Aragog: Scalable Runtime Verification of Shardable Networked Systems

Nofel Yaseen$^\diamond$, Behnaz Arzani†, Ryan Beckett†, Selim Ciraci§, and Vincent Liu$^\diamond$
$^\diamond$*University of Pennsylvania* †*Microsoft Research* §*Microsoft*

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 *Aragog*, a runtime verification system for distributed network functions that achieves high expressiveness, fidelity, and scalability. Given a property of interest, *Aragog* 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, *Aragog* 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 *Aragog* over several network functions including a NAT Gateway that powers Azure, 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 (NFs) 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 [4] 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 this system can lead to black holes, broken connectivity, forwarding loops, and more. Public incident reports from providers show multiple outages due to errors like these [4, 19].

To improve availability, recent proposals suggest using *static verification* to prove the correctness of these systems [21, 25, 29, 34, 40–42, 44]. While powerful, the need to reason about every possible interleaving of inputs and control flows presents a significant obstacle to the application of these techniques in today's network functions. Attempting to explore the full space of control flow paths often leads to state/path explosion [25, 29, 40]. Mitigations to this problem, broadly speaking, can be categorized in a few ways. The first is to require the use of special programming languages or other types of programmer interaction [21, 43]. 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 unordered [34], abstract [10], or restricted to a single machine [42, 44].

While effective in many cases, each of these approaches also comes 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.

An alternative approach to static verification is runtime verification of distributed systems. 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 [13, 14, 28, 30, 31, 33, 36, 39]. Compared to static verification, runtime verifiers only test inputs and control flows that are seen in practice, thus improving scalability and enabling verification of actual deployments running over actual data. In return, they sacrifice a principled exploration of the system's behavior and the ability to catch bugs early. We argue that these tradeoffs are a better fit for our operators' requirements.

We find today's runtime verifiers cannot be applied as-is to deployed network functions. The challenge (for 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 event, and (3) scale sub-linearly with the size of the original system—after all, 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 *Aragog* that overcomes the above challenges. *Aragog* provides a simple, but expressive language for describing violations of invariants, with a focus on supporting network functions. Examples of network-centric language features that are found in *Aragog*'s Invariant Violation (IV) specifications, but that are uncommon in other runtime verifiers 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.

Aragog translates these IV specifications 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, *Aragog* implements the core of these checks on top of production stream processing systems [2,3]. To efficiently coordinate between distributed verifiers, *Aragog* relies on hardware-supported time synchronization protocols like PTP. Finally, to minimize the overhead of the verification system, *Aragog* 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 IV specification, *Aragog* 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 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. *Aragog* includes an optimization scheme to suppress such messages before they ever leave the NF instance.

Aragog does not guarantee perfect accuracy under asynchrony—to do so would require atomicity guarantees in the critical path of the network functions. *Aragog* instead handles these situations speculatively and notifies users after-the-fact[1] about transient inconsistency (§7.3). Despite this, *Aragog* identified at least four bugs in an early (limited) deployment of a real distributed network function: Azure's new NAT gateway (NATGW). These bugs were detected within $\sim$100 ms of occurrence. Compare this to the hours our operators typically spend searching for similar bugs.

To summarize, our work makes the following contributions:

- We present a case study of the needs of a large modern network function from Microsoft's Azure. The system exhibits several interesting characteristics and suggests 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 *Aragog*, a system for at-scale runtime verification. When needed, *Aragog* can also run on traces (offline) and therefore complement static verification to find implementation bugs in distributed networked systems. Among other innovations, *Aragog* includes a novel method for computing location-dependent suppression of network events.
- We introduce a collection of *Aragog* invariant violations for a set of distributed network functions, and we evaluate *Aragog* on NATGW and a distributed firewall.

[1]This reporting happens in under 1 s. This delay is on the same order as other alerting systems used in our production networks.

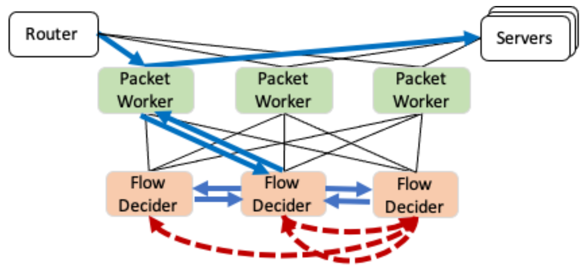


Figure 1: The architecture of our NATGW. The bolded blue arrows show the sequence of communication to handle the SYN packet of an incoming flow: it is sent to a random packet worker, which forwards it to the flow decider in charge of that flow. The flow decider chooses a target server and replicates the mapping to other deciders, then installs it in the original packet worker. The three dashed red arrows trace the allocation of the mapping for the reverse flow.

2 Motivation: A Cloud-scale NAT Gateway

Our work is grounded in experience with Azure's large-scale NF that we call NATGW. NATGW is a cloud-scale NAT gateway that balances incoming requests over available servers and supports almost all external traffic.

Like many other NFs of similar scale [16,35], 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 NATGW workers, which then rewrite the destination IP and port to point at a target server. A similar translation occurs for packets in the reverse direction (from the server to the client).

Figure 1 depicts the NATGW architecture. It is composed of two types of nodes: 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 deciders cache and replicate these mappings to other deciders to ensure availability.

Flow allocation. When a packet worker receives the first packet of a new flow, it 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 the target server to which to send the new flow 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. With its counterpart primary, greedily copies the mappings to the cache of other flow deciders in a manner akin to chain replication: decider i will try to copy to deciders $(i+1) \bmod N$ and $(i+2) \bmod N$, where N is the number of deciders. If one is down, it switches to $(i+3) \bmod N$.
4. Installs the mapping into the originating packet worker.

After the above flow allocation, the packet worker can process all subsequent packets of the flow without coordination with any other node. If the packet worker fails, anycast redirects the packet to another worker; the new worker will send the packet to the primary flow decider, fetching the existing mapping. If the flow decider fails, packet workers will 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 R_P, R_{P+1}, and R_{P+2}, where R_P is the primary. To delete the mapping, R_P would send a delete request to both of the other nodes. Now imagine the message to R_{P+1} is dropped. Rather than waiting for R_{P+1}, the others will go ahead and delete f. If, later, R_P fails, packet workers will contact R_{P+1} for the mapping, which will return a stale/inconsistent result until a timeout or periodic sync eliminates the inconsistency.

There are known mitigations to the above behavior (*e.g.*, querying a quorum on every packet or initiating a view change algorithm on R_P's failure); however, these come with significant performance costs. Instead, the NATGW 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. Instead, verification should be of the end-to-end system, *in situ*.

In the same vein, *Aragog* should not place undue burden on developers, *e.g.*, by requiring engineers to perform non-trivial proof writing (as mandated by many deductive reasoning techniques). NATGW has over 40 thousand lines of code—*Aragog* 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 [33]. In particular, it is challenging to specify invariants related to: (1) parametric variables over values 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. *Aragog* should provide syntax and semantic support for these behaviors.

Scalability. Just as a single machine cannot handle all traffic entering a large network, it also cannot be expected to verify the correctness of the entire network. Rather, the verifier should scale out to arbitrary size and require fewer resources than the original system. Therefore, *Aragog* should attempt to minimize the number of messages exported from each NF, *e.g.*, by exporting events (resulting from the execution of the NF) rather than packets (the inputs to the NF).

Graceful degradation of accuracy. As we describe in Section 7.3, perfect precision and recall is impossible in an asynchronous system without substantial overhead. Instead, *Aragog*'s correctness goal is in the same spirit as NATGW's: perfect recall under the assumption of 'partial synchrony' [15] and notifications of potential false positives/negatives after-the-fact. Our operators find this is sufficient for most cases.

Near-real-time alerts. Diagnosing bugs manually can take hours of operator time and the network could worsen the longer the bug persists: *Aragog* should raise alerts within seconds of observing the offending sequences of events.

4 *Aragog*'s Architecture

We present the design and implementation of a practical, expressive, and scalable verifier for large and complex NF deployments. Our system, *Aragog*, is a combination of a language for specifying invariant violations and a scale-out runtime system. *Aragog* takes a grey-box approach, requiring small changes to the underlying source code in order to export events of interest to the verifier. Thus, *Aragog* verifies by:

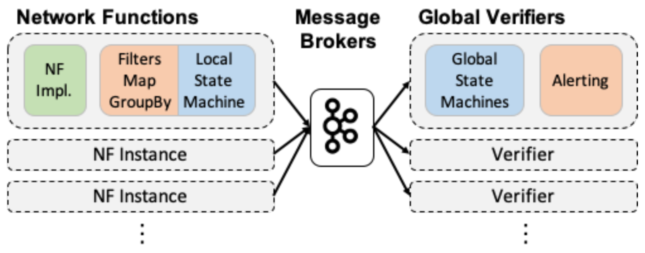


Figure 2: The architecture of *Aragog*. NF instances generate and feed events into a set of local state machines. The NF instances use these state machines to determine if they can hide unnecessary messages before exporting the rest to the global verifier. These messages pass through a Kafka cluster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

Specifying invariant violations over user-defined events. To provide operators with sufficient expressiveness to check network-level events, *Aragog* comes equipped with a new language for specifying invariant violations that is based on writing symbolic regular expressions over a global trace of events (and their locations) in the system. *Aragog*'s language includes a notion of parameterized "variables" that allows violations to be described in a way that holds for any combination of variable instantiations subject to constraints.

Checking for invariant violations. NF developers export any relevant events to *Aragog*. To scale up checking of the event stream, *Aragog* does two things. The first is to automatically analyze and split verification into local and global components. The local level resides at the NF instances themselves, where *Aragog* infers (only using the state of the local instance) whether it can safely suppress the event before exporting it to the global *Aragog* verifier. The second is to leverage the fact that most network invariants are defined across *related flows* rather than globally—for instance, on the granularity of a 5-tuple. As a result, events can be automatically sharded across a cluster of scale-out stream processing workers using Kafka [26] and Flink [11].

Note that, because the invariants are defined and checked only across related flows, we only need to know the correct ordering for events pertaining to those flows: event timestamps that use the sub-microsecond-scale synchronization of PTP suffices for our needs. For many production networks, these types of event exports are already common.

Overview. Figure 2 shows *Aragog*'s design. Users describe a set of invariant violations that identify classes of incorrect behavior. *Aragog* 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 *Aragog* agent filters, maps, and shards events The message brokers aggregate and compact those streams 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.

```
{ "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 NATGW JSON event schema.

5 Specification Language

Users define both events and policies over the events using two types of specifications that are inputs to *Aragog*: event definitions and Invariant Violation (IV) specifications. While both of these require the user to have some knowledge of the inner workings of the NF to specify how it can fail, our network operators determined that event-based violations struck a reasonable balance between precision and ease-of-use.

5.1 Event Definitions

Users specify the format of the event messages that arrive at the local verifier. *Aragog* 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, *Aragog* 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 or 128-bit `srcIP` field. These branches can define entire sub-headers and can contain nested conditionals.

Named constants. *Aragog* also allows users to define named constants representing integer values represented in decimal, hexadecimal, or binary notation. We show four such constants in Figure 3: two for values of the `eventType` field and one for the `nodeType` field. These are intended for use in IV specifications to make them more readable.

```
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 IV specification that ensures at most one primary is ever active for a given flow.

5.2 Invariant-Violation (IV) Specifications

Aragog parses incoming event messages and checks them against a set of user-defined policies that describe sequences of events that violate the invariants of the system. Operators specify these policies using *Aragog*'s domain-specific language, which we detail in this subsection.

Figure 4 shows an example specification for our NATGW. The policy only pertains to a subset of events (lines 1–3), and *Aragog* verifies it on a per-5-tuple basis (line 4). A violation occurs when some node `$X` adds a primary mapping (line 6) and then a different node (`NOT $X`) adds the same mapping (line 8) without `$X` removing it. The full grammar for IV specifications is shown in Figure 5. Briefly, an IV specification consists of (1) a collection of event transformations followed by (2) a regex-like expression over the generated events.

5.2.1 Transformations

Aragog allows users to define a set of policy-specific transformations. In addition to enabling greater flexibility and expressiveness, *Aragog* also uses these transformations to perform an initial filtering and aggregation as well as to identify valid sharding strategies. *Aragog* currently supports three transformations: **GROUPBY**, **FILTER**, and **MAP**.

Operators can use **GROUPBY** to indicate which events need to be considered together and which can be considered separately. For example, when an operator wishes to guarantee at most one primary is active (Figure 4) for *each flow*, the **GROUPBY** is used to classify events into unique flows. *Aragog* uses this transformation to both simplify policy logic and to assist in the sharding of verification.

Operators can also use the **FILTER** transformation to indicate which events should be considered at all and which should be ignored. In the above example, we only care about flow deciders—specifically when they add a flow as a primary and when they delete the flow mapping from the cache; we can filter events of any other type or from any other type of node. **FILTER**s are critical for reducing the number of events handled by the verification framework.

Finally, operators can use the **MAP** transformation to generate entirely new fields based on mathematical expressions over existing fields of the event message.

```
⟨IVspec⟩ ::= ⟨transformations⟩ 'MATCH' ⟨events⟩

⟨transformations⟩ ::= ⟨transformations⟩ ⟨transformations⟩
   | 'GROUPBY' '(' ⟨fields⟩ ')'
   | 'FILTER' '(' ⟨filter_matches⟩ ')'
   | 'MAP' '(' ⟨field_expression⟩ ',' ⟨field_name⟩ ')'

⟨fields⟩ ::= ⟨field_name⟩ [',' ⟨fields⟩]
   | 'LOCATION' [',' ⟨fields⟩]

⟨filter_matches⟩ ::= '(' ⟨filter_matches⟩ ')'
   | ⟨filter_matches⟩ '||' ⟨filter_matches⟩
   | ⟨filter_matches⟩ '&&' ⟨filter_matches⟩
   | ⟨filter_match⟩

⟨filter_match⟩ ::= ⟨field_name⟩ ⟨compare_op⟩ ⟨field_name⟩
   | ⟨field_name⟩ ⟨compare_op⟩ ⟨value⟩

⟨events⟩ ::= '.' '@' ⟨location_spec⟩
   | ['!'] '(' ⟨event_match⟩ ')' '@' ⟨location_spec⟩
   | '(' ⟨events⟩ ')'
   | ⟨events⟩ ⟨events⟩
   | ⟨events⟩ ⟨regex_op⟩
   | 'SHUFFLE' '(' ⟨events_list⟩ ')'
   | 'CHOICE' '(' ⟨events_list⟩ ')'

⟨events_list⟩ ::= ⟨events⟩ [',' ⟨events_list⟩]

⟨location_spec⟩ ::= 'ANY'
   | ⟨loc_matches⟩

⟨loc_matches⟩ ::= ['NOT'] '$'⟨loc_name⟩ [',' ⟨loc_matches⟩ ]

⟨event_match⟩ ::= ⟨field_match⟩ [',' ⟨event_match⟩]

⟨field_match⟩ ::= ⟨terminal⟩ ⟨compare_op⟩ ⟨terminal⟩

⟨terminal⟩ ::= ⟨field_name⟩
   | ⟨value⟩
   | '$'⟨variable_name⟩
   | 'TIME'
```

Figure 5: Grammar for *Aragog*'s IV specification language. Tokens ending in '_name' are identifiers that must begin with a letter; the 'compare_op' token refers to the class of operators '==', '!=', '<', etc; 'value' indicates a constant number; and 'field_expression' is a mathematical expression over fields.

5.2.2 Event Expressions

Users define invariant violations over the transformed event streams by specifying sequences of events that result in a violation of a particular policy. Users specify these sequences with a regular-expression-like language, which describes patterns over pre-defined elements. In *Aragog*'s case, the elements take the form of a set of matching operations over the fields of the event message; the example in Figure 4 shows matches on one such field, the `eventType`. A match can occur at any point in the stream of events and triggers on every occurrence of the match, not just the first. For example, if events $A \rightarrow B \rightarrow A$ form a violation and (at runtime) we observe the sequence *CABABAC*, *Aragog* will alert twice.

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

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

Figure 6: An example specification that checks that a stateful firewall does not drop reverse traffic for an open connection.

the contained expressions matches. In **SHUFFLE**, the contained events can arrive in any order, but must all arrive.

Event expressions come after the set of transformations and must appear after a **MATCH** statement.

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 specifiers. This is useful for specifying matches, but it is also important for determining how we might partition evaluation of the IV specification across both local and global verifiers (see Section 6). In both cases, the goal is to determine whether each pair of events are expected to occur at the same or at different NF instances.

Consider again the example in Figure 4. The example contains a single named location, `$X`, corresponding to the original primary node for the current flow. One way to use this named location is to specify that another event in the sequence must *also* occur at `$X`. Another, demonstrated in lines 7&8, is to specify that the event occurs at a location distinct from `$X`. Note that the syntax does not constrain the relationship between the locations of the events of lines 7&8.

Every event can reference one or more named locations, or it alternatively use the location `ANY`, which indicates no special semantic meaning of the location of the event. In the case of multiple locations, users specify multiple predicates (one per location). For example, to ensure three events with distinct locations: one could specify ev_1 at (`$X`, `NOT $Y`); ev_2 at (`NOT $X`, `$Y`); and ev_3 at (`NOT $X`, `NOT $Y`).

One possible method of implementing locations is to enumerate all possible locations in the system and expand the event expression accordingly. While this would allow the usage of more traditional state-machine evaluation techniques, it would also lead to an unacceptably inefficient implementation. Further, any change in membership would require us to fully recompile and re-install all IV specifications across the system. Instead, *Aragog* lazily tracks all potential candidates for location variables at runtime using a multi-leveled tree data structure, which we describe in detail in Section 6.

Variables. *Aragog* generalizes the state tracking afforded to locations in order to track other types of state in the IV specification. Examples of non-location stateful properties include the IP/port NAT mappings of the NATGW and connection tracking in a firewall. An example of the latter is shown in Figure 6, which verifies that if an outbound flow from source IP `$S` and destination IP `$D` is properly initialized, then packets in the reverse direction are also allowed.

```
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 7: An example of a timing violation specification that checks the behavior of TCP's TIME-WAIT state [22]. The SYN *must not* arrive by a deadline. This specification assumes that only packet sends are captured.

```
FILTER(flag == FIN || flag == FIN_ACK)
GROUPBY(IP1, IP2, port1, port2)
(eventType == FIN, TIME == $t) @ ANY
((eventType != FIN_ACK, TIME - $t <= 30000) @ ANY)*
(TIME - $t > 30000) @ ANY
```

Figure 8: An example of a timing-related IV specification that checks timely arrival of a FIN_ACK after a FIN. The FIN_ACK *must* arrive by a deadline.

As these variables do not indicate or impose restrictions on the location of the event, we do not use them for the partitioning procedure of Section 6.

Timing. Timeouts and deadlines are also common in NFs. To specify them, users can use parameterized variables in conjunction with a builtin `TIME` field to compare the time between multiple events. For example, Figure 7 defines a violation of the TIME-WAIT semantics of a TCP flow in which SYN packets should not be sent within 30 s of a passive closer's FIN/ACK. The same SYN packet 31 s after the FIN/ACK would not be a violation. On the other end of the spectrum, Figure 8 defines a violation where a FIN-ACK does not arrive in time (within 30 s of the FIN). Any intervening FIN-ACK will mean that the violation does not match.

6 State Machine Generation

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

Aragog, thus, generates its state machines in three stages. First, it creates a symbolic non-deterministic finite automaton (SFA) [12] whose alphabet is based around a theory of arithmetic and boolean algebra, and whose predicates can include the placeholder variables described in the previous section.

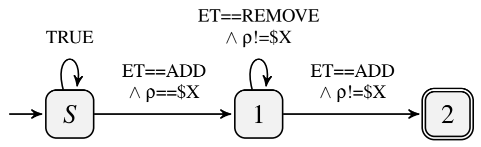


Figure 9: SFA for Figure 4 with some field names and constants abbreviated as well. ρ indicates location.

Second, it determinizes the SFA to a symbolic deterministic finite automaton (SDFA) to reduce runtime overhead of state machine execution. Finally, it constructs localized versions of the SDFA that can be used to infer the global state of the system from only locally observed events.

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.

Aragog 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 resulting predicates, *Aragog* 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, *Aragog* recursively builds out the state machine using Thompson's construction [38], treating **`CHOICE`** as a choice operator, and expanding **`SHUFFLE`** to all permutations. Figure 9 shows a (minimized) SFA for the example violation specification 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 will never lead to a match.

Aragog next determinizes the SFA: it generates an efficiently executable DSFA from the SFA using standard symbolic automata techniques [12]. The result is a state machine where all transitions are unambiguous and exhaustive. Figure 10 shows the DSFA for the example. Each state in the DSFA stores the correesponding set of SFA states the machine is in at that given point in time.

6.2 Local State Machines

Conceptually, the DSFA provides an efficient method for checking whether a stream of events leads to an invariant violation. In principle, we could simply funnel all events to a central verifier, which would then apply the relevant DSFA transition and report a violations upon reaching an accepting state. Unfortunately, doing so would require the verifier to process *all* unfiltered events in the system. Instead, we further improve *Aragog*'s scalability by generating a localized version of the state machine that is executed on the same machine as the NF before sending the event to the global verifier.

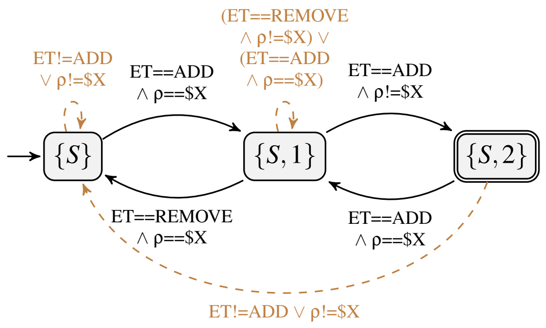


Figure 10: DSFA for the SFA in Figure 4. Colored, dashed edges represent suppressible transitions.

6.2.1 Suppressible Transitions

The local state machine needs to identify events that will not impact the detection (or lack of detection) of a user-specified violation whether or not it is sent to the global verifier. Our key observation is that there are transitions in the global DSFA that do not affect the end result of the state machine. We term these transitions *suppressible transitions*. More formally:

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

Definition 2. $q \xrightarrow{e} q'$ indicates that, 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 $q \xrightarrow{s} q_a$.

In the running example DSFA in Figure 10, the three dashed transitions are suppressible given the above definition. The two self-loops 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 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.

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)
```

6.2.2 Local State Machine Construction

Aragog 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 local state machine for every location variable in every IV specification such that each machine assumes it is playing the role of that location (*e.g.*, one machine for "I might be `$X` in a violation" and another for "I might be `$Y` in a violation"). In the example from Figure 10, there is only a single local state machine: the one for `$X`.

The first step in creating a local state machine, L, is to model the uncertainty other locations may introduce (Algorithm 1). The algorithm takes the global state machine G, the location variable V (*e.g.*, `$X`), and a predicate F corresponding to the user-defined **`FILTER`** statements. It 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 it is, 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 V. To model the fact that other NF instances might send events that use this transition, the algorithm adds to L an epsilon (ε) transition (line 9). An ε transition is one which the local SFA can take 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).

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 (line 11). It does so by partially evaluating the predicate with the assumption that ρ==V (line 10). In Figure 10, for example, the transition (`ET==REMOVE` ∧ `ρ==$X`) is simplified to `ET==REMOVE`.

Figure 11 shows the local SFA for location `$X` and its determinized (DSFA) form. By executing the DSFA in Figure 11 locally, an NF instance can learn some partial information about the state of the overall system. For example, after seeing

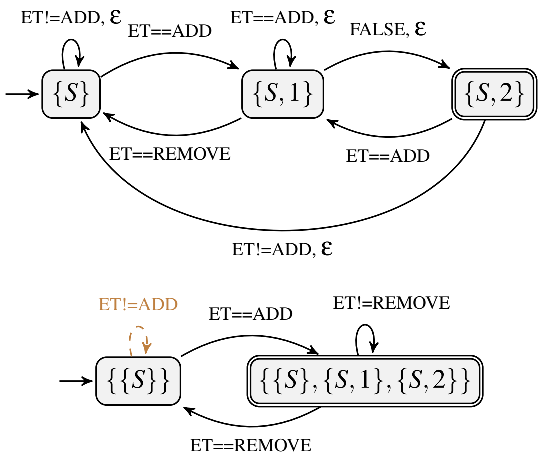


Figure 11: Local machine for `$X` from Figure 10. SFA is shown on top and its equivalent DSFA is shown below. Colored, dashed edges indicate locally suppressible transitions.

an `ADD` event, the NF instance recognizes that (if it is `$X`) 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 Suppressing Events Locally

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), explained below.

The algorithm works by iterating over every location or variable in the IV specification (line 5) and calling CreateLocalDFA to build the local state machine (line 6). It then walks over each local transition (T) and attempts to mark the transition as locally suppressible. To do so, it looks up all the possible global states corresponding to 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 all events that trigger T must be part of a suppressible transition in the global DSFA, so the event is suppressed.

In Figure 11, events matching `ET!=ADD` in state $\{\{S\}\}$ are suppressible: for each global state in the set ($\{S\}$), this event must be processed by a suppressible global transition.

Negated condition. The final part of the algorithm (lines 20 to 23) computes a "negated condition." This condition captures the case where the local NF may not correspond to any named location in the IV specification, *e.g.*, the NF instance is not `$X`, but it still may observe a relevant event as `NOT $X`. We observe, in such a case, the current machine can not possibly know anything about the global automaton

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 := ∅
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⟩
```

state since the other NF instances that also are not `$X` may be sending events that match `NOT $X` transitions. The fix 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 to the global verifier 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, *Aragog* 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, *Aragog* will simply not suppress any events, which is always safe.

7 Runtime System

We next describe the *Aragog* runtime.

7.1 Workflow Overview

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, computes supressions using local state machines, and then sends any 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, local *Aragog* agents co-located with the NF first apply any applicable transformations—**`FILTER`**, **`MAP`** or **`GROUPBY`**—to the raw stream. As each IV specification can have a different set of transformations, this may require *Aragog* to duplicate the incoming stream of raw events (it tries to avoid doing so when possible). The end result is a set of keyed event streams: one stream for each combination of policy and **`GROUPBY`** key.

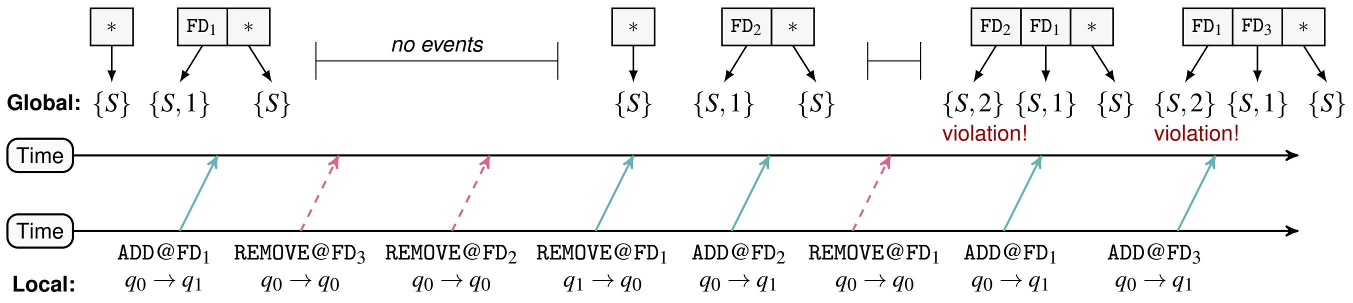
Computing suppression. The next step, also performed locally, is to determine whether events in each keyed stream are suppressible. *Aragog* passes the events through the localized state machines — one for each location referenced in each IV specification. For a given event and IV, *Aragog* suppresses the event when (1) all localized instances of the IV specification 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, *Aragog* sends the event to a Kafka queue for the given keyed event stream.

As a concrete example, Figure 12 shows processing of a series of events with the specification in Figure 4 and with the same **`GROUPBY`** key. The first event is an `ADD` event at flow decider $\mathtt{FD}_1$. After seeing this event, $\mathtt{FD}_1$ 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 $\mathtt{FD}_3$. 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 received by the global verifier. For example, with three replicas (including the primary), a correct execution of Figure 4 *Aragog* would receive—after suppression—just 2 out of 4 events (the add and remove at the primary but not the 2 suppressed removes at nodes other than `$X`).

Global state machines. Pulling from Kafka is a cluster of Flink instances running the global versions of the IV state machines. Both the Kafka and Flink instances are automatically provisioned, checkpointed, assigned **`GROUPBY`** keys, and load balanced to worker nodes. As Flink does not guarantee that events from different NF instances will arrive in order, *Aragog* temporarily stores and reorders events in the Flink workers 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 flows/policies and sending 'null'



*
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 12: Distributed execution for the example 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.

events to advance the timestamps of every partition would be costly. Instead, *Aragog* relies on the assumption of a maximum latency t_{max} and handles violations of this assumption with the techniques in Section 7.3.

Aragog will hold each event for t_{max} time before running it through the global DSFA. While processing events for a given IV specification, 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 IV specification's variables (see Section 7.2 for details). If any of the possible states is a 'final' state in the IV's DSFA, *Aragog* will raise an alert.

Consistent sampling. If scaling is still challenging despite sharding the verifier, filtering relevant events, and suppressing events locally, *Aragog* 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 keys that are not sampled.

7.2 (Location) Variable Tracking

Aragog tracks *all* possible instantiations of variables (location or otherwise) at runtime using a multi-level tree data structure (shown at the top of Figure 12). Intuitively, the tree captures the state the global automaton would be in for every possible instantiation, with the leaves of the tree as the state and the interior nodes as variable assignments. Every variable is assigned a single level of the tree.

Let the number of variables (location or otherwise) for an IV specification be n. When the system starts, the DSFA is in the start state, $\{S\}$, for all possible variable assignments. This is represented as a degenerate tree with height $n+1$ and a single leaf pointing at the start state $\{S\}$. The interior nodes are all set to $*$, indicating no constraints on the n variables. For every incoming event, we advance the DSFA using the state and variable assignments of every leaf. Whenever a predicate is encountered that references a variable, V_i, if $V_i = *$ is an ancestor of the current leaf we split execution into a case where V_i satisfies the predicate and a case where it does not. The $(n-i)$-height subtree under $V_i = *$ may need to be cloned.

In the example of Figure 12, there is only one variable (`$X`) and, thus, only two levels in the tree. The system starts in the degenerate case where `$X` $= *$. 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 ($\texttt{ET} == \texttt{ADD} \wedge \rho == \texttt{\$X}$) 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 *Aragog* to become desynchronized from the ground-truth state of the system. Even so, *Aragog* is able to guarantee both precision and recall of typical network violations under the assumption of 'partial synchrony' [15], *i.e.*, that there exists a time, t_s, after which there is some upper bound on message delivery time.

- *Recall:* Under a partial synchrony assumption, *Aragog*'s practice of creating a self loop in the initial state of the SFA means all violations whose trace begins *after* t_s are accurately modelled in the state machine and detected.
- *Precision: Aragog*'s precision guarantees are less complete, but still hold in practice. Specifically, we observe that all of the IV specifications we studied contained some property where flow state would eventually be dropped in reaction to a `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 guarantees that successfully pulled events are processed by the state machine

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 open only 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 [5]	`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 that *Aragog* can implement for several common network functions and systems.

exactly once. *End-to-end* guarantees of exactly once delivery between Flink and Kafka are also possible, but would incur the overhead of atomic exporting of NF events, transactions, and rollbacks. Instead, *Aragog* 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, two events arrive from an NF instance with a gap in their sequence numbers, or 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 *Aragog* with more than 6,500 lines of Java 8 code, packaged with Maven v3.6 and more than 2,000 lines of C++ code. The implementation consists of two major components: the compiler and runtime system. It can be found at: `https://github.com/microsoft/aragog`.

The compiler takes as inputs an event format specification as described in Section 5.1 along with a set of IV specifications in the format of Section 5.2. For each IV specification, 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 [1] parser generator, and the SFA construction and determinization use the open-source `symbolicautomata` library [6], but with the addition of a custom Z3-based [7] theory of Boolean Algebra designed to support our IV specification language.

We built the runtime system on top of Apache Flink [2] and Kafka [3]. 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, implemented in C++, ingest events directly, then filter, map, and suppress events as necessary before sending them to Kafka. The global verifiers, implemented in Java using Apache Flink, 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. We place the verifiers off of the critical path to avoid any impact on production traffic.

9 Evaluation

We evaluate *Aragog* in CloudLab [37] 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 *Aragog*. 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 export 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 IV specifications for NATGW (see Table 1).

A distributed firewall. We also execute a collection of microbenchmarks using an open-source, stateful, and distributed firewall implementation built on `iptables`, `conntrackd`, and `keepalived` [5]). On the firewall, we check various invariant violations, some of which were derived from [8]. The list of specific invariant violations 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 one primary and one hot standby each. Each primary-standby group shares a virtual IP with the VRRP protocol. We base the traffic between external hosts and internal servers on the traces provided in [9].

Invariant Violation	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.

DHCP. To show the flexibility of *Aragog* and its language, we also give examples of DHCP invariant violations in Table 1. With our current implementation, the operator needs to write just 6 lines to express the invariant violations. Each of the state machines uses a small number of states and transitions.

Evaluation metrics. We evaluate *Aragog* along a number of key dimensions: lines of code, throughput, latency, and CPU overhead. In addition, our micro-benchmarks show *Aragog*'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 *Aragog* is able to identify bugs in production systems. In particular, we were able to identify four bugs in the NAT gateway which were confirmed by our operators. Similarly, in the firewall, *Aragog* was able to find a series of injected configuration errors over real traffic traces.

9.1 Bugs Identified by *Aragog*

NATGW Bugs. Running the traces through *Aragog*, 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 observed for each.

`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 *Aragog* 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. *Aragog* confirmed the problem and helped the developers to formulate the test setup to reproduce the issue.

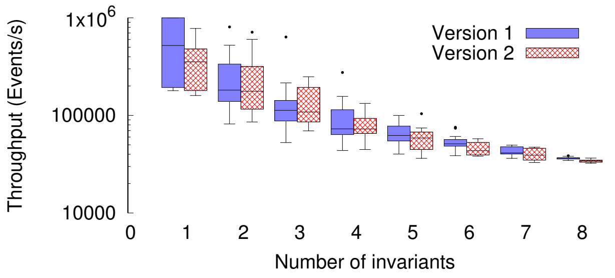


Figure 13: The throughput in events/second for an executor of *Aragog* on the trace.

Bugs in the distributed firewall rules. For the firewall, we manually injected bugs in the firewall configuration to test *Aragog*'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`. *Aragog* found all of them.

9.2 Throughput of *Aragog*

Aragog's global verifier keeps track of the set of possible states for each IV specification and the possible values for each variable/location. Thus, *Aragog*'s throughput is directly correlated with the number of IVs checked (Figure 13). To evaluate this scaling, we run the V1/2 traces through all the 8 NATGW IV specifications using a single Task Slot on the global verifier (running on an Intel(R) Xeon(R) E5-2450 processor CPU @ 2.10GHz machine). We upload the entire trace on Apache Kafka after local processing to measure the maximum throughput a single task slot of Apache Flink of the global verifier can process. In Figure 13 we randomly select n among the NATGW invariant violations and see the performance. As each type of invariant violation exhibits different resource requirements, we see more variance when the number of type of invariant violations selected is low.

With a single task slot, our optimizations allow *Aragog* to scale and process over 500,000 events per second for a single invariant violation type (over 30,000 for 8). Adding more task slots does not improve the performance as our implementation is parallel in nature and a single task slot is already using multiples core in a single machine.

Aragog scales linearly as we add more machines to the global verifier (Figure 14). Scaling with multiple machines avoids the bottleneck of CPU and I/O.

9.3 Overhead of *Aragog*

To measure the memory and CPU overhead of *Aragog*, we study its behavior while verifying the distributed firewall. In Figures 15, 16 and 17, data is divided into separate groups. 'Primary' represents the verifier running at the primary firewall. 'Backup' represents the verifier running at the hot-standby firewall. 'Manager' and 'executor' represent the Apache Flink job manager and executors, respectively. The global verifier runs on the executors.

Process/Location	Resource	Spearman correlation
job manager	CPU	0.14700
job manager	memory	−0.59379
executor	CPU	0.78481
executor	memory	−0.38373
primary	CPU	0.88916
primary	memory	−0.18253
backup	CPU	0.93618
backup	memory	0.24768

Table 3: Spearman Correlation between number of events/s and resource utilization at different locations of verifier while running the firewall.

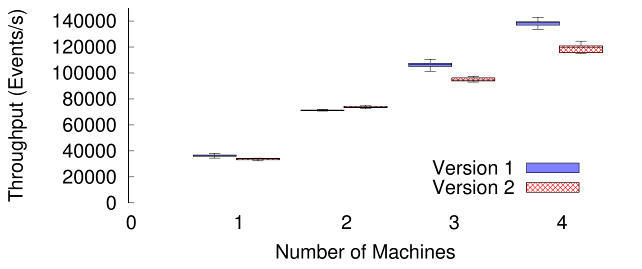


Figure 14: Throughput of multiple *Aragog* verification server checking all 8 types invariant violations

We see that in Figures 16 and 17, the overhead of the local verifiers is low. This is important as the local components are co-located with the production NF instances. To that end, the CPU utilization of the local verifier increases linearly with the number of flow events per second. We also observe the CPU and memory usage for the local verifier is higher at the primaries as they tend to generate more events. Memory at the local components is much less correlated (Table 3), partly due to *Aragog*'s small memory footprint (Figure 17).

The global verifier has higher CPU (Figure 15) and memory (Figure 17) than local verifiers as the global verifier is implemented in Java using Apache Flink. We have set the maximum memory of job manager to 1 GB and executor to 2 GB. In our graphs, we are plotting active memory in Java's heap for the global verifier rather than used memory to avoid including memory waiting to be cleaned up by the Java GC.

Figure 18 shows the CDF of *Aragog*'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 *Aragog* raises an alert.

9.4 Efficacy of Suppression

Each optimization in *Aragog* improves 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). Filters remove the need to send events that are not pertinent and reduce the number of events sent to the verifier by up to 61% for the NATGW (Table 4). Suppressible

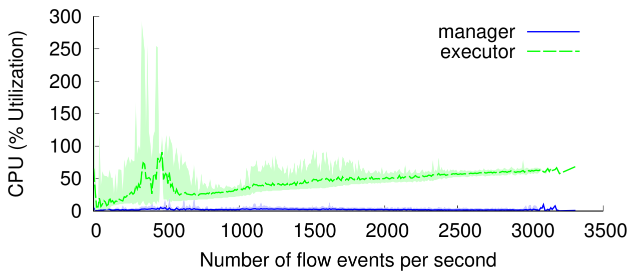


Figure 15: CPU utilization by *Aragog*'s global component. 'Manager' and 'executor' refer to the Flink node designations.

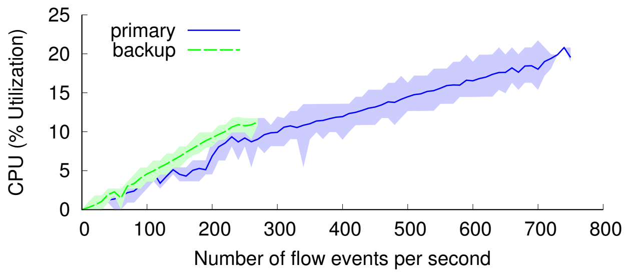


Figure 16: CPU utilization by *Aragog*'s local component. The graph shows CPU utilization of the local verifier at both the primary and backup firewall.

events can further reduce this number (by up to an additional 12% in our experiments).

10 Related Work

Runtime verification. Researchers have studied runtime verification extensively, with many papers dedicated to improving its expressiveness and performance. We find that, unfortunately, these existing systems are a poor fit for our setting. For example, D^3S [28] is a runtime verifier. Like *Aragog*, it focuses on identifying bugs in distributed systems at runtime, and its usage of C++ implementations to specify general-purpose properties means that it can check a wider range of properties than *Aragog*. On the other hand, *Aragog* is able to leverage its domain-specific IV specification language (based on regular expressions) to reduce overhead (*e.g.*, with event suppression). Similarly, while CrystalBall [39] can proactively steer a distributed system away from bad states, it imposes restrictions on the target system's architecture that make sense for a distributed system, but not necessarily for a large-scale NF. A third system, Pivot Tracing [31] tracks only causal relationships and not unrelated events at different machines—a property required by some of NATGW's uniqueness invariants. We emphasize that none of the above implies strict superiority. In particular, as *Aragog* is domain-customized for NFs, it should not be used for more general cases (*e.g.*, it may not be able to verify systems like Chord or Paxos efficiently).

We also note that *Aragog* borrows ideas from two areas within runtime verification. The first is verification of distributed systems, which is broadly separated into two categories based on whether the system assumes a synchronized

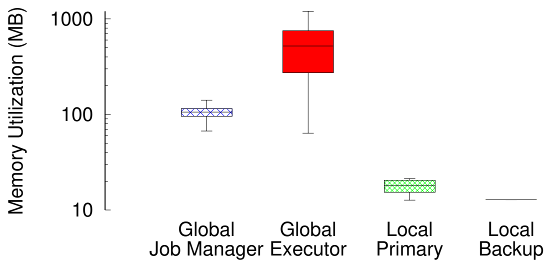


Figure 17: Memory utilization of verifier in MBytes.

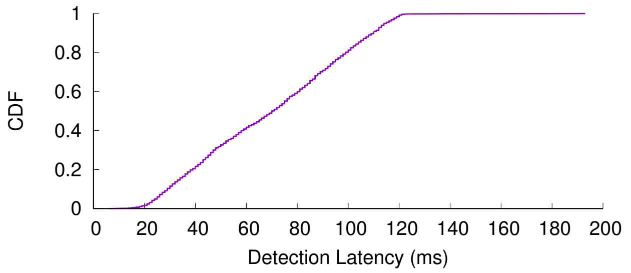


Figure 18: Latency (alert time – packet time) for detecting a violation in the distributed firewall.

global clock [17]. In this respect, *Aragog* would be considered a *decentralized* [14, 17] runtime verification system. The second is parametric verification, which focuses on checking universally or existentially quantified expressions [13, 20, 30, 36]. The location variables in *Aragog* are examples of parametric variables. The main distinction of *Aragog* from these systems is its combination of parametric and decentralized runtime verification through its support for location variables. Moreover, *Aragog*'s efficient implementation of this combination of features through its use of sharding and local symbolic state machine partitioning is new in this context.

Static verification of NFs and distributed systems. Static verification has as equally rich history, including in the domain of NFs and distributed systems [10, 34, 42, 44]. Static verification approaches may provide exhaustive guarantees of correctness, but often suffer from issues of scalability. For this reason, many static verifiers (*e.g.*, [42, 44]) assume single-machine middleboxes, while others (*e.g.*, [25, 29, 40]) may require checking an exponential number of states/paths. Leveraging hand-written NF models can improve scalability compared to verifying source code, but requires tedious and error-prone manual translation of NF models and divorces the verifier from the behavior of the actual deployed system [10, 34].

Aragog makes a different set of tradeoffs, opting to sacrifice principled exploration for improved scalability and giving up the ability to catch bugs early for the ability to test real implementations running over live data. We argue that these tradeoffs are a better fit for our operators' requirements.

Related to the above approaches is the use of semi-automated theorem provers such as Dafny [27]. Users can apply these tools to build systems that are provably correct. A good example of this approach is IronFleet [21], which was

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 4: Total number of generated events, events processed after filtering, and events processed after filtering and suppression for the NAT gateway with all 8 IV specifications.

used to build a verified, Paxos-based replicated-state-machine library. On the other hand, a drawback of this approach is that it requires significant development effort. IronFleet verification, for example, involved tens of thousands of lines of proof. In contrast, *Aragog* aims to be a lightweight (but sans proof-of-correctness) alternative, requiring little to no developer effort by catching bugs at run time.

Stateless dataplane verification. Dataplane verification tools such a HSA [23] and Anteater [32] verify the correctness of a static snapshot of network forwarding tables. Later tools such as Veriflow [24] 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, *Aragog* focuses on verifying complex temporal and stateful properties of general-purpose distributed NFs. For example, *Aragog* can ensure a stateful firewall correctly allows traffic only for connections that are established by an internal sender.

11 Discussion and Conclusion

Aragog is a lightweight verification framework for verifying distributed network functions. To scale to large systems with minimal overhead, *Aragog* 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 *Aragog* can verify any distributed system, its scalability will depend on whether the invariant violations of interest can utilize its sharding and suppression optimization effectively.

Finally, as *Aragog* 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., [18] or some other lightweight and precise event ordering mechanisms. Also for future work are innovations in atomic event export and transactions over streams in *Aragog*.

Acknowledgments

We gratefully acknowledge our shepherd Xi Wang and the anonymous OSDI reviewers for all of their thoughtful reviews, comments, and time. The authors would also like to thank Geoff Outhred for his feedback and support of this work. This work was funded in part by NSF grant CNS-1845749, DARPA contract HR0011-17-C0047, and a Microsoft internship.

References

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

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

[3] Apache Kafka. https://kafka.apache.org/.

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

[5] NetFilter. http://conntrack-tools.netfilter.org/.

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI '07)*, Cambridge, MA, April 2007.

[26] 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.

[27] 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.

[28] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^{3}S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, page 423–437, USA, 2008.

[29] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*, New York, NY, USA, 2019.

[30] 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.

[31] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016.

[32] 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*, pages 290–301, New York, NY, USA, 2011.

[33] 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.

[34] 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.

[35] 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*, pages 207–218, 2013.

[36] 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.

[37] 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.

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

[39] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, page 229–244, USA, 2009.

[40] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, page 213–228, USA, 2009.

[41] 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.

[42] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, page 275–290, New York, NY, USA, 2019.

[43] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *Proceedings of the ACM SIGCOMM 2017 Conference*, page 141–154, 2017.

[44] 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.

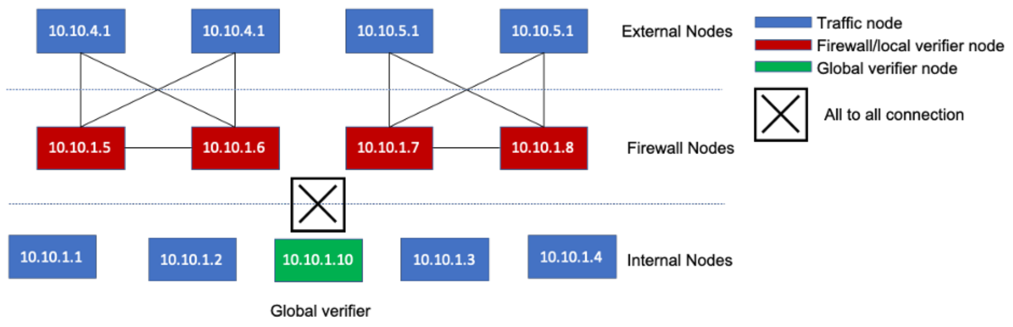


Figure 19: Topology for the distributed firewall demo.

A Artifact Appendix

Aragog is available at: https://github.com/microsoft/aragog. Instructions for installing and running the artifact can be found in the README of this repository.

A.1 Code Structure

A.1.1 SFA generation

SFA generation has three dependencies: `symbolic automata`, `z3` and `antlr`. The primary classes are:

GenerateSFA.java: This is the main class. It takes the event definition file and the IV specification file, and it outputs the SFA in a form that can be accepted by the runtime verifiers. `Antlr` is used to create the parse tree, which is then used to create the global SFA using the class `InvariantVisitor`, which recursively visits each node of the parse tree while constructing the SFA. A DSFA is generated from this automata and printed to a `.sm.g` file along with a DOT file representation.

GenerateLocalSFA.java: This class is called by GenerateSFA to create local versions of the global SFA. Specifically, it takes the SFA and locations as input and outputs the local SFA for each location. The end result of this step is a series of `.sm.[1-9][0-9]*` files, one series for each IV specification.

EventSolver: This class contains the theory of `BooleanAlgebra` logic required to create the SFA. Please refer to Section 6.1 of the paper for details.

A.1.2 Global Verifier

The global verifier has three dependencies: Apache Flink, Apache Kafka, and `antlr`. The primary classes are:

Verifier.java: This is the main class. The program creates state machines according to the provided `.sm.g` files and processes them. The input event messages can come either from a file, a socket, or Kafka. It parses the message, processes it, and raises alerts if required. Everything is done in streams to allow for parallelism.

Creation of the parser uses `ParserFactory.java`, which can parse according to `packetformat.json` or some user-specified custom parser.

GlobalSFAProcessor.java: This class is the runtime DSFA processor. It takes events as input and outputs alerts. Contained in this processor is functionality for reordering events based on their timestamp, tracking stateful variables across events, and advancing all possible instantiations of the DSFA. Critical to the function of the DSFA is an expression tree of binary/boolean operators that assist in evaluating the predicates attached to each transition in the DSFA. See the `expressions` sub-directory for details.

A.1.3 Local Verifier

The local verifier has three dependencies: `cppkafka`, `rapidjson` and `antlr`. The primary files are:

main.cpp: Like `GenerateSFA.java` of the global verifier, the local C++ version is responsible for constructing the state machine from the provided files and processing input events coming from either a file or a socket. The overall flow of the local verifier mirrors that of the global verifier, except that this one is implemented in C++ with none of the Flink support for automatic scaling and fault tolerance: after receiving an event, the event is parsed using the `PacketParser` class and sent to the local SFA processor (described below). The key difference is that the objective of this version is to decide whether the event should be suppressed and output it if not. Events are only suppressed if all state machines agree that they are suppressible.

SFAProcessor.cpp: This is the local, C++ version of `GlobalSFAProcessor.java`. Like other portions of *Aragog*'s local components, the local SFA processor implements a stripped-down, slightly modified version of the global verifier's functionality. In this case, the local node is tracking its view of the global state of the system, given only the locally observed events. As such, it does not need to worry about event reordering or location-variable tracking, which simplifies the implementation and leads to improved performance.

A.2 Firewall Demo

We include in the repository an example experiment involving firewalls and verifiers that emulates a portion of the experimental methodology of Section 9. This experiment expects the user to have a small cluster of machines that can play the role of each type of node. CloudLab is one viable option and we include configurations to assist in allocating such a cluster. The included code configures the topology of Figure 19.

The setup file, `Setup/setup.sh`, installs the required software on each machine in the user's cluster and also installs IP route rules that create an overlay corresponding to the topology referenced above.

Overall, the experiment consists of four external nodes, four internal nodes on a single LAN, and four firewall nodes interposing between the two groups. The firewalls are configured as two high-availability groups with one primary and one hot standby per group. Each primary-standby group shares a virtual IP with the VRRP protocol. We base the traffic between external nodes and internal nodes on traffic models from DCTCP [9].

The rules that are installed in the firewall are simple. Internal nodes can communicate with each other and initiate connections to external nodes. External nodes cannot initiate connections to internal nodes.

Alongside the firewall, each firewall node also runs the verifier, which computes filters and suppression. A single global verifier node runs both the Apache Kafka and Apache Flink deployments. Kafka is responsible for receiving and pipelining the events from all of the local verifiers. Flink is responsible for executing the global verifier.