The maximum throughput of a replicated KV store.

signature overhead. Finally, HotStuff, PBFT, and Zyzzyva-F are much slower, as they require more communication rounds in the fast path and incur signature-related overheads.

Key-Value store. We run a BTree-based Key-Value (KV) store as in NeoBFT [20], with 32B/128B keys/values, 1:1 `GET` to `SET` ratio, and 100K distinct keys. As shown in Figure 9, SwitchBFT's throughput is the same as NOPaxos, and only 1% lower than Unreplicated. Neo-HM and Neo-PK outperform the other baselines, but the rest are slower because batching is less effective for larger payloads.

9.2 Analysis

Scalability. The scalability of a BFT system is typically evaluated along two dimensions: increasing the number of clients and their request rates, and increasing the replica group size. The former highlights the maximum throughput of the BFT core mechanisms, and the latter demonstrates the efficiency of the I/O substrate. Ideally, we would like to characterize how SwitchBFT behaves across both dimensions. However, a full end-to-end scaling experiment cannot sufficiently stress the switch hardware, since the maximum throughput is inherently limited by the single-threaded CPU performance of an individual replica. As we show below, saturating the switch would require thousands of CPU replicas.

Therefore, to evaluate SwitchBFT under stress, we implement replicas directly on the switch. In this experiment, SwitchBFT logic is unchanged, but replicas are implemented in P4 and execute a simple `Echo` server on a switch. This setup allows us to measure the maximum throughput SwitchBFT can reach with unrealistically fast replicas. One switch pipeline executes the SwitchBFT protocol, while the other pipeline runs the replicas. The client is executed on the switch too, by sending requests using the switch's packet generator. It transmits minimum-sized requests (78B) at line rate (100Gbps). We use another switch to double the request rate.

The switch has a fixed processing capacity, which is shared among client requests, per-replica requests, and their acknowledgments. This creates a tradeoff: allocating bandwidth to maximize the client request rate requires fewer replicas, whereas increasing the replica group size stresses the multicast engine but reduces the achievable request rate.

To explore both dimensions, we evaluate two configurations: (1) 3 replicas, the minimum for BFT with one failure, and (2) 8K replicas, the maximum supported by our prototype. In the 3-replica experiment, two clients generate load while each replica receives requests through four distinct ingress ports. This setup introduces out-of-order arrivals, and to stress-test SwitchBFT's in-switch logic, we configure replicas to acknowledge requests without necessarily processing all preceding ones, while still enforcing in-order commit triggers. This design makes replicas respond faster, thereby increasing the load on SwitchBFT. Under this workload, SwitchBFT reaches a throughput of 300Mop/sec, fully utilizing the switch's packet-processing capacity.

In the 8K-replica experiment, SwitchBFT also saturates the switch's processing capacity, achieving the throughput of 135 Kops/sec. This experiment shows that SwitchBFT can easily scale to such large replica group sizes, and is constrained by the available in-switch bandwidth alone.

Finally, we validate our scalability analysis (§5) by fixing the request rate at 220Kop/sec and measuring the maximum replica group size the switch can sustain without packet loss. We find that the system supports up to 5K replicas, matching our analysis, and the throughput degradation observed when scaling to 8K replicas follows the expected linear trend, consistent with the bandwidth tradeoff discussed above.

Performance vs. message size. We measure performance while varying client message sizes, and also evaluate the performance impact of different hash functions for producing the message digest. These include 2 SHA256 implementations – the standard SHA256 [51], and an optimized SHA256 implementation utilizing AVX instructions – as well as BLAKE3 [23, 45]. The results are shown in Figure 10. For messages under 256B, all the implementations achieve similar throughput. However, for larger messages, SwitchBFT is up to 14% slower than NOPaxos with the fastest BLAKE3 hash function. This highlights that SwitchBFT works with any

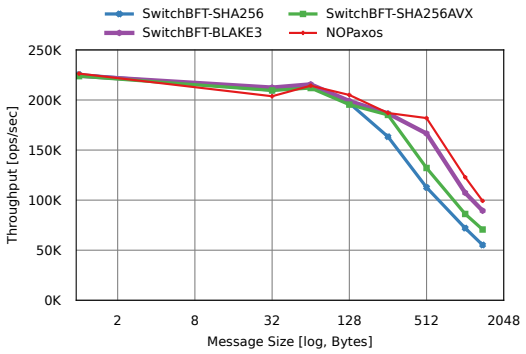

Figure 10: Throughput vs. message size for different hash algorithms to compute message digest. Higher is better.

#switches	1	2	3	4
Latency [μsec] (90/95/99p)	56/60/72	58/62/72	60/63/73	62/66/76

Table 3: Client-observable latency for a different number of switches in the data plane state replication chain. Adding three switches to the chain adds only 6 μsec.

hash function as switches do not execute one, unlike NeoBFT.

Latency overhead of multi-switch state replication. We vary the number of switches in the data plane state replication chain from 1 (no redundancy) to 4 and measure the latency of a single client request. To achieve strong consistency, all the messages from the clients and replicas traverse all switches in the chain from the head to the tail, and only then reach their destination. Table 3 shows that each switch in the chain introduces an additional latency of 2μsec at 90p.

Performance under switch failure. We run a chain with 4 switches and measure the performance when the head switch fails. Recovering from a leader switch failure requires electing a new leader and running Paxos to agree on the new chain configuration. We set the heartbeat timeout between the switches to 50msec and the client retransmission timeout to 10msec.

We start with four active switches in the chain. We run SwitchBFT at maximum throughput, and then simulate a failure by reconfiguring the head switch to function solely as a forwarding switch. The results are shown in Figure 11. When the head switch fails, throughput immediately drops to zero, as expected. After the timeout, the tail switch detects the failure, and the switches run the leader election protocol and agree on the new chain configuration. During this period, client requests are not forwarded to replicas by the chain. The recovery process completes right after the failure is detected. We validate this result by rerunning the experiment with a 30msec heartbeat timeout between the switches.

We also evaluate the worst-case scenario where the head switch fails after a middle switch failure. In this simulation, the failed head switch continues to increment sequence numbers, leading to gaps that require replicas to agree on NOPs after recovery. This added overhead doubles the duration of

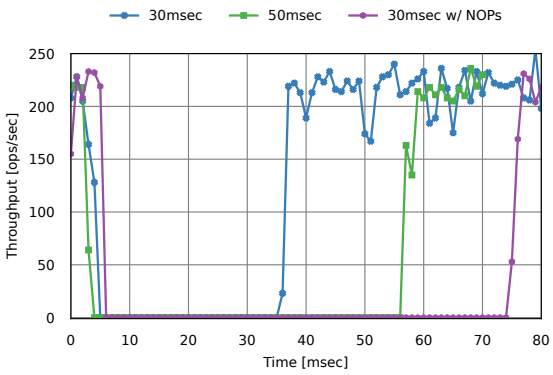

Figure 11: SwitchBFT's worst-case throughput under the chain head or middle switch failure for different timeout values.

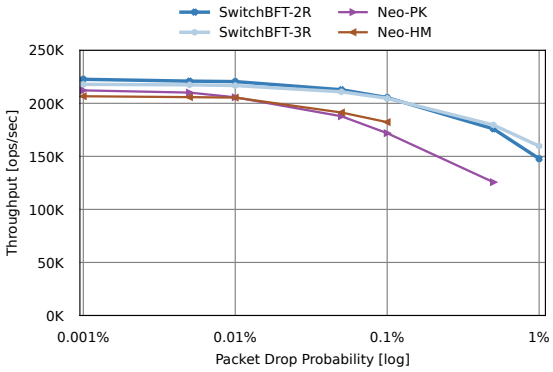

Figure 12: Throughput vs. packet drop probability, with two and three replicas in SwitchBFT compared to NeoBFT.

zero client-observable throughput. Replicas had to agree on 535 NOPs, taking 35ms (averaging 65μs per NOP agreement). This occurs because the process is fully serial, and can be reduced by batching recovery requests.

Resilience to packet drops. We vary the drop probability from 0.001% to 1% and report the throughput with one or two failures, denoted as SwitchBFT-2R (2 replicas), and SwitchBFT-3R (3 replicas).

The results are shown in Figure 12. SwitchBFT maintains consistent throughput irrespective of the replica group size. It retains ~93% of its throughput even at drop probabilities as high as 1:1000. However, at a drop ratio of 1:200, the throughput is reduced by approximately 20%. In addition, as expected, SwitchBFT-3R performs better at larger drop rates, because with more replicas, the probability of losing the same message in all replicas is lower, so the are fewer periods in which replicas are blocked while deciding on NOP.

Neo-HM's throughput cannot be measured beyond the drop probability of 1:1000, as the requests were dropped by all replicas and deemed unrecoverable due to the lack of the consensus recovery protocol in the author's implementation.

At a drop ratio of 1:1000, Neo-HM throughput suffers a 12% decline. In contrast, Neo-PK exhibits significant throughput degradation with more drops because replicas need to verify the signatures of lost messages before resuming operation.

10 Related Work

BFT consensus with trusted components. Utilizing trusted components to mitigate equivocation has been a longstanding line of research [5, 6, 8, 16, 18–20, 28, 33, 36, 55]. TrInc [16] and MinBFT [19] utilize trusted counters for message sequencing, A2M [9] utilizes append-only logs, TNIC [18] utilizes trusted NICs for message authentication. NeoBFT [20] and uBFT [35] leverage central trusted components, using a network switch and disaggregated memory respectively. However, none of the above reach the performance of CFT protocols in the fast path, and all suffer from a significant performance drop in the slow path.

Comparison with in-switch BFT. NeoBFT relies on a trusted network switch similar to SwitchBFT. It has two versions: a less secure one with a low-security HMAC computed in the switch, and a secure one with a trusted FPGA for computing signatures. The following table shows the benefits of SwitchBFT, in particular, support for more failures, faster `NOP` agreement, and no verification costs. Further, SwitchBFT is more resilient to Byzantine failures because it has no leader replica hence no costly view changes.

	NeoBFT [20]	**SwitchBFT**
Replicas	$3f+1$	**$2f+1$**
`NOP` agreement	PBFT	**2 message rounds**
Crypto-based authentication	Yes	**No**
Fast-path com. rounds	1	1
View change triggered by	Switch failure Byzantine leader	**No**

BFT without signatures. PBFT [10] proposed an optimization for their protocol that uses authenticators, HMACs vectors, instead of signatures. Still, it requires more replicas and communication rounds than SwitchBFT to commit a request.

Other works have studied the information-theoretic setting, in which no cryptography is used at all [1, 14, 38]. Most remain theoretical and require $3f+1$ replicas to tolerate f failures.

Clement et al. [13] showed that both signatures and equivocation prevention are needed to solve BFT agreement with $2f+1$ replicas. However, it only considers local trusted components. SwitchBFT shows that neither is necessary with a central trusted component, even with limited resources.

Accelerating the slow path. Aardvark [13] proposed a design for BFT protocols by avoiding optimizations that decrease the slow-path performance. There has been a recent surge of interest in *leaderless* protocols that avoid the slowdown due to leader failure [29, 37, 49, 58]. SwitchBFT achieves similar robust performance by relying on the trusted switch to coordinate agreement, without a leader. Thus, Byzantine failures do not affect the system's performance. Despite using a switch as a lightweight leader, the switch failover time is extremely quick.

In-network primitives. Strengthening network guarantees through in-network primitives has been extensively studied. For instance, NOPaxos [31] and Eris [25] demonstrated the efficacy of in-network sequencing in developing faster CFT and distributed transaction systems. Additionally, NeoBFT [20] illustrated the integration of in-network sequencing with message authentication. SwitchBFT builds upon this line of work and leverages another in-network primitive, packet source authentication, in its design.

11 Conclusions

SwitchBFT achieves performance that matches the state-of-the-art CFT protocols while tolerating Byzantine faults. By leveraging trusted switches and in-network packet source authentication, largely overlooked in prior BFT protocols, SwitchBFT offers robustness and efficiency in data center environments. Formal proofs and extensive evaluations demonstrate SwitchBFT's correctness, superior performance, and resilience to packet loss and switch failures.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Vincent Liu, for their insightful comments and constructive feedback. We gratefully acknowledge support from Israel Science Foundation (grants 1998/22 and 3673/25).

References

[1] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 381–391, 2022.

[2] Rashmi Agrawal, Ji Yang, and Haris Javaid. Efficient FPGA-based ECDSA Verification Engine for Permissioned Blockchains. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 148–155. IEEE, 2022.

[3] Aguilera, Marcos K and Ben-David, Naama and Guerraoui, Rachid and Papuc, Dalia and Xygkis, Athanasios and Zablotchi, Igor. Frugal Byzantine Computing. In *35th International Symposium on Distributed Computing*, 2021.

[4] Amazon. Blockchain on AWS. https://aws.amazon.com/blockchain/.

[5] 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*, pages 222–237, 2017.

[6] Naama Ben-David and Kartik Nayak. Brief Announcement: Classifying Trusted Hardware via Unidirectional Communication. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 191–194, 2021.

[7] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[8] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing Memory between Byzantine Processes using Policy-Enforced Tuple Spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):419–432, 2008.

[9] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

[10] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[11] Cisco. Port ACLs (PACLs) and VLAN ACLs (VACLs). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/vacl.html, 2023.

[12] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 301–308, 2012.

[13] Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. USENIX Association, April 2009.

[14] Tyler Crain. Two more algorithms for randomized signature-free asynchronous binary byzantine consensus with $t < n/3$ and $o(n^2)$ messages and $o(1)$ round expected termination. *arXiv preprint arXiv:2002.08765*, 2020.

[15] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking*, pages 1–13, 2020.

[16] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, April 2009. USENIX Association.

[17] Pedro Fouto, Nuno Preguiça, and Joao Leitão. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 575–592, Carlsbad, CA, July 2022. USENIX Association.

[18] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. TNIC: A Trusted NIC Architecture. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1282–1301, New York, NY, USA, 2025. Association for Computing Machinery.

[19] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine Fault Tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

[20] Guangda Sun, Mingliang Jiang, Xin Zhe Khooi, Yunfan Li, and Jialin Li. NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 239–254, 2023.

[21] Intel. Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[22] Intel. Tofino Native Architecture. https://github.com/barefootnetworks/Open-Tofino.

[23] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn. BLAKE3. https://github.com/BLAKE3-team/BLAKE3.

[24] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[25] 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*, pages 104–120, 2017.

[26] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable Network Support for Secure RDMA Systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2585–2600, 2022.

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[28] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308, 2012.

[29] Karolos Antoniadis, Julien Benhaim, Antoine Desjardins, Poroma Elias, Vincent Gramoli, Rachid Guerraoui, Gauthier Voron, and Igor Zablotchi. Leaderless consensus. *Journal of Parallel and Distributed Computing*, 176:95–113, 2023.

[30] kernel.org. IPVLAN Driver HOWTO. https://docs.kernel.org/networking/ipvlan.html.

[31] Jialin Li, Ellis Michael, Adriana Szekeres, Naveen Kr. Sharma, 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)*, Savannah, GA, USA, November 2016. USENIX.

[32] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed Shared State Abstractions for Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

[33] Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. Objects shared by Byzantine processes. *Distributed Computing*, 16:37–48, 2003.

[34] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

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

[36] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 174–183. IEEE, 2004.

[37] Iulian Moraru, David G Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[38] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with $t < n/3$, $O(n^2)$ Messages, and $O(1)$ Expected Time. *J. ACM*, 62(4), September 2015.

[39] NextPlatform. Hyperscalers ready to run barefoot in the datacenter. https://www.nextplatform.com/2017/01/30/hyperscalers-ready-run-barefoot-datacenter.

[40] NVIDIA. NVIDIA Spectrum-4. https://nvdam.widen.net/s/pjlcwnrdbn/ethernet-switches-spectrum-4-asic-datasheet-us.

[41] NVIDIA. HowTo Configure Filtering Rules on Mellanox Ethernet Switches (ACLs, IP Filtering). https://enterprise-support.nvidia.com/s/article/howto-configure-filtering-rules-on-mellanox-ethernet-switches--acls--ip-filtering-x, 2022.

[42] 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.

[43] 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*, pages 45–58, 2007.

[44] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *Fast Software Encryption: 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004. Revised Papers 11*, pages 371–388. Springer, 2004.

[45] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, November 2015.

[46] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[47] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[48] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking Packet Payload with P4. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, pages 274–281, 2020.

[49] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat: Multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[50] The P4 Language Consortium. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.0.html.

[51] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011.

[52] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, OSDI'04, page 7, USA, 2004. USENIX Association.

[53] Xsight Labs. X-Switch ISA (XISA). https://xsightlabs.com/wp-content/uploads/2025/03/XISA_Public-.pdf.

[54] Xsight Labs. X2 Programmable Ethernet Switch. https://xsightlabs.com/products/.

[55] Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[56] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio - Juniper Networks' Programmable Chipset - for Emerging In-Network Applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[57] Lior Zeno, Ang Chen, and Mark Silberstein. In-Network Address Caching for Virtual Networks. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 735–749, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

A Switch Resource Utilization

Table 4 summarizes the average per-stage resource utilization for the 64K history register size and up to 8K replicas configuration.

Resource	Utilization	Resource	Utilization
Match Crossbar	12.76%	TCAM	1.74%
Meter ALU	62.50%	VLIW Instruction	18.49%
Gateway	35.42%	Hash Bits	8.77%
SRAM	27.19%		

Table 4: The average per-stage resource utilization with a 64K register and up to 8K replicas support.

B Pseudocode for SwitchBFT

The pseudocode for VUAB is presented in Algorithm 1, and the pseudocode for SwitchBFT is provided in Algorithm 2.

C Correctness Proof

In the proof, we refer to the switch as a *sequencer* to signify its role in the protocol. We make the following assumptions for the proofs, all of which have been discussed and justified in the body of the paper.

(A1) The sequencer is correct and does not fail.

(A2) The hash function used for computing digests is a perfect hash function.

(A3) Messages are source authenticated by the network, i.e., the id of the sender of a message is always known.

(A4) Replicas and clients only receive messages directly from the sequencer.

Assumption (A1) is made solely for the purpose of the proof. We explicitly relax this assumption using a well-known state replication mechanism (chain replication) in §4.8. Assumption (A2) states that the cryptographic hash function is collision resistant, as is standard in BFT protocols [10,20,34,43].[5]

Note that Assumption (A4) can be enforced by having non-faulty replicas or clients ignore any message not received from the sequencer (by Assumption (A3), a receiver knows if a given message is from the sequencer).

C.1 Correctness of VUAB

We begin by proving the correctness of the VUAB protocol, with pseudocode provided in Algorithm 1.

[5]Quoted from PBFT [10]: "We assume that the cryptographic hash function is collision resistant: the adversary is unable to find two distinct messages m and m' such that $D(m) = D(m')$. These assumptions are probabilistic, but they are believed to hold with high probability for the cryptographic primitives we use. Therefore, we assume that they hold with probability one in the rest of the text.".

Lemma 2 (Validity)**.** *In executions of Algorithm 1 without message loss or reordering, if a non-faulty process broadcasts a message m, then all non-faulty processes deliver m.*

Proof. If a non-faulty process broadcasts a message m, then it calculates a correct digest $D = h(m)$ and sends $\langle m, D\rangle$ to the sequencer. In an execution in which there is no message loss, this eventually reaches the sequencer. Since, by assumption, the sequencer is correct (A1), it adds a new sequence number, larger than any used before, stores D and sends it to all processes on line 7. If messages are not lost, all non-faulty processes eventually receive this message and handle it on line 42. The condition of the while loop on line 43 is not met, and, since last only gets updated to the largest sequence number ever seen, in execution with no message reordering, $\langle m, seq\rangle$ has $seq = last + 1$ on line 46. Therefore, m is appended to the pending set of each non-faulty process, and since the hash function is assumed not to generate collisions (A2) and since messages are unique, m is eventually delivered. □

Lemma 3 (Integrity)**.** *For any broadcast message m, every non-faulty process executing Algorithm 1 delivers m at most once.*

Proof. Note that since the sequencer is correct (A1), it only ever forwards each message it receives once. Therefore, each process receives each broadcast message at most once. When a process receives a message, it places it in its `pending` list at most once, and therefore delivers it at most once. □

Observation 1 (Uniqueness)**.** *If m_i, S and m_j, S are messages received by non-faulty processes, where both have the same sequence number, then $m_i = m_j$.*

Proof. By assumptions (A3) and (A4), sequenced messages are only received from the sequencer. Moreover, since the sequencer is correct (A1) and assigns monotonically increasing sequence numbers to messages, two different messages cannot be assigned the same sequence number. □

Lemma 4 (Lossy Total Ordering.)**.** *For any two non-faulty processes executing Algorithm 1, if the k-th `deliver` invocation returns a message, it is the same message.*

Proof. Consider two non-faulty processes p_i and p_j and assume that p_i delivers message $m_i \neq \perp$ and p_j delivers message $m_j \neq \perp$ via the k-th `deliver()` invocation. Note that the number of entries in the `pending` list, and therefore the number of `deliver()` invocations, exactly matches the `last` variable. Furthermore, a non-$\perp$ message is only placed in the `pending` list if its sequence number is equal to `last`. Consequently, both share the same sequence number, k. Therefore, by line 1, they are the same message. □

Algorithm 1: The VUAB Protocol

```
1  Sequencer:
2  upon Init:
3      seq = 0
4      digests = [ ]
5
6  on receive ⟨m, D⟩:
7      seq = seq + 1
8      digests.insert(seq) = D
9      send ⟨m, D, seq⟩ to all processes
10
11 on receive ⟨VERIFY, seq⟩ from p:
12     send ⟨digests.find(seq), seq⟩ to p
13
14 on receive ⟨query⟩:
15     return seq
16
17 Replica:
18 upon Init:
19     last = 0; count = 0; pending = [ ]
20
21 def broadcast(m):
22     D = hash(m)
23     send ⟨m, D⟩ to sequencer
24
25 def deliver():
26     count = count + 1
27     ⟨m, D⟩ = pending.pop()
28     digest = hash(m)
29     if digest ≠ D || m == ⊥:
30         return ⊥
31     else:
32         return m
33
34 def verify(m, seq):
35     send ⟨VERIFY, seq⟩ to sequencer
36     on receive ⟨D, seq⟩:
37         digest = hash(m)
38         if D == digest:
39             return true
40         return false
41
42 on receive ⟨m, D, seq⟩:
43     while seq > last + 1:
44         pending.append(⟨⊥, ⊥⟩)
45         last = last + 1
46     if seq = last + 1:
47         pending.append(⟨m, D⟩)
48         last = last + 1
49     elif seq > count:
50         pending[seq − count − 1] = m
51
52 periodically:
53     send ⟨query⟩ to sequencer on receive ⟨S⟩ from sequencer:
54         while S > last + 1:
55             pending.append(⊥)
56             last = last + 1
```

Algorithm 2: Acknowledgement and Recovery

```
1  Sequencer:
2  upon Init:
3      states[seq, R] = ⊥
4      NOP_votes[seq] = 0
5      acks[seq, R] = 0
6      msg_votes[seq] = 0
7      recovering = [ ]
8      decisions[seq] = ⊥
9
10 on receive ⟨RECOVER, S⟩ from r1:
11     if decisions[S] == NOP:
12         send ⟨NOP_Decision, S⟩ to r1
13         return
14     recovering[S].add(r1)
15     send ⟨RECOVER, S⟩ to all replicas
16
17 on receive ⟨RECOVER, S, X⟩ from r2:
18     if states[S, r2] ≠ ⊥ or (X == NOP and
         decisions[S] == msg):
19         return
20     if X == NOP:
21         states[S, r2] = NOP_rep
22         NOP_votes[S]++
23         if NOP_votes ≥ f + 1:
24             decision[S] = NOP
25             send ⟨NOP_Decision, S⟩ to all replicas
26     else:
27         states[S, r2] = msg_rep
28         send ⟨RECOVER, S, X, digests.find(S)⟩ to
             all replicas in recovering[S]
29
30 on receive ⟨ACK, S, res, N⟩ from r1:
31     if N ≠ N_NOP:
32         NOP_seqs = {S | decisions[S] == NOP}
33         send ⟨reject, NOP_seqs⟩ to r1
34
35     else:
36         for i = 0; i <= S; i++:
37             if acks[i, r1] == 0:
38                 acks[i, r1] = 1
39                 msg_votes++
40             if msg_votes ≥ f + 1 and
                 decisions[S] == ⊥:
41                 decisions[S] = msg
42         send ⟨ACK, S, res⟩ to client of msg S
43
44 Replica:
45 upon Init:
46     log = [ ]; N_NOP = 0
47
48 on deliver m with seq num S:
49     start timer for S
50     log.append(m)
51     res = application.apply(m)
52     send ⟨ACK, S, res, N_NOP⟩ to sequencer
53
54 on deliver ⊥ with seq num S:
55     start timer for S
56     send ⟨RECOVER, S⟩ to sequencer
57
58 on receive ⟨RECOVER, S⟩:
59     while timeout for S not elapsed:
60         wait
61     send ⟨RECOVER, S, log[S]⟩ to sequencer
62
63 on receive ⟨RECOVER, S, m, D⟩:
64     digest = hash(m)
65     if digest == D:
66         log[S] = m
67         if log.length > S + 1:
68             redo(S)
69
70 on receive ⟨NOP_Decision, S⟩:
71     N_NOP ++
72     log[S] = NOP
73     if log.length > S + 1:
74         redo(S)
75
76 def redo(S):
77     rollback application to log entry S
78     for i = S; i < log.length; i++:
79         application.apply(log[i])
80         send ⟨ACK, i, res, N_NOP⟩
```

Lemma 5 (Loss Detection)**.** *If a message m is broadcast in Algorithm 1 by a non-faulty process, then either (1) none of the non-faulty processes deliver m or $\perp$ or (2) every non-faulty process delivers m or $\perp$.*

Proof. If m is lost before reaching the sequencer, the sequence number remains unchanged. In this situation, none of the non-faulty processes can detect the loss, resulting in the non-delivery of the message or $\perp$.

On the other hand, if the sequencer processes the message and assigns it a sequence number S, then eventually, for every non-faulty process p, p receives a message with sequence number $S' \geq S$; if p doesn't receive one, it queries the sequencer until it receives a reply. Once p receives a message with sequence number $S' \geq S$, it appends messages to its `pending` list until `last == S'`. Eventually, p will pop the S-th message from the list and deliver either a message value or $\perp$. □

Lemma 6 (Verifiability)**.** *In Algorithm 1, `verify(m,k)` returns true when executed by a non-faulty process if some non-faulty process delivered m as its kth delivery. Furthermore, if `verify(m,k)` returns true, then no non-faulty process delivered any message $m' \neq bot$ where $m' \neq m$ as its kth delivery.*

Proof. Let p be a non-faulty process that delivered m in its kth delivery. Then in particular, m must have had sequence number k, since otherwise p would not have put it in the kth position in its `pending` list. Furthermore, before delivering, p computed the digest of m and compared it to the digest D sent by the sequencer (line 28). Thus, $D = h(m)$ was the digest stored by the sequencer in line 8 for sequence number k. Therefore, if any non-faulty process q calls `verify(m, k)`, this is also the digest that the sequencer replies with in line 12. q then successfully confirms that the digest is correct in line 37, and returns true. Furthermore, if q successfully confirms that $D = h(m)$, no other non-faulty process can ever deliver a different message in its kth delivery, since all non-faulty processes also calculate the digest of a message before delivering it, and by Assumption (A2), no other message will have the same digest. □

Theorem 3. *Algorithm 1 is a correct implementation of VUAB.*

Proof. This follows directly from Lemmas 2, 3, 4, 5, and 6. □

C.2 The SwitchBFT Protocol's Safety

The main safety guarantees of an f-resilient BFT SMR algorithm are *validity*, *agreement* and *finality*. To formally define these, we start by defining what *commitment* means. For this, we make use of the notion of a replica r sending a *successful* acknowledgement, which means that the sequencer forwarded r's acknowledgement to the client.

Definition 1. *An operation O is* globally committed *if a client has received $f+1$ acknowledgement messages with the same `res` for that operation.*

We say a sequence number S is globally committed if S is the sequence number associated by the sequencer with a globally committed operation O.

We say that O (or S) is globally committed due to r *if r is one of the $f+1$ replicas that sent the agreeing acknowledgements.*

Definition 2. *A sequence number S is* locally committed *in a non-faulty replica r's log if r either (1) r received a $\langle \overline{COMMIT}, S', H, * \rangle$ message from the sequencer for some $S' \geq S$, where H is the digest of r's log from its last commit up to S', or (2) for some sequence number $S' \geq S$, S' is globally committed due to r or r has sent a successful acknowledgement of S' after S' was globally committed.*

We say a value v is locally committed at r if v is the value written in slot S in r's log and S is locally committed at r.

We can now define validity, agreement and finality, and prove the three properties.

- **Validity.** If a value v is locally committed at some non-faulty replica, then v is either $\perp$ or an operation sent by a client.
- **Agreement.** For any two non-faulty replicas r_1 and r_2 for which S is locally committed with values v_1 and v_2 respectively, $v_1 = v_2$.
- **Finality.** If a sequence number S is globally committed, then there is some non-faulty replica r for which the value in all slots $S' \leq S$ in r's log will never change.

We prove a validity property not just on locally committed values, but on all values that are in the log of a non-faulty replica.

Lemma 7. *If a value v is in the log of a non-faulty replica, then v is either `NOP` or an operation sent by a client.*

Proof. A value v enters the log of a non-faulty replica at an index S only under one of three conditions: (1) $v = m$ and replica received a message $\langle m, D, S \rangle$ from the sequencer and verified that $D = hash(m)$, (2) $v =$ `NOP` and the replica did not receive a message with sequence number S from the sequencer, and (3) the replica recovered this value in the recovery protocol. By the correctness of the sequencer, if r receives $\langle m, D, S \rangle$ from the sequencer, then the sequencer received $\langle m, D \rangle$ from the client. In case r recovered v in the recovery protocol, either $v =$ `NOP` or r verified that $D = hash(v)$ where D is the digest r received from the client in the $\langle \overline{\texttt{RECOVER}}, S, v, D \rangle$ message. By Assumption (A2), since the digest matched the hash, v must have been the client's original message. □

We now turn to proving agreement. For that, we first prove a useful lemma that shows why using the number of NOP decisions is a good way to ensure that a log matches the sequencer's known decisions.

Lemma 8. *The set of sequence numbers, $\mathcal{S}_r$ that contribute to a replica r's N_{NOP} count is a subset of the set sequence numbers, $\mathcal{S}$ for which the sequencer mad a NOP decision.*

Proof. A replica adds S to its local NOP decisions only upon receiving a NOP-decision message for S from the sequencer, which only sends such a message if S is decided to be NOP. □

A value can be locally committed for one of two reasons, and we handle each separately.

Lemma 9. *Once a non-faulty replica r receives a $\langle \overline{COMMIT}, S, H, N\rangle$ from the sequencer with $H = hash(r.log[0-S])$ no slot in r's log can ever change, and it has NOPs in exactly the sequence numbers in which NOP decisions have been made by the sequencer.*

Proof. For an entry S' in a non-faulty replica's log to change, it must be either (1) r's last log entry and currently NOP, with no NOP decision on the sequencer for S', or (2) non-NOP, where the sequencer decision for S' is a NOP.

Consider why the sequencer sent a $\langle \overline{\texttt{COMMIT}}, S, H, N\rangle$ message. There must have been some replica r' that initiated a log commitment, and at least $f+1$ replicas that replied with $\langle \overline{\texttt{COMMIT}}, S, H, N\rangle$. Thus, at least one non-faulty replica r' verified that its log's digest up to S is H, and that N equals the number of NOP decisions r' is aware of. By Lemma 8, the set of sequence numbers up to S with NOP decisions on r' and the sequencer is therefore the same. Furthermore, r' must have first ensured that it has no undecided NOP entry before sending a $\langle \overline{\texttt{COMMIT}}, S, H, N\rangle$. Therefore, for every entry $S' \leq S$, S' is either a decided NOP or a non-NOP entry in the log of r' at the time that $H = hash(r'.log[0-S])$.

If H equals the hash of r's log, by Assumption (A2), it must be the case that r's log is the same as r''s log. In particular, this means that it has NOP decisions in exactly the same slots as on the sequencer, and is not missing any values in slots that are not decided as NOP.

Finally, note that once a $\langle \overline{\texttt{COMMIT}}, S, *, *\rangle$ is sent by the sequencer, the sequencer never makes new decisions on any sequence number $\leq S$. □

Lemma 10. *The set of NOP decisions on the sequencer for sequence numbers $\leq S$ cannot change once S is globally committed.*

Proof. Once S is globally committed, the sequencer must have executed line 42 at least $f+1$ times with messages from different replicas. Therefore, it must have received acknowledgements for S from at least $f+1$ replicas, all of which failed the check on line 31 and entered the else clause. Therefore, for all sequence numbers smaller than or equal to S, it must have collected at least $f+1$ message votes, and passed the check on line 40. Therefore, for all sequence numbers $S' \leq S$, S' is decided (i.e. `decisions[S]` $\neq \perp$).

Note that for any S, `decisions[S]` never changes once it is no longer $\perp$. Therefore, the set of NOP decisions in the sequencer for sequence numbers $\leq S$ cannot change once S is globally committed. □

Lemma 11 (Agreement)**.** *For any two non-faulty replicas r_1 and r_2 for which S is locally committed with values v_1 and v_2 respectively, $v_1 = v_2$.*

Proof. Let S be locally committed at two non-faulty replicas r_1 and r_2. We consider the following cases.

Case 1. Replica r_1 received a $\langle \overline{\texttt{COMMIT}}, S_1, H_1, *\rangle$ message from the sequencer for $S_1 \geq S$, and H_1 is the digest of r_1's log up to S_1, and similarly, r_2 received a $\langle \overline{\texttt{COMMIT}}, S_2, H_2, *\rangle$ message from the sequencer for $S_2 \geq S$, and H_2 is the digest of r_2's log up to S_2. Without loss of generality, let $S_1 \leq S_2$. Then by Lemma 9, neither of their logs can change up to S_1, and the NOPs in their logs up to S_1 are the same. Note that by the Lossy Total Ordering property of VUAB, they must also have the same non-NOP entries up to S_2.

Case 2. Both replicas successfully acknowledged a message with sequence number $\geq S$ either after S was globally committed or S was globally committed due to them. Then both replicas must have had the same number of NOP decisions up to S, and by lemma 8, the same set of NOPs. Furthermore, by the Lossy Total Ordering property of VUAB, they must also have the same non-NOP entries up to S.

Case 3. One replica locally committed by acknowledgement and the other by log commitment. Then, just like in case 2, both replicas must have had the same number of NOP decisions up to S, and by lemma 8, the same set of NOPs. Furthermore, by the Lossy Total Ordering property of VUAB, they must also have the same non-NOP entries up to S. □

Lemma 12 (Finality.)**.** *If a sequence number S is globally committed, then there is some non-faulty replica r for which the value in all slots $S' \leq S$ in r's log will never change.*

Proof. If S is globally committed, then a client received $f+1$ acknowledgements for S's operation. Clearly, at least one of them is from a non-faulty replica r. Since, by lemma 10, the set of NOP decisions for sequence numbers $\leq S$ will never change, and r's local NOP count is the same as the sequencer's (since otherwise it would not have sent a successful acknowledgement), r's NOPs before S will never change. By the correctness of the VUAB protocol, none of its non-NOP slots can ever change either. □

Putting agreement, finality, and validity together, we arrive at the correctness of the SwitchBFT protocol.

Theorem 4. *The SwitchBFT protocol satisfies agreement, validity, and finality.*

C.3 Liveness of SwitchBFT

The liveness property we prove for SwitchBFT is, intuitively, that in periods in which the network behaves properly, any non-faulty client's operation will be committed.

More formally, we define the *Global Stabilization Time, GST* to be an a priori unknown point in time after which no packets are lost and all messages arrive within a timeout Δ. Equipped with this definition, we prove the following key theorem.

Theorem 5. *After GST, any operation sent by a non-faulty client will be committed.*

Note that this theorem implies that if a client persists in retransmitting messages for which it does not receive adequate acknowledgements after some appropriately chosen timeout, then eventually its operation will be committed.

To show this theorem, we rely on the correctness of VUAB. In particular, by the validity of VUAB, in such good executions, all non-faulty replicas receive the client's message and place it in their logs. However, this alone does not suffice; we must also show that a Byzantine replica cannot force a NOP decision when all non-faulty replicas received a message.

Lemma 13. *If all non-faulty replicas receive a non-faulty client's message at sequence number S within Δ time since their last received message, then a NOP decision cannot occur at S.*

Proof. Note that since, by Assumption (A1), for a NOP decision to occur at S, $f+1$ replicas must send a $\langle\overline{\texttt{RECOVER}}, S, \texttt{NOP}\rangle$. So, at least one of them must be non-faulty. Let that replica be r. For r to send a $\langle\overline{\texttt{RECOVER}}, S, \texttt{NOP}\rangle$ response at time t, it must have NOP in its log at t. However, note that r only sends $\langle\overline{\texttt{RECOVER}}, S, *\rangle$ after waiting for a timeout. By assumption, all non-faulty replicas receive a non-NOP value for S within the timeout. Therefore, r could not have sent a $\langle\overline{\texttt{RECOVER}}, S, \texttt{NOP}\rangle$, and no NOP decision could have been reached. □

C.4 Log Trimming

We have now proven that the SwitchBFT protocol is correct, as long as the sequencer is correct and does not lose stored metadata. We now consider the effects of sequencer log trimming; we show that if the sequencer collects its metadata for all sequence numbers smaller than S once a log commitment up to S is complete, this does not harm the correctness of the protocol.

In particular, note that if all replicas already have all values at sequence numbers $\leq S$ locally committed at the time of the log compaction, then there is clearly no effect on the protocol. However, it could be the case that some replica has not committed all sequence numbers $\leq S$, and must execute the recovery protocol after the log compaction has occurred. We therefore prove that the recovery protocol still works, and that any such non-faulty replica can later get to a state in which all its entries before S are locally committed.

Lemma 14. *A non-faulty replica r can successfully recover the value or the NOP decision of a sequence number S even after S's metadata is compacted by the sequencer.*

Proof. We first recall how a recovery after log compaction works. A replica r wanting to recover log slot S sends a $\langle\texttt{RECOVER}, S\rangle$ message. If S has already been compacted, the sequencer sends a special post-compaction recovery request to the replicas, requesting that they reply not with their entry for S in their $\langle\overline{\texttt{RECOVER}}, S, *\rangle$ message, but with their entire history instead. The sequencer then attaches its digest of the compacted history, and sends this to the recovering replica r. r then checks whether the attached digest matches the history's digest, and if so, adopts this history as its own, replacing its log. Since, before a compaction happens, there must have been at least $f+1$ replicas that sent $\langle\overline{\texttt{COMMIT}}, S, H, N\rangle$ with N matching the number of NOP decisions of the sequencer, there must have been at least one correct replica, r', that sent a $\langle\overline{\texttt{COMMIT}}, S, H, N\rangle$ message. r' must have verified that its history's digest matches H, and therefore, when r' sends its history in the $\langle\overline{\texttt{RECOVER}}, S, H\rangle$ message, r will successfully adopt its history. □

We must also show that if a non-faulty replica r has a non-NOP entry in a slot S in which a NOP decision was made before S was compacted by the sequencer, r can correctly discover which slots have a NOP decision. This can trivially be done by having r execute a recovery operation on any compacted slot, and have replicas reply with their entire log. However, we show that an optimization in which a replica only requests the NOP locations suffices.

Lemma 15. *Once a non-faulty replica r has successfully recovered all non-NOP entries in its log up to sequence number S, it can recover the NOP decisions even after the sequencer log has been compacted past S.*

Proof. Recall that, once r receives a $\langle\overline{\texttt{COMMIT}}, S, H, N\rangle$ from the sequencer in which N does not match its local NOP count, r sends a request to retrieve the NOP locations, and all replicas send back their NOP locations. With every received NOP list, r checks whether replacing those entries with NOP in its log yields a log whose digest matches H. If not, r rejects the response, and waits for another. Since there must have been at least $f+1$ $\langle\overline{\texttt{COMMIT}}, S, H, N\rangle$ messages from different replicas before the sequencer compacted the log, at least one non-faulty replica had an up-to-date log (i.e., a log whose digest matches H). That non-faulty replica will eventually send r its NOP list, and, since by assumption all of r's non-NOP messages are already up to date, r will then successfully create a log whose digest matches H. □