Stateful Network Testing with Concolic Network Execution

Anonymous authors
Paper under double-blind review

Abstract

As modern networks grow in complexity with virtualized software components, ensuring the correctness of such networks becomes a challenge. Conventional network verification relies on manually constructed models, which are difficult to build and maintain especially for the software components, thus limiting the viability of network verification. On the other hand, network testing through emulation provides high fidelity, but it lacks coverage of the packet header space and forwarding nondeterminism (e.g., ECMP, orderings of concurrent packets). In this work, we propose Neo, a hybrid data plane testing framework that combines model checking and emulation-based testing, with an aim to balance the benefits from both worlds. Neo provides and utilizes formal models for the standardized behavior where models can be easily maintained, and allows the software components in the network to be emulated where an accurate model is not readily available. As a result, Neo covers all possible execution where models are available, including the packet header space and forwarding nondeterminism of typical devices and protocols, while sacrificing these formal guarantees for the software components to obtain accuracy and applicability via emulation. We show that our approach can detect data planes issues that would otherwise be missed by the existing methods alone and that, with our optimizations, it performs reasonably well on various network datasets.

1 Introduction

Verification and testing are an essential part of network operation to ensure conformance to the desired correctness properties. Several techniques have been proposed to either verify or test data plane states [14,21,22,30,32–35,38,39,46,62–65,68]. Among these techniques, verification approaches [22, 30, 32–35, 39, 46, 63–65, 68] require an abstract model of the entire network being verified and provide strong guarantees about the properties of the verified states, assuming the model accurately represents the network. In contrast, testing approaches [14, 21, 38] validate correctness properties by sending traffic through test networks, which are typically emulated with similar configuration to real deployments. By emulating with software implementations instead of abstract models, these testing approaches can validate correctness properties with high accuracy without having to create models in advance, while potentially sacrificing soundness.

However, both approaches have fundamental limitations when applied to modern networks containing virtualized network functions (NFs), such as software firewalls, NATs, proxies, load-balancers (LBs), etc. On one hand, formal verification relies heavily on the accuracy of models. If a model does not accurately represent the real network, false positives and negatives can occur, which adversely impacts trust in the verifier. More importantly, the availability of models varies significantly in practice. For standardized behavior such as L2/L3 packet forwarding, ECMP, etc., models may be manually created with a reasonable one-time effort. However, for software components such as NFs, it is impractical to build and maintain high-fidelity models for each implementation, as these implementations may contain arbitrary code, evolve rapidly, and mostly do not have a formal specification. On the other hand, emulation-based testing requires no up-front effort for modeling, but it is known to produce results with potential false negatives when the system under test is nondeterministic or has an inexhaustible state space with testing alone [15, 25, 26], as modern networks do.

To strike a balance between verification and testing, we propose Neo, a stateful data plane testing framework that combines model checking and emulation-based testing, designed for modern networks with software NFs. The core idea is that it does not have to be all or nothing in either direction (i.e., either all formal verification or all emulation-based testing). Inspired by concolic testing in traditional program analysis [24, 25, 44], Neo allows any part of a network to be emulated for concrete execution where accurate models are not available while symbolically analyzing the remaining modeled components, for which we call *concolic network execution*. For standard components that constitute most of the network (e.g., switches and routers), Neo provides formal models to explore all behavior exhaustively. For the emulated components (e.g., software NFs), Neo creates isolated virtual environments where the emulations are executed as black boxes on appropriate inputs and the outcomes are interpreted accordingly back to the abstract domain. As a result, Neo enables testing for a whole spectrum of networks where previous verification approaches cannot be applied due to the lack of models for the software NFs in the network, and where previous network-wide testing approaches may lead to higher rates of false negatives due to the lower coverage of the packet header space and forwarding nondeterminism (e.g., ECMP, orderings of concurrent packets).

Several challenges arise from this hybrid approach. First, to integrate emulation-based testing with model checking, we need to be able to (1) run arbitrary code at each state transition and (2) modify any state variable during the process of model checking. To this end, we build a framework around the SPIN model checker [29] by dynamically instrumenting the verifier generated by the model checker and linking our own code as a library of callback functions invoked at each state transition (§3.3.1, §3.3.2). Second, since the model checking process is unaware of the emulated devices, we need to synchronize the state of the emulations with the state of their counterparts within the model. To do so, we design an efficient representation to keep track of the emulation states and introduce the ability to *rewind* the state of any emulation instances (§3.3.3). Third, as the emulated devices are treated as black boxes, by definition there is no direct way to know if a previously sent packet is dropped by an emulated device or if it is still being processed. We design three methods with varying degrees of applicability and reliability to detect packet drops within emulations (§ 3.3.4). Fourth, to translate packets between the abstract and the concrete domains and to reliably interpret the concrete execution results, we compute the network-wide packet equivalence classes (PECs) at the beginning of each run based on the input data plane configuration (§3.4). Lastly, with the state explosion problem of model checking [12, 40] compounded with emulation overhead, a naive implementation of Neo may not scale well, especially if we want to reason about concurrent packets and connections. Consequently, we devise a series of optimizations that make the prototype scalable to practical network sizes (§3.5).

We design and implement a prototype of Neo, showing that (1) it extends model checking for the networks with software NFs where models are unavailable, (2) it fully explores all possible behavior of the modeled components, thus reducing the rate of false negatives, and (3) it can detect problems where model-based and emulation-based approaches alone either cannot be applied or cannot guarantee to detect such issues. In addition, Neo scales well with both real-world network datasets and synthesized large-scale networks, where multiple cores can be utilized for testing in parallel. In most scenarios, invariant checks can be done within a few seconds. Compared to the unoptimized version, our optimizations improved the run time by 2× to 1648× faster (the improvement increases with the network scale).

2 Motivation

Difficulties of creating accurate formal models Let's begin with a simple example as shown in Figure 1, with the purpose of illustrating the difficulties of creating formal models that accurately represent real implementations. In this example, an internal host configured with multipath routing sends a web request via a stateful firewall to some external service. With a typical configuration, one may expect the following network invariants to be true: (1) all outbound traffic is allowed, and (2) all inbound traffic is blocked except for replies to previous requests. However, if a Linux-based software implementation is deployed as the firewall, invariant (1) may sometimes be violated. Specifically, when the kernel parameter `rp_filter` is enabled, the firewall performs reverse-path filtering (RPF) [6, 58], causing the outbound packets to be dropped if multipath routing happens to forward the reverse flow through a different router.

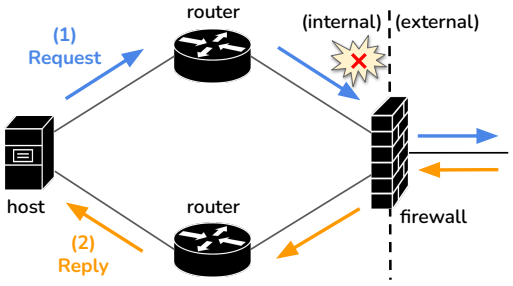


Figure 1: An example of an invariant violation due to reverse-path filtering (RPF).

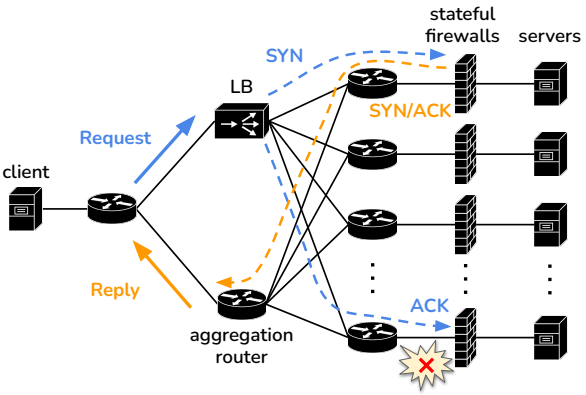


Figure 2: An example of an individually verified NF (LB) still containing bugs and causing end-to-end invariant violation.

To apply verification approaches to prove or disprove such invariants, one needs to first create models (or formal specifications) for each device, which would be virtually impossible given the presence of the software firewall. To accurately model the firewall's behavior, ultimately it is required to formally specify all behavior of the entire kernel networking stack, including the minute details of every kernel parameter that may affect the packet forwarding behavior, which takes an enormous amount of effort and does not scale with the network size and the number of software NFs. Moreover, this process of manual model specification often introduces bugs and inconsistencies between the model and the implementation, as is the case of BUZZ [22] described in [42], which makes it even more unattainable to keep all models updated with every implementation idiosyncrasy across revisions.

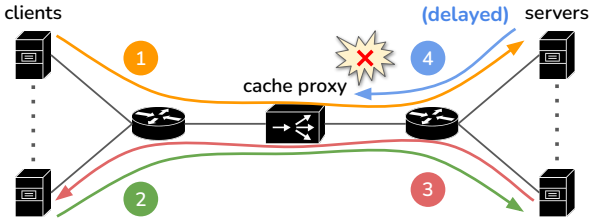


Figure 3: An example of an invariant violation that occurs only under specific event orderings.

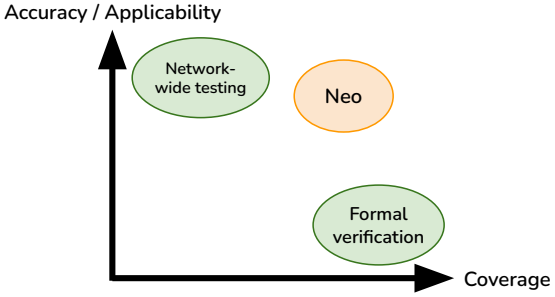


Figure 4: The scope of Neo compared to existing techniques.

Individually verified NFs In Figure 2, we show an example web service network consisting of the LB implementation from Klint [49], which demonstrates that having individually verified NFs is insufficient to guarantee end-to-end correctness at the network level due to the incompleteness of manual specification (i.e., models not capturing necessary details of implementations). Klint is an automated verifier that verifies the individual NF implementations conform to manually constructed formal specifications [49]. In this example, the LB realizes the Maglev algorithm [19]. However, when the LB is used together with an off-the-shelf stateful firewall, we discovered that occasionally the requests were dropped by the firewall, because the LB incorrectly split the packets within the same flow across different paths, making subsequent packets dropped by firewalls where the connection handshake had not been established. If we were to apply formal verification with a model of the LB, assuming it correctly implements the Maglev algorithm (flow-level load-balancing with ECMP), we would not have caught the issue, leading to false negatives. Interestingly, when we used the LB together with the Klint-verified firewall, we did *not* observe the packet drops, because the firewall did not strictly enforce stateful filtering, allowing packets to pass through before a connection is fully established. Similarly, we would have false positives if we were to use a model for the Klint-verified firewall, instead of the real implementation. This illustrates that, even with individually verified NFs, using manually specified models can still lead to incorrect results.

Flaky issues due to forwarding nondeterminism Figure 3 depicts an example that shows the importance of exploring as much nondeterministic behavior in the network as possible. In this example, a cache proxy is configured and expected to only query the backend servers for the first request (of the same asset), which seems to be a natural invariant in the situation. However, this invariant may be violated if the first request is somehow delayed, either due to congestion in the network path or queuing caused by a higher load at the queried backend server, and this violation only happens under specific event orderings. To fully test how NFs would and should behave in similar cases, one must examine *all* possible orderings of concurrent packets, which is one example of *forwarding nondeterminism*. As a result, network-wide testing approaches typically would not be able to detect such issues reliably, causing false negatives.

In fact, the examples above exhibit more than one concern. The invariant violations in Figure 1 and 2 are also flaky, as the issues only manifest when multipath routing happens to forward packets to specific next hops, which means testing approaches may not always exercise the problematic execution. The scenario in Figure 3 involves a software NF that is typically implemented in a general-purpose language and may contain arbitrary code, which makes it difficult to obtain an accurate model to apply formal verification.

Scope of Neo Hence, the goal of Neo is to extend high-quality testing to the types of networks that (1) involve software NFs that are hard to model, and (2) contain nondeterministic behavior in the part of the network that can be easily modeled. Figure 4 illustrates the scope of Neo. By hybridizing the approaches at two ends of the spectrum, Neo enables stateful testing for networks where the previous techniques alone either cannot be applied or cannot reliably detect the issues. We discuss the design details below.

3 System Design

3.1 Architecture Overview

At first, Neo takes an input network configuration file (`network.toml`) that describes (1) the initial data plane state, including the topology, the configuration of individual devices, and the configuration of the *control processes* if needed (§3.2), and (2) the correctness properties under test as a list of *network invariants* (§3.6). Figure 5 shows an excerpt of an input file. After parsing the input and setting up the initial state, Neo systematically and symbolically explores all possible execution of the modeled components while concretely executing the emulated NFs through a lightweight hypervisor whenever the network execution involves the emulated nodes (§3.3). Figure 6 shows the system architecture of Neo. At each reachable state, Neo checks for the configured network invariants whether the current execution path (up until the

```toml
# Example configuration of a modeled device.
[[nodes]]
name = "r1"
type = "model"
[[nodes.interfaces]]
name = "eth0"
ipv4 = "192.168.1.2/24"
[[nodes.routes]]
network = "0.0.0.0/0"
next_hop = "192.168.1.1"
# Example configuration of an emulated device.
[[nodes]]
name = "fw1"
type = "emulation"
driver = "docker"
[[nodes.interfaces]]
name = "eth0"
ipv4 = "192.168.1.1/24"
[nodes.container]
image = "<redacted>/firewall:latest"
command = ["/start.sh"]
# ...

[[links]]
node1 = "r1"
intf1 = "eth0"
node2 = "fw1"
intf2 = "eth0"
# ...

[[invariants]]
type = "reachability"
target_node = "server"
reachable = true
[[invariants.connections]]
protocol = "tcp"
src_node = ".*"
dst_ip = "10.0.0.1"
dst_port = [80]
# ...
```

Figure 5: Part of an input configuration file (`network.toml`).

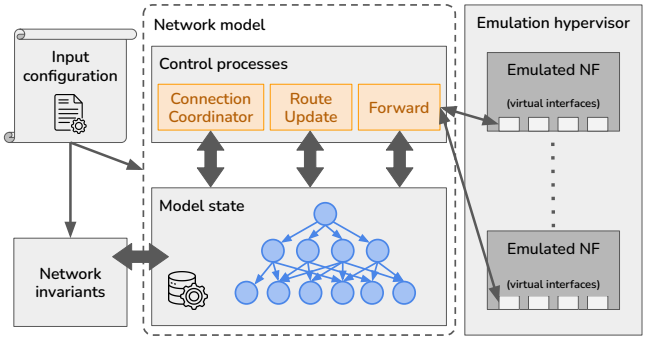


Figure 6: System architecture.

System-wide state variables	
C_i	State of the i^{th} connection, $i \in [0, n_{conn})$.
n_{conn}	Number of concurrent connections.
s_{emu}	State of all emulated nodes.
n_{choice}	Number of nondeterministic choices at the current step.
$choice$	The selected nondeterministic choice in $[0, n_{choice})$.
Per-connection state variables in each C_i	
P_{ij}	The j^{th} symbolic packet in C_i.
s_{conn}	Connection control state.
Per-packet state variables in each P_{ij}	
loc	Current packet location.
s_{proto}	Protocol state.
ip_{src}	Source IP address.
ip_{dst}	Destination IP address.
$port_{src}$	Source L4 port number (0 if no L4).
$port_{dst}$	Destination L4 port number (0 if no L4).

Table 1: A subset of the model state variables.

current state) constitutes an invariant violation. If a violation is found, the exploration halts with the violation reported. However, if the exploration exhausts all reachable states and no violation is found, Neo reports the test succeeded with all configured network invariants validated.

This design draws inspiration from prior verification approaches like VeriFlow, NetPlumber, and Plankton [33,35,52]. However, the key distinction between Neo and the verification approaches lies in the tight integration with emulation-based testing. For ease of discussion, we first introduce the symbolic network model which encodes all modeled components in the network (§3.2) to clarify and provide the basis for formal reasoning. Afterward, we discuss concolic network execution with the integration between symbolic model checking and testing (§3.3), assuming we have derived the packet equivalence classes (PECs). However, due to the black-box nature of the emulated NFs, deriving true PECs may not always be possible. We describe the process of PEC calculation at the beginning of each run and the necessary conditions where the calculation would result in true PECs, including the potential consequences (§3.4). Finally, we explain the optimizations which greatly improved the performance of Neo (§3.5), and the supported network invariants (§3.6).

3.2 Network Model

The network model in Neo is a shared-memory model defined as a nondeterministic finite automaton (NFA), which includes a set of state variables that may contain symbolic or concrete values, and a set of transition functions (provided by the *control processes*) that determine the valid transitions at each given state. Table 1 shows a subset of the state variables in the network model, which we describe below.

Packet equivalence classes (PECs) A packet equivalence class (PEC) defines a packet set where all elements exhibit an identical forwarding behavior across the network. The set of all PECs constitutes the abstract domain of our model, which are derived at the beginning of each run (§3.4). These PECs may be represented in various formats, including ternary bit-vectors [33, 34], multidimensional tries [35, 52], binary decision diagrams (BDDs) [27, 63, 68], sets of mutually disjoint intervals [30], or customized data structures [9]. In this work, we represent a PEC as a set of mutually disjoint intervals for its simplicity and ease of implementation. However, all other common representations will work with Neo's design. Specifically, a PEC is defined based on the 5-tuple values, in addition to s_{proto}, which encodes not only the protocol types but also any protocol-specific states such as TCP flags and ICMP types. A symbolic packet P_{ij} in Table 1 represents one PEC together with its current location *loc*.

Connection equivalence classes (CECs) To reason about stateful behavior with one or more concurrent connections, we extend the notion of PECs to connection equivalence classes (CECs). A CEC is uniquely defined as a set of PECs that occur within the same connection.[1][2] This notion does not fundamentally change the problem but helps assign semantic meaning across the PECs of the same connection. C_i in Table 1 encodes the state of a CEC, which includes its current symbolic packet P_{ij} and the *connection control state*, s_{conn}, which is crucial for coordinating concurrent connections (§3.5.1).

Model state For the system-wide state variables, n_{conn} denotes the number of concurrent CECs, which is determined by the input configuration at the beginning of each run, where C_i represents the state of i^{th} CEC. n_{choice} and *choice* represent nondeterministic decisions during the execution of the NFA. However, the exact semantic of n_{choice} and *choice* depends on the state of the system. They may represent the choice of forwarding next hops, concurrent connections, active control processes, or route updates. Lastly, s_{emu} represents the state of all emulated NF nodes in the network. We later expand on its definition and how Neo synchronizes s_{emu} with the actual emulation instances in § 3.3.3.

Control processes Control processes define the transition functions of the NFA, which mutate the state variables and progress the model to the next states. When more than one control processes are active, all valid transitions will be explored nondeterministically by the model checker. Three control processes are crucial to network execution, as shown in Figure 6. The *forward* process maintains a forwarding state machine for each P_{ij} and forwards the symbolic packets one step at a time, the *route update* process updates the data plane forwarding entries based on the configuration, and the *connection coordinator* process nondeterministically explores all possible orderings of concurrent packets and connections. During the execution of the NFA, each reachable state is then checked against the configured network invariants for potential violations.

The symbolic model described so far may seem comparable to those of the prior verification approaches [30, 33, 35, 63]. However, the key distinction of Neo is the integration with emulation-based testing and the support for reasoning about concurrent connections, as discussed below.

```
1  do
2  :: n_choice > 0 ->
3      select(choice: 0 .. n_choice - 1);
4      c_code { exec_step(&now); };
5  :: else -> break;
6  od
7  assert(!violated);
```

Figure 7: The model checker's view of the entire system in Promela. `select` implements nondeterministic decision. `now` is the global state vector for the model state. `violated` is a flag indicating an invariant violation. `exec_step` is an external function that is linked to our control processes.

3.3 Concolic Network Execution

3.3.1 Symbolic Model Checking with SPIN

At the core of Neo's design is a model checker that systematically explores all reachable model states in depth-first order. To have fine-grained control over the state variables and transitions, Neo leverages SPIN [29], an explicit-state model checker, to pilot model exploration given a skeleton model written in Promela [28], as shown in Figure 7. This allows running arbitrary code (including spawning emulations) and modifying the model state. Thus, the underlying network semantics are mostly hidden from the model checker and implemented as separate control processes (§3.2).

The model executes in a single loop with nondeterministic branching where one state transition occurs per iteration. At different model states, the nondeterministic choice may have different semantic meanings (§3.2). When a particular choice is made, Neo interprets this choice and invokes the corresponding transition function within the control processes to continue the execution. After each transition, the model state is checked against the configured network invariants to see whether a violation is possible. Therefore, eventually the loop ends when there is no more choice to make ($n_{choice} = 0$)

[1] We use the term connection loosely to refer to the communication sessions with fixed endpoints of both connection-oriented protocols like TCP and connectionless protocols like UDP or ICMP.

[2] In theory, it is possible for one PEC to belong to more than one CEC when two separate connections are initiated by the exact same but reverse endpoints at the same time. However, this rarely happens in practice.

because of either an invariant violation or all states having been explored (i.e., no valid transition to an unexplored state).

However, since SPIN operates on explicit, concrete values for state variables, to extend it for symbolic values such as P_{ij}, Neo maintains internal symbolic stores for each variable that may contain symbolic values, including P_{ij} and s_{emu}, and uses raw pointer values as identifiers for each distinct symbolic value. The raw pointer values are used as concrete values and stored in SPIN's state vector. However, inside `exec_step`, Neo reinterprets the raw pointer values via dynamic casting to the proper types (data structures) that contain the real symbolic values. These symbolic stores are duplicate-eliminated (§3.3.3) to ensure that each distinct raw pointer value corresponds to one distinct symbolic value and vice versa.

3.3.2 Concolic Execution with Emulations

Analogous to concolic execution in program analysis [25], to incorporate emulation into model checking, our emulation hypervisor must: (1) control and observe all network I/O of the emulation processes, (2) pause and resume the emulation processes, and (3) reset and restore the states of the emulation processes. To this end, Neo employs container virtualization for the emulated devices, each of which resides in an isolated network namespace [18] with virtual interfaces [36] created according to the input configuration. This allows our hypervisor to achieve (1) and (2). We expand on (3) in §3.3.3.

During state exploration, when a symbolic packet reaches an emulated device, since no accurate model is available, Neo has the packet processed by the real software by translating the symbolic packet from the abstract domain (PECs) to the concrete domain (packets with concrete values) and sending the concrete packet to the appropriate virtual interface (Figure 6). The *forward* process then interprets the outcome as the action performed on the symbolic packet by translating the concrete results back to the abstract domain.

To instantiate a concrete packet from a symbolic one, by definition, it suffices to pick one arbitrary packet from the PEC. If the set of all PECs we derive is true, then this concretization process would be sound, which means that using any concrete value within a PEC to represent the symbolic packet would result in the same behavior. To ensure deterministic execution traces, Neo picks the first (smallest) packet in the PEC as the concrete representative. Interpreting the concrete results back to the abstract domain can be achieved similarly. Assuming after sending a concrete packet, the emulated device emits one or more packets,[3] we can easily map the output concrete packets back to their corresponding PECs by looking up the set of all PECs. However, as the emulated devices are run as black boxes, without additional knowledge about the emulated devices, it cannot be guaranteed to always derive the set of true network-wide PECs, for which we discuss the necessary conditions and implications in §3.4.

[3]We discuss the cases of silent packet drops in §3.3.4.

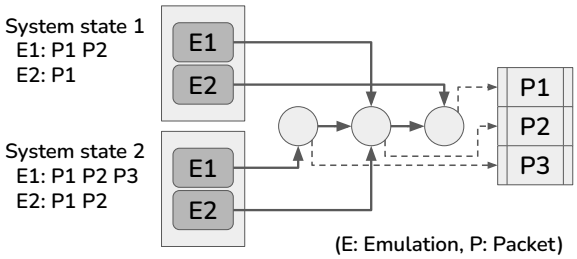


Figure 8: Duplicate-eliminated emulation states.

3.3.3 Managing Emulation States

One major challenge with this concolic design is representing the states of the emulated devices in the model state and the synchronization between the two. Since the model checker and the emulated devices are unaware of each other, during the depth-first traversal of the model state space, the emulation states may not be synchronized with model states. For example, after exploring one specific execution path of the NFA, the model checker would backtrack to a previous branching state and continue exploring the next branch, while the emulated devices remain in the state at the end of the last execution path without backtracking. Therefore, it is crucial to be able to record and restore the states of emulated devices.

To achieve this, Neo defines the state of each emulated device as the list of events the device has ever seen since the start of its execution, such as the list of all previously processed packets. The state of all emulated devices in a network is then defined as $s_{emu} = \{(i, e_i) \mid \forall i\}$, where e_i is the state of the i^{th} emulated device. Conceptually, s_{emu} is effectively a map that captures the entire snapshot of all histories of events of all emulated devices in the network. When a concrete packet is to be sent to an emulated device, the actual state of the emulation (tracked by the hypervisor) is first compared against the emulation state in the model (s_{emu}). If there is a mismatch, possibly due to state backtracking by the model checker, Neo resets the actual state of the emulated device to be aligned with the current model state, either by resetting and replaying the history of events, or simply fast-forwarding the emulation state if the current state is a prefix of the desired state in the model. This approach is relatively simple and more tractable than taking snapshots of the entire virtual memory.

However, a naive implementation would still lead to scalability issues due to the number of different combinations of event histories that need to be recorded for all emulated devices. We rely on two observations to make this approach scalable: (1) The same event may be part of multiple histories. (2) Many histories differ from each other only by a few events. As a result, we can leverage these redundancies for optimization. All events (e.g., concrete packets) are kept in a hash table and reused. No event is duplicated at any point during the entire state traversal. Moreover, any sequence and sub-sequence of events are also duplicate-eliminated. Fig-

ure 8 illustrates this layout. As a result, all elements of s_{emu} are effectively pointers to their event histories.

3.3.4 Drop Detection

Another challenge is to reliably interpret the outcome after sending a concrete packet to an emulated device, especially when the emulated device silently drops the packet. To tell packet drops from delayed responses, Neo provides three detection mechanisms: drop timeout, in-kernel drop monitor (`drop_mon`) [54], and drop tracing with eBPF [1,45].

Drop timeout estimate Waiting for a timeout is the more general approach applicable to any type of emulation. We dynamically estimate the timeout value based on past packet latency, similar to TCP's retransmit timeout [11,17,31,47,51]. At first, a relatively high timeout value is assigned to avoid false positives. During state exploration, each L_n, the latency of the n^{th} concrete packet, is recorded and the drop timeout estimate DTO is calculated as shown below. μ_n and σ_n refers to the arithmetic mean and mean deviation, respectively. α and β are gain factors, empirically set to 0.2 [31,47], while C is the mean deviation scalar used to account for system load. This method is the most applicable but may lead to potential false positives when there are sudden changes in packet processing latency in the emulated devices.

$$\begin{aligned} \mu_n &= \mu_{n-1} + \alpha \times (L_n - \mu_{n-1}) \\ \sigma_n &= \sigma_{n-1} + \beta \times (|L_n - \mu_{n-1}| - \sigma_{n-1}) \\ DTO &= \mu_n + \sigma_n \times C \end{aligned}$$

Drop monitor For better accuracy, Neo can utilize the in-kernel drop monitor via Netlink API [53] for emulated devices based on Linux. The drop monitor traces all `kfree_skb` invocations as well as hardware-level packet drops [54], and then sends alerts to the subscribed userspace processes. This ensures detection of packet drops in the kernel networking stack without false positives or negatives. However, since it is a system-wide API, *all* packet drops in the OS will generate an alert even for packets not related to Neo. This inefficiency leads us to explore a third option.

Drop tracing with eBPF To avoid unrelated alerts, Neo provides a drop tracing module based on eBPF [1,45], which allows fine-grained control to filter out irrelevant events directly within the kernel space. This filtering is carried out based on the inode number of each emulation's network namespace, the ingress interfaces, and other header fields based on the PEC of the concrete packet being processed. Similar to drop monitor, this method guarantees correctness when a packet is discarded along the networking stack, but it avoids unnecessary communication between the kernel and the userspace.

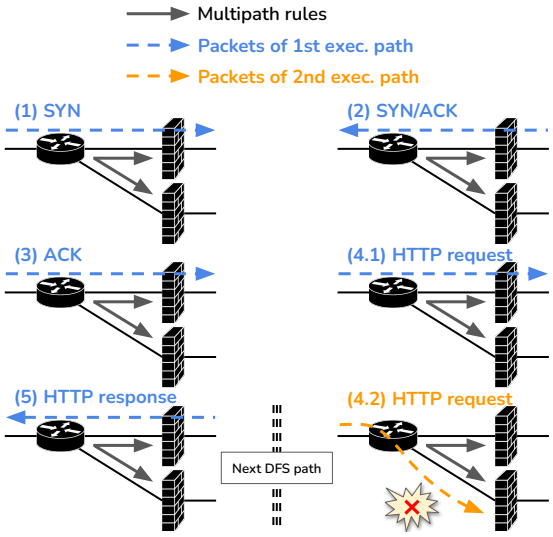


Figure 9: An example of an incorrectly modeled HTTP communication with multipath routing at the packet level.

3.3.5 Handling Multipath Routing

Naive modeling of multipath routing at the packet level can introduce false positives with model state exploration. Figure 9 shows an HTTP communication explored in depth-first order with packet-level multipath routing. The initial sequence of the TCP handshake and the HTTP request-response are modeled correctly. However, in the next execution path after backtracking, the HTTP request takes an alternate path and gets dropped by the firewall that did not observe the TCP handshake. Naturally, this is an invalid execution as multipath routing typically happens at the flow level. To avoid invalid executions and improve performance, we maintain additional states for devices with multipath configured, so that nondeterministic choices for multipath are available only for the first packet of a 5-tuple flow, ensuring all subsequent packets within the same flow follow the same path.

3.4 PEC Calculation

Before exploring the state space, Neo computes the PECs by first collecting all traffic specification values found in the input configuration file, such as IP prefixes and port numbers, and then carrying out a series of packet set intersections and differences to produce the PECs. This process captures the true PECs for the modeled devices. However, for the emulated devices, since no accurate model is available for such devices, we prioritize applicability over precision — Neo allows users to provide a list of configuration files inside each emulation's file system, which are scanned to extract any traffic specification values for PEC calculation.[4] Therefore,

[4]Neo also takes into account any packet-related values from the execution environment of each emulated device, including environment variables,

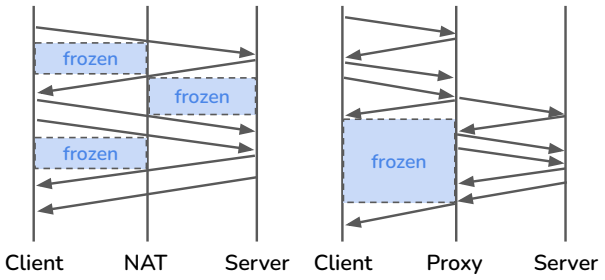


Figure 10: Two common patterns of dynamically created concurrent connections with a TCP session.

the derived PECs are generally an approximation for the emulated devices. This calculation process would lead to the true network-wide PECs if the distinctive packet forwarding behavior of each emulated device is solely affected by their execution environment (configuration files, environment variables, command-line arguments). In other words, the derived PECs would be sound if there is no hard-coded values within the implementations of the emulated devices.

Another approach of deriving accurate PECs from the emulated devices is through white-box program analysis, which essentially equates to constructing formal models for software implementations and suffers from similar limitations as described in §1 and §2.

3.5 Optimizations

3.5.1 Efficient Exploration of Concurrent Connections

To test stateful properties, it is crucial to efficiently explore all interleavings of concurrent packet flows or connections, as illustrated in §2. In Neo, there are two sources of concurrent connections. They may be specified by the network invariants right from the start, or dynamically created by the emulated devices in the network during execution. Figure 10 shows two common patterns of dynamically created concurrent connections, where a client initiates a TCP session to a server, and, along the paths, the TCP packets pass through a masquerading NAT or a proxy. Recall that, in our network model, we define a connection to be the communication between a set of fixed endpoints with the same 5-tuple flow in both directions (§3.2). Therefore, in both cases, the communication is modeled as two separate connections: one between the client and the NAT/proxy, one between the NAT/proxy and the server.

With multiple concurrent connections, a naive design of exploring every possible combination of packet orderings in all connections would not scale. Therefore, we introduce an optimization with partial-order reduction (POR) [10, 48] for concurrent connections based on these observations: (1) the forwarding behavior of stateless devices remains the same regardless of packet orderings, (2) most devices in a network are stateless, and (3) even with concurrent connections, a connection is sometimes *frozen* due to the lack of an active packet (Figure 10). Thus, we can soundly reduce the state space with Algorithm 1, where s_{conn} is defined as

$$s_{conn}(C_i) = \begin{cases} 2, & C_i \text{ is arbitrarily executable.} \\ 1, & C_i \text{ is about to enter a stateful device.} \\ 0, & C_i \text{ is frozen (no active packet).} \end{cases}$$

For the modeled devices, Neo knows whether a device is stateful or stateless based on the given model. However, for the emulated devices, since they are run as black boxes, Neo regards all of them as stateful to fully explore the packet orderings. This POR greatly reduces the state space and alleviates the state explosion problem with concurrent connections.

Algorithm 1: Interleaving connections with POR.

```
1  Function ChooseConnection():
2      if ∃i. s_conn(C_i) = 2 then
           // Deterministic execution.
3          n_choice ← 1;
4          i ← pick the first i where s_conn(C_i) = 2;
5          return {i};
6      else if ∃i. s_conn(C_i) = 1 then
           // Nondeterministic execution.
7          n_choice ← |{C_i | s_conn(C_i) = 1}|;
8          return {i ∈ ℤ | 0 ≤ i < n ∧ s_conn(C_i) = 1};
9      else
10         return ∅;
```

3.5.2 State Fragmentation and Hashing

In Neo, the model checker stores every state visited during exploration, which can lead to increased memory usage. To mitigate this, we introduce an optimization called state fragmentation, which stores portions of the state in separate memory chunks, with only pointers to these chunks kept in the model state. This reduces the memory cost for unused state variables and avoids storing copies of unchanged variables across states. For example, pointers of currently unused variables are set to null, and variables that do not change often contain only a single or a few copies.

State fragmentation alone already improves memory usage, as new states often differ only slightly from previous ones. However, the model checker may still produce multiple copies for the same value across different execution paths. To prevent this, we store all allocated chunks in hash tables, representing each distinct value with a single copy. This further reduces memory usage and allows representing symbolic values in an explicit-state model checker (§3.3.1). Figure 11 illustrates how state fragmentation and hashing conserve memory.

command-line arguments, etc.

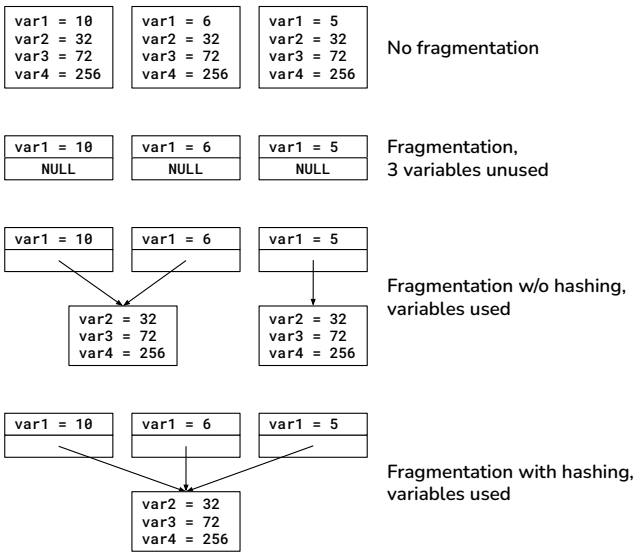


Figure 11: State fragmentation and hashing.

3.6 Network Invariants

Neo supports several invariants out of the box, as shown in Table 2. One might expect Neo to use linear temporal logic (LTL) [50] for invariant specification given our use of SPIN. However, LTL cannot capture some interesting invariants, such as ensuring consistency across all execution paths of multiple devices, since LTL operates over a linear sequence of states. CTL [16] or CTL* [20] could address this, but SPIN lacks native support for them. Our use of symbolic values also prevents us from using these temporal logics since SPIN could not interpret the semantics of symbolic values. Instead of relying on extensions [60, 61], we implement the checks for each invariant type (Table 2). Each type is parameterized with one or more traffic specifications and other type-specific parameters (e.g., target devices, sub-invariants). Given the depth-first state space traversal, Neo maintains and checks the states necessary for each invariant type. Most data plane properties can be expressed through these invariants [30, 34, 35, 39, 46, 65, 68], which is more practical and efficient than requiring network operators to use LTL. Neo can also be extended to support new invariant types if needed.

4 Implementation

We built a prototype of Neo in C++ and Promela, supporting container-based emulation with Docker [41] and common protocols like HTTP, TCP, UDP, and ICMP. We packaged various software NFs as container images for evaluation, including software routers, firewalls, NATs [3], load-balancers [2], and the DPDK-based, individually verified implementations from Klint [49]. Container images can be used in Neo by simply specifying the image name and tag in the input configuration file (Figure 5). While we primarily test software NFs, any software with visible network I/O can be packaged and used within Neo. It is also possible to incorporate other forms of emulation, such as virtual machines or hardware testbed, which we leave as future work. All experiments are conducted on a machine running Linux 6.6.2 with an Intel i7-1370P and 64 GiB RAM.

5 Evaluation

5.1 Dataset

To evaluate our design, we perform a series of experiments with various network datasets, as shown in Table 3, including the ISP topology data from Rocketfuel [57], the Stanford network dataset [37], our campus network,[5] synthesized fat-tree data center network for evaluating scalability, and a small web-service network with concurrent connections to demonstrate Neo's ability to soundly cover all packet orderings efficiently. The total number of CECs in Table 3 denotes all CECs computed from the input data plane configuration. However, the exact number of CECs required to be checked depends on each invariant specification.

5.2 Comparison with Full Emulation

5.2.1 Flaky tests from in-network nondeterminism

We first examine the qualitative difference between Neo and prior work that emulates the entire network. Note that we could not compare Neo with the prior formal approaches, as the lack of accurate models would prevent them from being applied in the first place, especially for networks with complex software components. To compare with full-network emulation, we adopt containerlab to use containers to form any specified network directly parsed from Neo's data plane input configuration. This use of container also aims to align with Neo's design to alleviate the performance overhead between different emulation techniques (e.g., virtual machines).

Recall the reverse-path filtering example from Figure 1, §2. Here we expand the number of hosts to 10, and repeatedly query a server behind the firewall, with `rp_filter` enabled, for 100 times. As a result, on average, 60% of the requests passed through the firewall, which means that, for this example, 60% of the time, simple testing with network emulators would *not* detect the configuration error and would lead to flaky false negatives. In contrast, since Neo would explore all potential paths of multi-path forwarding, we successfully detect the configuration issue 100% of the time.

[5] The campus network is anonymized for double-blind reviews and NDAs.

Type	Description
Reachability	The specified target endpoints should be always reachable.
Segmentation	The specified target endpoints should be always unreachable.
Reply-reachability	After querying the specified target endpoints, the replies should always reach the original sender.
Waypoint	The specified traffic should pass through (one of) the specified waypoints.
Loop absence	There should be no forwarding loop.
Load-balancing	The dispersion index across the specified target endpoints should be within the specified threshold.
Consistency	All specified sub-invariants should contain the same truth value. ("$\mathcal{A} = \mathcal{B} = \mathcal{C} = ...$")
Implication	The implication chain of invariants is true. ("$\mathcal{A} \implies (\mathcal{B} \implies (\mathcal{C}...))$" is true.)

Table 2: List of network invariants currently provided.

Network	Nodes	Links	Total CECs
ISP-1 (AS 3967)	79	147	441
ISP-2 (AS 1755)	87	161	483
ISP-3 (AS 1221)	108	153	459
ISP-4 (AS 6461)	141	374	1122
ISP-5 (AS 3257)	161	328	984
ISP-6 (AS 1239)	315	972	2916
ST-1 (Stanford AS-level)	103	239	717
ST-2 (Stanford AS-level)	1470	3131	9393
Anon. campus network (core1)	65	134	2426
Anon. campus network (core2)	74	172	2321
Anon. campus network (core4)	67	146	2038
Anon. campus network (core5)	38	81	1250
Anon. campus network (core8)	30	46	848
Anon. campus network (core9)	33	79	1336
Anon. campus network (all)	236	524	7349
4-ary fat-tree DCN	97	124	262–277
6-ary fat-tree DCN	312	428	908–961
8-ary fat-tree DCN	717	1020	2166–2293
10-ary fat-tree DCN	1372	1996	4240–4489
12-ary fat-tree DCN	2337	3452	7334–7765

Table 3: Networks used in evaluation

5.2.2 Real-world datasets with end-to-end connectivity

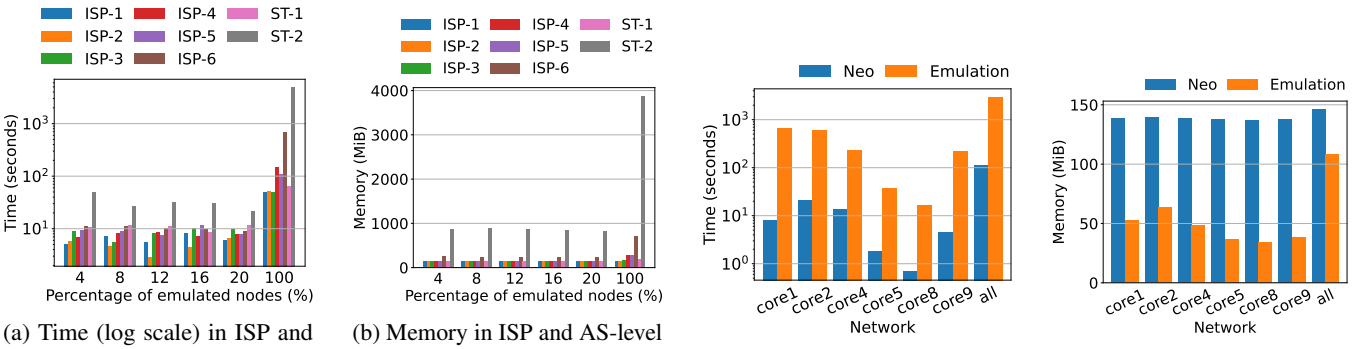
To examine Neo's performance compared to full emulation, we conduct identical network testing with both Neo and containerlab on the real-world datasets we collected (i.e., ISP topology from Rocketfuel [57], Stanford AS topology [37], and the anonymized campus network). Due to the lack of public data for detailed network configuration, for the ISP and AS topology, we sample a number of nodes as emulated firewalls and software routers running as containers, and check for the end-to-end connectivity between leaf nodes in the network. Within our anonymized campus network, however, we parsed the real device configuration files and deployed the parsed ACL rules and converged OSPF routes as realistic routing policies, and check for pair-wise connectivity between all buildings within the same network.

Figure 12 shows the overall time and memory required to run Neo and full emulation. In addition to the amount of network nodes running as emulations, one key distinction between Neo and full network emulation is the fact that Neo keeps track of the states of the emulated devices. If multiple packets may trigger state changes within an emulated node, Neo would explore all possible orderings of packet entrants by rewinding the emulation states, while full network emulators typically do not track the emulation states, which may further lead to potential false positives or negatives. Full-network emulators may implement similar state tracking mechanisms. However, in most network emulators, restarting a large portion of the network frequently can lead to severe performance degradation. We notice that, in Figure 12(d), although Neo utilizes slightly more memory, our design scales better with larger networks compared to full emulation.

5.3 Scalability

To evaluate our design at scale, Figure 13 shows a tenant subnet of a multi-tenant fat-tree DCN, inspired by [23]. Each ToR switch is connected to an individual tenant subnetwork, which carries two types of traffic, whitelisted and graylisted, and two types of servers, public and private. All traffic is allowed to access the public servers, but only the whitelisted traffic is allowed for private servers. A typical stateful filtering policy is enforced on graylisted traffic, where only replies to past requests are allowed. The invariant specifies that this policy should not be violated. A key distinction between Neo and the past work [23, 46] is the use of real implementation, which allows us to truthfully capture the behavior in reality. In our test, we assume that some tenants now wish to reclassify HTTP from graylisted to whitelisted, which is realized by updating the two routers. However, during the experiments, Neo finds the following violation where legitimate replies to past requests are dropped. $R2$ gets updated before seeing the request packet, which is then forwarded directly to $R1$ without passing through the firewall. Subsequently, the reply reaches $R1$ before $R1$ is updated with the new policy. Abiding by the old policy, $R1$ forwards the reply to the firewall. Finally, the firewall drops the reply for it has not seen the request. We perform the above experiment with different numbers of tenants and varying percentages of tenants (0%, 50%, and 100%) that go through the policy change. When at least one tenant is updating the whitelisting policy as described above, we correctly caught the violation and reported the steps leading to the violating state.

Figure 14(a) show the time and memory usage with increasing network scale. When tenants update their whitelisting



(a) Time (log scale) in ISP and AS-level networks. 4–20%: Neo, 100%: full emulation.

(b) Memory in ISP and AS-level networks. 4–20%: Neo, 100%: full emulation.

(c) Time (log scale) in campus networks.

(d) Memory in campus networks.

Figure 12: Comparison with full emulation for real-world networks.

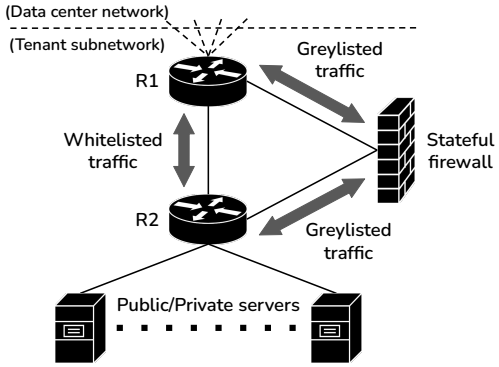


Figure 13: A tenant subnet of a fat-tree data center network.

policies, it leads to quick detection of invariant violations. When the model checker finds an invariant violation, the trace is reported immediately and the exploration is terminated, which means the model checker did not have to exhaust the state space. In contrast, the invariant checking is the most difficult when no issue occurs, since all executions need to be exhausted to confirm that no violations are possible. On the other hand, we observe that the memory requirement grows slowly with the network size regardless of whether the state space is fully explored.

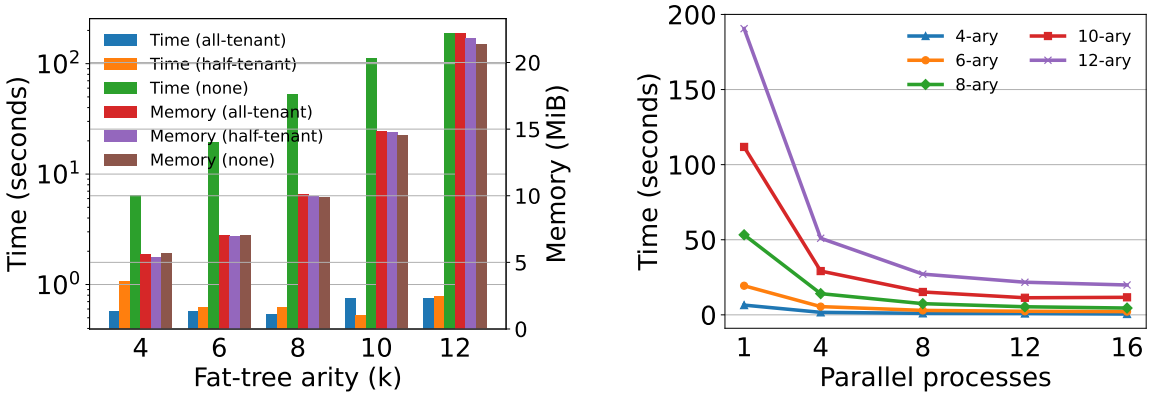
We also examined the scalability with parallel processes, as shown in Figure 14(b). Since independent CECs and the individual invariants can be checked in parallel, Neo accepts a command-line option for specifying the maximum number of parallel tasks. We see that as the number of parallel processes increases, the overall performance improves, though the improvement gradually slows down. The overhead comes from the fact that all the processes share the same underlying resources, including the container API service and the OS. However, the design allows the possibility of running emulations across machines, which we leave as future work.

5.4 Emulation and Drop Detection Overhead

Besides overall scalability, we recorded microbenchmarks for each packet injection and the use of emulations, as shown in Figure 15. Figure 15(a) summarizes the emulation and packet injection overhead for checking one CEC. "Total check time" refers to the time to check one CEC for one invariant. "Emulation overhead" includes the "emulation startup" time (creating and starting containers), "state rewind" time (rewinding the emulation states as described in § 3.3.3), "packet latency" (latency between sending the concrete packet(s) and interpreting the results), and other bookkeeping tasks such as updating emulation states. We can see that, for each CEC, the emulation overhead occupies the majority of the overall run time, while the emulation startup and state rewind, when it happens, constitute most of the emulation overhead. The actual packet latency only contributes a very small portion of the resource utilization.

When a packet injection generates one or multiple packets from the emulation (received by the emulation hypervisor), we observe that there is no significant difference in terms of latency across different drop detection methods (the lower 50% of Figure 15(c)), which is expected since the packets are received by a dedicated listening thread without triggering drop detection. However, when a packet is dropped, the eBPF method has the fastest reaction time among the three methods, with the drop monitor method being slightly slower and the timeout method being the slowest (the higher 50% of Figure 15(c)), which is because the timeout method requires Neo to wait until the timeout expires before it learns about the packet being dropped.

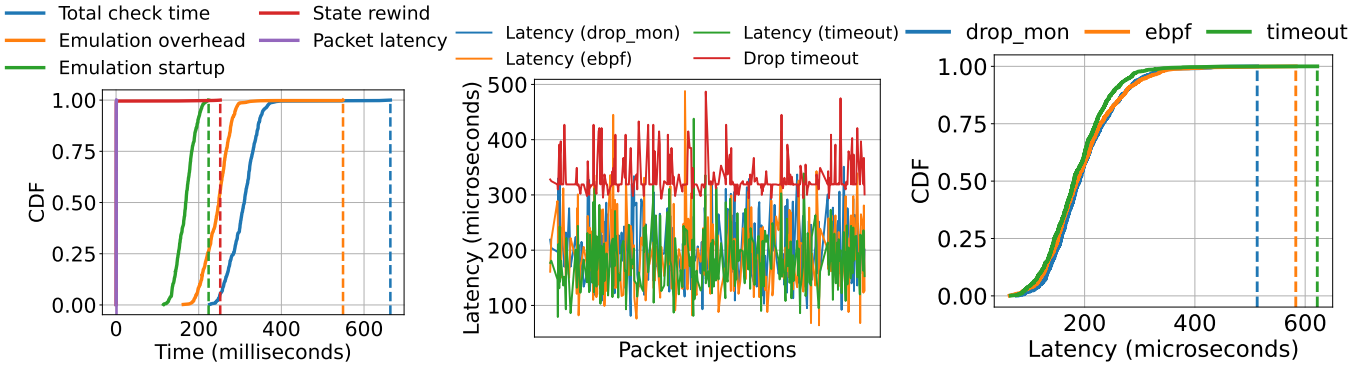
Figure 15(b) shows the latency of all packet reception events with different drop detection methods, and Figure **??** shows the latency when the packets are dropped. The drop timeout in both figures denotes the actual timeout values from the timeout estimation used for each packet injection. We see that when the packets are received (Figure 15(b)), there is no major difference in latency between drop detection meth-



(a) Time and memory with increasing network scale.

(b) Different degrees of parallelism and network sizes.

Figure 14: Scalability with the increasing sizes of a multi-tenant fat-tree data center network.



(a) Emulation overhead

(b) Receive latency

(c) Packet latency

Figure 15: Emulation and packet injection overhead (fat-tree DCN).

ods (except for the occasional spike when the system was under high load), which is expected because the listening thread would receive the packets before the drop detection mechanisms are triggered.

5.5 Optimizations

5.5.1 Load-balancing with concurrent connections

Figure 16 is a web service network designed to experiment with multiple concurrent connections. We create the networks with varying numbers of HTTP applications, each load-balanced through NAT with IPVS/LVS [2] across servers. We check that the incoming requests are distributed across servers evenly by computing the dispersion indices at each step to see if it satisfies the specified threshold. Different load-balancing algorithms were tested, such as source/destination-hashing, Maglev-hashing [19], and least-connection. Neo successfully detected the cases where the invariant was violated.

To assess the benefit of Neo's optimized design, we perform the experiments while disabling most of the optimizations, including POR for concurrent connections, state hashing, and

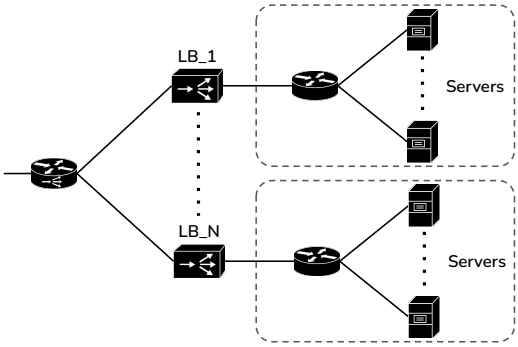


Figure 16: A web service network with LBs and concurrent connections.

Algorithm	Optimization enabled?	2 LBs, 4 servers (4 concurrent CECs)	4 LBs, 16 servers (8 concurrent CECs)	6 LBs, 36 servers (12 concurrent CECs)
Maglev [19]	Yes	0.73 s	0.50 s (violation)	0.50 s (violation)
	No	1.95 s	0.42 s (violation)	0.48 s (violation)
Destination-hashing	Yes	0.86 s	0.43 s (violation)	0.54 s (violation)
	No	2.60 s	0.51 s (violation)	0.55 s (violation)
Source-hashing	Yes	0.76 s	10.31 s	5 m 41 s
	No	1.97 s	> 4 h 43 m (killed)	> 5 h 23 m (killed)
Least-connection	Yes	0.78 s	10.64 s	5 m 40 s
	No	1.85 s	> 4 h 46 m (killed)	> 5 h 34 m (killed)

Table 4: Concurrent CECs with and without optimizations (N3)

the duplicate-eliminated emulation states. Table 4 summarizes the time needed to check the load-balancing invariant for N3 with and without optimizations. For the smaller network (2 LBs), the performance improvement is not as significant. However, for the relatively larger networks (4-6 LBs), Maglev and the destination-hashing resulted in invariant violations due to the higher dispersion indices larger than the given threshold,[6] and hence the violations were quickly reported. With source-hashing and least-connection algorithms, the unoptimized version did not finish the check and was killed for running out of memory (32G), which is to be expected because, without POR and state hashing, every ordering of connection interleavings needs to be explored and the values of the state variables would be copied several times. In summary, the optimizations improved not only the memory usage but also the run time by 2.35 to 3.01 times faster for the smaller network and 57 to 1648 times faster for the larger networks.

6 Limitations and Future Work

Support for emulation formats It is possible to extend the system to other formats of emulation such as virtual machines and hardware middleboxes (as long as Neo can send packets and reset states). However, our current implementation focuses on emulating software NFs as containerized processes. We plan to expand the support for other emulation formats in the future.

Coverage within emulation black boxes As described in § 3.3.3, certain types of internal nondeterminism in the emulated software (such as multithreading, random number generators, custom hash functions, etc.) may cause missed execution paths since we do not have visibility into the black boxes.[7] For this problem, Neo allows sending multiple packets for each injection to cover the common cases of potential behavior, which means Neo may not achieve complete coverage *within* the emulated software. In particular, we don't expect to catch corner cases in NFs that escape a robust and complete testing regime. For network-level nondeterminism from data plane dynamics, Neo is exhaustive in contrast with simple testing, while avoiding the need for a full behavioral model and the lack of fidelity that is inherent in model-based techniques.

[6] The dispersion index threshold of the invariant is configurable by the user and should be configured based on individual networks. We set it to 2.5 in this example for desmonstrative purposes.

[7] However, it may not be easier to create truthful models for these features with formal modeling.

7 Related Work

Data plane verification Data plane verification tools [30, 33–35, 39, 56] verify that a given data plane meets specified correctness requirements. These tools rely on having accurate models for software NFs in the network and sometimes assume deterministic or history-free networks. To achieve better accuracy and high-fidelity models, attempts have been made to simplify the creation of models using custom languages such as SEFL [59], NetSMC [65], or through FSM-based model extraction such as Alembic [42]. However, they either still require an understanding of the NFs and non-trivial manual effort to create the model, or are limited in scope of applicable NFs. With emulation, Neo simplifies data plane verification for a large fraction of real-world networks.

Configuration verification Configuration verification attempts to verify device configurations for a network before deployment, to catch issues ahead of time. Tools such as Tiramisu [5], Plankton [52], and Minesweeper [8] are examples of configuration verifiers. Such tools, at least the existing ones, are not capable of checking the evolution of network behavior under stateful forwarding of packets and interleavings of concurrent connections, as Neo can.

Network emulation Emulation-based testing [14, 21, 38] checks the correctness of networks by emulating them on a separate, usually virtualized, infrastructure with a configuration close to that of the deployment, and thus achieves high

fidelity. For networks with nondeterministic components, however, emulation-based techniques may not always exercise the executions where issues occur. We propose a tight integration between the emulation-based and model-based approaches by interpreting emulation behaviors directly within the model-checking process.

NF verification NF verification [49, 66, 67] focuses on verifying the properties of individual NF implementations, such as liveness, crash-free, and conformance to given specifications, via theorem proving or symbolic execution. The focus is fundamentally orthogonal, as we aim to check for data plane properties (Table 2), against the complex data planes where previous methods could not apply. One cannot reason about these properties without the knowledge of other devices in the network. More importantly, verifying NFs against specifications is different from model/specification synthesis, which means the existing work [49, 66, 67] cannot be extended for our purpose.

Model checking In the networking domain, past efforts with model checking have focused on individual software, protocol implementations [13, 43, 55], and configurations [52]. In contrast, Neo uses it for verifying data planes with stateful software components.

Symbolic and concolic execution Neo's approach of executing symbolic models as well as real software together to perform verification is similar to the concolic testing technique used in software verification [7, 24, 25]. Our motivation for using real software, however, is that it is impractical to obtain faithful models for complex NFs in the context of data plane verification. Namely, we focus more on the data plane issues manifested from the network-visible behavior of the software, rather than those within the software itself.

8 Conclusion

We proposed Neo, a concolic data plane testing framework for networks incorporating complex network functions. By combining emulation of real NF implementations with model checking for standardized data plane components, Neo strikes a balance between accuracy and coverage. Our experiments show that Neo can be a powerful tool in ensuring the correctness of heterogeneous networks in cases where the conventional methods fall short.

Artifacts

Please visit our anonymized repository [4] for detailed description of how to use the tool and how to reproduce certain experiments from the paper. The only experiment that cannot be reproduced is the one based on the anonymized campus network. The campus network dataset is under NDA and cannot be released due to security reasons. However, there are other comparable experiments based on publicly available datasets (e.g., the ISP and AS-level networks). Also, the aggregated performance statistics of the campus network tests can also be found in the repository, which have been sanitized and do not include specific configuration values or sensitive information.

References

[1] Extended berkeley packet filter. https://ebpf.io/. Accessed: 2023-04-16. URL: https://ebpf.io/.

[2] The linux virtual server project - linux server cluster for load balancing. http://www.linuxvirtualserver.org. Accessed: 2023-04-16. URL: http://www.linuxvirtualserver.org/.

[3] netfilter/iptables project homepage - the netfilter.org project. https://www.netfilter.org/. Accessed: 2023-11-30. URL: https://www.netfilter.org/.

[4] Anonymized repository of neo artifacts., August 2024. URL: https://anonymous.4open.science/r/neo-F701.

[5] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 201–219. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/abhashkumar.

[6] Fred Baker and Pekka Savola. Ingress filtering for multihomed networks. *RFC*, 3704:1–16, 2004. https://doi.org/10.17487/RFC3704.

[7] Sharon Barner, Cindy Eisner, Ziv Glazberg, Daniel Kroening, and Ishai Rabinovitz. Explisat: Guiding sat-based software verification with explicit states. In *Haifa Verification Conference*, pages 138–154. Springer, 2006.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM. https://doi.org/10.1145/3098822.3098834.

[9] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddNF: An Efficient Data Structure for Header Spaces. In *Haifa Verification Conference (HVC), 2016*, August 2016. URL: https://www.microsoft.com/en-us/research/publication/ddnf-an-efficient-data-structure-for-header-spaces/.

[10] Dragan Bošnački, Stefan Leue, and Alberto Lluch Lafuente. Partial-order reduction for general state exploring algorithms. In *International SPIN Workshop on Model Checking of Software*, pages 271–287. Springer, 2006.

[11] Robert T. Braden. Requirements for internet hosts - communication layers. *RFC*, 1122:1–116, 1989. https://doi.org/10.17487/RFC1122.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[13] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140, San Jose, CA, 2012. USENIX. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini.

[14] Jiamin Cao, Yu Zhou, Ying Liu, Mingwei Xu, and Yongkai Zhou. Turbonet: Faithfully emulating networks with programmable switches. In *28th IEEE International Conference on Network Protocols, ICNP 2020, Madrid, Spain, October 13-16, 2020*, pages 1–11. IEEE, 2020. https://doi.org/10.1109/ICNP49622.2020.9259358.

[15] Jianguo Chen, Hangxia Zhou, and Stefan D. Bruda. Combining model checking and testing for software analysis. In *International Conference on Computer Science and Software Engineering, CSSE 2008, Volume 2: Software Engineering, December 12-14, 2008, Wuhan, China*, pages 206–209. IEEE Computer Society, 2008. https://doi.org/10.1109/CSSE.2008.1025.

[16] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. https://doi.org/10.1007/BFb0025774.

[17] Wesley M. Eddy. Transmission control protocol (TCP). *RFC*, 9293:1–98, 2022. https://doi.org/10.17487/RFC9293.

[18] Jake Edge. Namespaces in operation, part 7: Network namespaces. https://lwn.net/Articles/580893/, January 2014. Accessed: 2023-04-16. URL: https://lwn.net/Articles/580893/.

[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 Katerina J. Argyraki and Rebecca Isaacs, editors, *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 523–535. USENIX Association, 2016. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud.

[20] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *J. ACM*, 33(1):151–178, January 1986. https://doi.org/10.1145/4904.4999.

[21] Will Fantom, Paul Alcock, Ben Simms, Charalampos Rotsos, and Nicholas J. P. Race. A NEAT way to test-driven network management. In *2022 IEEE/IFIP Network Operations and Management Symposium, NOMS 2022, Budapest, Hungary, April 25-29, 2022*, pages 1–5. IEEE, 2022. https://doi.org/10.1109/NOMS54207.2022.9789909.

[22] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, March 2016. USENIX Association. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/fayaz.

[23] Soudeh Ghorbani and Philip Brighten Godfrey. COCONUT: seamless scale-out of network elements. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 32–47. ACM, 2017. https://doi.org/10.1145/3064176.3064201.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA,

2005. Association for Computing Machinery. https://doi.org/10.1145/1065010.1065036.

[25] Patrice Godefroid and Koushik Sen. Combining model checking and testing. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 613–649. Springer, 2018. https://doi.org/10.1007/978-3-319-10575-8_19.

[26] Lin Gui, Jun Sun, Yang Liu, Yuanjie Si, Jin Song Dong, and Xinyu Wang. Combining model checking and testing with an application to reliability prediction and distribution. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 101–111. ACM, 2013. https://doi.org/10.1145/2483760.2483779.

[27] Dong Guo, Jian Luo, Kai Gao, and Y. Richard Yang. Poster: Scaling data plane verification with throughput-optimized atomic predicates. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 1141–1143. ACM, 2023. https://doi.org/10.1145/3603269.3610845.

[28] Gerard J. Holzmann. *The Design and Validation of Computer Protocols*. Prentice-Hall, Inc., USA, January 1991. URL: https://www.semanticscholar.org/paper/5dedd6706d8354f7200ed54193864f2bbc641b05.

[29] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. https://doi.org/10.1109/32.588521.

[30] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, Boston, MA, 2017. USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex.

[31] Van Jacobson. Congestion avoidance and control. In Vinton G. Cerf, editor, *SIGCOMM '88, Proceedings of the ACM Symposium on Communications Architectures and Protocols, Stanford, CA, USA, August 16-18, 1988*, pages 314–329. ACM, 1988. https://doi.org/10.1145/52324.52356.

[32] Karthick Jayaraman, Nikolaj S. Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C. Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 200–213. ACM, 2019. https://doi.org/10.1145/3341302.3342094.

[33] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian.

[34] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[35] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27. USENIX Association, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid.

[36] Maxim Krasnyansky, Maksim Yevmenkin, and Florian Thiel. Universal tun/tap device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt. Accessed: 2023-04-16. URL: https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[37] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In Robert Grossman, Roberto J. Bayardo, and Kristin P. Bennett, editors, *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187. ACM, 2005. https://doi.org/10.1145/1081870.1081893.

[38] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet:

Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 599–613, New York, NY, USA, 2017. ACM. https://doi.org/10.1145/3132747.3132759.

[39] 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. https://doi.org/10.1145/2018436.2018470.

[40] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, USA, 1992. UMI Order No. GAX92-24209.

[41] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), mar 2014. URL: https://dl.acm.org/doi/10.5555/2600239.2600241.

[42] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 699–718. USENIX Association, 2019. URL: https://www.usenix.org/conference/nsdi19/presentation/moon.

[43] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251175.1251187.

[44] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In David E. Culler and Peter Druschel, editors, *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association, 2002. URL: http://www.usenix.org/events/osdi02/tech/musuvathi.html.

[45] Andrii Nakryiko. Bpf co-re (compile once – run everywhere). https://nakryiko.com/posts/bpf-portability-and-co-re/, February 2020. Accessed: 2023-04-16. URL: https://nakryiko.com/posts/bpf-portability-and-co-re/.

[46] 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, 2017. USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths.

[47] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. Computing tcp's retransmission timer. *RFC*, 6298:1–11, 2011. https://doi.org/10.17487/RFC6298.

[48] Doron Peled. *Partial-Order Reduction*, pages 173–190. Springer International Publishing, Cham, 2018. https://doi.org/10.1007/978-3-319-10575-8_6.

[49] Solal Pirelli, Akvile Valentukonyte, Katerina J. Argyraki, and George Candea. Automated verification of network function binaries. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 585–600. USENIX Association, 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/pirelli.

[50] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. https://doi.org/10.1109/SFCS.1977.32.

[51] Jon Postel. Transmission control protocol. *RFC*, 793:1–91, 1981. https://doi.org/10.17487/RFC0793.

[52] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 953–967. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/prabhu.

[53] Jamal Hadi Salim, Hormuzd M. Khosravi, Andi Kleen, and Alexey Kuznetsov. Linux netlink as an IP services protocol. *RFC*, 3549:1–33, 2003. https://doi.org/10.17487/RFC3549.

[54] Ido Schimmel. Add drop monitor for offloaded data paths. *netdev@vger.kernel.org*, August 2019. Accessed: 2023-04-16. URL: https://lwn.net/Articles/796088/.

[55] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *2013 Formal Methods in Computer-Aided Design*. IEEE, October 2013. https://doi.org/10.1109/fmcad.2013.6679403.

[56] Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. Scalable verification of probabilistic networks. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 190–203. ACM, 2019. https://doi.org/10.1145/3314221.3314639.

[57] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring ISP Topologies with Rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004. https://doi.org/10.1109/TNET.2003.822655.

[58] Kotikalapudi Sriram, Doug Montgomery, and Jeffrey Haas. Enhanced feasible-path unicast reverse path forwarding. *RFC*, 8704:1–17, 2020. https://doi.org/10.17487/RFC8704.

[59] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 314–327, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2934872.2934881.

[60] Willem Visser and Howard Barringer. CTL* model checking for SPIN. In *SPIN*, volume 195, pages 32–51, 1999.

[61] Willem Visser and Howard Barringer. Practical CTL* model checking: Should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[62] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*, pages 2170–2183. IEEE, 2005. https://doi.org/10.1109/INFCOM.2005.1498492.

[63] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using Atomic Predicates. In *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, October 7-10, 2013*, pages 1–11. IEEE Computer Society, October 2013. https://doi.org/10.1109/ICNP.2013.6733614.

[64] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. Liveness verification of stateful network functions. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 257–272. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/yousefi.

[65] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. Netsmc: A custom symbolic model checker for stateful network verification. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 181–200. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/yuan.

[66] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. Verifying software network functions with no verification expertise. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 275–290. ACM, 2019. https://doi.org/10.1145/3341301.3359647.

[67] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 221–239. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/zhang-kaiyuan.

[68] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. APKeep: realtime verification for real networks. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 241–255. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/zhang-peng.