Correctness and Performance for Stateful Chained Network Functions

Junaid Khalid   Aditya Akella
University of Wisconsin - Madison

Abstract: Network functions virtualization (NFV) allows operators to employ NF chains to realize custom policies, and dynamically add instances to meet demand or for failover. NFs maintain detailed per- and cross-flow state which needs careful management, especially during dynamic actions. Crucially, state management must: (1) ensure NF *chain-wide* correctness and (2) have good performance. To this end, we built CHC, an NFV framework that leverages an external state store coupled with state management algorithms and metadata maintenance for correct operation even under a range of failures. Our evaluation shows that CHC can support ~10Gbps per-NF throughput and $< 0.6\mu s$ increase in median per-NF packet processing latency, and chain-wide correctness at little additional cost.

1 Introduction

NFV vastly improves network management. It allows operators to implement rich security and access control policies using NF chains [5, 14, 10, 6, 1]. Operators can overcome NF failure and performance issues by spinning up additional instances, and dynamically redistributing traffic [15, 29].

To be applicable to enforcing policies correctly, NFV must provide *chain output equivalence* (COE): given an input packet stream, at any point in time, the collective action taken by all NF instances in an NFV chain (Figure 1a) must match that taken by an hypothetical equivalent chain with infinite capacity always available single NFs (Figure 1b). COE must hold under dynamics: under NF instance failures/slowdowns, traffic reallocation for load balancing/elastic scaling, etc. Given that NFV is targeted for cloud and ISP deployments, COE should not come at the cost of *performance*.

These goals are made challenging by NFs' *statefulness*. Most NFs maintain detailed internal state that could be updated as often as per packet. Some of the state may be shared across instances. For example, the IDS instances in Figure 1a may share *cross-flow state*, e.g., per port counters. They may also maintain *per-flow* state, e.g., bytes per flow, which is confined to within an instance.
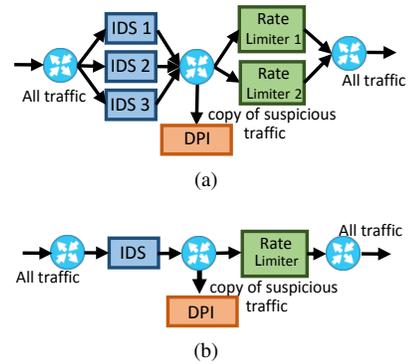

Figure 1: (a) Example NFV chain with many instances per NF (b) logical view with infinite capacity NFs/links for COE.

Ensuring COE under statefulness requires that, as traffic is being processed by many instances, or being reassigned across instances, updates to state at various NFs must happen in a "correct" fashion. For example, shared state updates due to packets arriving at IDS1 must be reflected at IDS2; likewise, when reallocating a flow, say f_1, from IDS1 to 2, f_1's state should be updated due to in-flight f_1 packets arriving at both IDSes 1 and 2. Finally, *how* the state is updated can determine an NF's action. For example, the off-path Trojan detector [12] in Figure 2 relies on knowing the exact order in which connection attempts were made. When there is a discrepancy in the order observed w.r.t. the true order – e.g., due to intervening NFs running slow or failing – the Trojan detector can arrive at incorrect decisions, violating COE.

Many NFV frameworks exist today [29, 25, 26, 20, 11, 17, 32]. Several of them focus on managing NF state migration or updates upon traffic reallocation during scaling or failover [29, 25, 26, 16, 32]. However, they either violate COE, or suffer from poor performance (or both).

First, most systems **ignore shared state** [29, 25, 26, 20]. They assume that NFs do not use cross-flow state, or that traffic can be split across NF instances such that sharing is completely avoided. Unfortunately, neither assumption is valid; many NFs [21, 30, 8, 22] have cross-flow state, and the need for fine-grained traffic partitioning for load balanc-

ing can easily force cross-flow state sharing across instances. Because shared state is critical to NF processing, ignoring how it is updated can lead to inconsistent NF actions under dynamics, violating COE (§2.2).

Second, existing approaches **cannot support chain-level consistency**. They cannot ensure that the order of updates made to an NF's state (e.g., at the Trojan detector [12] in Figure 2) are consistent with the input packet stream. This inability can lead to NFs arriving at incorrect decisions, e.g., missing out on detecting attacks (as is the case in Figure 2), violating COE. Similar issues arise in the inability to correctly suppress spurious duplicate updates observed at an NF due to recovery actions at upstream NFs (§2.1).

Finally, existing frameworks impose **high overhead** on state maintenance, e.g., 100s of milliseconds to move per-flow state across instances when traffic is reallocated (§2.2).

We present a new NFV framework, CHC ("correct, high-performance chains"), which overcomes these drawbacks. For COE, CHC uses three building blocks. CHC stores NF state in an in-memory **external state store**. This ensures that state continues to be available after NF instances' recover from failure, which is necessary for COE. Second, it maintains simple **metadata**. It adds a "root" at the entry of a chain that: (1) applies a unique logical clock to every packet, and (2) logs packets whose processing is still ongoing in the chain. At the store and NFs, CHC tracks packet clocks along with update operations each NF issues. Clocks help NFs to reason about relative packet ordering irrespective of intervening NFs' actions, and, together with datastore logs, help suppress duplicates. We develop failure recovery protocols which leverage clocks and logs to ensure correct recovery from the failure. In the extended version of our paper [18], we prove their correctness by showing that the recovered state is same as if no failure has occurred, thereby ensuring COE.

State externalization can potentially slow down performance of state reads/writes. Thus, for performance, CHC introduces **NF-aware algorithms** for shared state management. It uses scope-awareness of state objects to partition traffic so as to minimize cross-instance shared state coordination. It leverages awareness of the state access patterns of NFs to implement strategies for shared state caching. Because most NFs today perform a simple set of state update operations, CHC offloads operations to the state store, which commits them in the background. This speeds up shared state updates – all coordination is handled by the store which serializes the operations issued by multiple NF instances.

We built a multi-threaded C++ prototype of CHC along with four NFs. We evaluate this prototype using two campus-to-EC2 packet traces. We find that CHC's state management optimizations reduce latency overhead to **0.02μs - 0.54μs** per packet compared to traditional NFs (no state externalization). CHC failover offers **6X** better 75%-ile per packet latency than [29]. CHC is 99% faster in updating strongly

consistent shared state, compared to [16]. CHC obtains per-instance throughput of 9.42Gbps – **same as maximum achievable** with standalone NFs. CHC's support for chain-wide guarantees adds little overhead, but **eliminates false positives/negatives** seen when using certain security NFs in existing NFV frameworks. Thus, CHC is the only framework to support COE, and it does so at state-of-the-art performance.

2 Motivation

NFV allows operators to connect NFs together in chains, where each type of NF can use multiple instances to process input traffic demand. Use of software NFs and SDN [24] means that when incoming traffic load spikes, or processing is unbalanced across instances, operators can scale up by adding NF instances and/or reallocate flow processing across instances. Furthermore, hot-standby NFs can be used to continue packet processing when an instance crashes. Due to these benefits, cloud providers and ISPs are increasingly considering deploying NFV in their networks [4].

2.1 Key Requirements for COE

NFV chains are central to security and compliance policies, they must always operate correctly, i.e., ensure COE (§1). Ensuring COE is challenging: (1) NFs are stateful; they maintain state objects for individual and group of flows. These state objects may be updated on every packet and the value of these state objects may be used to determine the action on the packet. This requires support for fine gained NF state management. (2) In addition to this, COE also require that the per-NF and chain-wide state updates are consistent with the input packet stream. (3) Since chaining may create a dependency between the action taken in upstream instances and its downstream instances, it is important that the impact of a local action taken for failure recovery should be isolated from the rest of the chain. These challenges naturally map to three classes of requirements for supporting COE:

State Access: The processing of each packet requires access to up-to-date state; thus, the following requirement are necessary to ensure COE under dynamics:

- (R1) *State availability:* When an NF instance fails, all state it has built up internally disappears. For a failover instance to take over packet processing it needs access to the state that the failed instance maintained just prior to crashing.

- (R2) *Safe cross-instance state transfers:* When traffic is reallocated across NF instances to rebalance load, the state corresponding to the reallocated traffic (which exists at the old instance where traffic was being processed) must be made available at the reallocated traffic's new location.

Consistency: Action taken by a given NF instance may depend on shared-state updates made by other instances of the
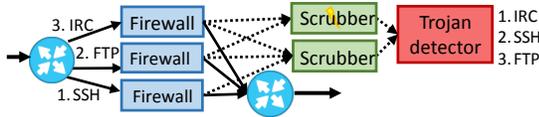
Figure 2: Illustrating violation of chain-wide ordering.

same NF, or state actions at upstream NFs in the chain. Ensuring that said NF instances' actions adhere to COE boils down to following requirements:

• (R3) *Consistent shared state:* Depending on the nature of an NF's state, it may not be possible to completely avoid sharing a subset of it across instances, no matter how traffic is partitioned (e.g., port counts at the IDSes in Figure 1a). Such state needs to be kept consistent across the instances that are sharing; that is, writes/updates made locally to shared state by different instances should be executed at all other instances sharing the state in the same global order. Otherwise, instances may end up with different views of shared state leading to inconsistent and hence incorrect actions.

• (R4) *Chain-wide ordering:* Some NFs rely on knowing the order in which traffic entered the network. Consider Figure 2. The off-path Trojan detector [12] works on a copy of traffic and identifies a Trojan by looking for this sequence of steps: (1) open an SSH connection; (2) download HTML, ZIP, and EXE files over an FTP connection; (3) generate IRC activity. When a Trojan is detected, the network blocks the relevant external host. A different order does not necessarily indicate a Trojan. It is crucial that the Trojan detector be able to reason about the true arrival order as seen at traffic input.

In Figure 2, either due to one of the scrubbers being slowed down due to resource contention or recovering from failure [29], the order of connections seen at the Trojan detector may differ from that in the traffic arriving at the input switch. Thus, the Trojan detector can either incorrectly mark Trojan traffic as benign, or vice versa. When multiple instances of the Trojan detector are used, the problem is compounded because it might not be possible to partition traffic such that all three flows are processed at one instance.

• (R5) *Duplicate suppression:* In order to manage straggler NFs, NFV frameworks can adopt the following approach: (a) deploy clones initialized with the state of a slow NF instance; (b) use packet replay to bring the clone up to speed with the straggler's state since state initialization; and (c) replicate packets to the straggler and clone (§5.3). Depending on when the clone's state was initialized, replay can lead to duplicate state updates at the straggler. Also, the original and clone instances will then both generate duplicate output traffic. Unless such duplicate updates and traffic are suppressed, the actions of the straggler and of downstream NFs can be impacted (spurious duplicates may trigger an anomaly). The need for duplicate suppression also arises during fault recovery (§5.4).

Isolation: NFs in a chain should not be impacted by failure recovery of other NFs. Specifically:

• (R6) *Safe chain-wide recovery:* When NF failures occur and recovery takes place, it is important that the state at each NF in the chain subsequent to recovery have the same value as in the no-failure case. In other words, actions taken during recovery should not impact the processing, state, or decisions of NFs upstream or downstream from the recovering NF — we will exemplify this shortly when we describe failings of existing systems in meeting this requirement.

The network today already reorders or drops packets. Our goal is to ensure that NF replication, chaining, and traffic reallocation together do not induce artificial ordering or loss on top of network-induced issues. This is particularly crucial for many important off-path NFs (e.g., DPI engines and exfiltration checkers) which can be thwarted by artificially induced reordering or loss.

2.2 Related work, and Our Contributions

A variety of NFV frameworks exist today [29, 16, 17, 25, 26, 20, 23, 11, 9, 28, 14, 32]. We review their drawbacks below.

Incomplete support for correctness requirements: Most existing frameworks focus on handling requirements R1 and/or R2. Split/Merge [26], OpenNF [16] and S6 [32] support cross-instance state transfers (R2). FTMB [29] and Pico Replication [25] focus on state availability (R1).

More fundamentally, Split/Merge, Pico Replication and FTMB focus on availability of the state contained entirely within an NF instance. They either ignore state shared across instances, or focus on the small class of NFs where such state is not used. Thus, these frameworks cannot handle R3.

Among existing frameworks, only OpenNF and S6 can support consistency for shared state (R3), but this comes at high performance cost. For example, OpenNF imposes a $166\mu s$ per packet overhead to ensure strong consistency! (§7). Similarly, S6 cannot support frequent updates to strongly consistent shared state.

Equally crucially, all of the above frameworks focus on a single NF; they cannot handle chains. Thus, none of them support chain-wide ordering (R4).

Support for R5 is also missing. StatelessNF [17] and S6 [32] update shared state in an external store or remote NF, respectively, but they do not support atomic updates to all state objects an instance can access. Thus, when a clone is created to mitigate a straggler off-path NF (as outlined above), the straggler may have updated other state objects that are not reflected in the clone's initialized state. Upon replay, the straggler can make duplicate state updates (likewise, duplicate packets can also arise). For the same reason, R6 is also violated: when an NF fails over, replaying packets to bring the recovery NF up to speed can result in duplicate processing in downstream NFs.

State management performance is poor: FTMB's periodic checkpointing significantly inflates NF packet processing latency (§7). As mentioned above, OpenNF imposes performance overhead for shared state. The overhead is high
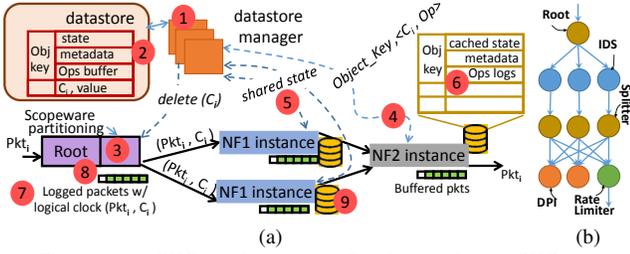
Figure 3: (a) CHC architecture; (b) Physical chain that CHC runs.

even for cross-instance transfers of per-flow state: this is because such transfers require extracting state from an instance and installing it in another while ensuring that incoming packets are directed to the state's new location.

Our contributions: How do we support requirements R1-R6 while ensuring good state management performance? Some NFs or operating scenarios may just need a subset of R1-R6. However, we seek a *single framework* that meets all requirements/scenarios because, with NFV becoming mainstream, we believe we can no longer trade-off general correctness requirements for performance or functionality (specific NFs). Thus, we identify *basic building blocks* and study how to synthesize them into one framework. We have set ourselves the ambitious goal of designing a single generic NFV framework to support all of these requirements, though some NFs may only need support for a subset of these requirements. Building such a framework is especially challenging because we must carefully deal with shared state and NF chaining.

Our system, CHC, has three building blocks (Figure 3a): We maintain NF state in an in-memory **state store external to NFs** (1; §4). NFs access the store to read/write relevant state objects. This ensures state availability (R1). The store's state object metadata simplifies reasoning about state ownership and concurrency control across instances (2; §4.3). This makes state transfer safety (R2) and shared state consistency (R3) simple and efficient (§5.1).

We propose **NF state-aware algorithms** for good state read/write performance which is a key concern with state externalization. These include (§4.3): automatic state scope-aware traffic partitioning to minimize shared-state coordination (3); asynchronous state updates for state that is updated often but read infrequently; this allows packet processing to progress unimpeded (4); NFs sending update operations, as opposed to updated state, to the store, which simplifies synchronization and serialization of shared-state updates (5); scope- and access pattern-aware state caching strategies, which balances caching benefits against making cache updates immediately visible to other instances (6).

Finally, we maintain a small amount of **metadata – clocks and logs**. We insert per packet logical clocks (7; §5) which directly supports cross-instance ordering (R4). We couple clocks with logs to support duplicate suppression (R5; §5.3) and COE under failover of NFs and framework components

(R6; §5.4). We log every packet that is currently being processed at some NF in the chain (8). Logged packets are replayed across the entire chain during failover. At the state store, we store logical clocks of packets along with the state updates they resulted in, which aids duplicate suppression. At each NF, we store packet clocks along with the update operations issued and the most recently read state value (9). Together with state store snapshots, these NF-side logs support COE under datastore recovery.

Though StatelessNF [17] first advocated for externalizing state, but it has serious issues. Aside from a lack of support for R4–R6, it lacks atomic state updates: when a single NF fails after updating some but not all state objects, a failover NF can boot up with incorrect state! It requires locks for shared state updates, which degrades performance. Also, it assumes Infiniband networks for performance.

3 Framework: Operator View

In CHC, operators define "logical" NF chains (such as Figure 1b) using a DAG API. We elide low level details of the API, such as how policies are specified, and focus on aspects related to correctness and performance. Each "vertex" of the DAG is an NF and consists of operator supplied NF code, input/output, configuration, and state objects. Edges represent the flow of data (packets and/or contextual output).

The CHC framework compiles the logical DAG into a physical DAG with logical vertex mapped to one or more *instances* (Figure 3b). For example, the IDS in Figure 1b is mapped to three instances in Figure 3b. The operator can provide default parallelism per vertex, or this can be determined at run time using operator-supplied logic (see below). CHC deploys the instances across a cluster. Each instance processes a partition of the traffic input to the logical vertex; CHC automatically determines the traffic split to ensure even load distribution (§4).

The CHC framework supports chain elastic scaling and straggler mitigation. Note that the logic, e.g., when to scale is not our focus; we are interested in high performance state management and COE during such actions. Nevertheless, we outline the operator-side view for completeness: operators must supply relevant logic for each vertex (i.e., scaling[1]; identifying stragglers[2]). CHC executes the logic with input from a "vertex manager", a logical entity is responsible for collecting statistics from each vertex's instances, aggregating them, and providing them periodically to the logic.

Based on user-supplied logic, CHC redirects traffic to (from) scaled up (down) NF instances or clones of straggler NFs. CHC manages state under such dynamic actions to ensure COE. CHC also ensures system-wide fault tolerance. It automatically recovers from failures of NFs or of

---

[1]e.g., "when input traffic volume increased by a certain θ"

[2]when an instance processing $\theta\%$ slower than other instances

CHC framework components while always preserving COE.

4 Traffic and State Management

We discuss how CHC processes traffic and manages state. The framework automatically partitions traffic among NF instances (§4.1) and manages delivery of packets to downstream NFs (§4.2). As packets flow, different NFs process them and update state in an external store; CHC leverages several algorithms for fast state I/O; the main challenge here is dealing with shared state (§4.3).

4.1 Traffic partitioning

CHC performs *scope-aware partitioning*: traffic from an upstream instance is partitioned across downstream instances such that: (1) each flow is processed at a single instance, (2) groups of flows are allocated to instances such that most state an instance updates for the allocated flows is not updated by other instances, and (3) load is balanced. #1 and #2 reduce the need for cross-instance coordination for shared state.

In CHC, state scope is a *first-class entity*. A function *.scope()* associated with a vertex program returns a list of scopes i.e., the set of packet header fields which are used to key into the objects that store the states for an NF; i.e., these are the different granularities at which states can be queried/updated. CHC orders the list from the most to least fine grained scope. Suppose the DPI vertex in Figure 1b has two state objects: one corresponding to records of whether a connection is successful or not; and another corresponding to the number of connections per host. The scope for the former is the 5-tuple (*src IP, dst IP, src port, dst port, protocol*); the scope for the latter is *src IP*.

CHC first attempts to partition traffic at instances immediately upstream (which, for the DPI in Figure 1b would be the IDSes) based on the most coarse-grained state scope (for the DPI this is *src IP*); such splitting results in no state sharing at the downstream (DPI) instances. However, being coarse grained, it may result in uneven load across instances. The framework gathers this information via the (DPI) vertex manager. It then considers progressively finer grained scopes and repeats the above process until load is even.

The final scope to partition on is provided in common to the *splitters* upstream. The framework inserts a splitter after every NF instance (Figure 3b). The splitter partitions the output traffic of the NF instance to instances downstream.

The *root* of a physical DAG is a special splitter that receives and splits input traffic. Roots can use multiple instances to handle traffic; in CHC, we fix root parallelism to some constant R. Network operators are required to statically partition traffic among the R roots such that the traffic processed by a root instance has no overlap in any of the 5-tuple dimensions with that processed by another instance.

Scope	Any	Per-flow	Cross-flow	Cross-flow
Access pattern	Write mostly, read rarely	Any	Write rarely (read heavy)	Write/read often
	Non-blocking ops. No caching	Caching \w periodic non-blocking flush	Caching \w callbacks	Depends upon traffic split. Cache, if split allows; flush periodically

Table 1: Strategies for state management performance

4.2 Communication

Inter-NF communication is asynchronous and non-blocking. Each NF's outputs are received by the CHC framework which is responsible for routing the output to downstream instances via the splitter. The framework stores all the outputs received from upstream instances in a queue per downstream instance; downstream instances poll the queue for input. This approach offers three benefits: (a) upstream instances can produce output independent of the consumption rate of downstream instances, (b) the framework can operate on queue contents (e.g., delete messages before they are processed downstream), which is useful for certain correctness properties, e.g., duplicate suppression (§5), (c) user logic can use persistent queues to identify stragglers/uneven load.

4.3 State Maintenance

CHC *externalizes* NF state and stores it in an external distributed key-value datastore. Thus, state survives NF crashes, improving availability and satisfying requirement R1 (§2). All state operations are managed by the datastore (Figure 3a). As described below, CHC incorporates novel algorithms and metadata to improve performance (Table 1).

State metadata: The datastore's client-side library appends metadata to the key of the state that an NF instance stores. This contains vertex_ID and instance_ID, which are immutable and are assigned by the framework. In CHC, the key for a per-flow (5 tuple) state object is: *vertex_ID + instance_ID + obj_key*, where obj_key is a unique ID for the state object. The instance_ID ensures that only the instance to which the flow is assigned can update the corresponding state object. Thus, this metadata simplifies reasoning about ownership and concurrency control. Likewise, the key for shared objects, e.g., *pkt_count*, is: *vertex_ID + obj_key*. All the instances of a logical vertex can update such objects. When two logical vertices use the same key to store their state, vertex_ID prevents any conflicts.

Offloading operations: Most NFs today perform simple operations on state. Table 2 shows common examples. In CHC, an instance can *offload operations* and instruct the datastore to perform them on state on its behalf (developed contemporarily with [32]). Developers can also load custom operations. The benefit of this approach is that NF instances do not have to contend for shared state. The datastore serializes operations issued by different instances for the same shared state object and applies them in the background (In

Operation	Description
Increment/ decrement a value	Increment or decrement the value stored at key by the given value.
Push/pop a value to/from list	Push or pop the value in/from the list stored at the given key.
Compare and update	Update the value, if the condition is true.

Table 2: Basic operations offloaded to datastore manager

[18], we prove that updates will always result in consistent state.). This offers vastly better performance than the natural approach of acquiring a lock on state, reading it, updating, writing it back, and releasing the lock (§7).

Non-blocking updates: In many cases, upon receiving a packet, an NF updates state, but does not use (read) the updated value; e.g., typical packet counters (e.g., [21, 22, 30]) are updated every input packet, but the updated value is only read infrequently. For such state that is written mostly and read rarely, we offer *non-blocking updates* (Table 1): the datastore immediately sends the requesting instance an ACK for the operation, and applies the update in the background. As a further optimization, NFs do not even wait for the ACK of a non-blocking operation; the framework handles operation retransmission if an ACK is not received before a timeout. If an instance wishes to read a value, the datastore applies all previous outstanding updates to the value, in the order NFs issued them, before serving the read.

Caching: For all the objects which are not amenable to non-blocking updates, we improve state access performance using novel caching strategies that leverage state objects' scope and access patterns (ready-heavy vs. not).

Per-flow state: CHC's scope-aware partitioning ensures that flows that update per-flow state objects are processed by a single instance; thus, these objects do not have cross-instance consistency requirements. The datastore's client-side library caches them at the relevant instance, which improves state update latency and throughput. However, for fault tolerance, we require local updates made to cached objects to be flushed to the store; to improve performance, these flush operations have non-blocking semantics (Table 1).

Cross-flow state: Cross-flow state objects can be updated by multiple instances simultaneously. Unlike prior works that largely ignore such state, CHC supports high performance shared state management. Some shared objects are rarely updated; developers can identify such objects as read-heavy. CHC (1) caches such an object at the instances needing them; and (2) the client-side library at each of these instances registers a *callback* with the store, which is invoked whenever the store updates the object on behalf of another instance. The NF developer does not need to provide callbacks to update state; they are handled by the client-side library.

The cached objects only *serve read requests*. Whenever an (rare) update is issued by an instance - operation is immediately sent to the store, The store applies the operation and sends back the updated object to the update initiator. At the same time, the client-side library of other instances re-

ceives callback from the store and updates the locally cached value (Table 1). We prove this approach results in consistent updates to shared state in [18]

For other cross-flow objects (not rarely-updated), the datastore allows them to be cached at an instance only as long as no other instance is accessing them (Table 1); otherwise, the objects are flushed. CHC notifies the client-side library when to cache or flush the state based on (changes to) the traffic partitioning at the immediate upstream splitter.

For **scale and fault tolerance** we use multiple datastore instances, each handling state for a subset of NF instances. Each datastore instance is multi-threaded. A thread can handle multiple state objects; however, each state object is only handled by a single thread to avoid locking overhead.

5 Correctness

So far, we focused on state management and its performance. We also showed how CHC supports requirement R1 (state availability) by design. We now show how it supports the requirements R2–R6. This is made challenging both by shared state and by chaining. To support R2-R6, CHC maintains/adds metadata at the datastore, NFs and to packets. We first describe how the most basic of the metadata – logical packet clocks and packet logs – are maintained. We describe other metadata along with the requirements they most pertain to.

Logical clocks, logging: The root (§4.1) attaches with every input packet a unique *logical clock* that is incremented per packet. The root also *logs* in the datastore each packet, the packet clock, and to which immediate downstream instance the packet was forwarded. When the last NF in a chain is done processing a packet, updating state and generating relevant output, it informs the CHC framework. CHC sends a "delete" request with the packet's clock to the root which then removes the packet from the log. Thus, at any time, the root logs all packets that are being processed by one or more chain instances. When any NF in the chain cannot handle the traffic rate, the root log builds in size; CHC drops packets at the root when this size crosses a threshold to avoid buffer bloat. When multiple root instances are in use (§4.1), we encode the identifier of the root instance into the higher order bits of the logical clock inserted by it to help the framework deliver "delete" requests to the appropriate root instance.

5.1 R2, R3: Elastic scaling

In some situations, we may need to reallocate ongoing processing of traffic across instances. This arises, e.g., in elastic scaling, where a flow may be processed at an "old" instance and reallocated to a "new" scaled up instance. We must ensure here that the old and new instances operate on the correct values of per- and cross-flow state even as traffic is reassigned (requirements R2 and R3).
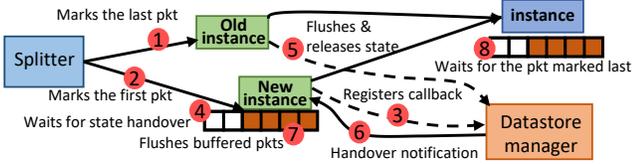
Figure 4: State handover.

Specifically, for cross-flow shared state, we require that: *updates made to the shared state by every incoming packet are reflected in a globally consistent order irrespective of which NF instance processed the corresponding packet.*

Existing systems achieve this at high overhead: OpenNF [16] copies shared internal state from/to the instances sharing it, each time it is updated by an incoming packet! In contrast, ensuring this property in CHC is straightforward due to externalization and operation offloading (§4.3): when multiple instances issue update operations for shared state, the datastore serializes the operations and applies in the background. All subsequent accesses to the shared state then read a consistent state value.

Per-flow state's handling must be correctly reallocated across instances, too (R2). One approach is to disassociate the old instance from the state object (by having the instance remove its instance_ID from the object's metadata) and associate the new instance (by adding its instance_ID). But, this does not ensure correct handover when there are in-transit packets that update the state: even if the upstream splitter immediately updates the partitioning rules and the traffic starts reaching the new instance, there might be packets in-transit to, or buffered within, the old instance. If the new instance starts processing incoming packets right away then state updates due to in-flight/buffered packets may be disallowed by the datastore (as a new instance is now associated with the state object) and hence the updates will be lost.

Thus, to satisfy R2, we require: *Loss-freeness, i.e., the state update due to every incoming packet must be reflected in the state object.* Furthermore, some NFs may also need *order-preservation: updates must happen in the order of packet arrivals into the network.*

These properties are crucial for off-path NFs, e.g., IDS. Such NFs cannot rely on end-to-end retransmissions to recover from lost updates induced by traffic reallocation [16]. Similarly, they may have to process packets in the order in which they are exchanged across two directions of a flow, and may be thwarted by a reordering induced by reallocation (resulting in false positives/negatives).

Figure 4 shows the sequence of steps CHC takes for R2: ① The splitter marks the "last" packet sent to the old instance to inform the old instance that the flow has been moved. This mark indicates to the old instance that it should flush any cached state associated with the particular flow(s) to the datastore and disassociate its ID from the per flow state, once it has processed the "last" packet. ② The splitter

also marks the "first" packet from the traffic being moved to the new instance. ③ When the new instance receives the "first" packet, it tries to access the per flow state from the datastore. If the state is still associated with the old instance_ID, it registers a callback with the datastore to be notified of metadata updates. ④ The new instance starts buffering all the packets associated with the flow which is being moved. ⑤ After processing the packet marked as "last", the old instance flushes the cached state and updates the metadata to disassociate itself from the state. ⑥ The datastore notifies the new instance about the state handover. ⑦ The new instance associates its ID with the state, and flushes its buffered packets.

The above ensure that updates are not lost *and* that they happen in the order in which packets arrived at the upstream splitter. In contrast, OpenNF provides *separate algorithms* for loss-freeness and order-preservation; an NF author has the arduous task of choosing from them!

Note also that packets may arrive out of order at a *downstream* instance, causing it to make out-of-order state updates. To prevent this: ⑧ The framework ensures that packets of the moved flow emitted by the new instance are not enqueued at the downstream instance, but instead are buffered internally within the framework until the packet marked as "last" from the old instance is enqueued at the new instance.

5.2 R4: Chain-wide ordering

To support R4, we require that: *Any NF in a chain should be able to process packets, potentially spread across flows, in the order in which they entered the NF chain.* CHC's logical clocks naturally allow NFs to reason about cross-flow chain-wide ordering and satisfy R4. E.g., the Trojan detector from §2.1 can use packets' logical clocks to determine the arrival order of SSH, FTP and IRC connections.

5.3 R5: Straggler mitigation

R5 calls for the following: *All duplicate outputs, duplicate state updates, and duplicate processing are suppressed.*

A key scenario in which duplicate suppression is needed is straggler mitigation. A straggler is a slow NF that causes the entire NF chain's performance to suffer. We first describe CHC's mechanism for straggler mitigation (which kicks in once user-provided logic identifies stragglers; §3), followed by duplicate suppression.

Clone and replay: To mitigate stragglers CHC deploys *clones*. A clone instance processes the same input as the original in parallel. CHC retains the faster instance, killing the other. CHC initializes the clone with the straggler's latest state from the datastore. It then replicates incoming traffic from the upstream splitter to the straggler and the clone.

This in itself is not enough, because we need to satisfy R2, i.e., ensure that the state updates due to packets that were
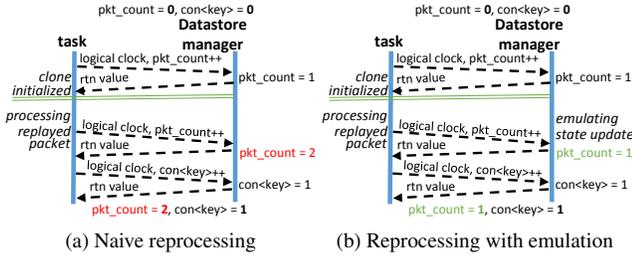
pkt_count = **0**, con<key> = **0**

task   **Datastore manager**

logical clock, pkt_count++

clone initialized   rtn value   pkt_count = 1

processing replayed packet

logical clock, pkt_count++

rtn value   pkt_count = 2

logical clock, con<key>++

rtn value   con<key> = 1

pkt_count = **2**, con<key> = **1**

(a) Naive reprocessing

pkt_count = **0**, con<key> = **0**

task   **Datastore manager**

logical clock, pkt_count++

clone initialized   rtn value   pkt_count = 1

processing replayed packet

logical clock, pkt_count++   *emulating state update*

rtn value   pkt_count = 1

logical clock, con<key>++

rtn value   con<key> = 1

pkt_count = **1**, con<key> = **1**

(b) Reprocessing with emulation

Figure 5: Duplicate update suppression

in-transit to the straggler at the time the clone's state was initialized are reflected in the state that the clone accesses. To address this, we *replay* all logged packets from the root. The root continues to forward new incoming packets alongside replayed ones. The clone processes replayed traffic first, and the framework buffers replicated traffic. To indicate end of replay traffic, the root marks the "last" replayed packet (this is the most recent logged packet at the time the root started replaying). When replay ends (i.e., the packet marked "last" was processed by the clone), the framework hands buffered packets to the clone for processing.

Given the above approach for straggler mitigation, there are three forms of duplicates that can arise. CHC suppresses them by maintaining suitable metadata.

1. Duplicate outputs: Replicating input to the clone results in duplicate outputs. Here, the framework suppresses duplicate outputs associated with the same logical clock at message queue(s) of immediate downstream instance(s).

2. Duplicate state updates: Some of the replayed packets may have already updated some of the stragglers' state objects. For example, an IDS updates both the total packet count and the number of active connections per host. A clone IDS may have been initialized after the straggler updated the former but not the latter. In such cases, processing a replayed packet can *incorrectly* update the same state (total packet count) multiple times at the straggler (Figure 5a). To address this, the datastore logs the state value corresponding to each state update request issued by any instance, as well as the logical clock of the corresponding packet. This is only done for packets that are currently being processed by some NF in the chain. During replay, when the straggler or clone sends an update for a state object, the datastore checks if an update corresponding to the logical clock of the replayed packet has already been applied; if so, the datastore *emulates* the execution of the update by returning the value corresponding to the update (Figure 5b). In [18], we describe how CHC handles non-deterministic state update operations.

3. Duplicate upstream processing: NFs upstream from the clone/straggler would have already processed some of the in-transit packets. In such cases, reprocessing replayed packets leads to incorrect actions at upstream NFs (e.g., an IDS may raise false alarms). To address this, each replayed packet is marked and it carries the ID of the clone where it will be processed. Such packets need special handling: the

intervening instances recognize that they are not suspicious duplicates; if necessary, the instances read the store for state corresponding to the replayed packet, make any needed modifications to the packet's headers, and produce relevant output; the instances can issue updates to state, too, but in such cases the datastore *emulates* updates as before. The clone's ID is cleared once it processed the packet.

5.4 R6: Safe Fault Recovery

Our description of R6 in §2 focused on NF failures; however, since CHC introduces framework components, we generalize R6 to cover other failures as well. Specifically, we require the following general guarantee:

Safe recovery Guarantee: *When an NF instance or a framework component fails and a recovery occurs, we must ensure that the state at each NF in the chain has the same value as under no failure.*

We assume the standard fail-stop model, that a machine/node can crash at any time and that the other machines/nodes in the system can immediately detect the failure.

First, we show how CHC leverages metadata to handle the failure of individual components. Then, we discuss scenarios involving simultaneous failure of multiple components.

NF Failover: When an NF fails, a failover instance takes over the failed instance's processing. The datastore manager associates the failover instance's ID with relevant state. Packet replay brings state up-to-speed (from updates due to in-transit packets). Similar to cloning (§5.3), we suppress duplicate state updates and upstream processing.

Since "delete" requests are generated after the last NF is done processing a packet, failure of such an NF needs special handling: consider such an instance T failing after generating an output packet for some input packet P, but before the framework sends a "delete" request for P. When P is replayed, T's failover instance produces output again, resulting in duplicate packets at the receiving end host. To prevent this, for the last NF in the chain, our framework sends the "delete" request for P *before* the NF generates the output packet. If the NF fails before the "delete" request, then P will be replayed, but this does not result in duplicate downstream processing since the NF did not generate output. If the NF fails after the "delete" request but before generating output, then P is not replayed, and hence the end host will not receive any output packet corresponding to P. To the host, this will appear as a packet being dropped by the network, causing P to be retransmitted from the source and resulting in correct overall behavior. In [18], we show that using this protocol an NF instance recovers with state similar to that under no failure.

Non-blocking operations: Non-blocking updates, where NF instances don't wait for ACKs, instead relying on the framework to handle reliable delivery, can introduce the following failure mode: a instance may fail after issuing state
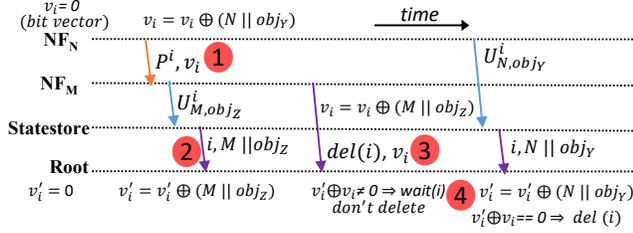
Figure 6: Recovery under non-blocking operations. Consider a packet P^i which is processed by NF_N, followed by NF_M, the last NF in the chain. NF_N and NF_M update objects obj_Y and obj_Z, respectively.

State operations and TS_{update} logs at the instance

I^1	U^9	U^{20}				U^{15}			U^{35}
I^2			U^{11}	U^{22}	U^{25}	R^{27} U^{30}			
I^3	U^8				U^{17}			R^{18} U^{23}	
I^4		U^{13}	R^{19}				U^{31}	U^{32}	

TS^{19}_{update} {20, 11, 8, 13} TS^{27}_{update} {15, 25, 17, 13} TS^{18}_{update} {15, 30, 17, 31}

Order of state update at the datastore   *Datastore crashed*

U^9 U^8 U^{13} U^{20} U^{11} R^{19} U^{22} U^{17} U^{25} U^{15} R^{27} U^{30} U^{31} R^{18} U^{23} U^{32} U^{35}

time: t   **time: $t + \delta$**

Figure 7: Recovering shared state at the datastore. I^k are instances. $U^{logical\_clock}$ and $R^{logical\_clock}$ represent "update" and "read".

update but before the update is committed and an ACK was received. In such cases, to ensure R6, we need that *the framework must re-execute the incomplete update operation.*

Suppose an instance N fails after processing packet P_i (i is the logical clock) but before the corresponding state update operation $U^i_{N,obj}$ (obj is the state object ID) completes. P_i may have induced such operations at a subset of NF instances $\{N\}$ along the chain. A natural idea to ensure the above property is to replay packets from the root to reproduce $U^i_{N,obj}$ at various N's. For this, however, P_i must be logged and should not have been deleted. If P_i is deleted it can't be replayed.

We need to ensure P_i continues to be logged as long as there is some N for which $U^i_{N,obj}$ is not committed. Our approach for this is shown in Figure 6: ① Each packet carries a 32-bit vector v_i (object ID and instance ID; 16b each) that is initialized to zero. Each NF instance where processing the packet resulted in a state update XORs the concatenation of its ID and the corresponding state objects' IDs into the bit vector. ② When committing a given NF's state update, the state store signals to the root the clock value of the packet that induced the update as well as the concatenated IDs. ③ The last instance sends the final vector along with its "delete" request to the root. ④ When a delete request and the final vector are received, the root XORs the concatenated IDs with the concatenated IDs reported by each signal from the state store in step 2. If the result is zero, this implies that updates induced by the packet at all NF instances $\{N\}$ were committed to the store; the root then proceeds to delete the packet from the log. Otherwise, the packet updated state at some NF, but the NF has not yet reported that the state was committed; here, the root does not delete the packet.

Root: To ensure R6 under root failover, we need that a new root must start with the logical clock value and current flow allocation at the time of root failure. This is so that the new root processes subsequent packets correctly. To ensure this, the failover root reads the last updated value of the logical clock from the datastore, and retrieves how to partition traffic by querying downstream instances' flow allocation. The framework buffers incoming packets during root recovery. We prove this approach ensures recovery with a state similar to that under no failure in [18].

Datastore instance: Recall that different NFs can store their states in different storage instances (§4.3). This ensures
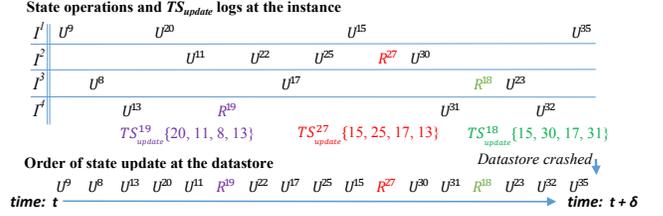
that store failures impact availability of only a portion of the overall state being tracked. Now, to ensure R6 under the failure of a datastore instance, we need that the recovered state in the new store instance must represent the value which would have resulted if there was no failure. The recovered state must also be consistent with the NF instances' view of packet processing thus far (i.e., until failure).

To support this property we distinguish between per-flow and shared state. For the former, we leverage the insight that all the NFs already maintain an updated cached copy of per-flow state. If a datastore instance fails, we can simply query the last updated value of the cached per-flow state from all NF instances that were using the store.

Recovering shared state is nuanced. For this, we use checkpointing with write-ahead logging [19]. The datastore periodically checkpoints shared state along with the metadata, "TS", which is the set of logical clocks of the packets corresponding to the last state operation executed by the store on behalf of each NF instance. Each instance locally writes shared-state update operations in a write-ahead log. Say the latest checkpoint was at time t and failure happens at $t + \delta$. A failover datastore instance boots with state from the checkpoint at t. This state now needs to be "rolled forward" to $t + \delta$ and made consistent with the NF instances' view of packet processing at $t + \delta$. Two cases arise:

(Case 1) If NF instances that were using the store instance don't read shared state in the δ time interval, then to recover shared state, the framework re-executes state update operations from the local write-ahead log on behalf of each NF, starting from the logical clocks included in the metadata TS in the checkpoint. Recall that in our design the store applies updates in the background, and this update order is unknown to NF instances. Thus, our approach ensures that the state updates upon re-execution match that produced by a *plausible* sequence of updates that the store may have enforced prior to failure. This consistency property suffices because, in Case 1, NFs are not reading shared state in the δ interval.

(Case 2) Say an NF instance issues a read between t and $t + \delta$; e.g., I^3 in Figure 7 issues R^{18}. Following the above approach may lead to an order of re-execution such that the actual state I^3 read in R^{18} is different from the state in the store after recovery. To ensure that the store's state is consistent with all I^k's current view, the framework must re-execute operations in such an order that the datastore would have pro-

	NF instance	Root
Store instance	✓*	✓*
NF instance	✓	✓

Table 3: Handling of correlated failures (Cannot recover if component and the store instance storing its state fail together).*

NF	Description of state object	Scope; access pattern
NAT	Available ports	Cross-flow; write/read often
	Total TCP packets	Cross-flow; write mostly, read rarely
	Total packets	Cross-flow; write mostly, read rarely
	Per conn. port mapping	Per-flow; write rarely, read mostly
Trojan detector	Arrival time of IRC, FTP and SSH flows for each host	Cross-flow; write/read often
Portscan detector	Likelihood of being malicious (per host)	Cross-flow; write/read often
	Pending conn. initiation req. along with its timestamp	Per-flow; write/read often
Load balancer	Per server active # of conn.	Cross-flow; write/read often
	Per server byte counter	Cross-flow; write mostly, read rarely
	Conn. to server mapping	Per-flow; write rarely, read mostly

Table 4: NFs and description of their state objects

duced the same value for each read in $[t, t + \delta]$.

To ensure this, on every read operation, the datastore returns TS along with the latest value of the shared state (e.g., TS^{19} is returned with I^4's R^{19}). The instance then logs the value of the shared state along with the corresponding TS. Re-execution upon failure then needs to select, among all TS's at different instances, the one corresponding to the most recent read from the store prior to the crash (i.e., TS^{18}, since R^{18} in the most recent read; most recent clock does not correspond to most recent read). How selection is done is explained shortly; but note that when the framework re-executes updates starting from the clock values indicated by this selected TS that would bring the store in sync with all NFs. In our example, TS^{18} is the selected TS; we initialize the store state with the value in the corresponding read (R^{18}). From the write-ahead log of each NF, the framework re-executes update operations that come after their corresponding logical clocks in TS^{18}. At instance I^1, this is the update after U^{15}, i.e., U^{35}. At I^3 and I^4 these are U^{23} and U^{32}, respectively. Shared state is now in sync with all NFs.

TS selection works as follows: first we form a set of all the TS's at each instance, i.e., $Set = \{TS^{18}, TS^{19}, TS^{27}\}$. Since the log of operations at an instance follows a strict clock order we traverse it in the reverse order to find the latest update operation whose corresponding logical clock value is in Set. For example, if we traverse the log of I^1, we find that the logical clock of U^{15} exists in Set. After identifying such a logical clock value, we remove all the entries from Set which do not contain the particular logical clock value (such TSs cannot have the most recent read); e.g., we remove TS^{19} as it does not contain logical clock 15. Similarly, we remove TS^{27}, after traversing I^2's log. Upon doing this for all instances we end up selecting TS^{18} for recovery. In [18], we prove that using this protocol the store recovers with state similar to that under no failure.

Correlated failures: Using the above approaches, CHC can also handle correlated failures (Table 3) of multiple NF instances, root, and storage instances. However, CHC cannot withstand correlated failure of a store instance with any other component that has stored its state in that particular instance. Replication of store instances can help recover from such correlated failures, but that comes at the cost of increasing the per packet processing latency.

6 Implementation

Our prototype consists of an execution framework and a datastore, implemented in C++. NFs runs in LXC containers [3] as multithreaded processes. NFs are implemented using our CHC library that provides support for input message queues, client side datastore handling, retransmissions of un-ACK'd state updates (§4.3), statistics monitoring and state handling. Packet reception, transmission, processing and datastore connection are handled by different threads.

For low latency, we leverage Mellanox messaging accelerator (VMA) [31] which enables user-space networking with kernel bypass similar to DPDK [13]. In addition to this, VMA also supports TCP/UDP/IP networking protocols and does not require any application modification. Even though we use VMA, we expect similar performance with other standard kernel bypass techniques. Protobuf-c [7] is used to encode and decode messages between a NF instance and the datastore. Each NF instance is configured to connect to a "framework manager" to receive information about it's downstream instances (to which it connects via tunnels), datastore instances and other control information.

The framework manager can dynamically change the NF chain by instantiating new types of NFs or NF instances and updating partitioning information in upstream splitters[3]. Our datastore implements an in-memory key-value store and supports the operations in Table 2. We reimplemented four NFs atop CHC. Table 4 shows their state objects, along with the state's scope and access patterns.

NAT: maintains the dynamic list of available ports in the datastore. When a new connection arrives, it obtains an available port from the datastore (The datastore pops an entry from the list of available ports on behalf of the NF). It then updates: 1) per-connection port mapping (only once) and, 2) (every packet) L3/L4 packet counters.

Portscan detector [27]: detects infected port scanner hosts. It tracks new connection initiation for each host and whether it was successful or not. On each connection attempt, it updates the likelihood of a host being malicious, and blocks a host when the likelihood crosses a threshold.

Trojan detector: implementation here follows [12].

Load balancer: maintains the load on each backend server. Upon a new connection, it obtains the IP of the least loaded server from the datastore and increments load. It then updates: 1) connection-to-server mapping 2) per server #connections and, 3) (every packet) per server byte counter.

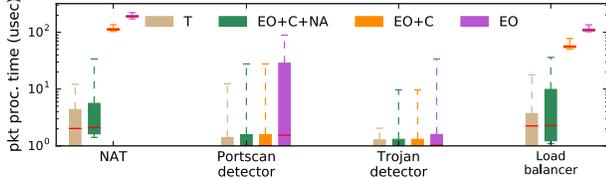[3] based on statistics from vertex managers

Figure 8: *5%ile, 25%ile, median, 75%ile and 95%ile pkt processing times. (T = Traditional NF, EO = Externalized state operations, C = with caching, NA = without waiting for the ACK)*

7 Evaluation

We use two packet traces (Trace{1,2}) collected on the link between our institution and AWS EC2 for trace-driven evaluation of our prototype. Trace1 has 3.8M packets with 1.7K connections and Trace2 has 6.4M packets with 199K connections. The median packet sizes are 368B and 1434B. We conducted all experiments with both traces and found the results to be similar; for brevity, we only show results from Trace2. We use six CloudLab [2] servers each with 8-core Intel Xeon-D1548 CPUs and a dual-port 10G NIC. One port is used to forward traffic, and the other for datastore communication and control messages. To process at 10Gbps, each NF instance runs multiple processing threads. CHC performs scope-aware partitioning of input traffic between these threads. Our datastore runs on a dedicated server.

7.1 State Management Performance

Externalization: We study three models which reflect the state access optimizations discussed in (§4.3): #1) All state is externalized and non-blocking operations are used. #2) Further, NFs cache relevant state objects. #3) Further, NFs do not wait for ACKs of non-blocking operations to state objects; the framework is responsible for retransmission (§4.3). The state objects per NF that benefit from #2 and #3 can be inferred from Table 1 and Table 4; e.g., for NAT, per-connection port mapping is cached in #2, and the two packet counters benefit from non-blocking updates in #3. We compare these models with a "traditional" NF where all state is NF-local. We study each NF type in isolation first.

Figure 8 shows the per packet processing times. The median times for traditional NAT and load balancer are $2.07\mu s$ and $2.25\mu s$, respectively. In model #1, this increases by $190.67\mu s$ and $109.87\mu s$, respectively, with network RTT contributing to most of this (e.g., NAT needs three RTTs on average per packet: one for reading the port mapping and other two for updating the two counters). We don't see a noticeable impact for scan and Trojan detectors (they don't update state on every packet).

Relative to #1, caching (#2) further lowers median processing times by $111.98\mu s$ and $55.94\mu s$ for NAT[4] and load balancer. For portscan and Trojan detector, reduces it by $0.54\mu s$ and $0.1\mu s$ (overhead becomes **+0.1μs** as compared to

---

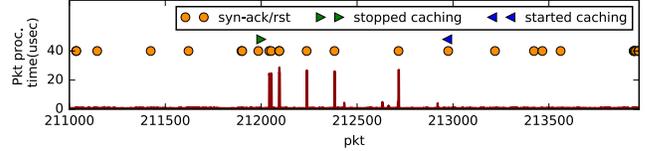[4]NAT needs 2 RTTs to update counters as port mapping is cached.

Figure 9: *Per packet processing latency with cross-flow state caching*

traditional NFs) as CHC caches the cross-flow state. Later, we evaluate the benefits of cross-flow caching in detail. Finally, #3 results in median packet processing times of **2.61μs** for NAT (which now needs 0 RTTs on average) and **2.27μs** for load balancer. These represent small overheads compared to traditional NFs: **+0.54μs** for NAT, and **+0.02μs** for the load balancer (at the median). Note that for portscan and Trojan detector the performance of #3 is comparable to #2 as they don't have any blocking operations.

We constructed a simple chain consisting of one instance each of NAT, portscan detector and load balancer in sequence, and the Trojan detector operating off-path attached to the NAT. With model #3, the median end-to-end overhead was **11.3μsec** compared to using traditional NFs.

Operation offloading: We compare CHC's operation offloading against a naive approach where an NF first reads state from the datastore, updates it, and then writes it back. We turn off caching optimizations. We now use two NAT instances updating shared state (available ports and counters). We find that the median packet processing latency of the naive approach is **2.17X** worse ($64.6\mu s$ vs $29.7\mu s$), because it not only requires 2 RTTs to update state (one for reading the state and the other for writing it back), but it may also have NFs wait to acquire locks. CHC's aggregate throughput across the two instances is >2X better.

Cross-flow state caching: To show the performance of our cross-flow state caching schemes (Table 1; Col 5), we run the following experiment: we start with a single portscan detector. After it has processed around 212K packets, we add a second instance and split traffic such that for particular hosts, captured by the set $\mathcal{H}$, processing happens at both instances. At around 213K packets, we revert to using a single instance for all processing. Figure 9 shows the benefits of caching the shared state. At 212K packets, when the second instance is added, the upstream splitter signals the original instance to flush shared state corresponding to hosts $\in \mathcal{H}$ (Table 4). From this point on, both instances make blocking state update operations to update the likelihood of hosts $\in \mathcal{H}$ being malicious on every successful/unsuccessful connection initiation. Thus, we see an increase in per packet processing latency for every SYN-ACK/RST packet. At packet number 213K, all processing for $\mathcal{H}$ happens at a single instance which can start caching state again. Thus, the processing latency for SYN-ACK/RST packets drops again, because now state update operations are applied locally and updates are flushed in a non-blocking fashion to the store.

Throughput: We measure degradation in per NF throughput for models #1 and #3 above compared to traditional NFs.
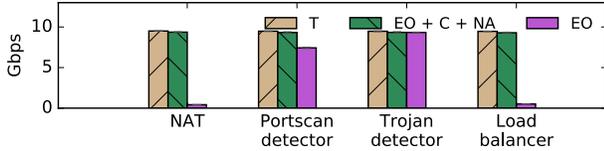
Figure 10: Per instance throughput. (T = Traditional NF, EO = Externalized state operations, C = with caching, NA = without waiting for the ACK)

Figure 11: State sharing.

Figure 12: Fault recovery.

Figure 10 shows that the max. per NF throughput for traditional NFs is around 9.5Gbps. Under model #1, load balancer and NAT throughput drops to 0.5Gbps. The former needs to update a byte counter (which takes 1 RTT) on every packet; likewise, the NAT needs three RTTs per packet. The port scan and Trojan detectors do not experience throughput degradation because they don't update state on every packet. Model #3 increases throughput to 9.43Gbps, matching traditional load balancer and NAT. We repeated our experiment with the aforementioned single-instance NF chain and observed similar maximal performance (9.25Gbps with both CHC and traditional NFs) in Model #3.

Datastore performance We benchmarked the datastore using the workload imposed by our state operations. We used 128bits key and 64bits value to benchmark the datastore. The datastore was running four threads. Each thread handled 100k unique entries. As discussed in §4.3, state is not shared between these threads. We found that a single instance of our datastore supports ∼**5.1M** ops/s (increment at 5.1M ops/s, get at 5.2M ops/s, set at 5.1M ops/s; Table 2). The datastore can be easily scaled to support a greater rate of operations by simply adding multiple instances; each state object is stored at exactly one store node and hence no cross-store node coordination is needed.

7.2 Metadata Overhead

Clocks: The root writes packet clocks to the datastore for fault tolerance. This adds a $29\mu s$ latency per packet (dominated by RTT). We optimize further by writing the clock to the store after every n^{th} packet.[5] The average overhead per packet reduces to $3.5\mu s$ and $0.4\mu s$ for $n = 10, 100$.

Packet logging: We evaluated two models of logging: 1) locally at the root, 2) in the datastore. The former offers better performance, adding $1\mu s$ latency per packet, whereas the latter adds $34.2\mu s$ but is more fault tolerant (for simultaneous root and NF failures). We also studied the overhead imposed by the framework logging clocks and operations at NFs, the datastore logging clocks and state, and the XOR-ing of identifiers (§5.4); the performance impact for our chain (latency and throughput overhead) was negligible ($< 1\%$).

XOR check and delete request: (§5.4) XOR checks of bit vectors are performed asynchronously in the background and do not introduce any latency overhead. However, ensur-

ing the successful delivery of "delete" request to root before forwarding the packet introduces a median latency overhead of $7.9\mu sec$. Asynchronous "delete" request operation eliminates this overhead but failure of the last NF in a chain may result in duplicate packets at the receiver end host.

7.3 Correctness Requirements: R1–R6

R1: State availability: Using our NAT, we compare FTMB's [29] checkpointing approach with CHC writing all state to a store. We could not obtain access to FTMB's code; thus, we emulate its checkpointing overhead using a queuing delay of $5000\mu s$ after every 200ms (from Figure 6 in [29]). Figure 12 (with 50% load level) shows that checkpointing in FTMB has a significant impact: the 75th%-ile latency is $25.5\mu sec$ – which is **6X** worse than that under CHC (median is 2.7X worse). FTMB's checkpointing causes incoming packets to be buffered. Because of externalization in CHC, there is no need for such checkpointing. Also, FTMB does not support recovery of the packet logger [29]. CHC intrinsically supports this (§5.4), and we evaluate it in §7.3.

R2: Cross-instance state transfers: We elastically scale up NAT as follows: we replay our trace for 30s through a single instance; midway through replay, we reallocate 4000 flows to a new instance, forcing a move of the state corresponding to these flows. We compare CHC with OpenNF's loss-free move; recall that CHC provides both loss-freeness and order preservation. CHC's move operation takes 97% **or 35X** less time (0.071ms vs 2.5ms), because, unlike OpenNF, CHC does not need to transfer state. It notifies the datastore manager to update the relevant instance_IDs. However, when instances are caching state, they are required to flush cached state operations before updating instance_IDs. Even then, CHC is 89% better because it *flushes only operations*.

R3: Cross-instance state sharing: We compare CHC against OpenNF w.r.t. the performance of strongly consistent shared state updates across NAT instances, i.e., updates are serialized according to some global order. Figure 11 (with 50% load level) shows that CHC's median per-packet latency is 99% lower than OpenNF's ($1.8\mu s$ vs 0.166ms). The OpenNF controller receives all packets from NFs; each is forwarded to every instance; the next packet is released only after all instances ACK. CHC's store simply serializes all instances' offloaded operations.

R4: Chain-wide ordering: We revisit the chain in Figure 2. Each scrubber instance processes either FTP, SSH, or

---

[5]After a crash, this may lead the root to assign to a packet an already assigned clock value. To overcome this issue, the root starts with $n + last$ $update$ so that clock values assigned to packets represent their arrival order.

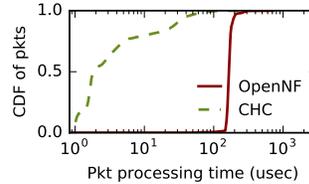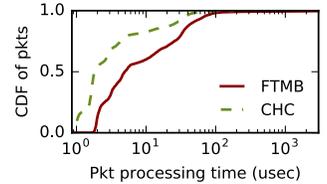|	30%load	50%load
Duplicate packets	13768	34351
Duplicate state updates	233	545

Table 5: Duplicate packet and state update at the downstream portscan detector without duplicate suppression.

Figure 13: Packet proc time.

Figure 14: Store recovery.

IRC flows. To measure the accuracy of the Trojan detector, we added the signature of a Trojan at 11 different points in our trace. We use three different workloads with varying upstream NF processing speed: W1) One of the upstream NFs adds a random delay between 50-100μs to each packet. W2) Two of the upstreams add the random delay. W3) All three add random delays. We observed that CHC's use of chain-wide logical clocks helps the Trojan detector identify **all 11 signatures**. We compare against OpenNF which does not offer any chain-wide guarantees; we find that **OpenNF misses 7, 10, and 11 signatures** across W1–W3.

R5: Duplicate suppression: Here, we emulated a straggler NAT by adding a random per packet delay between between 3-10μs. A portscan detector is immediately downstream from the NAT. CHC launches a clone NAT instance according to §5.3. We vary the input traffic load. Table 5 shows the number of duplicate packets generated by the NAT instances under different loads, as well as the number of duplicate state updates at the portscan detector – which happen whenever a duplicate packet triggers the scan detector to *spuriously* log a connection setup/teardown attempt. Duplicate updates create both false positives/negatives and their incidence worsens with load. No existing framework can detect such duplicate updates; CHC simply suppresses them.

R6: Fault Tolerance: We study CHC failure recovery.

NF Failure: We fail a single NAT instance and measure the recovery and per packet processing times. Our NAT performs non-blocking updates without waiting for the framework ACK; here, we use the 32bit vector (§5.4) to enable recovery of packets whose non-blocked operations are not yet committed to the store. To focus on CHC's state recovery, we assume the failover container is launched immediately. Figure 13 shows the average processing time of packets that arrive at the new instance at two different loads. The average is calculated over 500μs windows. Latency during recovery spikes to over 4ms, but it only takes **4.5ms** and **5.6ms** at 30% and 50% loads, respectively, for it to return to normal.

Root failure: Recovering a root requires just reading the last updated logical clock from the datastore and flow mapping from downstream NFs. This takes $< 41.2\mu$s.

Datastore instance failure: Recovering a datastore instance failure requires reading per-flow state from NFs using it, and replaying update operations to rebuild shared state. Reading the latest values of per-flow state is fast. Recovering shared state however is more time-consuming. Figure 14 shows the time to rebuild shared state with 5 and 10 NAT instances updating the same state objects at a single store instance. We replayed the state update operation logs generated by these instances. The instances were processing
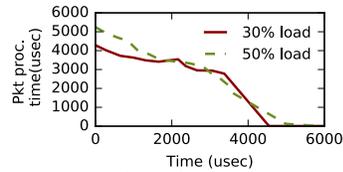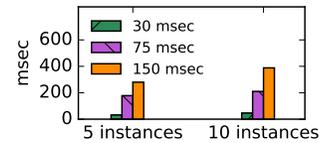
9.4Gbps of traffic; periodic checkpoints occurred at intervals of 30ms, 75ms, and 150ms. The recovery time is $\leq$ **388.2ms** for 10 NATs with checkpoints at 150ms intervals. In other words, a storage instance can be quickly recovered.

8 Conclusion

We presented a ground-up NFV framework called CHC to support COE and high performance for NFV chains. CHC relies on managing state external to NFs, but couples that with several caching and state update algorithms to ensure low latency and high throughput. In addition, it leverage simple metadata to ensure various correctness properties are maintained even under traffic reallocation, NF failures, as well as failures of key CHC framework components.

References

[1] Cisco Network Service Header: draft-quinn-sfc-nsh-03.txt. `https://tools.ietf.org/html/draft-quinn-sfc-nsh-03`.

[2] Cloud lab. `http://cloudlab.us/`.

[3] LXC - Linux containers . `https://linuxcontainers.org/lxc/introduction/`.

[4] Network functions virtualisation – update white paper. `https://portal.etsi.org/nfv/nfv_white_paper2.pdf`.

[5] Network functions virtualisation: Introductory white paper. `http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf`.

[6] NFV Management and Orchestration: An Overview. `https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf`.

[7] Protobuf-c. `https://github.com/protobuf-c/protobuf-c`.

[8] A. Anand, V. Sekar, and A. Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.

[9] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 49–60, 2012.

[10] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, P. Yegani, J. Guichard, and P. Quinn. Differentiated Network-Located Function Chaining Framework. Internet-Draft draft-boucadair-network-function-chaining-02, IETF Secretariat, July 2013.

[11] A. Bremler-Barr, Y. Harchol, and D. Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 511–524. ACM, 2016.

[12] L. De Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[13] Intel. Data Plane Development Kit. http://dpdk.org/.

[14] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 543–546, 2014.

[15] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical report, Technical Report, 2013.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174. ACM, 2014.

[17] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, 2017.

[18] J. Khalid and A. Akella. Correctness and performance for stateful chained network functions. https://arxiv.org/abs/1612.01497, 2018.

[19] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[20] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[21] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 3–3, 1998.

[22] PRADS. https://gamelinux.github.io/prads/.

[23] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42. ACM, 2015.

[24] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38. ACM, 2013.

[25] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 1. ACM, 2013.

[26] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. 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)*, pages 227–240, 2013.

[27] S. Schechter, J. Jung, and A. Berger. Fast detection of scanning worm infections. In *Recent Advances in Intrusion Detection*, pages 59–81. Springer, 2004.

[28] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, 2012.

[29] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, pages 227–240. ACM, 2015.

[30] Squid. `http://www.squid-cache.org/`.

[31] Mellanox. Messaging Accelerator (VMA). `http://www.mellanox.com/page/software_vma`.

[32] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI'18*, 2018.