Efficient and Correct Test Scheduling for Ensembles of Network Policies

Yifei Yuan, Sanjay Chandrasekaran, Limin Jia, and Vyas Sekar, Carnegie Mellon University

https://www.usenix.org/conference/nsdi18/presentation/yuan

This paper is included in the Proceedings of the
15th USENIX Symposium on Networked
Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-931971-43-0

Open access to the Proceedings of
the 15th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by USENIX.

Efficient and Correct Test Scheduling for Ensembles of Network Policies

Yifei Yuan Sanjay Chandrasekaran Limin Jia Vyas Sekar
Carnegie Mellon University

Abstract

Testing whether network policies are correctly implemented is critical to ensure a network's safety, performance and availability. Network operators need to test ensembles of network policies using a combination of native and third-party tools in practice, as indicated by our survey. Unfortunately, existing approaches for running tests for ensembles of network policies on stateful networks face fundamental challenges with respect to correctness and efficiency. Running all tests sequentially is inefficient, while naïvely running tests in parallel may lead to incorrect testing results. In this paper, we propose Mikado, a principled scheduling framework for scheduling tests generated by various (blackbox) tools for ensembles of policies. We make two key contributions: (1) we develop a formal correctness criteria for running tests for ensembles of policies; and (2) we design a provably correct and efficient test scheduling algorithm, based on detecting read-write test conflicts. Mikado is open source and can support a range of policies and testing tools. We show that Mikado can generate correct schedules in real-world scenarios, achieve orders of magnitude reduction on the test running time, and schedule tests for thousands of network policies in large networks with 1000+ nodes within minutes.

1 Introduction

Network policies, such as reachability (Can A talk to B?) and dynamic service chaining [14], are implemented via complex network configurations. Testing whether these policies are correctly implemented is becoming increasing important to ensure the security, performance and availability of networks [13, 47, 42, 44, 29, 19].

In recent years, a number of testing techniques have been developed which focus on a variety of network policies, such as ATPG [47], BUZZ [13] and Symnet [42]. Each of these tools can efficiently generate test traces for a single network policy. Our survey indicates, however, that operators have a broad spectrum of policies that the network must implement in practice and thus they need efficient techniques for testing *ensembles* of network policies. Testing such ensembles of policies involves incorporating multiple third-party testing tools (e.g., ATPG for reachability, Symnet for network function correctness, BUZZ for service chaining) as they may offer complementary capabilities and tradeoffs. Looking forward, with emerging trends such as intent-based networking [7, 28, 23], we expect that the policies and testing tools will increase both in diversity and in number.[1]

Unfortunately, testing such ensembles raises fundamental conflicts with respect to *efficiency* and *correctness*. Today, operators often run all tests sequentially in the live network, as indicated by our survey. However, this approach cannot scale up to large-size networks that enforce hundreds or even thousands of network policies. On the other hand, running all tests in parallel may produce incorrect testing results due to interference among the tests. For instance, if a firewall enforces a policy *A* based on connection state, then any test of another policy *B* that changes the relevant connection state will induce incorrect test results for *A* when executing the two tests in parallel (See §2 for more examples).

To achieve both correctness and efficiency, we need a principled framework for *scheduling* such ensembles of test cases. In this respect, strawman solutions such as avoiding specific middleboxes or randomizing the parallelization strategy lead to suboptimal and/or incorrect results; and trivial exhaustive search for optimal schedules may take exponential time. Thus, there are two key challenges in realizing such a framework: 1) how to reason about the correctness of a schedule for ensembles of tests generated by a variety of testing tools; and 2) how to

[1] An alternative to testing is to statically verify networks [37] or synthesizing configurations from intents [43]. However, given the dynamic, stateful nature of processing, the large state space of possible behaviors, testing will still be needed even with these advances to: (1) check correctness for scenarios that cannot be statically verified and (2) to diagnose possible disconnects between the models in the verification/synthesis tools and the real network implementations.

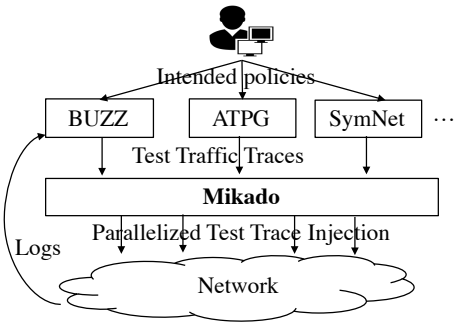


Figure 1: Mikado overview.

efficiently generate optimal schedules for large networks with thousands of nodes and policies.

To this end, we design and implement Mikado, a framework that offers a provably correct and efficient scheduling for tests of ensembles of network policies. As illustrated in Fig. 1, Mikado logically sits between the test generation tools and the network (live or shadow) and is agnostic to the testing tools. Given the test traces for the intended policies, Mikado generates a test schedule that can (near-optimally) reduce the test running time needed to cover those test traces. We note that an alternative design is to generate and schedule test cases in one go. However, given the wide spectrum of network policies, it is unlikely for a single testing tool to support all kinds of policies. Thus, we design Mikado with a blackbox approach to support multiple test generation tools specialized for various policies.

Mikado's design makes two key contributions. First, we develop a formal model for stateful network testing to reason about the correctness of a schedule of test cases (§4). Second, we use our formal model to detect read-write interference among tests and use an efficient graph coloring heuristic on an interference graph to generate provably correct and near-optimal schedules (§5). We also identify opportunities for optimizing the test running time via a combination of random packet header fuzzing and an iterative refinement technique that reduce the likelihood of interference among test traces (§6).

We evaluate Mikado based on both real and synthetic network topologies and policies (§7). We show that: (1) Mikado can generate correct schedules in real-world scenarios; (2) Mikado achieves orders of magnitude reduction on the test running time for thousands of policies on real network topologies; (3) Mikado is scalable to network with 1000+ switches and middlebox and thousands of policies; (4) the proposed extension to reduce interference is both effective and efficient.

2 Motivation

We first describe motivating examples to illustrate the challenges of running tests in stateful networks today. Then we describe our recent survey over network operators, which further confirms these challenges in practice.

2.1 Motivating Examples

Multi-stage IPS: Fig. 2 shows a multi-stage intrusion prevention deployment which consists of a light-weight IPS (IPS1) and a heavy-weight IPS (IPS2). The network operator wants to enforce a set of policies for each department, where traffic from all departments should be sent to IPS1 but traffic from suspicious hosts labeled by IPS1 (e.g., generating 10 bad connections) is sent to IPS2 for payload signature analysis. The topology of the network is at the bottom of Fig. 2 and the policy ensemble is illustrated on top of Fig. 2, which we call policy graph.

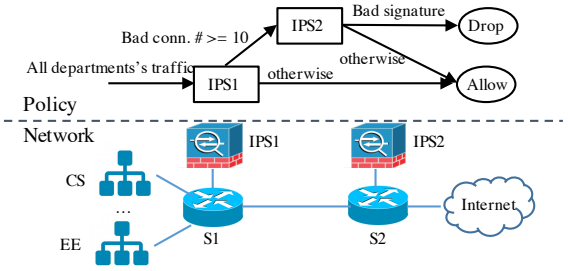


Figure 2: Multi-stage IPS.

To test the intended policies in a live network, a testing tool may generate three test traces for a department, each of which corresponds to a path in the policy graph. For instance, to test the policy ensemble for traffic from the CS department, the tool may generate three test traces from a host H, one containing 6 bad connections and the other two containing 11 bad connections. Further, the second test trace only includes good signatures and the last test trace includes bad signatures. We write $trace_H^6$ to denote the first test trace, $trace_H^{11}$ for the second, and $trace'^{11}_H$ for the third. If the policies are correctly implemented, by injecting the test traces into the network, the network operator would expect to see that packets in the first two traces are successfully forwarded to the Internet and only the latter is directed to IPS2, while $trace'^{11}_H$ is directed to IPS2 and blocked.

Today, operators often need to inject a trace, obtain the results, and then repeat for the next trace. However, such a sequential approach may generate *incorrect* testing results due to the local state maintained by IPS1. For example, injecting $trace_H^6$ after $trace_H^{11}$ will cause $trace_H^6$ to be (mistakenly) directed to IPS2, because of IPS1's stale counter value. An improved sequential testing may wait for a sufficient time T_{to} for the state to expire between each two injections. However, this scheduling can be very inefficient: suppose the time of injecting a trace and waiting for timeout is 30 seconds, running tests for $1,000$ policies would take 8 hours!

Injecting all test traces in parallel can reduce the testing time, however, such scheduling could generate incorrect testing results: e.g., injecting $trace_H^6$ and $trace_H^{11}$ together may cause both traces to be directed to IPS2 as

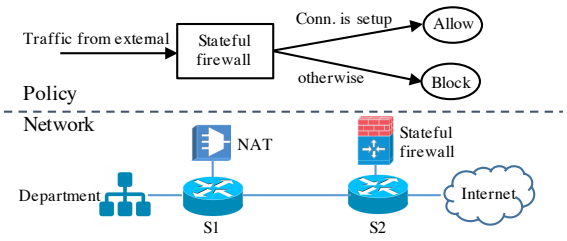


Figure 3: Stateful firewall.

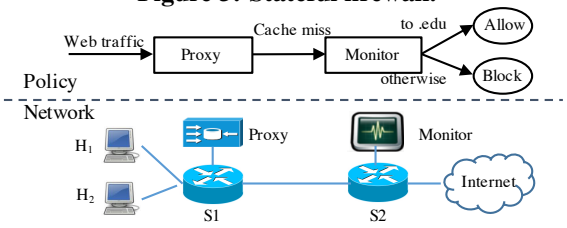


Figure 4: Web monitoring.

the IPS1's counter value reflects the collective connection counts of the traces, not individual ones.

Stateful Firewall: In practice, operators use multiple tools to test different policies. We show that running multiple tools may also lead to incorrect results. Consider the stateful firewall example Fig. 3, where the intended policies are to 1) allow an external IP to access internal network after an internal host is connected to that IP, and 2) block external traffic otherwise.

A tester (e.g., BUZZ [13]) aimed at stateful policies may generate a test trace $trace_{BUZZ}$ that contains a connection from an internal client C_1 to an external server S followed by packets from S to C_1. Meanwhile, a network operator may also use tools like ATPG [47] to test basic reachability. ATPG may generate a test traffic trace $trace_{ATPG}$ from S to a client C_2. However, running the two traces in parallel may produce incorrect testing results: if $trace_{ATPG}$ reaches the firewall after the connection has been set up between C_1 and S by $trace_{BUZZ}$, $trace_{ATPG}$ would go through the firewall while it should have been blocked if it was injected alone. This example demonstrates that simple composition of multiple testing tools may not be safe even if one (ATPG in this example) is aimed at stateless policies such as basic reachability.

Traffic monitoring: Running multiple tests not only raises false alarms, but can also mask configuration bugs in the network. In Fig. 4, a proxy is used to improve the web performance, and a monitoring system is configured to monitor web traffic. The intended policy is to allow `.edu` web traffic for all internal hosts. However, switch S_2 is mistakenly configured to block traffic from H_2.

To test the intended policy, one may run tests that request `xyz.edu` from all hosts. Running the tests in arbitrary order may not uncover the misconfiguration on S_2. For example, consider a test trace containing requests to `xyz.edu` from host H_1, and a second trace containing requests to `xyz.edu` from host H_2. The second trace can uncover the misconfiguration when injected alone to the network, as the request will be blocked by S_2. However, when injected together with the first trace (or immediately after it), the request from H_2 may get a response from the proxy, which caches the content from the first test trace, and the bug is not revealed.

# Policies	%
< 10	19.23%
10-100	57.69%
100-1000	11.54%
$>$1k	11.54%

# Tools	%
< 5	35%
5-10	50%
$>$10	15%

Table 1: Number of policies and testing tools.

Summary: We observe key correctness and efficiency challenges in testing an ensemble of policies, and natural strawman solutions face one or both of the challenges. For example, simple isolation heuristics, such as avoiding running tests that access the same middleboxes, will run tests for EE and CS separately for the example in Fig. 2. However, an optimal scheduling can safely run tests from EE and CS together even if they all access IPS1. Exhaustively searching for optimal schedules is not applicable as it takes exponential time and thus incurs prohibitive overhead to the testing workflow.

2.2 Survey on Network Testing

To understand the reality of the aforementioned challenges in network testing today, we conducted a survey in Sept. 2017 among subscribers to the North American Network Operators Group. Among all 30 respondents, 4 manage small networks ($<$ 1k hosts), 6 medium networks (1k-10k hosts), 11 large networks (10k-100k hosts), and 9 very large networks ($>$ 100k hosts). Questions and responses can be found in the link [3]. Here we highlight a few key observations.

Sequential live testing is the dominant testing methodology: When asked the ways of running network tests, 72.97% report running testing on live networks, which is significantly higher than other methodologies (24.32% for emulated network based testing and 2.7% for others). Among those who responded live testing, 86.21% run tests in a sequential way (i.e. run each test case one by one).

Ensembles of policies need to be tested: Table 1 shows that network operators need to test varied number of network policies. While the majority (57.68%) reports policy numbers ranging from 10-100, a significant portion (accounts for 12%) reports the number to be several thousands. A large variety of policies are also reported, ranging from reachability, service chaining, access control, routing, latency/throughput among others.

Multiple tools are used: When asked the number of tools used in testing, all responses report at least 2 different tools. As shown on the right of Table 1, 50% re-

Concerns	%
Test cases may not match the policy intent	30.23%
Testing result maybe incorrect	27.91%
Testing traffic may conflict	25.58%
Testing is slow	16.28%

Table 2: Concerns when running tests

spondents use 5-10 tools for network testing, while some use 10+ tools. While ping, traceroute, iperf are still popular testing tools, we also see responses on using expert-crafted scripts, third-party tools, and custom tools.

Top concerns: Given the large variety and number of policies and testing tools, we hypothesize that correctness of testing results and test conflicts may be one of the concerns for running tests. This is confirmed by our survey. Table 2 lists the concerns network operators report for running network tests under a multi-choice question. A large number (53%) of responses report correctness of testing results and test conflicts as their concerns.

3 Overview

Our goal is to schedule an ensemble of test traces efficient while guaranteeing the correctness of the tests. In other words, we seek to inject as many test traces as possible in parallel such that these traces do not interfere with each other. To this end, we need: (1) a systematic way to reason about the potential for interference of test cases and (2) efficient techniques to identify and schedule non-interfering test cases.

Figure 5 depicts the workflow of our tool Mikado, zoomed in view of Figure 1. The input to Mikado is a set of test traffic traces that are generated from *any* testing tools (e.g., Symnet, ATPG, Buzz), and Mikado outputs an efficient and correct test scheduling plan. To realize the requirements discussed above, Mikado consists of the following components (the dashed are extensions).

Interference Checker: To reason about the potential for interference, we develop a formal model to capture the behavior of the stateful networks (§4). Using this model, Mikado first checks the potential source of interference among the test traces. This offers a provable guarantee that scheduling non-interfering test traces in parallel preserves the testing results as if each test trace was injected on a separate or isolated network.

Trace Scheduler: Base on the pairwise interference relations, Mikado builds an interference graph, where each node in the graph corresponds to a test trace, and an edge connects two interfering traces. Mikado then uses a graph coloring algorithm to generate an optimal scheduling of the test traces by assigning each node in the interference graph a color such that two end nodes of an edge are assigned different colors.

Given a colored interference graph, Mikado runs all tests in k rounds, where k is the number of colors on the interference graph. In the i-th round, Mikado injects

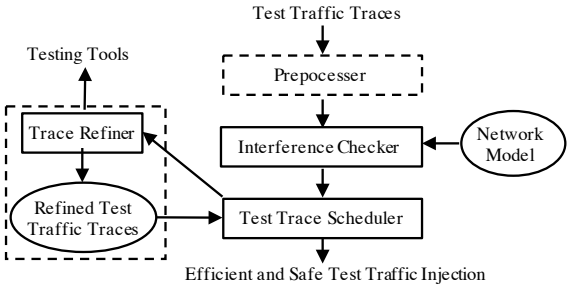


Figure 5: Mikado architecture.

all test traces with the i-th color and reports the executing results to the testing tools. Between two consecutive rounds, Mikado waits for a sufficient amount of time in order for all state to reset.

Preprocessor and Trace Refiner: Optionally, Mikado can further improve the scheduling by random packet fuzzing and test traces refinement.

Before checking interference, Mikado can run heuristic-based preprocessing to reduce the chance of interference among test traces. This process rewrites the test traces using randomly selected values for given fields according to the policy being tested.

After obtaining a correct schedule, Mikado can further refine the test traces to reduce the number of edges in the interference graph, and thus reduce the number of colors to color the refined interference graph. In particular, given a trace that is interfering with a set of traces, Mikado reruns the testing tool for that trace to generate another test trace which is not interfering with the set of traces. For this purpose, Mikado automatically generates a configuration file for the testing tool, without modifying the internal logic of the testing tool.

Illustrative Example: We use the multi-stage IPS example from §2 to illustrate the end-to-end workflow. For brevity, we do not discuss the preprocessing here and only consider the testing of two policies for each department. Suppose $trace_H^6$ and $trace_H^{11}$ are the two traces generated for the CS department and $trace_{H'}^6$ and $trace_{H'}^{11}$ are the two traces for the EE department. First, the interference checker automatically detects the interferences, and builds the interference graph as shown in the left subfigure in Figure 6, where circular nodes correspond to $trace_H^6$ and $trace_H^{11}$, and rectangular nodes correspond to test traces for the EE department. As discussed in §2, $trace_H^6$ and $trace_H^{11}$ cannot be injected together, and thus there is an edge in the interference graph between them. Second, the trace scheduler colors each node with the goal to minimize the number of colors. An example coloring is shown in the middle subfigure. Based on this colored interference graph, Mikado can safely inject blue traces (i.e., $trace_H^6$ and $trace_{H'}^6$) in the first round, and then inject the red traces (i.e., $trace_H^{11}$ and $trace_{H'}^{11}$) in the second round. Additionally, the trace refiner can be used to refine a test trace to further reduce the interference.

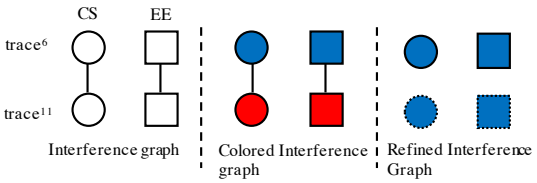


Figure 6: Example run of Mikado on multi-stage IPS.

For instance, to reduce the interference between the two circular nodes, the refiner generates a configuration file that asks the testing tool to generate another test trace that is not from H to refine the trace $trace_H^{11}$. If the testing tool succeeds in doing so, the old node will be replaced with the newly generated trace in the interference graph, and edges from the old nodes are removed. The right subfigure illustrates the final results, where the nodes with dashed line are the refined traces. After the refinement, all test traces can be safely injected in a single round.

4 Problem Formalization

Before we design any tool for scheduling, first we need a formal basis for reasoning about the correctness of a schedule. To this end, we develop a formal notion of correctness and define the test scheduling problem.

4.1 Stateful Network Model

We extend prior work on formal network modeling used in defining stateless networks (c.f. [22, 40]) to model the stateful behaviors we encountered in §2. This model provides a necessary building block for our test correctness definition and proofs for our scheduling algorithm. Our model may also be used in providing semantics to other network testing and verification problems. Note that this model abstracts concrete network functions and thus subsumes models in BUZZ [13] and Symnet [42]. We currently do not model multicast and time.

Syntax. The syntactic constructs used for our model are summarized in Figure 7. We write *pkt* to denote network packets, which are bitstrings. We model locations in the network as ports, denoted p. We use natural numbers n to represent concrete network locations. A special port *exit* models all locations outside the network of concern. To model dropped packets we include a special port *drop*. A located packet, denoted lp, is a pair of a port and a packet. (p, pkt) means that the packet pkt is at port p waiting to be processed. Packets at port *exit* are the ones that have left the network. All dropped packets are at port *drop*. Multiple packets can co-locate at one port. A packet queue, denoted Q, maps each port p to a list of packets located at p.

Each state in a stateful network (e.g., connection state remembered by a firewall) is labelled with a unique state location, denoted s. We assume s is drawn from a finite set of location symbols Σ. A state map, denoted

Packet	pkt	::=	$[b_1 \dots b_l]$
Port	p	::=	$n \mid exit \mid drop$
Located Packet	lp	::=	(p, pkt)
Port Queue	Q	$\in$	$Port \to Packet\ list$
State. Loc.	s	$\in$	Σ
State Map	M	$\in$	$StateLoc \to Value$
Network Env.	E	::=	(Q, M)
Topo. Func.	ρ	$\in$	$Port \to Port$
Trans. Func.	δ	$\in$	$StateMap \times LocatedPkt \to StateMap \times LocatedPkt$
Configuration	C	::=	(ρ, δ)
Network State	S	::=	(E, C)
Network run	r	::=	$E_1 \xrightarrow{lp/lp'}_C \cdots E_k$
Observation	o	::=	$[lp_1/lp_1; \cdots; lp_n/lp'_n]$

Figure 7: Stateful network model: syntax

M, maps each state location to a value, drawn from a finite set of values *Value*. Intuitively, the state locations represent the variables and data structures that are internally maintained by stateful middleboxes and SDN controllers. We write E to denote network environments, which are pairs of the packet queue and the state map. We write M_{init} to denote the initial state map. M_{init} maps each state location to its pre-defined initial value. The initial port queue, Q_{init}, maps each port to an empty list. We call (Q_{init}, M_{init}) an *initial environment*. A *terminating environment* is one where the port queue is empty for all port p s.t. $p \notin \{exit, drop\}$.

The topology of the network is determined by connections between ports. We use a function ρ to map a port p to another port p', where packets at p are sent to. The process of transforming and forwarding packets are modeled as transfer functions, denoted δ. A transfer function takes as arguments a pair of a state map and a located packet and returns a pair of a new state map and a new located packet. A configuration of the network is a pair (ρ, δ), which include the topology and transfer function of the network. A network state S is a pair of the environment E the configuration C.

Operational semantics. As packets are processed and traverse the network, network states change and affect the processing of future packets. We define small-step operational semantics to model the transitions of network states. We write $E \xrightarrow{lp/lp'}_C E'$ to denote the transition from E to E' using configuration C. Here, lp and lp' above the arrow denote the located packet processed at the transition and the resulting located packet respectively. We assume that the configuration C does not change during the testing of policies in the network.

The only transition rule NET-TRANS (shown below) states that if the first packet at port p is pkt, the transition function changes the located packet (p, pkt) to (p', pkt') and the state map from M to M', and p' is connect to p'', then after pkt at p is processed, the state map is M', and the port queue is updated to reflect that pkt is no longer

at p and pkt' is added to port p''.

NET-TRANS

$$\frac{\begin{array}{c} pkt = head(Q(p)) \\ \delta(M,(p,pkt)) = (M',(p',pkt')) \qquad \rho(p') = p'' \\ Q' = Q[p'' \mapsto (Q(p'')++[pkt'])][p \mapsto tail(Q(p))] \end{array}}{(Q,M) \xrightarrow{(p,pkt)/(p',pkt')}_{(\rho,\delta)} (Q',M')}$$

We call a sequence of network transitions a *run*, denoted r. All transitions in a run r from the state (E,C) use C as the configuration. A *complete run* is a run, whose last environment is a terminating one. Given a network state (E,C), we write $\mathscr{R}(E,C)$ to denote the set of complete runs starting from (E,C). We sometimes omit the configuration C from runs as it never changes during the testing and is often clear from the context.

4.2 Correctness of Network Tests

We formalize the correctness criteria for network testing and define the test scheduling problem.

Network testing via packet trace. When testing policies, a network operator first generates a sequence of located packets (using existing test generation tools), then injects the packets into the network, and finally makes a judgement of whether the policy is violated based on observations derived from a complete run of the network.

We call the sequence of located packets $(lp_1,\ldots,lp_m)$ generated by the testing generation tools a *test trace*, denoted t. Injecting a trace t into a network state with port queue Q results in a new port queue where each located packet in t is enqueued at the appropriate port. A test always runs from an initial environment, so we write $queue(t)$ to denote the port queue resulted from injecting t to Q_{init}. Executing the test trace corresponds to generating a complete run from the environment $(queue(t),M_{init})$. One subtlety is that the order in which packets in t are processed matters (e.g., reply to a request can only happen after the request). Our model takes care of this by defining valid runs that comply with the injection order requirement. Given a run r in $\mathscr{R}(queue(t),M_{init})$, we say r is a valid test run w.r.t. t and an ordering requirement $\varphi(t)$, r satisfies $\varphi(t)$. Different test traces have different ordering requirements based on protocol conventions. We assume that such requirements are given as input; our approach and algorithm is agnostic to the specific type of ordering requirement.

We define the observation of a run, written $\mathscr{O}(r)$, as the sequence of pairs of located packets. Given a run r where $r = S_0 \xrightarrow{lp_{i1}/lp_{o1}}_C S_1 \cdots \xrightarrow{lp_{ik}/lp_{ok}}_C S_k$, the observation $\mathscr{O}(r)$ is simply $[lp_{i1}/lp_{o1};\cdots;lp_{ik}/lp_{ok}]$. This corresponds to the trace routes that the network operator sees.

Correct network test. Given a set of test traces $\{t_1,\ldots,t_k\}$ for an ensemble of policies $(pol_1,\ldots,pol_k)$, we have shown in §2 that test traces may interfere with each other if they are allowed to execute concurrently. We provide a strong *correctness* definition of running multiple tests based on a tester's observation of each test.

Intuitively, if it is correct to concurrently run a set of test traces, the observation of each test trace should be the same as the test trace running alone. For the IPS example in §2, the observation of injecting $trace^6_H$ and $trace^{11}_H$ together for $trace^6_H$ may be different from that of injecting $trace^6_H$ alone. In the former case, packets in $trace^6_H$ may be sent to IPS2, while in the latter case all the packets in the trace land in the *exit* port.

To formalize this intuition, we define a projection of a run given a packet trace, written $r\downarrow_t$, as the observation containing only the located packets that are related to t, that is, either it is in t, or it is (transitively) transformed from a located packet in t. The projection of a run r from $(queue(t),M_{init})$ given t is $\mathscr{O}(r)$. Then the correctness definition can be formalized below:

Definition 1 (Correctness) *Given a set $T = \{t_1,\cdots,t_n\}$ of test traces, we say that it is correct to schedule T (T is correct for short) if $\forall r \in \mathscr{R}(queue(\cup_{i=1}^n t_i),M_{init})$ s.t. r is valid w.r.t. each $\varphi_j(t_j)$ where $j \in [1,n]$, $\forall i \in [1,n]$, $\forall r' \in \mathscr{R}(queue(t_i),M_{init})$, $r\downarrow_{t_i} = \mathscr{O}(r')$.*

Test scheduling problem. Given an ensemble of test traces, a natural scheduling is to run all tests in sequential rounds. In each round, we want to run as many tests in parallel as possible, in order to reduce the total runtime of testing. Between two rounds, we can wait sufficiently long for the network state to reset so the execution of each round starts from a clean (the initial) state.

Definition 2 (Correct scheduling) *Given a set of test traces T generated for ensembles of policies by a set of testing tools, a* correct schedule *of T is a partition $\{T_1,..,T_k\}$ of T, such that running each T_i is correct.*

We assume that a round of testing takes roughly the same time which is much less than the waiting time between rounds. Therefore, our goal is to correctly minimize the number of rounds and the *test traces scheduling problem* is defined as follows. That is, given a set of test traces T, the *test trace scheduling problem* seeks a correct schedule with minimal number of rounds.

5 Test Traffic Scheduling

A naïve solution to the test trace scheduling problem is enumerating all possible partitions and then checking whether each set of traces in the partition is correct using Definition 1. However, this is extremely inefficient. First, the number of partitions is exponential to the number of test traces in the set. Second, checking whether a set of test traces is correct directly by comparing projections of every possible run of the composed test traces is also prohibitively expensive. For instance, there are

$\frac{(2m)!}{m!m!}$ possible complete runs of a one-port network and two test traces of size m; and $n!$ runs for a one-port network and n traces of size 1.

Next, we outline our key insights (§5.1) and then present our algorithm and discuss the algorithm's time complexity and correctness guarantees (§5.2).

5.1 Key Insights

Even though checking correctness directly may be complex, we can solve an approximation of the problem much more efficiently. For our three examples from §2, it is not too hard to see why test traces may interfere with each other. The main cause of the problem is that concurrently executing test traces do not preserve the values of each other's state locations that determine their behavior. If we are able to identify the conditions under which test traces may interfere with each other, we can solve the scheduling problem by selecting test traces that do not interfere with each other to be scheduled together.

First, we observe that by examining a run of a test trace, we can identify the fragment of the state map that is key to decide how packets are processed by the network. (Recall from §4.1, a state map M maps each state in the network to a value.) For instance, the value of the bad connection counter of IPS1 (denoted s_H) decides how packets in $trace^6_H$ are processed. Changing the value of s_H changes the observation of the test trace and, more importantly, changing the value of other state locations (e.g., the bad connection counter for host H_1) in the state map will not affect how packets in $trace^6_H$ are processed.

A natural corollary of this observation is that given two test traces, we can determine whether they interfere with each other by examining the state maps that decide the behavior of each trace from independent runs of the test trace. For the IPS example, let $trace^6_{H1}$ be the test trace containing 6 bad connections from host *H1* and $trace^{11}_{H2}$ be the test trace containing 11 bad connections from host *H2*. They modify and depend on two different state locations in IPS1 and thus, don't interfere with each other.

Finally, we observe that instead of considering arbitrary partitions, we can construct a correct schedule from pair-wise noninterfering test traces as none of the test traces interfere with each others' key state maps. Those state maps remain the same as an independent run of a test trace, and therefore, result in the same observation.

Given these observations, we can encode the correct scheduling problem as a graph coloring problem where the nodes correspond to test traces and the edges connect interfering test traces (the lack of edges between two nodes implies compatibility between the two traces). Then, we can use an efficient graph coloring algorithm to generate a near-optimal coloring of the graph, which corresponds to a near-optimal correct schedule: nodes of the same color can be scheduled together.

Algorithm 1 Correct scheduling

```
1:  function GEN_INTF_GRAPH(T, C, M_init)
2:      G ← {}
3:      (ρ, δ) ← C
4:      for each trace t_i ∈ T do
5:          create a node v_{t_i} for t_i in graph G
6:          r_i ← a run from (Q^{t_i}, M_init) under C
7:      for each pair of traces t_1, t_2 ∈ T do
8:          (S^1_0 --lp^1_0/..--> ··· --lp^1_k/..--> S^1_{k+1}) ← r_1
9:          (S^2_0 --lp^2_0/..--> ··· --lp^2_l/..--> S^2_{l+1}) ← r_2
10:         let M^a_b be the state map for S^a_b
11:         if there exist i ∈ [1,...,k+1], j ∈ [0,...,l]
        and s, such that s ∈ dtMap(δ, M^2_j, lp^2_j), M^1_i(s) ≠
        M^2_j(s), and M^1_{i-1}(s) ≠ M^1_i(s) then
12:             G ← G ∪ (v_{t_1}, v_{t_2})
13:     return G
14:
15: function SCHEDULE(T, C, M_init)
16:     G ← GEN_INTF_GRAPH(T, C, M_init)
17:     G_c ← GRAPH_COLORING(G)
18:     for each color i in G_c do
19:         T_i ← nodes in G_c of color i
20:     return (T_1, ··· , T_k)
```

5.2 Algorithm

The main function of our algorithm (Alg. 1) is SCHEDULE (lines 15-20), which takes a set of test traces T, a network configuration C, and the initial state map M as input and returns a partition of T. The SCHEDULE function calls GEN_INTF_GRAPH to generate the interference graph of T and GRAPH_COLORING to color the graph. The latter can be any efficient coloring algorithm, which we omit. The output partition corresponds to a schedule of the test traces (line 20).

The GEN_INTF_GRAPH function creates a node in the graph for all traces in T (line 5). The edges are supposed to connect two nodes representing test traces that interfere with each other. The key is to decide whether two test traces interfere (not compatible) with each other, which relies on the following two functions: $dtMap(\delta, M, lp)$ and $upd(\delta, M, lp)$. At a highlevel, $dtMap(\delta, M, lp)$ returns a state map M' containing a subset of the mappings in M that determines the result of $\delta(M, lp)$. $upd(\delta, M, lp)$ returns a state map M' that maps the subset of state locations in the domain M that are updated by the transition $\delta(M, lp)$ to new values.

For each concrete transition function δ, it is straightforward to define $dtMap(\delta, M, lp)$. Using the stateful firewall example from §2, the state map remembering whether there exists a prior connection from a host within

the department should be returned by this function. We require that all instances of $dtMap(\delta, M, lp)$ return the *determinant state map* of the transition from M and lp under δ, formally defined below. Intuitively, state locations that are in the determinant state map uniquely determines the result of $\delta(M, lp)$; state locations that are not in the determinant state map are allowed to be altered by other test traces without changing the behavior of the current test trace. Use the stateful firewall example again, other state locations such as the state of TCP connections are not in the determinant state map and can be altered by other test traces while keeping the observation of the current test trace the same.

We write $diff(M_1, M_2)$ to denote the set of state locations that are in the domain of M_1 and M_2 and mapped to different values by M_1 and M_2. The determinant state map of the transition from M and lp under δ is M_d if we construct another state map M' by changing the values that state locations in M but not in M_d are mapped to, the transition from M' and lp generates the same located packet ($lp_1 = lp_2$), and the resulting state maps M_1 and M_2 only differ at state locations that are not updated by the transition and not in the determinant state map. The last condition essentially forces updates to the state maps to be determined by M_d as well.

Definition 3 (Determinant state map) *We say a state map M_d determines (is the determinant state map of) the transition from M and lp under δ if $M = (M_d, M_n)$ and for all M'_n s.t. $\mathsf{dom}(M_n) = \mathsf{dom}(M'_n)$, let $M' = (M_d, M'_n)$, $(M_1, lp_1) = \delta(M, lp)$, $(M_2, lp_2) = \delta(M', lp)$, it is the case that $lp_1 = lp_2$ and $diff(M_1, M_2) \subseteq \mathsf{dom}(M') \setminus \mathsf{dom}(upd(\delta, M, lp) \cup upd(\delta, M', lp))$.*

To build the interference graph, the algorithm executes each test trace t_i independently and stores the run (lines 4-6). Only one run is needed and we pick the one where a located packet in t_i is only processed when the located packet before it has left the network. We call such a run a *sequential* run, written $R_{seq}(queue(t_i), M)$. The if condition on line 11 checks whether a run r_1 contains a state location s that determines the execution and run r_2 writes to s with a different value from the one that determines the transition in r_1. If so, then r_1 and r_2 interfere with each other. This is because the modification by r_2 could change the behavior of r_1. This is a conservative check.

Let us revisit the example shown in Figure 6. We explain how our algorithm detects the interference between $trace^6_H$ and $trace^{11}_H$. The middlebox IPS1 maintains a state of the number of bad connections for each host. Consider both traces, the determinant state map $dtMap(\delta, M, lp)$ and $upd(\delta, M, lp)$ are both the bad connection counter for H. We write s_H to denote this state location. For the run of $trace^6_H$, the determinant state maps are $s_H \mapsto 0$, to $s_H \mapsto 6$. The update state maps for the run of $trace^{11}_H$ are $s_H \mapsto 0$, to $s_H \mapsto 11$. Obviously, the updates by the second trace don't always agree with the determinant state maps of the first trace. Condition on line 11 of Algorithm 1 is true. Therefore, the two traces are interfering with each other.

We prove that Algorithm 1 is correct (Theorem 1), and detailed proofs can be found in Appendix.

Theorem 1 (Correctness)
SCHEDULE(T, C, M_{init}) *is a correct schedule.*

6 Extensions to Basic Algorithm

So far, we have discussed how to schedule the test given a set of test traces for ensembles of policies. As we see in previous examples, testing tools may generate interfering test traces for ensembles of policies. However, it is possible that the cause is not that the policies conflict, but the optimization heuristics that the testing tools employ to improve the efficiency of test trace generation. For instance, BUZZ attempts to pick concrete field values (same values across policies) for test traces to reduce the number of symbolic variables. In this section, we investigate the opportunities to guide test trace generation to further improve scheduling efficiency.

6.1 High-level Idea

Recall the multi-stage IPS example in §2. The three test traces ($trace^6_H$, $trace^{11}_H$, and $trace'^{11}_H$ for each policy path) interfere since they share the same counter on IPS1 for the source H. Intuitively, by altering the source of each test trace, we can obtain test traces for each policy path while avoiding the interference. For example, $trace^6_{H1}$, $trace^{11}_{H2}$, and $trace'^{11}_{H3}$ are three test traces for each policy path and can be safely injected in parallel.

To generalize this intuition, we identify a set of *influencing fields* for a located packet lp (in a test trace t_1) given a test trace t_2 (denoted $I_{lp}(t_2)$). The property of $I_{lp}(t_2)$ is that by altering values for some fields in it, it could be possible to avoid the interference between t_1 and t_2 related to lp. For instance, the set of influencing fields for packets in $trace^6_H$ includes the source, since changing the sources in $trace^6_H$ from H to $H1$, interference with other test traces are avoided.

The precise fields in $I_{lp}(t)$ may require involved static analysis of the model and the test trace. Here, we propose two heuristics for identifying $I_{lp}(t)$ and new field values: one is to fix the set to commonly used fields such as the 5-tuples and randomly select the value of them; the other is to leverage results learned from the interference analysis (Alg. 1). We discuss each in detail next.

6.2 Random Packet Fuzzing

For flow-based policy testing (e.g., flow-based service chaining [39, 13]), Mikado employs a light-weight preprocessing for the test traces to reduce the chance of

interferences. This preprocessing picks random values from the flow space specified in the policy and rewrites the 5-tuple of each packet in the test trace using these random values. In addition, to preserve the flow semantics, this preprocessing ensures that fields with the same original values will be substituted with the same values.

This heuristic does not ensure the coherence of the test trace and the policy to be tested in general, as it may pick wrong fields and field values. However, as we will see in §7, this heuristic works reasonably well in practice for the targeted policies.

6.3 Test Trace Refinement

Mikado also employs a more rigorous analysis to *refine* test traces using information obtained from the initial scheduling process. The refinement process uses concrete runs of test traces to guide the selection of $I_{lp}(t)$ and the new values for fields in $I_{lp}(t)$. This process aims to tell the test generation tool not to use values that are known to cause interference from previous analysis.

In the rest of this section, we first describe how to refine t to be non-interfering with all traces in T, followed by the algorithm that refines given correct schedules.

Test Generation with Configurations. Given a testing tool A, let $GenTest_A(pol, N, Config)$ denote the test traces generated by A for policy pol and network N under the tool configuration $Config$. Different tools can be tuned differently. One can treat $Config$ as constraints on packet header fields for the test traces to be generated. Given the sequential run r_1 and r_2 obtained from the scheduling process for test trace t_1 and t_2 respectively, we first apply an analysis to identify $I_{lp}(t_2)$ for all $lp \in t_1$, which are essentially the fields in lp that are used as meaningful input in the transition function δ. For instance, the source address in the multi-stage IPS example. This is more precise than fixing a set of fields a priori.

Next, given the set of influencing fields $I_{lp}(t_2)$ for all $lp \in t_1$, we generate the following constraint $CS(t_1,t_2)$: $\bigwedge_{lp_i \in t_1} \bigvee_{f \in I_{lp_i}(t_2)} lp_i'(f) \neq lp_i(f)$. That is, every packet lp_i' in the refined trace by A should not have values exactly the same for fields in $I_{lp_i}(t_2)$. This constraint will be translated to suitable configuration files that the test generation tool understands (see §7). Similarly, to refine a trace t_1 for a set of traces T, the configuration is the constraint $CS(t_1,T) = \bigwedge_{t_2 \in T} CS(t_1,t_2)$. For instance, for the multi-stage IPS example, if we refine $trace_H^{11}$, the constraint is $\bigwedge_i src_i \neq H$, where src_i is the source address of the i-th packet in the refined trace.

With the above settings, we use Algorithm 2 to refine a trace t for a policy P, such that the refined trace is non-interfering with all traces in T. In the algorithm, we try to refine t for at most max_num_try times. In each try, we invoke $GenTest_A$ to generate a possible trace t'. We return t' if it is non-interfering with all traces in T, otherwise, we invoke $GenTest_A$ again for another trace that is not t'.

Algorithm 2 refineTrace(t, T)

```
1: CS = CS(t, T))
2: for i = 1 to max_num_try do
3:     if GenTest_A(P, N, CS) generates a new trace t' then
4:         if t' is non-interfering with T then return t'
5:         else CS = CS ∧ {t'' ≠ t'} // t'' is the next generated trace
6:     else return FAIL
7: return FAIL
```

Refinement Algorithm. Given a correct schedule for a set of policies $\{T_1, .., T_k\}$ as input, our refinement algorithm outputs a refined correct schedule for the policies with potentially a smaller number of injection rounds. At a highlevel, our refinement algorithm takes a greedy heuristic and iterates all rounds T_i, starting from the round with least number of traces. Then it attempts to refine all traces in T_i using the algorithm described above. If all traces in T_i can be refined, we remove T_i and obtain a correct schedule with fewer injection rounds. We omit the details of the algorithm in interest of brevity.

7 Evaluation

We evaluate Mikado via a testbed-based emulation and large-scale simulations and show that Mikado: (1) is able to detect test interferences in a range of scenarios and generate *correct* schedule; (2) achieves orders of magnitude reduction in the test running time compared with alternative test scheduling mechanisms; (3) is scalable to networks with 1000+ switches/middleboxes and thousands of policies.

Implementation: We implement a prototype of Mikado in Python. We consider four testing tools in our framework, namely ATPG [47], BUZZ [13], Pingmesh [19], and Symnet [42]. For ATPG and BUZZ, we reuse their constructs for routing tables and middleboxes; and for Symnet, we manually encoded the middleboxes that are not in the code repository in their language SEFL. To support the refinement extension, we also implement a light-weight helper function (≈100 LoC each tool) that translates the generated constraints to each tool's configuration (e.g., Z3 [9] formulas for Symnet). All experiments are conducted on a server with 20 cores (2.8Ghz) and 128GB RAM.

7.1 Validation

End-to-end correctness. We first validate Mikado's correctness in a variety of use cases. On our testbed, we emulate hosts and (software) middleboxes as separate KVM-based virtual NFs, connected with OpenVSwitch.

We use OpenDayLight [33] as the controller for all emulated networks.

• To emulate the multi-IPS scenario in Fig. 2, we use Snort as both IPS1 and IPS2, and use three hosts to emulate the departments and the Internet. We configure IPS1 to enforce two policies: 1) forwarding a host's traffic to IPS2 if the host issues more than 5 connections in a minute; 2) otherwise the traffic is sent to the Internet. We run BUZZ to test the two policies in parallel, and log the traffic at the interface of IPS2 to check the testing results. We run the experiment 100 runs, and BUZZ reports violation of policy 1) in 80 runs. All reported policy violations are validated to be false positives: the root cause is that BUZZ injects traffic from a single host for two policies, and causes conflicts.

• For the scenario in Fig. 3, we use iptables as both the NAT and the stateful firewall, and use Symnet and ATPG to test the connectivity between the Internet and the department in parallel. In all 100 runs, the security policy on the stateful firewall was reported violated. However, we verify that the testing results are false positives: test cases interfere on the stateful firewall.

• We use a simple "blue team-red team" test to simulate the scenario in Fig. 4. We use Squid as the proxy and Snort as the monitor device to emulate the network. To bind proxy's requests with the origin hosts, we use FlowTag [14] to configure the network. One student ("red") intentionally misconfigures the switch to drop web requests from H2, and the other student ("blue") use BUZZ to test HTTP connectivity of each host. Of 10 times of this experiment, the blue team could only uncover the bug in 5 times.

In all three scenarios, Mikado can successfully detect the interference among the generated tests, and correctly schedule the tests in separate runs. We repeat the three experiments with the schedule from Mikado, and no false positives or negatives are produced.

Interference detection. Next, we evaluate Mikado's capability to detect test interference in practice. We consider two popular types of network policies in our survey: reachability and service chaining policies. To collect real-world service chaining policies, we conduct a survey from a set of industrial and academic sources [38, 14, 13, 44, 20, 1, 2, 27, 30], and build a library of 38 service chaining policy templates. The number of network functions on each service chain ranges from 1 to 5, and the library involves 14 types of network functions in total. Our library is a superset of what has been considered in the prior work in this space.

Table 3 summarizes the key metrics for five different topologies we consider. For each network, we assign hosts into a number of policy groups and enforce 1000 service chaining policies using our library of service chaining templates for randomly selected pairs of policy groups. We use BUZZ and Symnet to generate test cases for each policy (500 policies for each tool). For reachability policies, we consider basic reachability policies using ATPG[2] and TCP reachabilities inspired by Pingmesh [19]. Figure 8 shows the number of test interferences with the number of policies under test.

	# Switches	# Middleboxes	# Links
Internet2	9	9	37
Stanford	16	16	37
Sprint	11	10	28
Oxford	20	20	46
Chinanet	42	39	105

Table 3: Summary of the dataset.

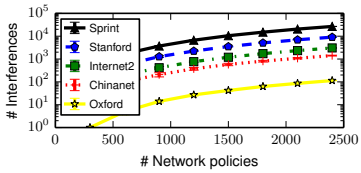


Figure 8: Number of potential interferences.

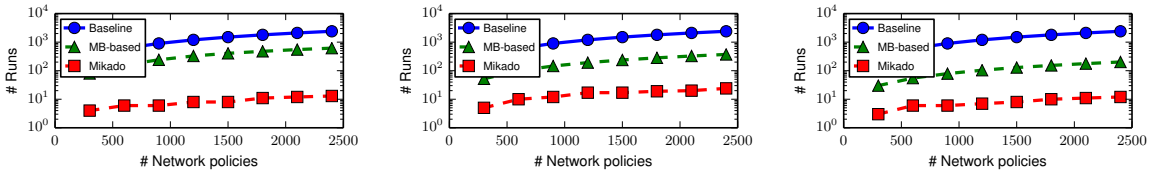
As expected, as the number of policies increases, Mikado detects an increasing number of interferences among policy tests. Further analysis confirms that interferences are caused mostly by multiple tools. For example, when testing all service chaining and TCP reachability policies on Internet2, 3504 interferences are detected and 77% of them happen across different tools. The results further confirm that simply randomizing the parallelization is not likely to generate correct schedules.

7.2 Test Time Reduction

We evaluate the test time reduction using Mikado's scheduling based on the same setup as above. Figure 9 shows the (average) number of runs to complete the tests for different number of policies under test. Recall that all test traffic in each run can be injected in parallel, while multiple runs have to be conducted in sequential. Therefore, the number of runs serves as a reasonable proxy for the test running time. For comparison, we consider two alternative approaches: BASELINE is the basic scheduling approach that runs tests sequentially; MB-based is the heuristic which schedules test traces that do not traverse the same middleboxes together; and Mikado is the proposed scheduling approach (we build a checker that validates the correctness of the schedule). Each data point on the figure is obtained by repeating 100 times.

First, we observe that Mikado significantly reduces the testing running time across all networks (Sprint and Oxford in Appendix) compared to both alternative approaches. For example, when testing 2560 service chaining and TCP reachability policies in Internet2, Mikado

[2]Since ATPG's source code only supports Internet2 and Stanford networks, we only apply ATPG to the two networks.



(a) Internet2 (b) Stanford (c) Chinanet.

Figure 9: Test running time.

can complete all tests in 13 runs by parallelizing most non-interfering traces. In contrast, BASELINE has to inject each trace in a single round (i.e. 2560 runs in total), and MB-based approaches can only complete all test in 663 runs. Both approaches take orders of magnitude more time than Mikado. Putting this result in perspective, if a single run can be complete in 30 seconds, Mikado can effectively reduce the test running time from 21 hours (sequential) to 10 minutes. We also note that the overhead of Mikado's scheduling algorithm is negligible compared with the test generation time: generating all test cases for Internet2 takes 14 hours, while Mikado only takes 2 minutes to generate the test schedule.

Second, Mikado's extensions are effective in further reducing the testing time. Fig. 10 shows the testing running time for the Internet2 network with Mikado's extensions. In general, we observe that our refinement technique achieves greater reduction compared to the random fuzzing heuristic. This is because that the refinement can systematically explore available field values by incorporating with testing tools via simple configurations. With the extensions, the number of runs in Internet2 can be reduced from 13 to 7.

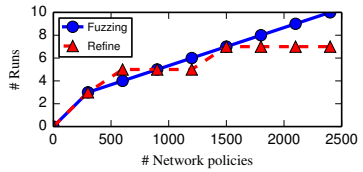


Figure 10: Test running time with Mikado extensions.

Sensitivity to topologies. To evaluate how sensitive Mikado is w.r.t. the network topology, we run our scheduling algorithm on 20 different topologies from TopologyZoo [26]. Figure 11 shows the CDF of the test running time reduction for 200 test traces on each topology. Here, we consider the testing time reduction as the ratio between the test running time using Mikado's correct schedule and that of the sequential testing.

We observe that our scheduling algorithm achieves high reduction for most cases. In particular, for more than half of the topologies, our basic scheduling algorithm achieves at least 87% reduction, while it has 96%+ reduction with the refinement technique.

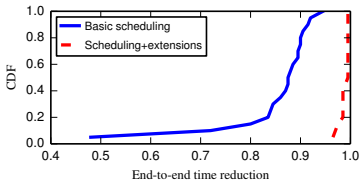


Figure 11: Test running time reduction CDF

7.3 Scalability

Next, we evaluate the scalability of Mikado with respect to the network size, the number and the complexity of policies. Specifically, we evaluate the running time of our scheduling algorithm, including the time of the interference graph generation and graph coloring[3].

With network size. We generate Fattree topologies [4] with varying sizes and augment the topology by adding a middlebox to every switch. We define the network size as the number of switches in it. We run Mikado on each topology with 5000 randomly generated test traces, and each test trace traverses two middleboxes.

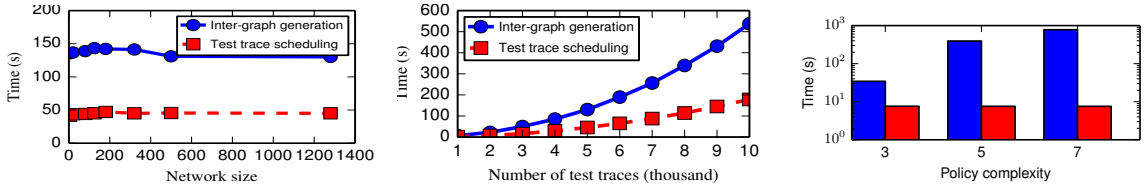
Figure 12a plots the running time of each component of our scheduling algorithm vs. network size. Both the interference graph generation and the graph coloring algorithm can scale up to 1000 switches with negligible increase in the running time. This is expected as the running of both algorithms are dominated by the number of test traces, as shown in §5. We also observe that the running time of the interference graph generation algorithm slightly decreases as the network size increments. This is because on larger networks the chance that two test traces interfere with each other is lower, and leads to faster interference checking.

With the number of test traces. We further run our algorithm on the Fattree topology with 500 switches, and vary the number of test traces from 1000 to 10000.

Figure 12b shows the running time of each algorithm with the number of test traces. As we show in §5, both algorithms run quadratically with the number of test traces; and for 10 thousand test traces, it takes less than 10 minutes for both algorithms to generate a correct schedule.

To put the above results in perspective, we further compare our scheduling algorithm with the naive scheduling approach discussed in §5. Recall that the

[3]We do not consider the refinement heuristic since it relies on other testing tools and not Mikado per se



(a) Network size. (b) Number of test traces. (c) Policy complexity.

Figure 12: Mikado scalability.

naive scheduling approach needs to consider all possible partitions of the test traces, and for each partition it needs to check for all possible runs due to the concurrent injection of all the traces in the partition. The naive approach takes more than 8 hours just for checking compatibility for 10 test traces on one switch.

With policy complexity. We define the policy complexity as the number of middleboxes appeared on the service chain of each policy. We run our algorithm on 2000 test traces, and vary the number of middleboxes each test trace must traverse. Figure 12c shows that both algorithms scales well w.r.t. policy complexity. Even in the case where each test case traverses 7 middleboxes, generating the interference graph takes 13 minutes, while the scheduling algorithm only takes 8 seconds.

8 Related Work

Our work draws ideas from several related areas including network testing, software testing, and formal modeling. We discuss related work in those areas.

Network testing and verification. The high-level goals of network testing [47, 13, 42] and network verification [22, 24, 21, 31, 32, 46, 6, 37, 12, 8, 6, 10, 44] are similar: check whether networks have implemented intended policies correctly. Different from network verification, network testing is better suited for bug finding and in general does not provide guarantees that policies are correctly implemented.

Our work builds upon existing work on network testing and focuses on generating correct and efficient scheduling of test cases. Even though we tested our framework on four existing testing tools, it can be extended as new testing tools are proposed.

Network modeling and abstractions. Formal models of networks are necessary building blocks for formal analysis of the network. There have been a number of existing formal network models (c.f [22, 40, 15, 34, 35, 5, 25, 36]). Some model stateless dataplanes, some model SDN control and dataplanes, and some like ours, model stateful dataplanes. All of the models are defined to facilitate analysis or verification methods that rely on the model. Ours is no exception. Our model, even though straightforward, serves the purpose of providing the basic constructs for defining the correct test ensemble scheduling problem, a key contribution of this work.

Service-chaining policies. The majority of the policies that we use to test Mikado are service-chaining policies. Recently, much work has been done surrounding enforcing service-chaining policies. For example, Simple [39] proposes a static service chain enforcement; FlowTag [14] uses tag bits in packet to implement dynamic service chaining; and [38] offers a composition for service-chaining policies. Rather than enforcement or composition of policies, our work stays at the level of scheduling test cases for those policies and is complementary to the above mentioned projects; Mikado can be used to perform extensive tests of networks that aim to enforce those policies.

Software testing. Our randomized packet fuzzing is borrowed from the software testing literature. Similar to software fuzzing, we also aim to achieve good coverage of the input space, but we do not really care about the coverage of the portions of the network tested. Testing network models is very similar to testing [18, 17, 41, 45, 11, 16], where network model is encoded as a program. However, the type of network testing that we are interested in are live tests (i.e., inject test while into the live network). This provides a set of unique challenges. To carry out an ensemble of tests on software, one can simply perform each test on a copy of the software in parallel. This is not possible for live network tests, since live networks cannot be easily duplicated.

9 Conclusions

We present Mikado, a framework that generates efficient and correct scheduling of test traces for ensembles of network policies. Using a formal model for stateful networks, we develop rigorous definitions of correctness for safely injecting test traces in parallel. We develop an efficient and provably correct algorithm for the test trace scheduling problem. Mikado employs additional heuristics to further improve the testing time reduction. We validate Mikado in a variety of scenarios, and show that Mikado can easily handle large networks with thousands of test traces.

Acknowledgment

We thank all anonymous reviewers and our shepherd Timothy Roscoe for their helpful suggestions and comments. This work is partially supported by NSF CNS-1513961, CNS-1552481, and Intel Labs University Research Office.

References

[1] Cisco NSH Service Chaining Configuration Guide. *https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/wan_nsh/configuration/xe-16/wan-nsh-xe-16-book.html*.

[2] ODL: Service Function Chaining. *http://events.linuxfoundation.org/sites/events/files/slides/odl%20summit%20sfc%20v5.pdf*.

[3] Survey on Network Testing. *http://www.andrew.cmu.edu/user/yifeiy2/mikado/survey.pdf*.

[4] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 63–74.

[5] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2014), ACM, pp. 113–126.

[6] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 31.

[7] BUTLER, B. What is intent-based networking? *https://www.networkworld.com/article/3202699/lan-wan/what-is-intent-based-networking.html*.

[8] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., REXFORD, J., ET AL. A NICE Way to Test OpenFlow Applications. In *NSDI* (2012), pp. 127–140.

[9] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[10] DOBRESCU, M., AND ARGYRAKI, K. Software Dataplane Verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 101–114.

[11] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic Test Input Generation for Database Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 151–162.

[12] FAYAZ, S. K., SHARMA, T., FOGEL, A., MAHAJAN, R., MILLSTEIN, T., SEKAR, V., AND VARGHESE, G. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 217–232.

[13] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 275–289.

[14] FAYAZBAKHSH, S. K., CHIANG, L., SEKAR, V., YU, M., AND MOGUL, J. C. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 533–546.

[15] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 279–291.

[16] GODEFROID, P. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 47–54.

[17] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 213–223.

[18] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)* (November 2008).

[19] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[20] JOSEPH, D. A., TAVAKOLI, A., AND STOICA, I. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 51–62.

[21] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. 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), USENIX, pp. 99–111.

[22] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 113–126.

[23] KHAN, F. Intent-based Networking - A Must for SDN. *http://resources.solarwinds.com/intent-based-networking-not-an-option-but-a-must-for-sdn/*.

[24] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 15–27.

[25] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 59–72.

[26] KNIGHT, S., NGUYEN, H., FALKNER, N., BOWDEN, R., AND ROUGHAN, M. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on 29*, 9 (october 2011), 1765 –1775.

[27] KUMAR, S., TUFAIL, M., MAJEE, S., CAPTARI, C., AND S, H. Service Function Chaining Use Cases In Data Centers. *https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06* (2014).

[28] LERNER, A. Intent-based Networking. *http://blogs.gartner.com/andrew-lerner/2017/02/07/intent-based-networking/*.

[29] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. MirrorNet: Faithfully Emulating Large Production Networks. In

Proceedings of the 26th Symposium on Operating Systems Principles (New York, NY, USA, 2013), SOSP '17, ACM.

[30] LIU, W., LI, H., HUANG, O., BOUCADAIR, M., LEYMANN, N., CAO, Z., AND HU, J. Service Function Chaining (SFC) Use Cases. *https://tools.ietf.org/html/draft-liu-sfc-use-cases-01* (2014).

[31] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 499–512.

[32] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 290–301.

[33] MEDVED, J., VARGA, R., TKACIK, A., AND GRAY, K. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a* (2014), IEEE, pp. 1–6.

[34] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices 47*, 1 (2012), 217–230.

[35] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX Association, pp. 1–13.

[36] NELSON, T., FERGUSON, A. D., SCHEER, M. J., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. *NSDI, Apr* (2014).

[37] PANDA, A., LAHAV, O., ARGYRAKI, K., SAGIV, M., AND SHENKER, S. Verifying Reachability in Networks with Mutable Datapaths.

[38] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 29–42.

[39] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 27–38.

[40] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 323–334.

[41] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2005), ESEC/FSE-13, ACM, pp. 263–272.

[42] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 314–327.

[43] SUBRAMANIAN, K., D'ANTONI, L., AND AKELLA, A. Genesis: synthesizing forwarding tables in multi-tenant networks. In *POPL* (2017), pp. 572–585.

[44] TSCHAEN, B., ZHANG, Y., BENSON, T., BENERJEE, S., LEE, J., AND KANG, J.-M. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference* (2016).

[45] VISSER, W., PĂSĂREANU, C. S., AND KHURSHID, S. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2004), ISSTA '04, ACM, pp. 97–107.

[46] XIE, G. G., ZHAN, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On Static Reachability Analysis of IP Networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE* (2005), vol. 3, IEEE, pp. 2170–2183.

[47] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic Test Packet Generation. *IEEE/ACM Trans. Netw. 22*, 2 (Apr. 2014), 554–566.

Appendix

Evaluation on Test Running Time

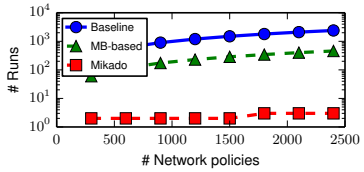


Figure 13: Test running time for Oxford

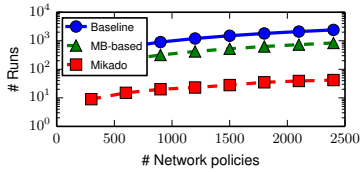


Figure 14: Test running time for Sprint

Proof of Theorem 1

Theorem 1 is a corollary of a stronger lemma (Lemma 2), which we explain next.

We first define the compatibility of two tests traces t_1 and t_2 using just the sequential run of each test trace as follows. The definition essentially is the negation of the condition on line 12 of Algorithm 1.

Definition 4 *Test trace t_1 is compatible with test trace t_2 w.r.t. M_{init} and (ρ,δ) iff $r^1 = R_{seq}(queue(t_1),M_{init})$, $r^2 = R_{seq}(queue(t_2),M_{init})$, $\forall i \in [0,|r^1|-1]$, $\forall j \in [0,|r^2|-1]$, $\forall x \in \mathsf{dom}(dtMap(\delta,M_i^1,lp_i^1)) \cap (upd(\delta,M_j^2,lp_j^2))$, $dtMap(\delta,M_i^1,lp_i^1)(x) = upd(\delta,M_j^2,lp_j^2)(x)$, where M_i^k and lp_i^k denotes the state map and the located packet processed at the i^{th} state of r^k respectively.*

Prove disconnected nodes in the graph returned by the GEN_INTF_GRAPH are compatible is straightforward (Lemma 1).

Lemma 1 *Let G =GEN_INTF_GRAPH(T,C,M_{init}). If v_{t_1} and v_{t_2} are not connected by an edge in G then t_1 is compatible with t_2 w.r.t. M_{init} and C.*

To simplify the proofs, we represent the set of runs $\mathscr{R}(Q,M)$ as an execution tree R, which can be either a leaf node of one network environment, or a node whose content is a network environment E and each of its children is another execution tree obtained from making a transition from E and processing packet lp_i.

$$\begin{array}{llll} \text{Exec. tree} & R & ::= & Lf(E) \\ & & \mid & Nd(E, \xrightarrow{lp_1/lp_1'} R_1, \cdots, \xrightarrow{lp_n/lp_n'} R_n) \\ \text{Path} & \pi & ::= & \bullet \mid n:\pi \end{array}$$

We can identify a unique run in R using a path π, which is a list of numbers indicating which transition to take. For instance, path 1:1:1 identifies the run that takes three transitions following the first branch at each step.

Next we define $r \lesssim_{\pi_1,\cdots,\pi_i,\cdots,\pi_n} R_1;\cdots;R_n$ in Figure 15. The meaning of this judgment is that r simulates an interleaving of runs, each of which is indexed by the path π_i in R_i. For our proofs, R_i is the execution tree for test case t_i and r is a run of the ensemble of tests t_1 to t_n.

The base case is when the indices all point to the initial state. We only need to check that r contains only one state, where the state map is the initial map and the queue only contains the test traces. Here, we $Q_1 \uplus Q_2$ represents an interleaving of packets from Q_1 and Q_2 that preserves the order of packets enforced by Q_1 and Q_2. The inductive case checks that (1) the last transition of r matches a transition in R_i (2) the port queue of the last state of r is an order-perserving interleaving of all the port queues in each R_j at the correct indices and (3) each state location in the state map M in the last state of r preserves the determinant state maps of each R_i. The last condition is the most complex, as we need to reason about who last updates a location s in M. If s is last updated by a packet related to test trace t_m, and s is part of the determinant location of another test trace t_l ($l \neq m$), then $M(s)$ should be the same as the value in that determinant state map. Otherwise, m may interfere with l. However, a test trace can modify its own determinant state locations. It would be too strong to require $M(s)$ to be equal to the values of all determinant state location of the test trace m itself. Instead, we only guarantee that $M(s)$ is equal to the corresponding state in R_m.

Lemma 2 is the key to our correctness proof. It states that any run starting from the ensemble of test traces maintains a simulation relation with each individual runs. Here, $\mathscr{R}_m$ denotes a run of length m.

Lemma 2 *Give a set of test traces $T = \{t_1,\cdots,t_n\}$, s.t. $\forall\ i\ j \in [1,n]$ and $i \neq j$, t_i is compatible with t_j w.r.t. M_{init} and C, let $R_i = \mathscr{R}((queue(t_i),M_{init}),C)$ then let $R = \mathscr{R}_m((queue(flatten(T)),M_{init}),C)$, $\forall r \in R$, $\exists \pi_1,\cdots,\pi_n$, $r \lesssim_{\pi_1,\cdots,\pi_n} R_1;\cdots;R_n$.*

The proof is by induction of $\sum_{i=1}^{n} |\pi_i|$. We rely on the fact that the set of valid runs have the same determinant state maps and updates (Definition 5) to generalize the compatibility conditions on sequential runs to other valid runs.

Definition 5 *We say that the valid runs of test trace t form an equivalence class iff let $r_{seq} = R_{seq}(queue(t),M_{init})$, $\forall r \in \mathscr{R}(queue(t),M_{init})$, $\forall lp \in t$, $r \downarrow_{\{lp\}} = r_{seq} \downarrow_{\{lp\}}$ and $\forall Q,M,lp_1,lp_1',Q',M'$ s.t. $(Q,M) \xrightarrow{lp/lp_1'} \in r_{seq}$, and $(Q',M') \xrightarrow{lp/lp_1'} \in r$, imply*

$$\frac{R_i = Lf(Q_i, M_i) \text{ or } Nd((Q_i, M_i), \_) \qquad M = M_1 = \cdots = M_n = M_{init} \qquad Q = Q_1 \uplus \cdots \uplus Q_n}{(Q, M) \lesssim_{\bullet,\cdots,\bullet} R_1; \cdots; R_n}$$

$$\frac{\begin{array}{l}\pi_i = \pi_i' : \eta \qquad r \lesssim_{\pi_1,\cdots,\pi_i',\cdots,\pi_n} R_1; \cdots; R_n \\ \forall j \in [1,n] \text{ let } R_j \backslash_{\pi_j} = Lf(Q_j, M_j) \text{ or } Nd((Q_j, M_j), \_) \\ Q' = \uplus_{j=1}^{n} Q^j \qquad R_i \backslash_{\pi_i'} = Nd((Q_a, M_a), Ch) \qquad Ch.\eta = \xrightarrow{lp/lp'} R_i', \\ \forall x \in \mathsf{dom}(M), x \text{ is last updated by } t_m, \\ \quad \forall l \in [1,n], s.t.\ l \neq m, \text{if } x \text{ is in the determinant state map of a state map } M_\alpha \text{ in } R_l, \text{then } M(x) = M_\alpha(x). \\ \quad \text{let } r_m = R_m|_{\pi_m} \text{ let } M_\beta \text{ be the last state in } r_m \text{ where } x \text{ is updated from the previous state}, M(x) = M_\beta(x) \\ \forall x \in \mathsf{dom}(M), x \text{ is not updated by any thread}, \forall j \in [1,n], M_j(x) = M(x) = M_{init}(x)\end{array}}{r \xrightarrow{lp/lp'} (Q, M) \lesssim_{\pi_1,\cdots,\pi_i,\cdots,\pi_n} R_1; \cdots; R_n}$$

Figure 15: Simulation relation

$upd(\delta, M, lp) = upd(\delta, M', lp)$ *and* $dtMap(\delta, M, lp) = dtMap(\delta, M', lp)$.

The correspondence relation established in Lemma 2 ensures that the determinant state maps of individual runs are preserved by the combined run. Theorem 1 follows straightforwardly because observations are determined by the transitions, which in turn, are determined by the state maps.