REINFORCE: Achieving Efficient Failure Resiliency for Network Function Virtualization based Services

Sameer G Kulkarni[*], Guyue Liu[‡], K.K. Ramakrishnan[†], Mayutan Arumaithurai[*], Timothy Wood[‡] and Xiaoming Fu[*]

[*]University of Göttingen, Germany, [‡]George Washington University, [†]University of California, Riverside.

ABSTRACT

Ensuring high availability (HA) for software-based networks is a critical design feature that will help the adoption of software-based network functions (NFs) in production networks. It is important for NFs to avoid outages and maintain mission-critical operations. However, HA support for NFs on the critical data path can result in unacceptable performance degradation. We present REINFORCE, an integrated framework to support efficient resiliency for NFs and NF service chains. REINFORCE includes timely failure detection and consistent failover mechanisms. REINFORCE replicates state to standby NFs (local and remote) while enforcing correctness. It minimizes the number of state transfers by exploiting the concept of external synchrony, and leverages opportunistic batching and multi-buffering to optimize performance. Experimental results show that, even at line-rate packet processing (10 Gbps), REINFORCE achieves chain-level failover across servers in a LAN (or within the same node) within 10ms (100μs), incurring less than 10% (1%) performance overhead, and adds average latency of only ~400μs (5μs), with a worst-case latency of less than 1ms (10μs).

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; **Network management**; *Network reliability*; • **Computer systems organization → Availability**;

KEYWORDS

Network Functions (NF), Service Function Chains (SFC), Fault-tolerance, Availability, Resiliency.

ACM Reference Format:

Sameer G. Kulkarni, Guyue Liu, K. K. Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, Xiaoming Fu. 2018. REINFORCE: Achieving Efficient Failure Resiliency for Network Function Virtualization based Services. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281441

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

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281441

1 INTRODUCTION

Fault Tolerance (FT) and High Availability (HA) are important concerns for various network services. A number of studies show that middleboxes fail [16, 35] and software failures [17, 18, 39] occur often. Recent work [35] estimates roughly 40% of network failures are caused by middleboxes, and the measurements on network failures by Gill *et al.* [16] indicate that load balancers have the highest failure probability. Nearly a third (31%) of device failures are attributed to software related issues. Since these middleboxes operate inline with the network forwarding path, a software failure can significantly disrupt network operations.

Failure recovery time and the overhead for providing resiliency depends on the type of failure. For example, a crash in a software component can be quickly detected and fixed locally by the host operating system within a few microseconds, while recovery from operating system failures may take at least a few milliseconds (e.g., 10-50ms for lightweight unikernels like ClickOS [28] and Mirage [27]) to reboot and restore the device. Hardware failures such as link and node failures may take seconds or more.

Network Function Virtualization (NFV) implements network services and middlebox functions (e.g., load balancers, firewalls, NATs, caching proxies) in software which can then be run on off-the-shelf commodity servers, avoiding the use of dedicated purpose-built hardware. However, an NFV-based data plane must compensate for the potential lower reliability of commodity hardware [11]. In addition, the presence of multiple layers of software including hypervisors or container libraries, guest OSes, system and application software, increase the chance of software failures. In this paper, we present REINFORCE, an integrated framework to support efficient resiliency for NFs and NF service chains.

Multiple NFs may be composed into a service chain run on a single node, either as consolidated functions in a single process [26, 32], or in a pipelined fashion [28, 44]. Of course, scale-limitations may require the service chain to span multiple nodes. Our failure resiliency framework addresses both cases, and we seek to coherently deal with all different kinds of failures, e.g., software failures including the failure of an individual NF instance, and hardware failures such as node and link failures, or power outages.

Previous work such as FTMB [40] and Pico Replication [37] have tried to address fault tolerance and high availability for individual network functions. Such approaches introduce excessive overhead when adopted for service chains, as shown in our evaluation (§5.3). Some chain-level approaches [15, 24] seek to provide reliability guarantees across several NFs, but incur very high latency due to packet buffering delays. Further, intra-node commit operations

(such as the evaluation with FTMB [40] where the logging and storage components are co-located on the same node) may not factor the network latency that can impact overall system performance. Hence, a design choice that compensates for the network round trip latency incurred for the inter-node commit operations is desirable.

Designing comprehensive failover mechanisms that can *efficiently* provide *fast* failure recovery for single NFs *and* service chains, either within a single node or spanning multiple nodes, is the goal of our work. Additionally, REINFORCE aims to guarantee consistency properties for NF state and packet content under all failure situations. REINFORCE ensures the external view of the coherent state of an NF or a service chain with its backup is consistent while achieving external synchrony [29, 30].

In order to ensure chain-wide correctness of operation and to address non-determinism, we maintain two distinct types of information for effective failure resiliency in an NFV environment: The application state (state for an NF or chain of NFs); and the packet processing progress (which can be characterized by a per-flow logical timestamp, as long as packets can be replayed after a failure). Thus, when a chain of NFs is backed up on a different node, we employ *lazy checkpointing* of application state to reduce overhead during normal operation and buffer input packets at a predecessor node in-between the checkpointing instants. These input packets are replayed to the backup node in case of a failure. Keeping track of packet processing progress of all the flows at least requires a per-flow timestamp, which is the critical information necessary to enforce correctness when the packets are replayed. The application state (state of NF or chain of NFs) can then be correctly recovered through replay. This allows us to commit the minimum amount of lightweight per-flow timestamp information at a finer timescale, while committing the more heavyweight application state at a coarser timescale. This distinction enables REINFORCE to achieve high performance under normal, failure-free operation.

A key insight of our work is to carefully separate how resiliency is provided for deterministic packet processing (replay and lazy checkpoints) from non-deterministic behavior (which requires full checkpoints to ensure consistency). Deterministic packet processing occurs when the replay of the same set of input packets at an NF or chain of NFs has the same result, both in terms of the state of NF(s) as well as the packets that are output on the wire from the NF (and/or NF chain). In contrast, non-determinism may not produce the same result upon replay. Non-determinism (ND) is not uncommon, *e.g.,* a NAT or load balancer may make a random choice for a flow's first packet [5, 33]. We annotate code paths which exhibit ND, and only when replay is not possible, trigger more expensive checkpoints to save overhead.

REINFORCE guarantees correctness and achieves external synchrony by speculatively processing packets and compactly committing per-flow timestamps (recorded at checkpoints) to the standby. We precisely replay to the backup only those input packets that had been processed by the primary between the checkpoint instant and a failure and coordinate checkpointing with non-deterministic actions to avoid inconsistencies. Thus, unlike FTMB [40], REINFORCE incurs no overhead for deterministic packet processing, eliminating the need for per-packet access logs at NFs and also the need to enforce strict ordering of packets while replaying packets at the backup. Unlike Pico Replication [37], REINFORCE hides the replication latency and improves throughput by batching and overlapping multiple commit transactions, while allowing NFs to continue speculative execution with the judicious use of multiple buffer stages. These improvements result in a dramatic performance improvement over existing approaches. To summarize, our key contributions include:

- **Integrated resiliency framework**: We present an efficient NFV resiliency framework for DPDK [1] based network functions and service chains with distinct local and remote redundancy schemes (§3.1).
- **Lightweight application checkpointing**: We design mechanisms to minimize the state that needs to be replicated to the backup by taking advantage of *logical clocks*, *external synchrony*, *2-phase commit*, and *dirty state tracking* to enforce correctness before releasing packets from an NF service chain (§4.1-§4.2).
- **Chain wide recovery**: We develop low overhead and low latency approaches for consistent recovery of all network functions in a chain within or across hosts (§3.2-§3.3).
- **Fast failure detection**: We devise ways to quickly detect NF (in order of μs), link and node failures (in ms) (§3.4).
- **Optimization techniques**: We exploit non-blocking, pipelined NF processing with judicious batching and buffering to maximize throughput, minimize latency and avoid overheads during normal operation of network functions (§3.6).

Source code and artifacts of REINFORCE are shared online.[1].

2 DESIGN CONSIDERATIONS

There are a number of key requirements for building an NF resiliency framework for service chains:

Correctness and recovery transparency: NF state must be preserved and consistently recovered across the replica nodes in the event of a failure. In addition, for a service chain, it is necessary to ensure that all the NFs in the chain are able to process flows without interruption, by preserving the necessary processing state of each of the NFs in the chain.

Low overhead: NFs are typically expected to process millions of packets per second and serve large numbers of flows. CPU cycles and memory bandwidth are at a premium. It is necessary to ensure that the performance impact of resiliency support on NFs during normal operation (processing rate and latency) is minimal.

Generality: Given the diversity of types of network services and different deployment patterns, it is necessary to ensure that resiliency solution can be easily adopted for different types of NFs; it is important that resiliency support be incorporated with minimal modifications to the NF's code.

2.1 Deployment and State Management

Deployment: Our implementation focuses on NFs run inside containers, although many of our techniques can be generalized to other approaches. Containers enable cheap snapshots using tools like CRIU (Checkpoint Restore In Userspace) [2]. Unfortunately, these cannot be trivially applied to NFs because they do not interact cleanly with user-space I/O frameworks like DPDK [1, 40]. Further, they cannot provide consistent checkpoints across groups of NFs

[1] https://github.com/sameergk/REINFORCE_Supplements

run in different containers. For this reason, we develop a resiliency abstraction in the NF framework that can identify just the key NF state that needs to be backed-up in each container.

Service chaining: NFs are typically chained to efficiently process flows through multiple functional components. For example, we may have a Service Function Chain (SFC) for HTTP traffic to traverse through a NAT, Firewall, IDS, and Load-balancer NFs [36]. The ordering of NFs needs to be preserved, even when failures cause flows to be routed to a replica. The NF chain (ordered list) needs to be treated as a unit of processing rather than as individual NFs in isolation.

State characterization: NFs keep a variety of state information, including configuration parameters, counters, flow connection status, and application specific variables. We focus on stateful NFs, *e.g.,* NAT, DPI or IDS, which may maintain global configuration state, as well as per-flow or per-connection state. We further classify state updates as either deterministic or non-deterministic (see more details in §2.4). Although many common middleboxes (*e.g.,* firewall, IDS, IPS) do not modify packet headers at all, or if they do, modifications are deterministic for the given input packet headers [19], we must also consider the packets traversing a service chain as state themselves, because other NFs may modify their data (*e.g.,* NAT, load balancers), and we must track their progress through the service chain using logical timestamps. For correctness, all of this state must be properly synchronized to the backup for every NF in a chain.

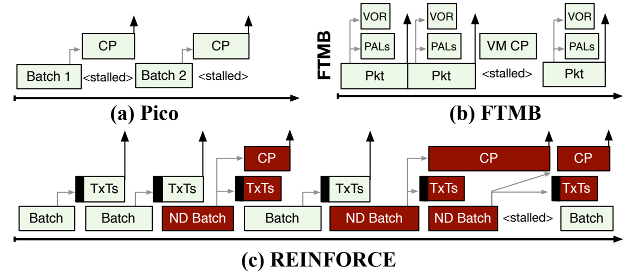
2.2 Failure Model and Detection Schemes

Fast failure detection is a key to providing fast failover. Here, we only consider fail-stop software and hardware failures: software crashes, link status changes, power outages, *etc.*

Software failures: For software failures, we rely on low level kernel events like signals, traps, and syslogs that can be effectively checked (queried or polled) to determine the status of individual software NFs. REINFORCE assumes that we can recover from such failures by reloading the NF with a checkpoint of its recent state and reprocessing any intermediate packets.

Link and server failures: For hardware failure detection, we considered various state-of-the art Layer-2/Layer-3 schemes such as Link Aggregation Control Protocol (LACP) and Open Shortest Path First (OSPF), including Software-defined networking (SDN) and Openflow-based Echo and Fast Failover schemes. Ultimately, we selected Bidirectional Forwarding Detection (BFD) [21–23], which is a lightweight, protocol-independent liveness detection protocol that can detect link failures in millisecond timescales. By examining the status of multiple links we can also use BFD to detect server failures[2]. As a layer-3 failure detection protocol, BFD is also widely supported in hardware with ASIC-based forwarding planes.

Although the timers can be set to a value as low as 1μs, we observed that such aggressive timeout values can result in excessive false-positives. We experimented with the default BFD values across a link connecting two nodes back-to-back with about 50% background traffic and observed that a timeout value below 1000μs still resulted in occasional false-positives.



(a) Pico (b) FTMB

(c) REINFORCE

Figure 1: Comparison of NFV resiliency mechanisms

2.3 Recovery: Replay vs. No-replay

Pico Replication [37] first proposed NF resiliency with a pure checkpointing (i.e., no-replay) scheme that buffers all the output and stops NF processing until the completion of checkpointing (CP, to assure state consistency) as shown in Figure 1(a). However, this buffering results in high latency and degraded throughput during normal operation if checkpointing costs are high. This is because we need to pause NF packet processing and also hold the processed buffer until state is replicated, in order to ensure correctness of the state replicated to the backup.

An alternative proposed in FTMB [40] is to maintain input packet logs (at a predecessor node) and replay the log to reconstruct lost state after a failure. With this approach, output packets can be preemptively released before creating a full NF checkpoint since the state can be recreated via replay. However, to ensure state correctness during replay, it becomes necessary to track and commit all possible source of non-deterministic processing before releasing the packets. FTMB logs each access to each shared variable as packet access log (PAL), and vector clock (VOR) of PALs across all the threads to ensure the correct ordering of accesses to the shared variables during replay as shown in Figure 1(b). FTMB can output packets without waiting for NF state to be checkpointed. By replaying the log of input packets, and the packet-access logs, the replica can recover the lost state and be reinstated correctly in the role of the primary NF. This approach overcomes the latency impact for a majority of the packets, but adds complexity to NF development and can incur high overhead to enforce sequential ordering. FTMB needs to do full system checkpointing periodically and halt processing for the duration of checkpointing (order of a few milliseconds), which is shown to result in high tail latencies and impact overall throughput.

REINFORCE uses a combination of infrequent (lazy) checkpointing and replay of packets. The key is to maintain external synchrony [29]: rather than provide strict synchronization where NFs block until replication completes, REINFORCE allows NFs to continue speculative execution of packets while a backup is performed. Packets will only be released out of an NF service chain once the backup has the minimum information necessary for recovering from a failure. Relaxing the constraint of synchronous replication and adopting external synchrony means that processing can continue through the service chain, and for subsequent packets in the flow, while still providing consistency guarantees to clients receiving the packets. When a failure occurs, the backup node can replay

[2]BFD, by itself, does not take any remedial action when the failure is detected; instead on failure detection, it informs the registered clients (higher level protocols) of the non-responsive adjacency.

packets that have to be processed since the last checkpoint snapshot and update the NF application state on the backup. A logical timestamp is used to determine the packets that have been released since the checkpoint so that the replay process does not transmit unnecessary, duplicate packets downstream while updating the backup state.

2.4 Non-Determinism

NFs operating on the same input (flow of packets) can still diverge in their internal state across multiple executions due to implicit or explicit non-determinism in the processing [7, 40]. Non-determinism (ND) can occur due to i) dependence on hardware whose outcome cannot be predicted, such as hardware clocks, random number generators, *etc.* , ii) race conditions in accessing shared variables among NF threads, and iii) when the intermittent packets are marked for ECN/lost/dropped, then the order of packet arrival and subsequent processing may become nondeterministic [41]. For example, a load balancer (even with the "Active:Active" redundancy configuration) that assigns one server from a pool of backend servers for each TCP connection can end up choosing different backend servers for the same flow when the selection logic is based on system specific calls like `random()`. Similarly, a rate limiter that restricts the number of maximum sessions for a given client can end up rejecting/terminating different connections due to races in the NF threads accessing a shared connection variable during replay.

FTMB [40] overcomes non-determinism by rigorously tracking and ensuring that all the events that can potentially lead to non-determinism (any shared state access and outcomes of unpredictable system calls) are captured and committed to the stable log before releasing the packets. This way, even benign accesses to shared variables or non-deterministic calls (*e.g.,* shared counters) whose impact is unrelated to packet processing (*i.e.,* do not impact the external view) are logged and enforced at the replay node. The result is not only excessive logging overheads but also a limit on an NF's throughput during normal, failure-free operation. Further, with multi-threaded NFs, during replay FTMB enforces a strict ordering for accesses to any shared resources across multiple processing threads. Enforcing this ordering requires more intricate instrumentation of the NF's code and affects both normal and recovery mode performance.

As non-determinism may still occur, we present an alternative, simpler approach to tackle it without the need for per packet access logs or enforcing the strict ordering of packet access to shared variables. We exploit the fact that deterministic packets can be replayed without the need to capture the NF state. Non-deterministic updates may typically be tied to specific packets, e.g., the first packet in a flow that causes non-deterministic updates at several NFs, while subsequent packets do not. However, we do not make any assumption on when non-determinism can occur. For example, L4-L7 NFs (say load balancing) may exhibit non-determinism after receiving/processing a specific byte stream and can be anywhere in the middle of the flow. When an NF performs a non-deterministic state update (for which we require the programmer to annotate such operations) we link it to the packet (batch) which triggered it. Then, taking advantage of external synchrony, we only need to ensure that by the time the packet reaches the end of the service chain and is ready to be sent out, all of its dependent non-deterministic state has been checkpointed to the standby, avoiding the need to replay it after a failure. For example, in the load balancer example, it is sufficient to track the initial connection state update at the start of the flow, rather than tracking and enforcing the access to shared global counters for every packet processed by load balancer NF threads.

3 ARCHITECTURE AND DESIGN

We present the key components of REINFORCE and briefly discuss their roles. We then describe how REINFORCE handles local and remote failures, performs failure detection, and guarantees correctness.

3.1 REINFORCE Components

The key components in the REINFORCE framework are shown in Figure 2.

The NF Orchestrator is responsible for provisioning the NF Manager nodes and designating the active and standby nodes for different service chains. It also configures the BFD settings on each of the NF Managers in the cluster.

The SDN Controller is responsible for populating the flow entries and forwarding rules at each of the NF Manager nodes. In addition, it pro-actively configures the back-up path options: a) in the case of multiple links, it configures the alternate output ports on the predecessor nodes of the designated active NF manager node; and b) configures the flow rules on designated replica standby nodes.

The NF Manager is the core component of REINFORCE. It acts as the in-host controller of coordinating NF functionality, using DPDK's framework for zero-copy delivery of packet data to and between NFs of a service chain within the host. The NF manager tracks the liveness of associated network ports (links) and the NFs provisioned on it. It also provisions and provides the shared memory pools to the NFs to exchange packets, shared memory state, and message notifications. In addition, NF manager implements the "packet logger" module to log and timestamp all incoming packets, and "RSync" module to provide consistent state replication service to the NFs. We leverage both proactive and reactive configuration schemes along the lines of [11, 12].

Active NFs process incoming packets delivered to NFs via the NF Manager. Each NF is integrated with a "*libnf*" library that provides the necessary hooks to facilitate state checkpointing and recovery, thus minimizing changes required on the NF.

NF Standbys can be run on either the primary host (for local failover) or a secondary host (for remote failover). We choose an "Active–Hot Standby" configuration for NF resiliency, where the state updates from the active NFs are consistently committed on the corresponding standby NFs. Software failures that can be recovered by NF instances within the same host can be provisioned for 1:1 redundancy of active:standby NFs. We protect each individual NF instance in a chain, thus allowing REINFORCE to be resilient to multiple NF failures on the same node. We also support failures at the chain level, where all network function standbys of a chain are provisioned on a remote node (which can also host other active NFs). This supports link and node failures for both hardware and software. Multiple NF chains on different active nodes can be configured to share the same standby node.

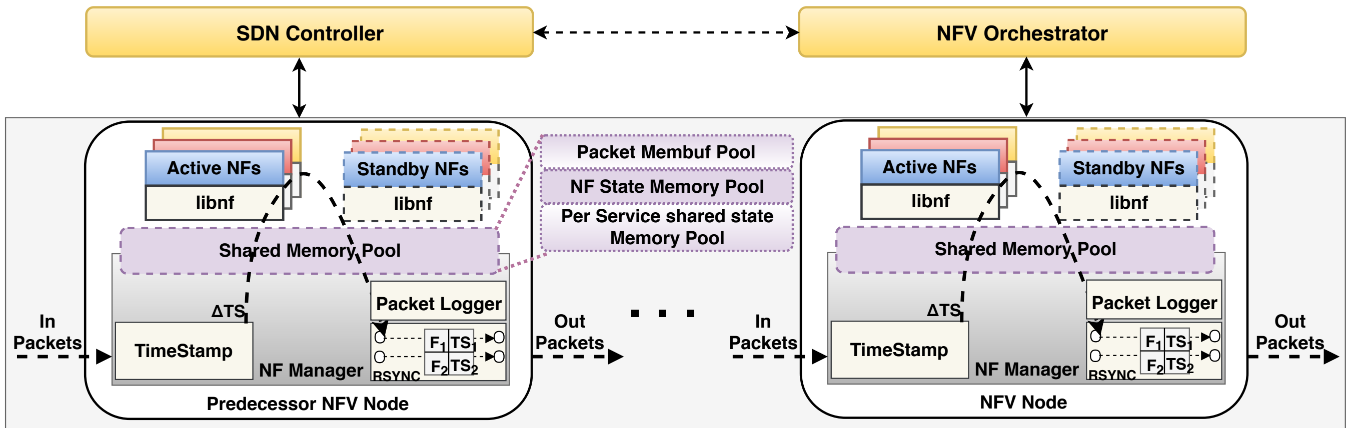


Figure 2: Architecture of REINFORCE (Operational symmetry retained across all the nodes in the chain). Each NFV node hosts multiple NFs (constituting either entire or part of NF chain). Shared memory pool accessible to NF Manager, active and local standby NFs hold the local, global and packet pools. Input packets are timestamped at the start of the chain and logged before transmitting out to the subsequent NFV node in the chain.

The Predecessor Node is a server prior to the node hosting active NFs, which is responsible for logging the incoming packet stream. This is used to handle server or link failures that require the packets to be replayed to the standbys on a remote node for recovery. If the NF chain spans more than one node, the symmetry in our design allows for multiple predecessor nodes like the one shown in Figure 2. However, REINFORCE primarily addresses single node failure (*i.e.,* any node in the chain, except the very first predecessor node, source or destination host) Also, only the first predecessor node is responsible for time-stamping packets.

The *libnf* routine exports the necessary interfaces and provides the common state replication/management functions to NF developers. NF developers need to annotate/set the bit field in the packet headers to indicate the occurrence of non-deterministic updates (`pkt->header.nd = 1`) for any packet(s) in a batch. *libnf* checks for non-deterministic updates and correspondingly updates the NF processing state machine to decide whether to continue the NF processing or block/stall the processing. NF processing is blocked if there is another non-deterministic state update while an outstanding non-deterministic state update remains to be committed. Further, *libnf* combines the batch processing of packets to the NFs and handles the selective NF state synchronization (*i.e.,* only the dirtied memory is copied) to a local standby NF at the end of processing the batch of packets. NFs only need to specify the memory size and offset for the state update, then the *libnf* routine updates the corresponding bitmap (64-bit integer) indicating the dirtied/updated NF state. We expend additional CPU cycles in the active NF to synchronize the modified NF state to the local standby NF. Nevertheless, this is transparent to the NF and is completely handled by a set of library functions implemented in *libnf.*

3.2 Local Resiliency

In scenarios where only software crashes have to be tolerated, the standby NF (also termed replica or backup) is provisioned locally by the NF Manager to provide resiliency from NF instance failures as shown in the left part of Figure 3. After initialization, the backup remains in a 'Paused' state until the NF Manager issues signals to wake up the NF.

NF state checkpointing: We use a "no-replay" scheme to synchronize the active and standby NFs when they are on the same host node. We strictly enforce an "output commit" property: *i.e.,* no output (packets) are released by an NF until all of the corresponding NF state is checkpointed to the standby instance. The helper library, *libnf*, performs checkpointing by copying only the modified regions of application state from the primary to standby NF memory as described in Section 4.1. To ensure the NF memory is in a consistent state while the copy is performed, we trigger checkpoints after an NF finishes processing a batch of packets and suspend that NF's processing while copying is performed. The CPU overhead of checkpointing varies based on the size of data to be synchronized. Fortunately, since the standby is local to the same node, our evaluation illustrates that checkpointing can be done very efficiently even with a batch size of only 32 packets.

Failover: Once the NF Manager detects the failure of an active NF, it wakes up the standby, which is guaranteed to have a checkpoint of application state consistent with the last batch of successfully released packets. The standby then restarts processing from the most recent batch. While the primary may have speculatively processed a portion of the batch, REINFORCE's output commit property ensures that the packets would not have been released, preventing duplicate packets or any inconsistencies related to non-deterministic packet processing. Later, if the original active NF is restored, the standby NF is reverted back to the paused state, allowing the active NF to continue processing packets.

Chaining: Protection of NF chains within a local node is done on a per-NF basis. Output commit buffering is enforced at each NF in the chain, and if one or more NFs fail, standbys will be initiated only for the failed NFs.

3.3 Remote Resiliency

We employ both checkpointing and packet "replay" to provide resiliency from host node failures and link failures (that result in

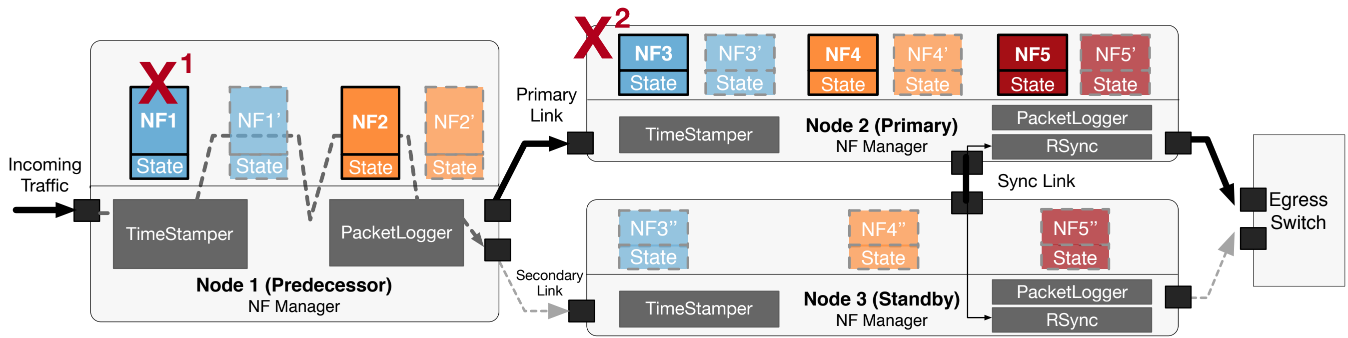


Figure 3: On the left side is the local failover of NF Instance. Upon NF1 failure in Node-1, NF Manager initiates failover seamlessly to local replica NF1'. The right side represents the remote failover of NF chain (NF3, NF4, NF5) to the remote standby node. Upon failure of Primary (Node-2), the predecessor nodes (Node-1) initiates failover by replaying the packets from its logged buffer and also redirects the subsequent packets to the standby node.

loss of connectivity) when the backup is on another node, as shown at right in Figure 3.

Standby server: The NF Orchestrator designates a standby node and notifies the NF Managers at both the node with the active NFs of the chain (Primary) and the predecessor node serving the NF chain. The node with the Active NFs and the predecessor node monitor the liveness status using BFD (more detail in §3.4). If an alternate route to the primary server exists after a link failure (*i.e.,* an alternate output port has been configured by SDN controller), the predecessor node simply redirects the traffic. If a link or node failure makes the primary unreachable, the predecessor node initiates the replay mode on the designated standby/backup node.

Chain-wide state checkpointing: REINFORCE relies on five key concepts, i.e., i) Packet logging with timestamps, ii) Latch buffers for external synchrony, iii) Pipelined replication, iv) Atomic state updates and v) Replay-based recovery to assure consistent and efficient failure resiliency of chains replicated to a secondary host. We describe these now.

(1) **Packet logging with logical time stamping**: In REINFORCE, all the incoming packets at the predecessor node are appended with a logical timestamp (*e.g.,* simple 64 bit packet counter)[3]. And all the outgoing packets are logged (buffered) in per-port rotating log buffers at each of the NFV nodes (predecessors).

The input packet log at the predecessor node is used to replay packets to the standby node when an active node fails. At the active NFV node, the timestamped value of each packet is used to track the packet processing progress for a flow. This information is maintained in a Transmit Timestamp (*TxTs*) table replicated across the primary and backup nodes. The input logger flushes buffered packets upon notification by the active node's NF Manager that they have been successfully sent out.

(2) **Transmit latch buffers**: Buffering packets are needed to provide external synchrony in the face of failure, but excessive buffering increases latency. Packets are stored in a latch buffer at the end of the service chain on the primary server. If all packets processed within a batch are deterministic (which is often the case), then they can be released more quickly since the standby must only update its TxTs table in order to know which packets must be replayed in the event of a failure. Once a TxTs table 'commit' acknowledgment arrives from the standby's RSync component indicating the timestamps for deterministic packet batches are recorded, packets are released to downstream external nodes. On the other hand, replay is unsafe for packets with non-determinism, so REINFORCE proactively pushes checkpoints for any batch that contains non-determinism as described next.

(3) **Pipelined replication**: Our remote replication scheme simplifies consistency and improves performance by leveraging the local checkpoints that we already provide for software failures on the primary host. The local replicas have their state updated at the end of each batch of packets, as described in Section 3.2, which gives a consistent version of the state that can be copied to the remote server without any need to pause the primary replica. As discussed previously, an important feature is that REINFORCE differentiates between deterministic and non-deterministic updates to either NF state or packet data. Deterministic updates can be recovered via replay on the remote host, so state checkpoints can be replicated in a lazy fashion to reduce overhead. On the other hand, non-deterministic state updates cannot be replayed, so packet batches with non-determinism need to have a checkpoint replicated to the backup before they are released from the primary. Fortunately, this replication can be parallelized in two ways. First, it can be performed concurrently with subsequent packet processing in the remainder of the chain. Second, as shown previously in Figure 1(c), an NF can continue to speculatively execute deterministic batches of packets while a checkpoint completes, only stalling its processing if a second non-deterministic batch occurs. To maintain packet ordering, deterministic packets that are processed after the non-deterministic packets are also not released until the non-deterministic packets are released. This gives the ability to continue making progress subsequent to a non-deterministic packet as long as no other non-deterministic packet processing occurs before state corresponding to the first non-deterministic packet processing is complete. Otherwise, we have to stall packet processing to ensure correct recovery of the state at the remote replica in the event of failures while the state is being checkpointed and copied to the replica.

[3] A single nondecreasing counter is sufficient on the Logger; this, in turn, gives monotonic per-flow counters when packets are demultiplexed on the primary node

(4) **Atomic State Updates**: REINFORCE follows a 2-phase commit protocol to provide atomicity between state updates at the backup and packets released at the primary. Our commit protocol begins when the primary sends its updated Transmit Timestamp counters and any necessary non-deterministic state updates. The secondary associates the logical clock values (flow-specific Transmit Timestamps) with the arriving checkpoint state, and ensures that both of these are fully received for all NFs in the chain before acknowledging back to the primary. Then, the primary can release packets in its Latch Buffer to be transmitted towards their destination. The primary then notifies the secondary, so that the latter can commit the checkpoint state. State updates resulting from deterministic operations are transmitted periodically; once this state has been received, the predecessor node can be notified to clear its input log.

(5) **Replay:** The use of latch buffers and atomic state updates guarantees "external synchrony," i.e., the state maintained at the backup can be made consistent with the output packets that have been released from the primary server. Note that since deterministic application-level state is only replicated periodically, it is possible for the standby to recover to a state where the TxTs table says that some packets have been released, but the standby's state does not yet reflect the deterministic updates they should have produced. Thus, in the event of failure, the standby NF chain must rollback to the last checkpoint and replay any subsequent packets so that the standby's state matches the external view of the system (outside of the chain) irrespective of the failure. However, since a chain has multiple NFs and their state updates may arrive at different times, it is possible for a packet to be replayed through some NFs which have already processed it. We believe that NFs are already designed to be robust to receiving duplicate packets—duplicate transmissions are a regular occurrence in wide-area networks, and thus this does not require special handling. The exception to this is processing packets involving non-determinism, which is why we ensure tight state consistency for them—such packets are only released once their state has been confirmed by the standby, avoiding replay.

3.4 Failure Detection

NF instance Failure Detection: NF Managers are responsible to track the liveness of all provisioned NF instances. The NF manager detects NF instance failures in two ways. First, it captures 'voluntary' NF instance failures, by registering for event notification and messages that are triggered via OS (Linux) signals and NF instance-specific messages, when any catchable exception occurs at an NF instance. Second, for involuntary NF terminations, the NF manager performs periodic (every 100 μseconds) checks via the `kill(nf_pid,0)` signal to check and deduce the status of all the registered active NFs. This operation is carried out by the NF Manager's "Monitor thread" which is also responsible for other tasks such as NF registration, de-registration and logging of statistics. The 100 μseconds probe interval is system configurable and can be tuned with a REINFORCE macro at the time of compilation. Even at this frequency, the CPU overhead is less than 1%, to track the liveness of 64 NFs.

Link and Node Failure Detection: We leverage BFD [21] and adapt its configuration settings to mirror those of S-BFD [34] for both link and host failure detection.

BFD configuration and Tunable parameters: During node initialization, the NF Manager is configured to initiate BFD in active mode for each of the host's ports. We configure the BFD minimum Rx and Tx transmit timer intervals to 500μs and the detection timeout multiplier to 3. We operate BFD probing in two modes. When a BFD session is initiated, BFD probes are generated at a very low frequency, *i.e.,* once every 100ms (10Hz) for the remote end to establish the connection (discovery mode). Once the session is established, the connectivity "echo" probe frequency on an idle link is increased to 2000Hz (*i.e.,* once every 0.5ms).
Further, as an optimization, we do not send any BFD probes when there is active traffic and instead piggyback on the presence of active traffic to indicate the port liveness. Explicit BFD liveness detection by generating echo probes is initiated only when the link is observed to be idle for a period of 500μs. With this setup, we can detect link or remote node failures in less than 3*ms*. Not that a link failure can be distinguished from a node failure and can be recovered quickly at the link layer with very little overhead, rather than initiating a software failover of the NF chain to another node. We tune the BFD values recommended for normal network BFD operation [21] for our NFV context, and are more aggressive, mainly because the BFD probes are generated only on idle links. These BFD parameters (probe interval and timeout multipliers) are system tunable parameters, exported as macro variables, and set at the compile time. Even in the worst case, with the link operating in active mode and a frequency of 2000Hz, BFD packets (60 bytes) account to less than .01% of the 10Gbps link bandwidth.

Table 1: Effect of Tx Hold ring buffer sizing

Buffer size (KB)	1	4	8	16
Throughput (Mpps)	2.04	4.8	5.81	6.16
99% ile Latency (ms)	0.942	1.062	1.132	1.26

3.5 Chain-wide correctness

The goal of REINFORCE is to ensure external synchrony and enforce chain-wide correctness. Here we describe how correctness is ensured under local and remote replication, with and without non-determinism.

Local Replicas: With local standby NFs, the state of the active-NF is committed using a local memory copy for each batch of packets before allowing the packets to proceed to the next NF in the chain. This automatically ensures state consistency w.r.t. the external view of the packets released from the NF. This provides correctness regardless of non-determinism since a batch of packets is only released if its state (reflecting any non-deterministic updates) has been committed to the local replica.

Remote Replicas: For the case of replication and failover to a remote node, we separately consider two distinct modes of operation and how REINFORCE ensures state consistency. First, for *deterministic packet processing*, packets from the primary are released from the per-chain latch buffer only upon committing updates of packet-processing progress to the standby's Transmit Timestamps (TxTs table). However, the standby's NF chain state can be out-of-sync and lag the primary/external state. Upon a failure, replaying the packets from the predecessor node makes the standby NFs' state roll-forward and be synchronized to what was in the primary before the failure. All the NFs process the packets to update their state.

Based on the committed TxTs table, the NF Manager on the standby discards duplicate packets that have already been sent. Thus, the standby node's NFs are synchronized to the external view of the state, which was that of the primary at the time of the failure.

If a batch of packets results in *non-deterministic processing*, then the packets from primary are released only when *both* the NF state checkpoint and the TxTs are committed to the remote standby for the entire chain. This ensures that the external view is in sync with the state at *both* the primary and standby nodes across the NF chain. Note that when *any* NF in the chain results in non-deterministic state updates, the state for the entire NF chain is committed before releasing the packets. Thus standby NFs are always in sync with the primary for released packets. Unlike FTMB, REINFORCE does not strictly enforce the processing order on the standby NFs to be identical to the primary in case of a failure before a commit. In this context, the external state corresponds only to the last packets released and not the speculatively processed packets in this batch. This allows the standby and primary states to differ, but the external view remains consistent when the standby becomes active. Thus, the standby NFs can behave differently (i.e., have different resulting state) than the primary for packets that were processed but not yet released from the primary's latch buffer, and still provide an externally synchronous view of the chain-wide state.

3.6 Optimization and parameter tuning

Increasing the batch size improves throughput but also increases latency because the packets spend a longer time in the queue if they are processed in larger batches. To amortize per packet processing cost, we perform state replication and packet release tasks across a batch (32) of packets. Earlier work with DPDK has observed that a batch size of 32 is sufficient to achieve the highest throughput [13]. We also find on our system that the sweet spot balancing throughput and latency is with a batch size of 32. The NF state update on the local standby is done after processing a batch of packets. This has multiple benefits: a) reduces the number of memory copy operations required to synchronize the state on the standby NF; b) minimizes redundant updates by taking advantage of temporal locality and burstiness of packets in a flow, thus benefiting from any over-writes on the same state information. Finally, the state transfer to a remote replica is also batched. The task of copying the packet processing progress (the transmit timestamp table), the NF application state transfer, and the two-phase commit for remote replication when needed, are all batched across this batch of packets. We also tune the size of the Latch ring buffers which hold outgoing packets. Table 1 examines the impact on round-trip latency as the size of the output ring buffer varies, for simple forwarding. With a 200μs RTT and an input rate of 10 Gbps, increasing the buffer size from 1K to 4K (8K) can double (triple) the throughput, while incurring less than a 20% increase in tail latency. By using multiple latch buffers, we achieve concurrency between the replication of state and packet processing. Multiple buffers are used to hold the outgoing packets while the timestamp and non-deterministic state are replicated, allowing for the primary NF to continue uninterrupted packet processing for packets from other flows (thus ensuring correct ordering for packets of the same flow). The algorithm for using multiple buffering stages is illustrated in Figure 4. In order to limit latency, we drain the output transmit ring buffers as

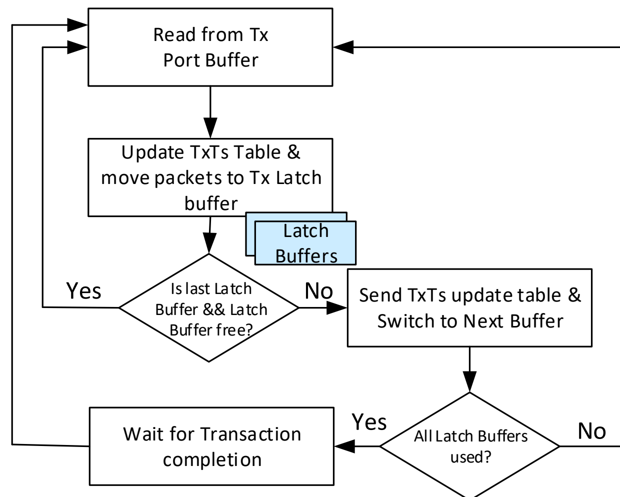


Figure 4: Flow chart describing Multi-transaction buffers.

often as possible. Each stage of buffering helps increase the amount of concurrency possible for NF packet processing to be done in parallel with state updates, while buffering packets at that stage. Only when the second ring buffer in front of the output link buffer becomes full does the third ring buffer get populated, and so on. These stages help minimize the latency of holding up packets while the 2-phase commit transaction is complete. If on the other hand, we only have a single large latch buffer, that buffer can increase latency significantly. Note that the multiple stages of buffering are utilized only when the RTT to the backup is very high.

3.7 Assumptions and Limitations

REINFORCE assumes failstop errors as the fault model for NFs, hosts, and links. Despite such failures, we assume replaying packets through a local or remote replica of the NF will result in correct behavior. During replay mode, upstream NFs of an NF chain may process duplicate packets for various reasons. We assume NFs are able to safely handle duplicate packets without impacting correctness. Similarly, since REINFORCE depends on logical packet timestamps, we can tolerate packet re-ordering. However, we need to check the timestamps during the packet replay mode to avoid re-releasing duplicate packets downstream. For supporting timers, which is common in networks, NFs must explicitly annotate them so that the remote standby can initiate those timer events after a failure. Similarly, we assume NF developers are able to set a flag to indicate which packets involved non-deterministic updates, e.g., at the start of a flow when a load balancing NF randomly selects a destination.

4 IMPLEMENTATION

REINFORCE is built on OpenNetVM [44], a DPDK [1] based NFV platform that enables to run the NFs in containers or as separate processes. We implemented the following modules i) Packet logging: to add a logical timestamp to all input packets and to log all outgoing packets to a stable store; ii) RSync: to enforce external synchrony and perform the two-phase commit transaction using multiple latch buffers; and iii) Liveness Monitoring: to monitor the liveness of locally provisioned NFs and BFD sessions across the configured

links. We dedicate 1 CPU core each for the RSync, packet logging, and liveness monitoring functions.

The control framework coordinates failover via pause and resume event notifications to active and standby NFs, and performs failover actions for remote link failures. To account for the transmit state timestamp of each flow and enforce 2 phase commit transactions, packets leaving the last NF in a chain are stored in latch buffers in the RSync component before being released to the DPDK NIC ports.

4.1 Local Failover

Shadow rings: In a typical NF platform implementation, after the NF processing, packets are handed to a transmit ring to be sent out on the network link or forwarded to another NF in a service chain. REINFORCE introduces the concept of a "Shadow Ring" on both receive (Rx) and transmit (Tx) ends of the NF processing pipeline. Shadow rings are shared ring buffers between active and standby (replica) NFs. Rx shadow rings buffer the batch of packets that the NF needs to process and Tx shadow rings buffer the batch of processed packets for which the state updates have not yet been reflected on the replica NF. There are two benefits of shadow rings. First, they enable enforcing "output commit" for state update on the local standby NF by allowing the transmission of a batch of processed packets only after the NF's state is updated on the standby. Second, when a local failover is required due to an NF failure, the receive side shadow ring enables the replica NF to immediately pick up from the first unprocessed packet, while allowing us to discard previously processed packets for whom the state-update has already been completed.

Shared memory pool: Network functions have two kinds of application state, i) External or the shared state across all the NF instances, and ii) Internal state including the per-flow state and instance specific configurations for each NF. To account for shared state, the NF Manager allocates and maintains a memory pool per service type. The size of this memory pool is a configurable parameter. In our experiments, we set this to 4 MB (our most complex NF, based on the nDPI library, uses 2.8MB). To account for internal NF specific state, the NF manager allocates and reserves a dedicated memory pool for maintaining this state for each instantiated NF instance. The size of this memory pool is again a configurable parameter. In our experiments, we set this to 64 KBytes.

Tracking dirty state: To ease development, we do not require explicit APIs for NFs to interact with their state, as is done in prior work. Instead, we define memory pools for each NF state type and allow NFs to perform arbitrary operations to these regions. REINFORCE automatically detects dirty state regions by scanning small chunks of the NF and service state memory pools to detect changes, similar to FaRM [9]. In our evaluation we configure the number of state chunks to 64, allowing the minimum transferable chunk size for NF specific state to be 1KB and for the total shared service state to be 64KB. The only API that NFs must use when manipulating state is setting a per-packet flag that indicates if it caused non-deterministic updates.

4.2 Remote Failover

Atomic two-phase commit transaction: We use a simple UDP like best-effort connectionless transport to deliver updates to the backup and use sequence numbers to identify any missing packets (while it is desirable to have a reliable transport, we wish to avoid the connection setup and initial handshake overheads of TCP). We use a custom Ethernet type to differentiate state update packets from regular NF destined packets associated with the DPDK port. If packets are lost, we abort the transaction and resend new updates. State transfer packet headers include the fields to indicate the type of packet transfered (either state transfer or acknowledgement packet), type of state (NF state, service configuration information, or Tx Timestamp), size of the packet, base offset address, packet sequence number, and 'last packet' flags. **Accounting for failed transactions**: When the Tx state update commit acknowledgement is not delivered to the primary, the NF manager may be blocked, resulting in port buffers getting full and subsequent processing by NFs being discarded or stalled. To avoid this, we have a transaction timeout after which the NF manager aborts the current transaction and continues to process the subsequent packets and send new updates. To ensure continuity, we choose to continue processing subsequent packets and drop the packets corresponding to the failed transaction. End-to-end retransmission of these dropped packets is expected to update the standby NF state appropriately.

Tx timestamp state update overhead: We opportunistically perform the transmit timestamp (TxTs) table state updates as often as possible. The frequency of operation is limited by the RTT and number of configured latch buffers on the system. Assuming a best case RTT (between two directly connected nodes) of 100μ seconds, performing each Tx timestamp checkpoint in the worst case needs to transfer the entire TxTs table of 64KB (64 1KB packets). This is an overhead of less than 5.25% on the 10Gbps link. For checkpointing, using large latch buffers (8K), we can checkpoint at a slow rate (roughly once every 5 RTTs) *i.e.,* performing checkpoints once every 500μ seconds, reducing the 10Gbps link overhead to less than 1.05% at the cost of added latency.

5 EVALUATION

Our experimental testbed used five Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz servers, each with 157GB RAM, two sockets with 28 cores each, running Ubuntu SMP Linux kernel 3.19.0-39-lowlatency. The topology for the primary, standby and predecessor nodes is as shown in Figure 3. In addition, we have a source and sink node at the two ends. For these experiments, nodes were connected back-to-back with dual-port 10Gbps DPDK compatible NICs to avoid any switch overheads. We use the DPDK-based high speed traffic generator, Moongen [10] to generate line rate traffic consisting of UDP and TCP packets, apache-bench [6], and wrk [4] to flood HTTP download requests. We vary the number of flows and the NF chain setup as needed for each experiment. We configured the number of stages of latch buffers (*i.e.,* multiple transaction buffers) to 3, with each stage having 4K packet buffers.

5.1 Overhead Analysis

Our profiling indicates the cost of memory scan and updating of the dirty state for a 64KB memory and 1KB chunks to be 55-80 CPU ticks. The copy overhead for a 4KB page is measured to be 2315-2590 CPU ticks. Copy operation for a batch of processed packets drastically reduces the overhead during normal processing. Next, we consider DPI, a compute-intensive NF, and lightweight monitor

Table 2: NFs and NF Chains used in experiments.

NF	Type	Characteristic	
		Shared Memory	Non Determinism
Simple Forward (SF)	Stateless	-	-
Basic Monitor (MON)	Stateful	64 KB	5 SV, 1 ms
Vlan Tag(QoS)	Stateful	64 KB	3 SV, 1 ms
Load Balancer (LB)	Stateful	64 KB	1 SV, 1 ms
DPI	Stateful	4 MB	2 SV, 1 ms
Chain 1	Stateful	QoS, BM	
Chain 2a	Stateful	QoS, BM, LB	
Chain 2b	Stateful	QoS, BM, SF	
Chain 3	Stateful	QoS, BM, SF1, SF2	

(SV) Shared variables and rate of non-deterministic updates.

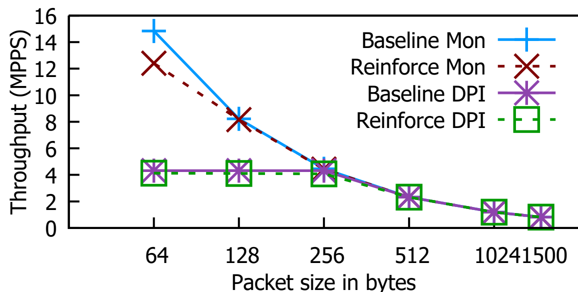


Figure 5: Throughput of different packet sizes

(MON) NFs, and subject them to line rate traffic for different packet sizes. We observe from Figure 5 that REINFORCE is able to achieve performance identical to baseline for all DPI. For MON, even the worst case performance impact is less than 15% (12.6 Mpps with REINFORCE compared to 14.88Mpps baseline).

Table 3: Performance impact of non-determinism

Time (ns)	1	10^3	10^5	250us	500us	10^6	10^7	10^8	10^9
Throughput (Mpps)	0.22	0.22	0.24	7.03	11.19	12.65	13.34	13.34	13.34
Max. Latency (us)	1370	1370	1370	1361	790	698	670	617	617

Impact of Non-Determinism rate: REINFORCE performs a 2-phase commit of the chain-wide packet-processing progress and NF states. To ensure correctness, any non-deterministic updates result in chain-wide NF state checkpointing. The table above shows the impact on performance for different non-determinism rates, which is varied from 1 pkt. every nano second up to one pkt. every second. REINFORCE is able to provide near line-rate processing for the non-deterministic rate that is less frequent than one every 250μsec, while more frequent non-determinism reduces the throughput, eventually dropping to 0.22Mpps. This is due to the round-trip latency of 2-phase commit, causing the NF processing to stall if a previous non-deterministic batch of packets has to be processed and state committed to the standby. In Sec. 5.4 we demonstrate that, with multiplexing, unused resources can be utilized to serve other NFs when one NF's processing stalls due to non-determinism.

5.2 Operational Correctness and Performance

We demonstrate the operational correctness of REINFORCE with both Graybox and Blackbox tests.

Graybox tests: We first validate the correctness of failover operation through instrumented template NFs that check for consistency of NF state updates and packet processing. If any packets

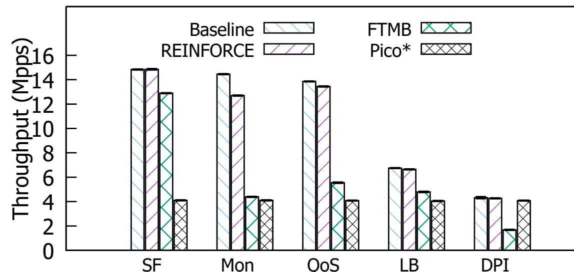


Figure 6: Performance impact of different FT systems on the normal operation for different NFs.

are obtained with incorrect content (inconsistency between state embedded in the NF and packet content), then the NF flags the error to the NF Manager and the NF terminates. We run a script to perform 10K forced terminations and re-activation of the active NF. Each time the active NF fails, the failover to the backup NF happens automatically. When the active NF is re-instantiated, the NF state is updated and flows are routed back to the active NF. These tests successfully validated the correctness conformance of REINFORCE for the local failover scenario.

Blackbox tests: We also assess the application level of failover through the following end-to-end tests.

i) DPI based protocol detection: We demonstrate REINFORCE with a DPI NF and feed it a set of PCAP traces (MPEG, Hangout, Youtube-upload, Snapchat, QUIC) available at NTOP [3, 8]. We observe that the DPI NF identifies the protocols correctly both when there are no failures and when the primary fails and REINFORCE fails over to the standby NF.

Table 4: Effect of Failure on HTTP downloads

	Baseline	Resiliency	
	w/o failure	Local Failover	Remote Failover
Requests/sec	10.52	10.32	10.22
Transfer/sec (GB)	1.08	1.06	1.02

ii) HTTP downloads: We route HTTP downloads through a service chain of 2 NFs (QoS and MON). We start repeated HTTP download requests for a period of 60 seconds, and trigger failures at the 30 second mark. a) We induce a QoS NF instance failure to account for local NF failover, and b) we induce, NF MGR failure on the primary node to trigger a chain-wide remote failover. We compare the *baseline* operation *i.e.,* failure-free operation with both NF instance failure (local failover) and node failure (remote failover) cases. We observe with REINFORCE that HTTP downloads succeed for both the NF instance failure (local failover) and node-failure (remote failover) cases. We also observe very little impact on the application, resulting in just 3-4% reduction in the total number of requests serviced per second, and a negligible reduction in throughput as shown in Table 4.

Failover Times: We measured the time for local and remote failovers from the instant we induce a failure. For local failover: mean= 56μs and maximum= 114μs over 100 iterations. For remote failover: mean= 3280μs and maximum= 3517μs over 10 iterations. This includes failure detection time with BFD and for the predecessor node

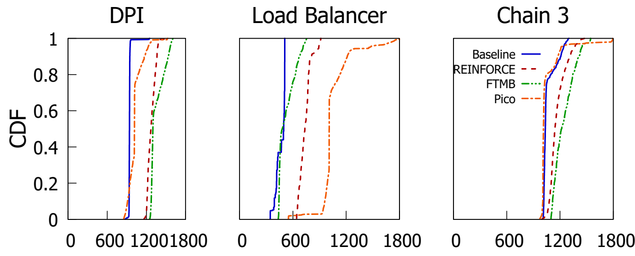


Figure 7: RTT with different FT Systems.

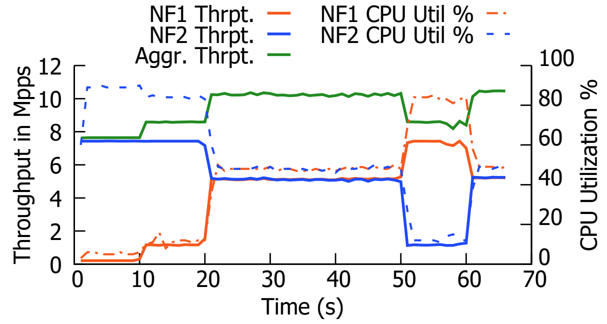


Figure 8: Two isolated NFs with varying Non-Determinism rates, running on the same CPU core.

to initiate the failover at the backup by starting replay of buffered packets. We do not account for the time needed to complete the replay and send the first new packet, as it varies based on the processing chain and non-determinism intervals. But, we did measure the time needed to initiate and prepare for replay at the predecessor node (*i.e.,* to notify the standby node, open a pcap file and start replay). The average time taken was 60-100μs, and just below 2ms to replay approximately 3K packets from the predecessor node to the standby. Replay execution of the 3K packets (20Mbits) on the standby took approximately 2ms.

5.3 Failure-free Operation

We compare REINFORCE with FTMB [40] and Pico Replication [37][4] for a number of different NFs in terms of i) overhead during normal operation, reflected in the throughput, and ii) latency of packet processing (additional state update operation), for individual NF instances. Figure 6 shows throughput for normal operation, in Mpps (with error bars showing the standard deviation in throughput). REINFORCE performs almost as well as baseline (no resiliency) case, achieving near line rate (~13.5Mpps) throughput for most NFs. REINFORCE's remote replication outperforms Pico replication by 2 orders of magnitude. Fig. 7 shows the impact on packet latency for two selected NFs. Local replication adds less than 5μs to the baseline case, while remote replication adds roughly 400μs.

Table 5: Multitenancy and resiliency modes.

Mode	Min	Median	Max
Simple Forward NF			
Baseline	210	221	373
Local Replication	211	229	402
Remote Replication	540	601	789
Basic Monitor NF			
Baseline	268	334	695
Local Replication	270	338	699
Remote Replication	596	623	840

(a) Latency (μs) with different resiliency modes.

Time (sec)	NF1 (μs)	NF2 (μs)
1-10	1	1000
11-20	100	750
20-30	250	500
30-40	500	500
40-50	750	250
50-60	1000	100
60-70	1000	1000

(b) Non Deterministic rates for NF1 and NF2 at different time intervals.

5.4 Multi-tenancy and Resiliency Levels

We now demonstrate the benefits of the NF management framework of REINFORCE in supporting multi-tenant NF execution and in providing performance isolation for flows configured with different resiliency levels.

5.4.1 Multi-tenancy. In a typical multi-tenant environment, network functions from different tenants can be co-located and share the same CPU. We expect that NFs from different tenants see different workloads and are consequently subject to different rates of non-deterministic at different time intervals. When the non-deterministic rate is high, the CPU may be idle/underutilized because of frequent stalls. But, REINFORCE takes advantage of efficiently multiplexing NFs that exhibit non-determinism to improve CPU utilization and overall system throughput. We multiplex 2 NFs, NF1 and NF2, from different tenants on the same core. They exhibit different non-deterministic rates at different time intervals as shown in Table 5b. Figure 8 shows the ability of REINFORCE to efficiently multiplex these 2 NFs, resulting in the improvement in both the CPU utilization and aggregate throughput.

5.4.2 Differing resiliency levels. We demonstrate the benefit of REINFORCE's ability to support different flows configured with different resiliency levels. REINFORCE provides the desired resiliency while isolating the flows from each other. We have two Monitor NF instances, configured as: (a) one NF instance with only local resiliency (backup on the same node) for flow-1; and (b) a second NF instance with node-level resiliency (remote standby) for flow-2. We also perform a similar experiment with the Simple Forward NF as well. The latencies for the two flows differ, as shown in Table 5a. We observe the impact on latency is minimal for the flows configured with only local resiliency (less than 30ms in the worst case). But, flows configured with remote resiliency (local + remote) incur nearly 2x higher latency for both the Simple Forward and Basic Monitor cases. This shows the ability of REINFORCE in providing different levels of resiliency while isolating one flow from another – an essential and desirable characteristic for multi-tenancy.

5.5 Impact of Chain Length

We consider experiments with multiple chains having different lengths as described in Table 2. We compare four cases: baseline (no resiliency), REINFORCE, FTMB, and Pico Replication[5]. Figure 9 shows that performance of REINFORCE remains consistent with the baseline for varying chain lengths, unlike FTMB and Pico. In fact, with increased chain lengths, the overheads of REINFORCE are amortized allowing the throughput to be closer to the baseline. With FTMB and Pico, to ensure correctness, the output commit

[4] We implement a) simplified FTMB logic with parallel releases, by storing packet access logs (PALs) per shared variable *i.e.,* NFs transmit all the associated PALs before releasing packets to the NF manager; the NF manager simply transmits the packets without blocking for output commit, and b) Pico Replication: NF state checkpointing with output commit policy.

[5] For both FTMB and Pico Replication, we implement the state replication only at the end of the chain, rather than at every NF. This serves as a simplified, optimized approach, but does not ensure correctness.

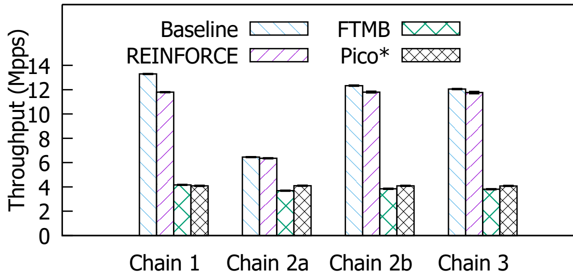


Figure 9: Comparison of chain-wide processing performance for different NF chains.

must be performed individually for each NF in the chain. In fact, the throughput we show for FTMB and Pico is likely to be optimistically high compared to a full implementation.

6 RELATED WORK

NF state migration: Split/Merge [38] defines state access APIs to read and update the internal state of virtualized NFs being moved across hosts. It relies on the ability to identify per-flow state to provide consistent migration. Stratos [14] provides an orchestration layer for NFs by using an SDN controller to migrate the instances and redistribute the traffic to less congested nodes. Likewise, we take advantage of SDN controller to set up the forwarding rules but rely on NF Managers to efficiently migrate the NF instances. OpenNF [15] presents a control plane architecture to have loss-free transfer of NF state. But, because of a controller-based orchestration and event buffering mechanism, it has high per packet latency for migration. In contrast, REINFORCE relies on the NF Manager (Remote Sync) to perform the NF state migration across the designated active and standby NF nodes while allowing NFs to simultaneously process packets. Unlike [38], REINFORCE does not require the NFs to depend on specific state update APIs, but only requires the NFs to annotate the state updates sufficiently to distinguish between deterministic and non-deterministic changes. StateAlyzr [25] complements our work by enabling the NF developers to programmatically analyze and to identify just the right amount (minimal) of NF state, and correctly annotate the NF state that needs to be migrated to ensure consistent state replication. S6 [43] provides a framework for elastic scaling of NFs. It implements NF state as distributed shared state objects, and supports object replication to facilitate NF state sharing across multiple NF instances.

Fault tolerance and high availability: Pico Replication [37] is an application level NF state checkpointing based high availability framework built on top of Split/Merge. It provides fine-grained flow level state replication and employs flow group-based NF state transfers. To enforce correctness, it buffers all output packets during NF state checkpointing, thus delaying outputs even during failure-free operation.

FTMB [40] is a replay based framework that logs all input packets and the per packet access log of all the components (*i.e.,* the shared variables in NF that account for non-determinism) that are necessary to restore the state on the replica during replay. In addition, to amortize the cost of input logging, it also employs periodic check-pointing of NFs. Thus, FTMB guarantees correctness of operation during replay mode by ensuring strict ordering of packet processing (guided by the packet access logs) at the replay node. In essence, FTMB's notion of correctness emulates strict idempotent per packet behavior across the active and replica nodes. This comes at the cost of maintaining multiple per packet access logs, which becomes a potential bottleneck for NFs with 5+ shared variables, for packet rates of 1.25Mpps (ref. §5 of [40]), resulting in more than ~30% overhead traffic. In addition, due to periodic VM checkpointing, the tail latencies drastically increase from less than 100μs, at the 50th%-ile, to nearly 810 μs, at 95th%-ile and 18ms at 99th%-ile. Also, both of these works do not account for chains of network functions. REINFORCE fills this gap by presenting an efficient chain level replication mechanism that does not excessively impact (introduce latency) failure-free operation, nor does it induce limits on processing rates to enforce correctness at the replica state. Plover [42] presents a virtualized state machine replication system to address general VM fault tolerance. By enforcing the same total order of inputs for a VM replicated across hosts, it can keep most memory pages updated and only transfer the few divergent pages between primary and secondary, which effectively alleviates checkpointing overhead while maintaining external consistency. *Alternative architectures:* StatelessNF [20] and StreamNF [24] are alternative approaches of externalizing the state of the NFs to in-memory databases like RAMCloud [31]. While this may be feasible for some NFs, the database can become a bottleneck. Also, substantial refactoring is required. REINFORCE focuses on traditional middlebox architectures and limits changes needed in NF development.

7 CONCLUSION

REINFORCE is the first to address chain-wide network function resiliency, supporting fast detection and recovery of software as well as server and link failures. REINFORCE can detect NF failure and failover to a local standby within 150μs. More importantly, it provides chain-wide failover to a remote node within 5ms. In addition, REINFORCE results in minimal overhead for normal operation and achieves 2X better performance than the state-of-the-art. We distinguish the minimum state information needed to achieve efficient and consistent remote replication. The key is REINFORCE's separation of deterministic and non-deterministic NF processing, and only incurring the overhead of checkpointing and a 2-phase commit of state on the standby for non-deterministic NF processing. REINFORCE automatically tracks and replicates state to standby NFs while enforcing correctness. The amount of state replicated is minimized by using 'lazy' replication of NF application state across hosts, and packet replay is used to speed up the recovery of deterministic NF processing. Even for reasonably frequent non-deterministic packet processing, REINFORCE's performance is far superior to the other alternatives.

Acknowledgement: We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their thoughtful feedback. This work was supported by EU FP7 Marie Curie Actions CleanSky ITN project Grant No. 607584, and US NSF grants CRI-1823270, CRI-1823236, CNS-1422362, CNS-1522546 the ARO DURIP grant W911NF-15-1-0508, and grants from Hewlett Packard Enterprise Co. and Futurewei Technologies, Inc.

REFERENCES

[1] Data plane development kit. http://dpdk.org/, 2014. [online].

[2] Criu: Checkpoint restore in userspace. http://criu.org/, 2017. [online].

[3] ndpi test pcap traces. https://github.com/ntop/nDPI/tree/dev/tests/pcap, 2018. [online].

[4] wrk: a http benchmarking tool. https://github.com/wg/wrk, 2018. [online].

[5] Alpernas, K., Manevich, R., Panda, A., Sagiv, M., Shenker, S., Shoham, S., and Velner, Y. Abstract interpretation of stateful networks. In *International Static Analysis Symposium* (2018), Springer, pp. 86–106.

[6] Bench, A. ab-apache http server benchmarking tool.

[7] Cachin, C., Schubert, S., and Vukolić, M. Non-determinism in byzantine fault-tolerant replication. *arXiv preprint arXiv:1603.07351* (2016).

[8] Deri, L., Martinelli, M., Bujlow, T., and Cardigliano, A. nDPI: Open-source high-speed deep packet inspection. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)* (Aug. 2014), pp. 617–622.

[9] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 401–414.

[10] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. Moongen: a scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference* (2015), ACM, pp. 275–287.

[11] ETSI-GS-NFV-002. Network Functions Virtualization (NFV): Architectural Framework. http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf, 2013. [online].

[12] ETSI-GS-NFV-REL-001. Network Functions Virtualization (NFV): Resiliency Requirements. http://www.etsi.org/deliver/etsi_gs/NFV-REL/001_099/001/01.01.01_60/gs_NFV-REL001v010101p.pdf, 2015. [online].

[13] Gallenmüller, S., Emmerich, P., Wohlfart, F., Raumer, D., and Carle, G. Comparison of frameworks for high-performance packet io. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems* (2015), IEEE Computer Society, pp. 29–38.

[14] Gember, A., Krishnamurthy, A., John, S. S., Grandl, R., Gao, X., Anand, A., Benson, T., Akella, A., and Sekar, V. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR abs/1305.0209* (2013).

[15] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. Opennf: Enabling innovation in network function control. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 163–174.

[16] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev. 41*, 4 (Aug. 2011), 350–361.

[17] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 7:1–7:14.

[18] Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., and Eliazar, K. J. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 1–16.

[19] Jackson, E. J., Walls, M., Panda, A., Pettit, J., Pfaff, B., Rajahalme, J., Koponen, T., and Shenker, S. Softflow: A middlebox architecture for open vswitch. In *USENIX Annual Technical Conference* (2016), pp. 15–28.

[20] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 97–112.

[21] Katz, D., and Ward, D. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[22] Katz, D., and Ward, D. Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop). RFC 5881, June 2010.

[23] Katz, D., and Ward, D. Generic Application of Bidirectional Forwarding Detection (BFD). RFC 5882, June 2010.

[24] Khalid, J., and Akella, A. Streamnf: Performance and correctness for stateful chained nfs. *CoRR abs/1612.01497* (2016).

[25] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., and Akella, A. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 239–253.

[26] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[27] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. *SIGPLAN Not. 48*, 4 (Mar. 2013), 461–472.

[28] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 459–473.

[29] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 1–14.

[30] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. *ACM Trans. Comput. Syst. 26*, 3 (Sept. 2008), 6:1–6:26.

[31] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[32] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[33] Panda, A., Lahav, O., Argyraki, K., Sagiv, M., and Shenker, S. Verifying isolation properties in the presence of middleboxes. *arXiv preprint arXiv:1409.7687* (2014).

[34] Pignataro, C., Ward, D., Akiya, N., Bhatia, M., and Networks, J. Seamless Bidirectional Forwarding Detection (S-BFD). RFC 7880, July 2016.

[35] Potharaju, R., and Jain, N. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference* (New York, NY, USA, 2013), IMC '13, ACM, pp. 9–22.

[36] Quinn, P., and Nadeau, T. Problem Statement for Service Function Chaining. RFC 7498, Apr. 2015.

[37] Rajagopalan, S., Williams, D., and Jamjoom, H. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 1:1–1:15.

[38] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 227–240.

[39] Sahoo, S. K., Criswell, J., and Adve, V. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 485–494.

[40] Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J. a., Ratnasamy, S., Rizzo, L., and Shenker, S. Rollback-recovery for middleboxes. *SIGCOMM Comput. Commun. Rev. 45*, 4 (Aug. 2015), 227–240.

[41] Velner, Y., Alpernas, K., Panda, A., Rabinovich, A., Sagiv, M., Shenker, S., and Shoham, S. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2016), Springer, pp. 811–830.

[42] Wang, C., Chen, X., Jia, W., Li, B., Qiu, H., Zhao, S., and Cui, H. PLOVER: Fast, multi-core scalable virtual machine fault-tolerance. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 483–489.

[43] Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S., and Shenker, S. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 299–312.

[44] Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., and Wood, T. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (New York, NY, USA, 2016), HotMIddlebox '16, ACM, pp. 26–31.