Horde: Scalable Runtime Verification of Shardable Networked Systems

Paper # 162, 12 pages + references + appendix

Abstract

Network functions like firewalls, proxies, and NATs are instances of distributed systems that lie on the critical path for a substantial fraction of today's cloud applications. Unfortunately, validating these systems remains difficult due to their complex stateful, timed, and distributed behaviors.

In this paper, we present the design and implementation of *Horde*, a runtime verification system for distributed network functions that achieves high expressiveness, fidelity, and scalability. Given a property of interest, *Horde* efficiently checks running systems for violations of the property with a scale-out architecture consisting of a collection of global verifiers and local monitors. To improve performance and reduce communication overhead, *Horde* includes an array of optimizations that leverage properties of networked systems to suppress provably unnecessary system events and to shard verification over every available local and global component. We evaluate *Horde* over several network functions including a NAT Gateway that powers a large cloud provider, identifying both design and implementation bugs in the process.

1 Introduction

An emerging bottleneck to correctness and availability in modern cloud systems are the various network functions (*e.g.*, firewalls, NATs, and load balancers) that interpose on the majority of application requests flowing to, from, and between servers in the cloud. Over time, these network functions have become increasingly complex. Today, many of these functions are full-fledged distributed systems whose correctness depends on the coordination of multiple devices as well as on stored state and system timing.

Configuration errors and software bugs in these components can have an outsized impact on SLAs [1, 2] not only because of the complexity of these systems, but also because they are on the critical path of most application requests. For instance, a production NAT gateway we verify in this work manages (replicated) states for millions of flows and errors in it can lead to black holes, broken connectivity, forwarding loops, and more. Internal incident logs over the course of just a few weeks (late October to early November of 2019) show errors like these causing at least four outages, and public incident reports from providers show similar effects [1, 3, 4].

To improve availability, recent proposals suggest using *static verification* to prove the correctness of these systems [5–8]. A key challenge for static verification techniques is the problem of state explosion: checking the validity of the system for every possible input (*i.e.*, packets) and ordering of events is prohibitively expensive. Broadly speaking, these proposals respond in one of a few ways. The first is to require the use of special programming languages or other types of programmer interaction [8, 9]. The second is to use model checking techniques to more efficiently explore all possible system behaviors. Finally, many systems—to reduce the state space they must verify and to make verification more tractable—limit the set of verifiable behaviors, *e.g.*, to those that are compact [5], unordered [6], or abstract [10].

All of these approaches come with significant drawbacks. With the first, programmers are saddled with a substantial burden that can overwhelm the development of the system. With the second, model checking still typically relies on hand-written models of functionality, which may be difficult to provide for a rapidly evolving or complex system. Finally, limiting the scope of verification fails to extend to the increasingly complex services found in modern networks—services that arguably need verification the most.

Runtime verification is an alternative approach that sidesteps many of the above limitations. In runtime verification, a tool extracts information about the current state of a running system—testbed, canary, or production—to verify that invariants hold throughout execution [11–15]. Compared to static verification, runtime verification tests control flow, workloads, and implementations actually used in practice and does not require hand-written models. It is often used in conjunction with other testing and verification tools as a backstop for finding bugs. The challenge, particularly in network functions, is the need, at runtime, to: (1) reason about the coordination between events issued at different locations, (2) efficiently aggregate global state after each individual event, and (3) scale sub-linearly with the size of the original system—a verifier that requires the same amount of resources as the system itself is untenable for most production environments.

In this paper, we present the design of a scale-out, runtime verification tool for network functions called *Horde* that overcomes the above challenges. *Horde* provides a simple, but expressive language for specifying invariants with a focus on supporting network functions. Examples of network-focused language features included in *Horde*, but uncommon in other runtime verification systems are support for properties that are parametric over the "location" of events, properties that reference stateful variables, the ability to execute partial matches over packet fields, and support for temporal predicates.

Horde translates these invariants to a set of symbolic automata that can efficiently verify the current global state of the system. In addition, to ensure that the system can scale out to a near-unlimited number of machines, *Horde* implements the core of these checks on top of production stream processing systems [16,17]. To efficiently coordinate between distributed verifiers, *Horde* relies on hardware-supported time synchronization protocols like PTP. Finally, to minimize the overhead of the verification system, *Horde* leverages observations that network events/invariants are typically:

Flow- or connection-based: For most network functions, correctness is defined on a per-flow or per-connection basis. From the invariants, *Horde* derives sharding keys that allow it to distribute the verification task across independent workers. These shards also expose boundaries on which we can gracefully scale down the system to a sampled subset of the input.

Partially suppressible: Rather than aggregate all events in the system to a logically centralized verifier, most network events have limited windows of relevance depending on the state of the system, e.g., only if the connection has recently been closed. *Horde* includes an optimization scheme to suppress such messages before they ever leave the NF instance.

We note that *Horde* does not guarantee perfect accuracy under asynchrony—to do so would require atomicity guarantees in the critical path of the network functions. *Horde* instead chooses to handle these situations speculatively and notify users after-the-fact about transient inconsistency (Section 7.3). Despite this, *Horde* identified at least four bugs in an early (limited) deployment of a real distributed network function: a NAT gateway (NATGW) of a large public cloud.

To summarize, our work makes the following contributions:

- We present a case study of the needs of a large modern network function from our production cloud. The system exhibits several interesting characteristics and suggest key requirements for verifier design.
- We synthesize ideas from timed regular expressions, symbolic automata, and parametric verification. To the best of our knowledge, ours is the first to demonstrate a concrete need and method for combining these concepts.
- We introduce the design and implementation of *Horde* a system for at-scale runtime verification. Among other innovations, *Horde* includes a novel method for computing location-dependent suppression of network events.
- We introduce a collection of invariants over distributed network functions and an evaluation of *Horde* using these functions and invariants.

2 Motivation: A Cloud-scale NAT Gateway

Our work is grounded in our experience with a large-scale production network function (NF). The function, which balances requests over a set of available servers, supports almost all incoming external traffic in one of the largest cloud providers in the world. Its design and requirements will serve as a motivating example for the remainder of this paper.

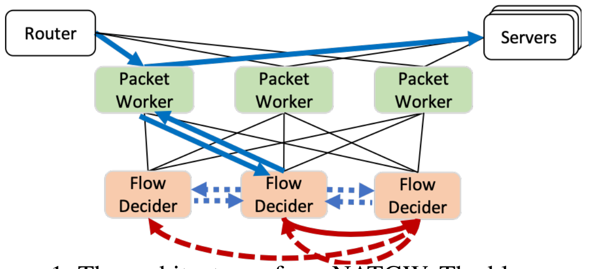


Figure 1: The architecture of our NATGW. The blue arrows show the path of the SYN packet of an incoming flow: it is spread across a set of packet workers, which send the packet to the flow decider in charge of that flow. The flow decider chooses a server to send the flow to, replicates the mapping, and installs it in the original packet worker. Red arrows trace the allocation of the mapping for the reverse flow.

Like other network functions of similar scale [18, 19], NATGW is implemented entirely in software, is distributed across a pool of servers, and replicates state for fault tolerance. Routers use ECMP-based anycast to randomly direct packets to a NATGW node, which then rewrites the destination IP and port to a target server. The same translation occurs for packets in the reverse direction (from the receiver to the sender).

The NATGW architecture is composed of two types of nodes (Figure 1): packet workers and flow deciders. Packet workers process every packet passing through the NATGW, parsing its header, looking up the target server, and rewriting the packet header to point to that target. The mapping of a flow to a target server is decided with the help of a sharded set of flow deciders. The flow deciders cache and replicate these mappings to other flow deciders to ensure availability.

Flow allocation. When the first packet of a new flow arrives at a packet worker, the packet worker uses a hash of the 5-tuple to identify the "primary" flow decider that owns the flow and forwards the packet to that decider. The primary then:

1. Decides which server to send the new flow to and installs the mapping in the local flow cache.
2. Sends the reverse mapping to the flow decider that "owns" the other end of the flow. Together, these two mappings cover translation for both incoming and outgoing traffic.
3. These primary deciders greedily copy their mappings to the cache of other flow deciders in a manner akin to chain replication: decider i will try to copy to deciders $(i+1)$ mod N and $(i+2)$ mod N, where N is the number of deciders. If one is down, it switches to $(i+3)$ mod N.
4. Next, the primary flow decider will install the mapping into the originating packet worker.

After the initial flow allocation, a packet worker can process all subsequent packets of the flow without coordination with other nodes. If the packet worker fails, anycast redirects the packet to another packet worker. The new worker sends the packet to the primary flow decider, fetching the existing mapping. If the flow decider fails, packet workers query the next deciders in the sequence until they find the mapping.

Flow mapping timeouts. All components time out their flow mappings to ensure stale entries are eventually removed.

To ensure NATGW maintains mappings for active flows, packet workers periodically send a keepalive message to the primary decider. The primary forwards the keepalive to all replicas, refreshing the timeout on every instance of the mapping in the system. In parallel, the primary forwards the keepalive to the primary in charge of the *reverse* mapping.

Eventual consistency. This NATGW design exhibits some interesting properties. One of them is a choice to allow for temporary inconsistency in the presence of node failures in order to satisfy certain practical and performance constraints.

For example, consider three replicas of a flow mapping P, A, and B, where P is the primary. To delete the mapping, P would send a delete request to both of the other nodes. Now imagine the message to A is dropped. Rather than waiting for A, the others will go ahead and delete f. If, later, P fails, packet workers will contact A for the mapping, which will return a stale/inconsistent result until a timeout or periodic sync eliminates the inconsistency.

We note that there are known mitigations to the above behavior (*e.g.*, querying a quorum on every packet or initiating a view change algorithm on P's failure); however, these come with significant performance costs. Instead, this system is an example of a *deployed* architecture that chooses eventual consistency after careful consideration of its drawbacks and alternative solutions. Our work is motivated by our operators' experience with such behaviors.

3 Design Goals

Our runtime verifier targets the following design goals:

Practicality. Network functions are complex, written in a variety of languages, and frequently rely on external libraries, drivers, and other components. NATGW, for example, is built using libraries like DPDK and interacts with an ecosystem of networking hardware and configurations. The intricacies of the systems, the richness of their dependencies, and the rapid evolution of all the associated components mean the system is not easily modeled or accurately simplified. Verification should be of the end-to-end system, *in situ*.

Horde should not place too much additional burden on developers: we should not require engineers to perform non-trivial proof writing (as mandated by many deductive reasoning techniques). NATGW has over 40 thousand lines of code—*Horde* should avoid incurring a proportional overhead.

Expressiveness. Prior work has observed a gap between state-of-the-art verification tools and the requirements of modern networks [11]. In particular, it is challenging to specify invariants having to do with: (1) parametric variables over values

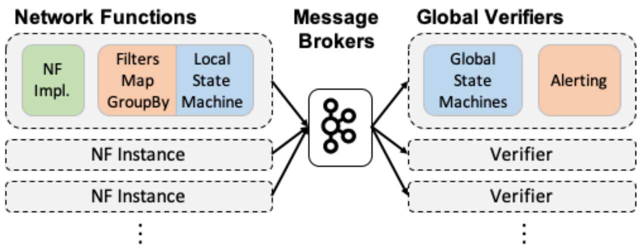


Figure 2: The architecture of *Horde*. NF instances generate and feed events into a set of local state machines. The NF instances use these state machines to determine if they can hide unnecessary messages before exporting the rest to the global verifier. These messages pass through a Kafka cluster and are streamed to a set of Flink-based verification engines.

like locations or identifiers, (2) coordination between network devices, and (3) timing of events. Moreover, since the number of devices (*e.g.*, flow deciders) may vary over time as the system scales out, it is useful to express properties in a way that does not require explicitly naming components.

Scalability. Just as a single machine cannot hope to handle all traffic entering a large network, it also cannot be expected to verify the correctness of the entire network. Rather, the verifer should be able to scale out to arbitrary size and require fewer resources than the original system. In pursuit of that goal, the verifier should attempt to minimize as much as possible the number of messages exported to a logically centralized aggregator, e.g., by exporting events (resulting from the execution of the NF) rather than packets (the inputs to the NF).

Graceful degredation of accuracy. As we describe in Section 7.3, perfect precision and recall is impossible in an asynchronous system without substantial overhead. Instead, our correctness goal is in the same spirit as NATGW's: perfect recall under the assumption of 'partial synchrony' [20] and notifications of potential false positives/negatives after-the-fact. Our operators find this is sufficient for most cases.

4 *Horde*'s Architecture

We present the design and implementation of a practical, expressive, and scalable verifier for large and complex NF deployments. Our system, *Horde*, is a combination of an invariant specification language and a scale-out runtime system. *Horde* takes a grey-box approach, requiring minimal changes and access to the underlying source code to export events of interest to the verifier. It verifies NF executions by:

Specifying invariants and exporting events. Operators write invariants over events. To provide them with sufficient expressiveness to check network-level events, *Horde* comes equipped with a new invariant language based on writing symbolic regular expressions over a global trace of events (and their locations) in the system. *Horde*'s language includes a notion of parameterized "variables" that allows invariants to be written in a way that holds for any combination of variable

```
{ "fields" : [
    {"eventType" : 16},
    {"nodeType" : 8},
    {"sourceIPv4or6" : 8},
    {"sourceIPv4or6==4" : [ {"srcIP" : 32} ],
     "sourceIPv4or6==6" : [ {"srcIP" : 128} ]},
    ...
],
"constants" : {
    "NAT_ALLOCATION" : 1,  // eventTypes
    "FLOWCACHE_CONSENSUS" : 769,
    "PACKET_WORKER" : 0,    // nodeTypes
    ...
}}
```

Figure 3: A snippet of the event schema for the NATGW.

instantiations subject to constraints.

NF developers export any relevant events, along with their PTP timestamp, to the verifier. Note that, because invariants are defined and checked *only* across related flows, we only need to know the correct ordering for events pertaining to those flows: the sub-microsecond-scale synchronization of PTP suffices for our needs. For many production networks, these types of event exports are already common.

Verifying the invariants. To scale up checking of these invariants, *Horde* does two things. First, it *Horde* automatically analyses and splits verification into local and global components in order to scale. The local level resides at the NF instances themselves, where *Horde* infers from local events whether it can suppress the event before exporting it to the global *Horde* verifier. Second, *Horde* leverages the fact that most network invariants are defined across *related* flows rather than globally—for instance, on the granularity of a 5-tuple. Events are automatically sharded across a cluster of scale-out stream processing workers using Kafka [21] and Flink [22].

Overview. Figure 2 shows *Horde*'s architecture. Users write a set of invariants that describe classes of incorrect behavior. *Horde* translates these to a set of symbolic automata and then splits the automata into local and global components. It then deploys these to NF instances and global verifiers.

At runtime, NF instances stream events into the pipeline. The local *Horde* agent filters and shards events, the message brokers aggregate and compact those streams, and the global verifiers determine, for the shard, whether a violation occurred. Kafka and Flink will automatically allocate resources and load balance requests to ensure scalability.

5 Specification Language

Users define system events and invariants over those events using two types of specifications that are inputs to *Horde*.

5.1 Event Definitions

Users specify the format of the event messages that arrive at the local verifier. *Horde* expects these messages to be in the form of packed arrays of raw binary data whose format is defined with a JSON configuration file. For example, Figure 3 shows a selected subset of the definition for NATGW event messages. 'fields' contains the ordered list of expected fields in the message. Each field is defined by a JSON dictionary specifying the field's name and its length in bits—for instance, the first 16 bits of the event message is an `eventType`.

Conditionals. In addition to specifying the length of each field and their ordering, *Horde* allows users to implement simple conditional parsing logic. The example event definition shows one such use where *srcIP* can be either IPv4 or IPv6. In the configuration shown, event messages include a 8-bit field that specifies the IP version number. Depending on the value of that version number, the next field is either a 32-bit `srcIP` field or a 128-bit field. These branches can define entire sub-headers and can contain nested conditionals.

Named constants. *Horde* allows users to define named constants (to use in invariant policies) which are integer values represented in decimal, hexadecimal, or binary notation. We show four such definitions of constants: two for values of the `eventType` field and two for the `nodeType` field.

```
1  FILTER((eventType == FLOWCACHE_PRIMARY_ADD
2        || eventType == FLOWCACHE_REMOVE_ENTRY)
3        && workerType == FD)
4  GROUPBY(srcIP, dstIP, srcPort, dstPort, proto)
5  MATCH
6  (eventType == FLOWCACHE_PRIMARY_ADD) @ $X
7  ((eventType == FLOWCACHE_REMOVE_ENTRY) @ NOT $X)*
8  (eventType == FLOWCACHE_PRIMARY_ADD) @ NOT $X
```

Figure 4: An example invariant specification that ensures at most one primary is ever active for a given flow.

5.2 Invariant Policy Definitions

Horde parses incoming event messages and checks them against a set of user-defined invariants. Operators write these invariants using *Horde*'s domain-specific language. Operators define invariants separately and their specification includes transformations and expressions.

We show an example invariant for our NATGW in Figure 4 which only pertains to a subset of events (lines 1 and 2). *Horde* checks it per-5-tuple (line 4). A violation occurs when a node `$X` adds a primary mapping (line 6) and then a different node (`NOT $X`) adds the same mapping as the primary (line 8) without `$X` removing it. The dollar sign syntax differentiates an arbitrary-valued variable from event fields and constants.

We show an example of an invariant for a stateful firewall in Figure 5. The invariant matches on a per-location basis where the firewall initializes a traffic for outbound flows from source IP `$S` and destination IP `$D` and later sees traffic in the reverse direction dropped. Any number of intervening events can occur between the initial and the dropped packet (line 4).

Appendix A illustrates the full invariant policy grammar.

```
FILTER(eventType == INIT || eventType == DROP)
GROUPBY(LOCATION)
MATCH
 (eventType==INIT, srcIp==$S, dstIp==$D) @ ANY
 (. @ ANY)*
 (eventType==DROP, srcIp==$D, dstIp==$S) @ ANY
```

Figure 5: An example invariant that ensures a stateful firewall does not drop reverse traffic for an open connection.

5.2.1 Transformations

In order to scale, *Horde* uses a set of user-defined and invariant-specific transformations. *Horde* uses these transformations to perform an initial filtering and aggregation and to identify valid sharding strategies. *Horde* currently supports three transformations: `GROUPBY`, `FILTER`, and `MAP`.

Operators can use `GROUPBY` to indicate which groups of flows the invariant pertains to. For example, when we wish to guarantee at most one primary is active (Figure 4) for *each flow*, the `GROUPBY` identifies unique flows. *Horde* uses `GROUPBY` to both simplify the invariant specifications and to assist in the sharding of invariant checking.

`FILTER` indicates which events the invariant cares about and the worker type at which those events occur. In our running example, we only care when a flow decider adds a flow as a primary and when they delete the flow mapping from the cache; we can filter all other types of events. These two transformations are critical for reducing the number of events handled by the verification framework.

To these, we add `MAP`, which computes a new field based on a mathematical expression over fields of the event message.

5.2.2 Invariant Expressions

We define invariants through a sequence of events that result in a violation of a particular policy. Users specify these sequences with a regular-expression-like language over the fields of the event message. In our example (Figure 4), these expressions come after the set of transformations and must appear after a `MATCH` statement.

As in regular expressions, the language describes patterns over pre-defined elements. In *Horde*'s case, elements are defined as a set of matching operations over the fields of the event message—the example shows matches on event types. A match can occur at any point in the stream of events and triggers on every occurrence of the match, not just the first.

As in other regular languages, users can list the sequence of expected elements and use operators like '*', '+', and '?' to signify repetitions. Users can also leverage the functions `CHOICE` and `SHUFFLE`. In `CHOICE`, an occurrence of any one of the contained expressions matches. In `SHUFFLE`, the contained events can arrive in any order, but must all arrive.

Locations. In distributed NFs, an important feature is that correct behavior is defined not only on the events and their order, but on *where* the events occurred. Therefore, every event match is accompanied by a location specifier. This is useful

```
MAP(srcIP < dstIP ? srcIP : dstIP, IP1)
MAP(srcIP < dstIP ? dstIP : srcIP, IP2)
MAP(srcIP < dstIP ? srcPort : dstPort, port1)
MAP(srcIP < dstIP ? dstPort : srcPort, port2)
FILTER(flag==FIN || flag==ACK || flag==FIN_ACK)
GROUPBY(IP1, IP2, port1, port2)
MATCH
  (flag == FIN) @ $X
  SHUFFLE(
    (flag == FIN, TIME == $s) @ $Y,
    (flag == ACK, TIME == $t) @ $Y)
  (flag == SYN, TIME - min($s, $t) <= 30000) @ $X
```

Figure 6: An example of a timing invariant that checks the behavior of TCP's TIME-WAIT state [23].

for specifying matches, but is also important for determining how we might partition evaluation of the invariant across local and global verifiers (see Section 6). In both cases, the goal of forcing the user to specify the location of the events is to determine whether each pair of events are expected to occur at the same or at difference NF instances.

We have one named location in Figure 4 (`$X`). Lines 7-8 indicate that events must occur at a different location than those of lines 6. But the invariant does not constrain the relationship between the locations of the events of lines 7-8. Users can name multiple locations and specify multiple predicates, *e.g.* to ensure event 1 is at `$X`; event 2 is at `$Y` (not `$X`); and event 3 is at `NOT $X`, `NOT $Y`, or event 4 is at location of `ANY`.

If a pattern match is ambiguous, *i.e.*, there are multiple ways to match the invariant to the pattern, then this could result in multiple (different) 'first occurrences' — *Horde* tracks all of them. For example, consider the invariant: `(A @ $X) (A @ $Y)` This invariant specifies that, for any two adjacent events, `A`, if the events occur at different locations, we have a violation. Thus, if we observe three events, `(A @ W1)`, `(A @ W2)`, and `(A @ W3)`, the invariant will match twice: once for {`$X=W1`, `$Y=W2`} and once for {`$X=W2`, `$Y=W3`}.

Simply enumerating all possible assignments of NF instances to variables would lead to an unacceptably inefficient implementation. Further, any change in membership would require us to fully recompile and re-install all invariants across the system. Instead, we lazily track potential candidates for location variables at runtime using a multi-leveled tree data structure, which we describe in detail in Section 6.

Variables. *Horde* generalizes the state tracking afforded to locations in order to track other types of state in the NF. Examples of non-location stateful properties include the IP/port NAT mappings of the NATGW and connection tracking in a firewall. As these variables do not indicate or impose restrictions on the location of the event, we do not use them for determining the partitioning of the invariant execution.

Timing. Timeouts and deadlines are common in NFs. To specify them, users can use parameterized variables to match an event timestamp field and leverage arithmetic operations to compare the time between multiple events. For example,

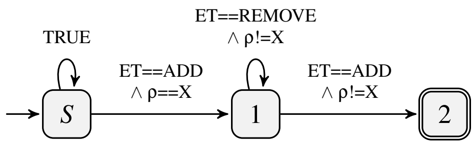


Figure 7: SFA for Figure 4 with some field names and constants abbreviated as well. ρ indicates location.

Figure 6 defines violations of TIME-WAIT semantics of TCP flows, assuming only packet sends are captured. This defines a violation where a SYN occurs within 30 s of the passive closer's FIN/ACK (more examples in Appendix B).

6 State Machine Generation

Horde checks for violations of invariants efficiently by translating each of the invariants into a state machine. In contrast to traditional finite-state automata, *Horde* requires a combination of complex features, e.g., timing, arithmetic, field/location variables, and regular expression event patterns.

Horde, thus, generates its state machines in three stages: it constructs a symbolic non-deterministic finite automaton (SFA) [24] that matches on a single specified violation, determinizes the SFA to a symbolic deterministic finite automaton (SDFA) to reduce runtime overhead, and constructs local state machines from the global SDFA.

6.1 Constructing the SFA

We first convert all predicates on events into boolean logic with equalities/inequalities by taking the conjunction of all event field matches and the location specifier. For example, we transform an event match `(A==B, C==D) @ NOT $X` to the predicate `(A==B ∧ C==D ∧ ρ!=X)`, where ρ is the placeholder for the event's location, which we determinize at runtime. A '`!`' modifier on the event would negate this predicate.

Horde performs an additional check on the sequence of generated predicates to facilitate efficient variable checking (Section 7.2). Specifically, it checks via reachability analysis that all uses of variables in either an arithmetic expression or non-equality comparison ($<$, $\leq$, $>$, and $\geq$) strictly follow after their introduction via an equality comparison.

With the checked predicates, *Horde* constructs the SFA by creating a start state, S, with a self-loop for any event (`TRUE`). This self-loop ensures the pattern will match starting from anywhere in the event trace. From the initial state S, *Horde* recursively builds out the state machine using Thompson's construction [25], treating **`CHOICE`** as a choice operator, and expanding **`SHUFFLE`** to all permutations. Figure 7 shows a (minimized) SFA for the example invariant from Figure 4. We mark the final state in the SFA as the accepting state, which indicates a violation when reached.

The specified transitions may not cover the complete space of possible events. All events that do not match any transition out of the current state can not lead to a match.

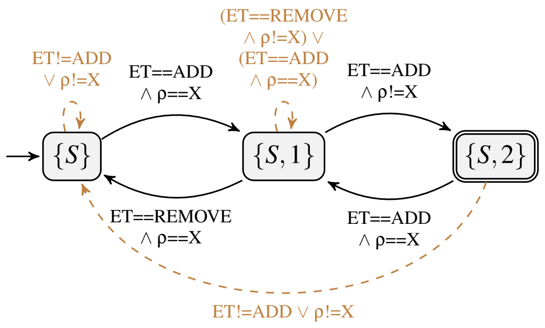


Figure 8: DSFA for the SFA in Figure 4. Colored, dashed edges represent suppressible transitions.

Horde next determinizes the SFA: it generates an efficiently executable DSFA from the SFA using standard symbolic automata techniques [24]. The result is a state machine where all transitions are unambiguous and exhaustive. Figure 8 shows the DSFA for the example. Each state in the DSFA stores the set of SFA states the machine is in at any given time.

6.2 Local State Machines

Given a DSFA, we could timestamp and send all events to the verifier, which would then apply the relevant transition and report a violations upon reaching an accepting state. However, doing so would require the verifier to process *all* events. We scale *Horde* and reduce the load on the network by generating a localized version of the state machine that is executed before sending the event to the global verifier.

6.2.1 Suppressible Transitions

By localizing the global state machine we find the set of events that will not impact the detection (or lack of detection) for a user-specified violation before sending them to the global verifier. Consider the DSFA in Figure 8. The three dashed transitions in the state machine represent *suppressible transitions*. Suppressible transitions match events we can hide from the verifier without impacting the detection of a violation:

Definition 1. An event stream s is either empty $s = \varepsilon$ or an event followed by another stream $s = e \cdot s'$.

Definition 2. We write $q \xrightarrow{e} q'$ to mean: from state q, event e transitions to state q'. We lift this to event streams inductively as $q \xrightarrow{\varepsilon} q$, and $q \xrightarrow{e \cdot s} q''$ iff $q \xrightarrow{e} q'$ and $q' \xrightarrow{s} q''$.

Definition 3. Transition t is suppressible if for any event e matching t from state q, then (1) $q \xrightarrow{e} q'$ means q' is not an accepting state, and (2) for any event stream s, and accepting state q_a then $q \xrightarrow{e \cdot s} q_a$ iff $q \xrightarrow{s} q_a$.

The two self-loops in Figure 8 are clearly suppressible (satisfy Definition 3) since an event processed by such a loop will not change the global state: (not) observing the event has no effect, and the loops do not occur on accepting states. Perhaps

Algorithm 1 Create a local state machine for a variable

```
1:  input: Global DSFA G, variable V, filter F
2:  output: Local DSFA L
3:  procedure CREATELOCALDFA(G, V, F)
4:      L := CopyStates(G)
5:      for S ← States(G) do
6:          for T ← Transitions(G, S) do
7:              P := Predicate(G, T)
8:              if SAT($(F \wedge P) \not\Rightarrow (\rho = V)$) then
9:                  AddTransition(L, TargetState(T), ε)
10:             P′ := Simplify(P, ρ==V)
11:             AddTransition(L, TargetState(T), P′)
12:     return Determinize(L)
```

less obvious is that the bottom-most edge is also suppressible since from either state $\{S\}$ or $\{S,2\}$ one needs to see the same two events to get back to the accepting state $\{S,2\}$. For example, an `ADD` event at `X` followed by another at `NOT X` will take either state $\{S\}$ or $\{S,2\}$ back to $\{S,2\}$. We never mark transitions with time constraints as suppressible—we assume the timing of an otherwise irrelevant event might still be significant. Using this approach we mark the suppressible edges of the global DSFA (Figure 8).

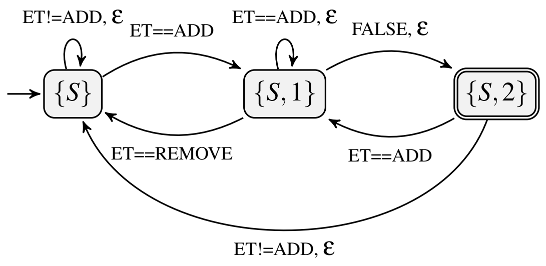
One can detect suppressible events using any number of heuristics, with better ones leading to more suppressed events.

6.2.2 Local State Machine Construction

Horde uses local knowledge to determine whether an event will be processed by a suppressible transition. Since each local component is unaware of what might be happening at other components, it must conservatively account for all possibilities. To determine (locally) whether an event is suppressible, we create a new state machine for each location variable where each state machine assumes it is playing the role of that location in matching some violation (*e.g.*, one machine for "I might be `X` in a violation" and another for "I might be `Y` in a violation"). Building off the previous example from Figure 8, we start by constructing a single local state machine for `X`:

The first step in creating a local state machine is to model the uncertainty other locations may introduce (Algorithm 1). The algorithm takes the global state machine G, a variable V (*e.g.*, `X`), and a predicate F corresponding to the user-defined **`FILTER`** statements, and returns a new localized SDFA.

The algorithm considers each transition T in G where T has predicate P, and checks whether the formula $(F \wedge P) \not\Rightarrow (\rho = V)$ is satisfiable (line 8). If so, then there exists a potential event that makes it through the filter F and uses transition T but which takes place at a location other than `X`. To model the fact that other NF instances might send events that use this transition, the algorithm adds a transition with ε to the local SFA L (line 9). An epsilon (ε) transition is one which the local SFA takes immediately and unconditionally. It accounts for the possibility of concurrent execution of other NF instances to represent that the global state could be in either state (the one before or the one after the ε transition).



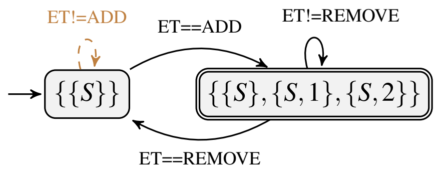


Figure 9: Local machine for location `X` from Figure 8. SFA is shown on top and its equivalent DSFA is shown below. Colored, dashed edges indicate locally suppressible transitions.

In either case, the algorithm then adds a local transition to L by simplifying the existing transition predicate (P) to account for the fact that the location is known (*e.g.*, `X`, line 11). It does so by partially evaluating the predicate with the assumption that ρ==V (line 10). In Figure 8, for example, the transition (`ET==REMOVE` ∧ ρ==`X`) is simplified to `ET==REMOVE`. Figure 9 shows the local SFA for location `X` and its determinized (DSFA) form. We will use this local DSFA throughout the rest of the running example.

By executing the DSFA in Figure 9 locally, an NF instance can learn some partial information about the state of the overall system. For example, assuming the NF is `X`, then after seeing an `ADD` event, the NF instance recognizes that the global state machine can be in any state: $\{S\}$, $\{S,1\}$, or $\{S,2\}$. However, after locally processing a `REMOVE` event, the local machine now knows it must be in state $\{S\}$ once more.

6.2.3 Local Event Suppression

The local machine can hide events when it can prove they would otherwise be processed by suppressible transitions in the global machine. Algorithm 2 is used to create all the data structures needed to suppress events locally. It takes the global state machine G as input along with the user-defined filters F, and produces as output a collection of local state machines (L_i) as well as a negated condition (NC).

The algorithm works by iterating over every location variable in the invariant (line 5) and calling CreateLocalDFA to build the local state machine (line 6). It walks over each local transition (T) and attempts to mark the transition as locally suppressible. To do so, it looks up all the global states the system can be in for this local state (line 11) and checks whether the local transition can process an event that is also processed by, and is not suppressible for, some global transition T′ from one of these states (line 16). If not, then every event must

Algorithm 2 Construct local state machines

```
1: input: Global DSFA G, filter F
2: output: Local state $\Theta = \langle\{L_1, \ldots, L_k\}, NC\rangle$
3: procedure LOCALIZE(G, F)
4:     NC := false, LS := $\emptyset$
5:     for V ← Variables(G) do
6:         L := CreateLocalDFA(G, V, F)
7:         for S ← States(L) do
8:             for T ← Transitions(L, S) do
9:                 suppress := true
10:                P := Predicate(L, T)
11:                for S′ ← GlobalStates(L, S) do
12:                    for T′ ← Transitions(G, S′) do
13:                        if CanSuppress(G, T′) then
14:                            continue
15:                        P′ := Predicate(G, T′)
16:                        if SAT($P \wedge (\rho = V) \wedge P'$) then
17:                            suppress := false
18:                if suppress then MarkSuppressed(L, T)
19:        LS := LS ∪ {L}
20:    for S′ ← States(G) do
21:        for T′ ← Transitions(G, S′) do
22:            if CanSuppress(G, T′) then continue
23:            NC := NC ∨ Simplify(Predicate(G, T′), ρ==Fresh())
24:    return ⟨LS, NC⟩
```

trigger a suppressible transition, so the event is suppressed.

In Figure 9, events matching `ET!=ADD` in state $\{\{S\}\}$ are suppressible: for each global state in the set ($\{S\}$), this event must be processed by the suppressible self-loop transition.

Negated condition. The final part of the algorithm (lines 20 to 23) computes a "negated condition." The local NF may not correspond to any named locations (*e.g.*, `X`) in the invariant: *Horde* must handle the case where the NF instance is not `X`, but may observe a relevant event (*e.g.*, as `NOT $X`). We observe, in such a case, the current machine can not possibly know anything about the global automaton state since the other NF instances that also are not `X` may be sending events that match `NOT $X` transitions. The strategy is simple: the algorithm computes the disjunction of all the transition predicates in the global state machine subject to the knowledge that the location ρ does not match any variable (line 23).

In the running example, the algorithm computes: (`ET==ADD` ∧ `Z==X`) ∨ (`ET==ADD` ∧ `Z!=X`) ∨ (`ET==REMOVE` ∧ `Z==X`), where `Z` is a fresh variable that is guaranteed to not match any location in the predicate. The above condition simplifies to **`ET==ADD`**. This means that the local machine *must* send any `FLOWCACHE_PRIMARY_ADD` events regardless of its local state. Note that non-location variables may introduce some uncertainty at the local verifier, which may not be sure what other NF instances have observed for their value. To address this, *Horde* first tries to generate a predicate that accounts for any possible variable assignment by enumerating all possible assignments from their $==/!=$ expressions, replacing their occurrences in the negated condition, and computing the disjunction of the resulting predicates. If any variables or arithmetic operations remain in the disjunction, *Horde* will simply not suppress any events, which is always safe.

7 Runtime System

We next describe the *Horde* runtime.

7.1 Workflow

We begin with the common case: NF instances synchronized via PTP send events, at runtime, to a co-located local agent via traditional IPC mechanisms. This local agent applies transformations, executes each of its local state machines, and keeps track of variables. The local agent sends non-suppressible events to the global verifier via a set of Kafka brokers.

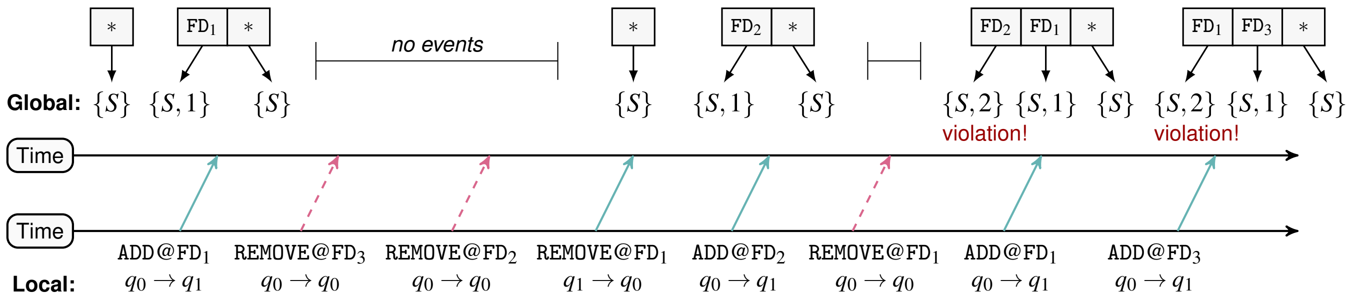
Filtering, mapping, and grouping. After ingesting the stream of PTP-timestamped events at each NF instances' local agent, the first task of *Horde* is to apply any applicable transformations—**`FILTER`**, **`MAP`** or **`GROUPBY`**—to the raw stream. As each invariant can have a different set of transformations, this may require *Horde* to duplicate the incoming stream of raw events (although it tries to avoid doing so when possible). The end result is a set of keyed event streams: one stream for each combination of invariant and **`GROUPBY`** key.

Computing suppression. The next such step, also preformed locally, is to determine the suppressibility of events in each keyed stream. *Horde* will pass the events through the localized state machines generated in Section 6.2.2—one for each location referenced in each invariant. For a given event and invariant, *Horde* suppresses the event when (1) all localized instances of the invariant would take a suppressible transition when fed the current event and (2) the event does not satisfy the negated condition. If either constraint is false, *Horde* sends the event to a Kafka queue for the given keyed event stream.

As a concrete example, Figure 10 shows processing of a series of events in the same **`GROUPBY`** group and the invariant in Figure 4). The first event is an `ADD` event at flow decider $\texttt{FD}_1$. After seeing this event, $\texttt{FD}_1$ will transition locally from state q_0 ($\{S\}$) to state q_1 ($\{\{S\},\{S,1\},\{S,2\}\}$). Since this transition is not suppressible, the event is sent to the verifier. The next event is a `REMOVE` event that takes place at $\texttt{FD}_3$. This particular transition *is* suppressible and the negated condition (`ET==ADD`) is not satisfied, thus, the event is suppressed.

This suppression can substantially reduce the number of events the verifier receives. For example, if we had 12 flow deciders then in a correct execution we would expect to receive—after suppression—just 2 out of 13 events (the primary add, the primary remove, and 11 suppressed non-primary removes). This number of exported events would remain constant regardless of the number of flow deciders.

Global state machines. Pulling from Kafka is a cluster of Flink instances running the non-localized versions of the invariant state machines. Although each keyed stream can contain events from every NF instance in the system, all the events



∗
FD1 ∗
no events
∗
FD2 ∗
FD2 FD1 ∗
FD1 FD3 ∗
Global:
{S} {S,1} {S}
{S} {S,1} {S}
{S,2} {S,1} {S}
violation!
{S,2} {S,1} {S}
violation!
Time
Time
ADD@FD1 REMOVE@FD3 REMOVE@FD2 REMOVE@FD1 ADD@FD2 REMOVE@FD1 ADD@FD1 ADD@FD3
Local:
q0 → q1 q0 → q0 q0 → q0 q1 → q0 q0 → q1 q0 → q0 q0 → q1 q0 → q1

Figure 10: Distributed execution for the invariant from Figure 4 on an example sequence of events for N flow deciders. Time progresses from left to right. Local events are shown along the bottom line with the local state of the flow decider. We use $q_0 = \{\{S\}\}$ and $q_1 = \{\{S\},\{S,1\},\{S,2\}\}$. The global verifier's state is shown at the top. Red, dashed edges indicate suppressed events.

are funneled to a single Kafka topic and Flink partition, each of which is automatically provisioned, checkpointed, and load balanced to worker nodes. As Flink does not guarantee that events from different NF instances will arrive in order, *Horde* stores these events and reorders them with an efficient priority queue before passing them to the associated state machine.

One challenge is how long to wait for delayed events. One approach is to maintain a list of all NF instances along with the timestamp of the last event they sent to this partition and only process time t when we have seen events from all instances up to $t + latency$. Unfortunately, most NF instances do not interact with most invariants and sending 'null' events to every partition would be costly. Instead, *Horde* relies on the assumption of a maximum latency T and handles violations of the assumption with the techniques in Section 7.3.

Horde will hold each event for T local time before passing it to the DSFA for processing. While processing events for a given invariant, the verifiers will track all of the possible states in which the associated state machine could be, as well as all potential values of the invariant's variables (see Section 7.2 for details). If any of the possible states is a 'final' state in the invariant's DSFA, *Horde* will raise an alert.

Consistent sampling. If scaling is still challenging despite sharding the verifier, filtering relevant events, and suppressing events locally, *Horde* provides a final mechanism that lets users trade performance for completeness by sampling a consistent set of events with consistent hashing based on the `GROUPBY` key (*e.g.*, a 5-tuple for NATGW). In this way, each group is itself complete though false negatives remain possible when violations occur for 5-tuples that are not sampled.

7.2 Variable Tracking

Horde needs to track the value of locations and other variables at runtime. *Horde* tracks *all* possible instantiations simultaneously using a multi-level tree data structure (the global state in Figure 10). Since there is only one variable (`X`) in the example, there is only one level in the tree. Intuitively, the tree captures the state the global automaton would be in for every possible instantiation of `X`. When the system starts, we are in state $\{S\}$ for any `X` (denoted by $*$). After the first `ADD` event arrives at the verifier from $\texttt{FD}_1$, we fork the tree to separate out the old case and a new case for `X`=$\texttt{FD}_1$. When `X` is $\texttt{FD}_1$, the verifier takes the transition ($\texttt{ET} == \texttt{ADD} \wedge \rho == X$) to state $\{S,1\}$: the current location ρ is $\texttt{FD}_1$, and `X` is also $\texttt{FD}_1$. Otherwise if `X`!=$\texttt{FD}_1$, it takes the self-loop transition to remain in $\{S\}$.

For the next event from $\texttt{FD}_1$ (`REMOVE`), there is no new case to fork, and applying the transition to both cases in the tree leads to both being in state $\{S\}$ once more. Therefore, the states are collapsed together back to $*$. This process continues until the second to last event where a violation is detected for the case where $X = \texttt{FD}_2$ due to a duplicate add at $\texttt{FD}_1$. The final event (`ADD` at $\texttt{FD}_3$) leads to a second violation, where now $X = \texttt{FD}_1$, and is subsequently caught by the implementation.

7.3 Fault Tolerance

Failures and message drops/delays can cause *Horde* to become desynchronized from the ground-truth state of the system. Even so, *Horde* is able to guarantee both precision and recall of typical network violations under the assumption of 'partial synchrony' [20], i.e., that there exists a time, T, after which there is some upper bound on message delivery time.

- *Recall* Under a partial synchrony assumption, *Horde*'s practice of creating a self loop in the initial state of the SFA means that all violations whose trace begins *after* T will be accurately modelled in the state machine and detected by *Horde*.
- *Precision: Horde*'s precision guarantees are less complete, but still hold in practice. Specifically, we observe that most of the invariants we studied contained some property where flow state would eventually be dropped in reaction to `REMOVE_ENTRY` or TCP `FIN`/`RST` event; such transitions are common in networked systems and ensure that any desynchronized state machine instances will eventually transition back to the initial state.

In addition to the above, Flink provides some guarantees that NF events are processed by the state machine exactly once. Flink and Kafka also provide support for an *end-to-end* guarantee of exactly once delivery, but with the overhead of

Network Function	Invariant Description	LoC	States	Transitions
NAT Gateway	`nat_decider_open`: After a PW goes into closed state, at least one replica also goes into closed state.	14	4	10
	`nat_consensus`: All TCP flows are only open after consensus.	5	2	4
	`nat_open_to`: Open flows are timed out after 4 minutes of inactivity.	5	4	12
	`nat_primary_single`: There is a single primary per flow.	10	3	7
	`nat_primary_to`: The NATGW does not start an idle timeout for active flows.	13	6	18
	`nat_same_consensus`: After TCP flow *U* is terminated, the next flow for *U* achieves consensus.	12	5	15
	`nat_syn_to`: Flows with a TCP handshake in progress timeout after 5 seconds of inactivity.	5	4	12
	`nat_udp_same_consensus`: If UDP flow *U* times out, the next flow for *U* achieves consensus.	12	6	17
Firewall [26]	`fw_consistency`: *all* Firewall instances should block suspicious IPs after a block rule is added.	6	4	12
	`fw_client_init`: Ensure a flow can only be open after a client initiates it.	4	2	4
	`fw_syn_first`: Data packets are only allowed after a SYN is sent.	4	2	4
DHCP	`dhcp_reuse`: Leased addresses are not re-used until expiration or release.	6	4	12
	`dhcp_overlap`: Leases should not overlap between DHCP servers.	6	3	7

Table 1: List of example invariants *Horde* can implement for several common network functions and systems.

atomic exporting of NF events, transactions, and rollbacks. Instead, *Horde* chooses to rely on partial synchrony and to alert users after the fact when false positives may have occurred. This can happen when an event arrives with a timestamp earlier than the last processed event or when two events arrive from an NF instance with a gap in their sequence numbers. It can also happen with an NF instance (and its local agent) fail. Upon restarting, the agent can immediately resume exporting events, but the local state machine may be out of sync. In this case, it can temporarily export all events (which is always safe) until it can synchronize with the global verifier to rebuild the local state machines from the global verifier's state.

8 Implementation

We have implemented *Horde* with more than 6,500 lines of Java 8 code, packaged with Maven v3.6. The implementation consists of two major components: the compiler and runtime system. We plan to make our implementation open source.

The compiler takes as inputs an event format specification as described in Section 5.1 along with a set of invariant specifications in the format of Section 5.2. For each invariant, it generates the global state machine, the resulting local state machines, information about suppressible events, and a slew of other metadata about variables, filters, and partitioning. The lexer and parser use the ANTLR v4.7 [27] parser generator, and the SFA construction and determinization use the open-source `symbolicautomata` library [28], but with the addition of a custom Z3-based [29] theory of Boolean Algebra designed to support our invariant language.

We built the runtime system on top of Apache Flink [16] and Kafka [17]. These frameworks are designed for scalable and robust stream processing and provide, intrinsically, fault-tolerant and stateful processing, exactly-once semantics, load balancing, flexible membership, checkpointing, etc. The local agents ingest events directly, then filter, map, and suppress events as necessary before sending them to Kafka. The global verifiers pull from Kafka into a timestamp-based priority queue from which events are dequeued after waiting for a maximum delay; violations are logged to disk.

9 Evaluation

We evaluate *Horde* in CloudLab [30] with a number of network functions and along a number of dimensions.

The deployed NAT gateway (§2). We use two event traces captured from two different builds of the NAT gateway to evaluate *Horde*. The builds capture the introduction of a set of bugs that arose from the change of an interface between two internal components, with V1 from before the change and V2 from after. The traces are both for 7 flow deciders over a 30 minute interval, but they contain a different number of packets (V1: 23.7M; V2: 9.0M) owing to changes in the protocol. The production deployment of NATGW does not yet support fine-grained clock synchronization, but our operators plan to add it in the system's next version. Instead, we capture the event traces and correct for time drift using a set of known synchronization points within the event stream. In total, there are eight invariants for NATGW (see Table 1).

A distributed firewall. We perform a collection of micro-benchmarks using an open-source, stateful, and distributed firewall implementation built on `iptables`, `conntrackd`, and `keepalived` [26]). On the firewall, we check various invariants, some of which were derived from [31]. The list of specific invariants we check are listed in Table 1.

We deploy this firewall on a topology with four clients, four internal hosts on a single LAN, and four firewall nodes interposing between the two groups. The firewalls are configured as two high-availability groups with two primaries and two hot standbys. Each primary-standby group shares a virtual IP with the VRRP protocol. Traffic between external hosts and internal servers is based on the traces provided in [32].

Evaluation metrics. We evaluate *Horde* along a number of key dimensions: lines of code, throughput, latency, and CPU overhead. In addition, our micro-benchmarks show *Horde*'s ability to scale as the number of nodes in the NF deployment increase by demonstrating the benefits of our event suppression scheme. Finally, we find *Horde* is able to identify four bugs in the NAT gateway which were confirmed by our operators. Similarly, it identified all bugs in the distributed firewall.

Invariant	Version 1	Version 2
nat_decider_open	0	0
nat_consensus	0	0
nat_open_to	1	45019
nat_primary_single	0	0
nat_primary_to	1	29964
nat_same_consensus	536	259
nat_syn_to	0	2697
nat_udp_same_consensus	0	0

Table 2: Violations found in traces for NATGW versions. Note that V1's trace contains more events than V2's, which may account for the difference in nat_same_consensus violations.

9.1 Bugs Identified by *Horde*

NATGW Bugs. Running the traces through *Horde*, we discovered violations of nat_open_to, nat_primary_to, nat_same_consensus, nat_syn_to, all of which were confirmed as caused by bugs by the NATGW team. Table 2 shows the absolute number of violations of each. A few notes about this process follow.

nat_open_to was by far the most frequent violator in V2. Discussions with our operators revealed that in V2, this violation (and that of nat_syn_to) were due to related bugs in the code: it had taken operators over an hour to identify the issues while *Horde* identified it in under a minute. Although nat_open_to also had a violation in V1, further examination revealed that the violation in V1 was due to an expected consequence of eventual consistency—specifically one of the replicas was getting update messages from the packet worker but the primary did not and therefore started a timeout for the flow. This led us to start checking for nat_primary_to.

Also prominent in both systems were violations of nat_same_consensus. This violation occurred because the flow was not closed or removed properly from one of the replicas. The operators suspected this could be an issue, but never had a method to test that hypothesis. *Horde* confirmed the problem and helped the developers to formulate the test setup to reproduce the issue.

Bugs in the distributed firewall rules. For the firewall, we manually injected bugs in the firewall configuration to test *Horde*'s ability to identify this category of errors. The injected issues, for instance, always allowed external traffic from a particular address range into the internal network, violating fw_client_init. *Horde* found all of them.

9.2 Throughput of *Horde*

Horde's global verifier keeps track of the set of possible states for each invariant and the possible values for each value/location. Thus, its throughput is directly correlated with the number of invariants (Figure 11). To evaluate this scaling, we run the V1/2 traces thorugh a random subset of the 8 NATGW invariants using a single Task Slot on the global verifier (running on an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz machine). Even with this single-core execution, our optimizations allow *Horde* to scale and process over 60,000 events per second for a single invariant (over 10,000 for 7 invariants).

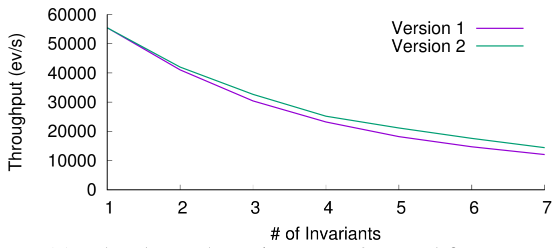


Figure 11: The throughput in events/second for an executor of *Horde* on the two NATGW traces.

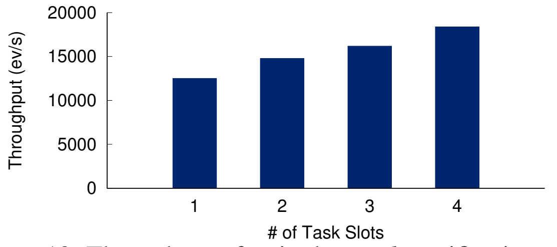


Figure 12: Throughput of a single *Horde* verification server checking 7 invariants with different numbers of Task Slots.

Horde scales linearly as we add additional cores to the global verifier (Figure 12). On a single machine, however, the bottleneck is typically memory and I/O. Scaling with multiple machines avoids this bottleneck.

9.3 Overhead of *Horde*

Horde imposes minimal CPU and memory overhead (Figures 13 and 14, respectively). This is especially important for the local components as they are co-located with the production NF instances. *Horde*'s CPU utilization remains around 0.1% on average for the production NATGW while its memory usage is always bellow 1.4 GB. As we expect, we observe the CPU and memory usage for the local verifier is higher at the primary NF instances as they tend to generate more events.

Figure 15 shows the CDF of *Horde*'s *time to detection* for violations in the distributed firewall function. The time to detection is low: in the median it takes roughly 70 ms from the time the event was executed (the violation occurred) at the NF instance until *Horde* raises an alert.

9.4 Efficacy of Suppression

Each optimization in *Horde* improves its scalability by reducing the number of events sent to the global verifier (reducing the network overhead and the number of events processed at the global verifier). Filter, removes the need to send events that do not pertain to the invariants and reduces the number of events sent to the verifier by up to 61% for the NATGW (Table 3). Suppressible events can further reduce this number (by up to an additional 12% in our experiments).

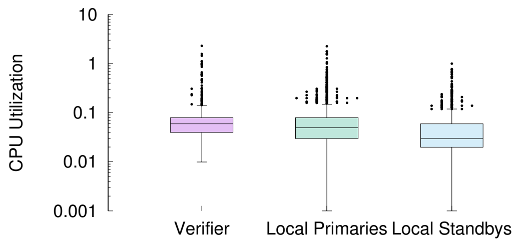


Figure 13: CPU utilization of *Horde* running with a single executor per machine. A utilization of 1 indicates full utilization of a single core.

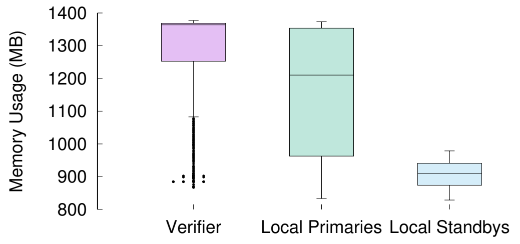


Figure 14: Memory utilization of *Horde* in MBytes.

10 Related Work

Runtime verification. Runtime verification has been well-studied, with many papers dedicated to improving its expressiveness and performance. *Horde* borrows ideas from two lines of work in the area. The first, is runtime verification of distributed systems, which is broadly separated into two categories based on whether the system assumes a synchronized global clock [33]. In this respect *Horde* would be considered a *decentralized* [15, 33] runtime verification system. The second, is parametric runtime verification, which focuses on checking universally or existentially quantified expressions [12–14, 34]. The location variables in *Horde* are examples of parametric variables. The main distinction of *Horde* from prior work is its combination of parametric and decentralized runtime verification through its support for location variables. Moreover, *Horde*'s efficient implementation of this combination of features through its use of sharding and local symbolic state machine partitioning is new in this context.

NF and distributed systems verification. The concept of NF verification is not new. Many prior works use verification techniques to check/enforce correctness. Most of these approaches are based on static verification of hand-written NF models [5, 6, 10]. Such static checking can give strong guarantees of correctness, but suffers from several drawbacks: Unlike *Horde*, it can not check complex properties such as temporal properties, can not catch implementation bugs, and requires tedious and possibly error-prone hand-translation of models for NF functionality.

Yet another approach to verifying complex distributed software is through the use of semi-automated theorem provers such as Dafny [35]. A good example of this approach is IronFleet [8]. A major drawback of this approach is that it requires significant development effort. For example, verifying IronFleet involved tens of thousands of lines of proof. In contrast *Horde* aims to be lightweight, requiring little to no developer effort by catching bugs at runtime.

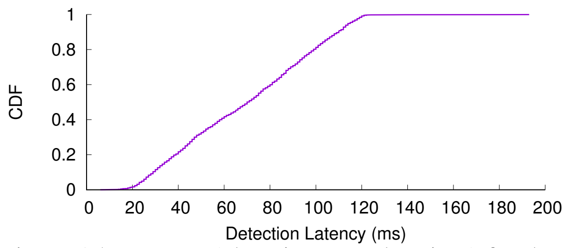


Figure 15: Latency (alert time - packet time) for detecting a violation in the distributed firewall.

Version	Generated	After Filter	After Suppression
V1	189M	92.9M (**49.1%**)	70.2M (**37.1%**)
V2	72.2M	36.7M (**50.8%**)	28.0M (**38.8%**)

Table 3: Total number of generated events, events processed after filtering, and events processed after filtering and suppression for the NAT gateway with all 8 invariants.

Stateless dataplane verification. Dataplane verification tools such a HSA [36] and Anteater [37] verify the correctness of a static snapshot of network forwarding tables. Later tools such as Veriflow [38] perform runtime verification by constantly re-verifying the network state as changes occur. Each of these tools reasons about all packet behaviors—a challenging task—however, their reasoning is limited to verification of *stateless* network forwarding. In contrast, *Horde* focuses on verifying complex temporal and stateful properties of general-purpose distributed NFs. For example, *Horde* can ensure a stateful firewall correctly allows traffic only for connections that are established by an internal sender.

11 Discussion and Conclusion

Horde is a lightweight verification framework for verifying distributed network functions. To scale to large systems with minimal overhead, *Horde* leverages a two-tiered setup with local monitors at each NF instance sending events to (and hiding events from) a collection of sharded global verifiers. While *Horde* can verify any distributed system, its scalability will depend on whether the invariants of interest can utilize its **GROUPBY** and suppression optimization schemes.

Finally, as *Horde* is the first to verify distributed network functions at scale (and at runtime), there are a number of aspects where follow up work may be needed. Included in this set are explorations of other time synchronization protocols, e.g., [39] or some other lightweight and precise event ordering mechanisms. Also for future work are innovations in atomic event export and transactions over streams in *Horde*.

References

[1] Maglev outage. https://status.cloud.google.com/incident/cloud-networking/18013.

[2] Microsoft load balancer outage. https://lucian.blog/why-is-the-azure-load-balancer-not-working/.

[3] Microsoft public incident reports. https://status.azure.com/en-us/status/history/.

[4] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72, 2016.

[5] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 221–239, Santa Clara, CA, February 2020. USENIX Association.

[6] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, March 2017. USENIX Association.

[7] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. Netsmc: A custom symbolic model checker for stateful network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 181–200, February 2020.

[8] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob Lorch, Bryan Parno, Michael Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60:83–92, 06 2017.

[9] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 141–154. Association for Computing Machinery, 2017.

[10] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract interpretation of stateful networks, 2017.

[11] Tim Nelson, Nicholas DeMarinis, Timothy Adam Hoff, Rodrigo Fonseca, and Shriram Krishnamurthi. Switches are monitors too! stateful property monitoring as a switch design criterion. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 99–105, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 341–356. Springer Berlin Heidelberg, 2014.

[13] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. Marq: Monitoring at runtime with qea. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 596–610, 2015.

[14] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin ŞerbănuŢă, and Grigore Roşu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 285–300, 2014.

[15] M. Ali Dorosty, Fathiyeh Faghih, and Ehsan Khamespanah. Decentralized runtime verification for ltl properties using global clock, 2019.

[16] Apache flink: Stateful computations over data streams. https://flink.apache.org/.

[17] Apache kafka. https://kafka.apache.org/.

[18] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM 13)*, pages 207–218, 2013.

[19] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 523–535, 2016.

[20] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[21] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[23] Information Sciences Institute. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[24] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV'17)*, July 2017.

[25] Guangming Xing. Minimized thompson nfa. *International Journal of Computer Mathematics*, 81:1097 – 1106, 2004.

[26] Netfilter. http://conntrack-tools.netfilter.org/.

[27] Antlr. https://www.antlr.org/.

[28] A symbolic automata library. https://github.com/lorisdanto/symbolicautomata.

[29] Z3. https://github.com/Z3Prover/z3.

[30] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[31] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, and Masum Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE journal on selected areas in communications*, 23(10):2069–2084, 2005.

[32] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[33] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. *Runtime Verification for Decentralised and Distributed Systems*, pages 176–210. 2018.

[34] Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zălinescu. *Monitoring Events that Carry Data*, pages 61–102. 2018.

[35] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

[36] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[37] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[38] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[39] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 81–94, 2018.