Systematically Testing OpenFlow Controller Applications

Peter Perešíni[⋆], Maciej Kuźniar[⋆], Marco Canini[†☆], Daniele Venzano[‡], Dejan Kostić[•], and Jennifer Rexford[ᵀ]

[⋆] *EPFL, Switzerland,* [†] *Université catholique de Louvain, Belgium,* [‡] *EURECOM, France*

[•] *KTH Royal Institute of Technology, Sweden,* [ᵀ] *Princeton University, United States*

Abstract

The emergence of OpenFlow-capable switches enables exciting new network functionality, at the risk of programming errors that make communication less reliable. The centralized programming model, where a single controller program manages the network, seems to reduce the likelihood of bugs. However, the system is inherently distributed and asynchronous, with events happening at different switches and end hosts, and inevitable delays affecting communication with the controller. In this paper, we present efficient, systematic techniques for testing unmodified controller programs. Our NICE tool applies model checking to explore the state space of the entire system—the controller, the switches, and the hosts. Scalability is the main challenge, given the diversity of data packets, the large system state, and the many possible event orderings. To address this, we propose a novel way to augment model checking with symbolic execution of event handlers (to identify representative packets that exercise code paths on the controller). We also present a simplified OpenFlow switch model (to reduce the state space), and effective strategies for generating event interleavings likely to uncover bugs. Our prototype tests Python applications on the popular NOX platform. In testing three real applications—a MAC-learning switch, in-network server load balancing, and energy-efficient traffic engineering—we uncover thirteen bugs.

Keywords: Software-defined networking; OpenFlow; reliability; model checking; symbolic execution

1. Introduction

While lowering the barrier for introducing new functionality into the network, Software-Defined Networking (SDN) also raises the risks of software faults (or *bugs*). Even today's networking software—written and extensively tested by equipment vendors, and constrained (at least somewhat) by the protocol standardization process—can have bugs that trigger Internet-wide outages [1, 2]. In contrast, programmable networks will offer a much wider range of functionality, through software created by a diverse collection of network operators and third-party developers. The ultimate success of SDN, and enabling technologies like OpenFlow [3], depends on having effective ways to test applications in pursuit of achieving high reliability. In this paper, we present NICE, a tool that efficiently uncovers bugs in OpenFlow programs, through a combination of model checking and symbolic execution.

1.1. Bugs in OpenFlow Applications

An OpenFlow network consists of a distributed collection of switches managed by a program running on a logically-centralized controller, as illustrated in Figure 1. Each switch has a flow table that stores a list of rules for processing packets. Each rule consists of a pattern (matching on packet header fields) and actions (such as forwarding, dropping, flooding, or modifying the packets, or sending them to the controller). A pattern can require an "exact match" on all relevant header fields (*i.e.*, a *microflow* rule), or have "don't care" bits in some fields (*i.e.*, a *wildcard* rule). For each rule, the switch maintains traffic counters that measure the bytes and packets processed so far. When a

[☆] Corresponding email: marco.canini@uclouvain.be

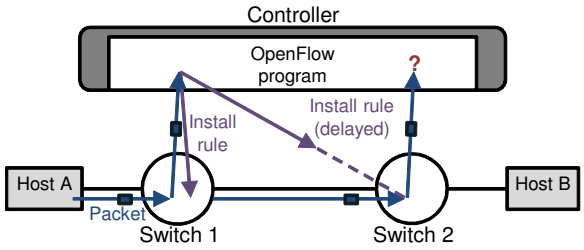


Figure 1: An example of OpenFlow network traversed by a packet. Due to delays between controller and switches, the packet may not encounter an installed rule in the second switch.

packet arrives, a switch selects the highest-priority matching rule, updates the counters, and performs the specified action(s). If no rule matches, the switch sends (part of) the packet to the controller and awaits a response on what actions to take. Switches also send event messages, upon joining the network, or when links go up or down.

The OpenFlow controller (un)installs rules in the switches, reads traffic statistics, and responds to events. For each event, the controller program defines a handler, which may install rules or issue requests for traffic statistics. Many OpenFlow applications[1] are written on the NOX controller platform [4], which offers an OpenFlow API for Python and C++ applications. These programs can perform arbitrary computation and maintain arbitrary state. A growing collection of controller applications support new network functionality [5, 6, 7, 8, 9, 10], over OpenFlow switches available from several different vendors. Our goal is to create an efficient tool for systematically testing these applications. More precisely, we seek to discover violations of (network-wide) correctness properties due to bugs in the controller programs.

On the surface, the centralized programming model should reduce the likelihood of bugs. Yet, a software-defined network is inherently distributed and asynchronous, with events happening at multiple switches and inevitable delays affecting communication with the controller. To reduce overhead and delay, applications push as much packet-handling functionality to the switches as possible. A common programming idiom is to respond to a packet arrival by installing a rule for handling subsequent packets in the data plane. Yet, a race condition can arise if additional packets arrive while installing the rule. A program that implicitly expects to see just one packet may behave incorrectly when multiple arrive [11]. In addition, many applications install rules at multiple switches along a path. Since rules are not installed atomically, some switches may apply new rules before others install theirs. Figure 1 shows an example where a packet reaches an intermediate switch before the relevant rule is installed. This can lead to unexpected behavior, where an intermediate switch directs a packet to the controller. As a result, an OpenFlow application that usually works correctly can misbehave under certain event orderings.

1.2. Challenges of Testing OpenFlow Apps

Testing OpenFlow applications is challenging because the behavior of a program depends on the larger execution environment. The end-host applications sending and receiving traffic—and the switches handling packets, installing rules, and generating events—all affect the program running on the controller. The need to consider the larger environment leads to an extremely large state space, which "explodes" along three dimensions:

Large space of switch state: Switches run their own programs that maintain state, including the many packet-processing rules and associated counters and timers. Further, the set of packets that match a rule depends on the presence or absence of other rules, due to the "match the highest-priority rule" semantics. As such, testing OpenFlow applications requires an effective way to capture the large state space of the switch.

Large space of input packets: Applications are *data-plane* driven, *i.e.*, programs must react to a huge space of possible packets. Older OpenFlow specifications allow switches to match on MAC and IP addresses, TCP/UDP port numbers, and the switch input port; newer generations of switches match on even more fields. The controller can perform arbitrary processing based on other fields, such as TCP flags or sequence numbers. As such, testing OpenFlow applications requires effective techniques to deal with large space of inputs.

Large space of event orderings: Network events, such as packet arrivals and topology changes, can happen at any switch at any time. Due to communication delays, the controller may not receive events in order, and rules may not be installed in order across multiple switches. Serializing rule installation, while possible, would significantly reduce application performance. As such, testing

[1] In this paper, we use the terms "OpenFlow application" and "controller program" interchangeably.

OpenFlow applications requires efficient strategies to explore a large space of event orderings.

To simplify the problem, we could require programmers to use domain-specific languages that prevent certain classes of bugs [12, 13, 14]. However, the adoption of new languages is difficult in practice. Not surprisingly, most OpenFlow applications are written in general-purpose languages, like Python and Java. Alternatively, developers could create abstract models of their applications, and use formal-methods techniques to prove properties about the system. However, these models are time-consuming to create and easily become out-of-sync with the real implementation. In addition, existing model-checking tools like SPIN [15] and Java PathFinder (JPF) [16] cannot be directly applied because they require explicit developer inputs to resolve the data-dependency issues and sophisticated modeling techniques to leverage domain-specific information. They also suffer state-space explosion, as we show in Section 8. Instead, we argue that testing tools should operate directly on *unmodified* OpenFlow applications, and leverage *domain-specific knowledge* to improve scalability.

1.3. NICE Research Contributions

To address these scalability challenges, we present NICE (*No bugs In Controller Execution*)—a tool that tests unmodified controller programs[2] by automatically generating carefully-crafted streams of packets under many possible event interleavings. To use NICE, the programmer supplies the controller program, and the specification of a topology with switches and hosts. The programmer can instruct NICE to check for generic correctness properties such as no forwarding loops or no black holes, and optionally write additional, application-specific correctness properties (*i.e.*, Python code snippets that make assertions about the global system state). By default, NICE systematically explores the space of possible system behaviors, and checks them against the desired correctness properties. The programmer can also configure the desired search strategy. In the end, NICE outputs property violations along with the traces to reproduce them. Programmers can also use NICE as a simulator to perform manually-driven, step-by-step system executions or random walks on system states. By combining these two features – gathering event traces that lead to bugs and step-by-step execution – developers can effectively debug their applications.

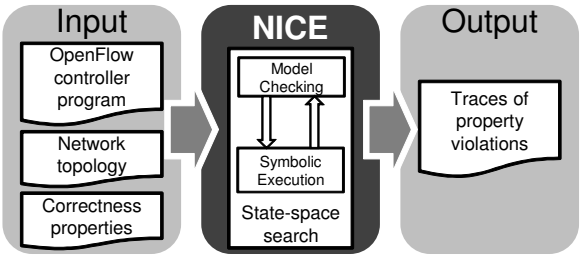


Figure 2: Given an OpenFlow program, a network topology, and correctness properties, NICE performs a state-space search and outputs traces of property violations.

Our design uses explicit state, software model checking [16, 17, 18, 19] to explore the state space of the entire system—the controller program, the OpenFlow switches, and the end hosts—as discussed in Section 2. However, applying model checking "out of the box" does not scale. While simplified models of the switches and hosts help, the main challenge is the event handlers in the controller program. These handlers are data dependent, forcing model checking to explore all possible inputs (which doesn't scale) or a set of "important" inputs provided by the developer (which is undesirable). Instead, we extend model checking to *symbolically execute* [20, 21] the handlers, as discussed in Section 3. By symbolically executing the packet-arrival handler, NICE identifies equivalence classes of packets—ranges of header fields that determine unique paths through the code. NICE feeds the network a representative packet from each class by adding a state transition that injects the packet. To reduce the space of event orderings, we propose several domain-specific search strategies that generate event interleavings that are likely to uncover bugs in the controller program, as discussed in Section 5.

Bringing these ideas together, NICE combines model checking (to explore system execution paths), symbolic execution (to reduce the space of inputs), and search strategies (to reduce the space of event orderings). The programmer can specify correctness properties as snippets of Python code that assert global system state, or select from a library of common properties, as discussed in Section 6. Our NICE prototype tests unmodified applications written in Python for the popular NOX platform, as discussed in Section 7. Our performance evaluation in Section 8 shows that: (*i*) even on small examples, NICE is five times faster than

[2] NICE only requires access to the controller state.

approaches that apply state-of-the-art tools, (*ii*) our OpenFlow-specific search strategies reduce the state space by up to 20 times, and (*iii*) the simplified switch model brings a 4-fold reduction on its own. In Section 9, we apply NICE to three real OpenFlow applications and uncover *13* bugs. Most of the bugs we found are design flaws, which are inherently less numerous than simple implementation bugs. In addition, at least one of these applications was tested using unit tests. Section 10 discusses the trade-off between testing coverage and the overhead of symbolic execution. Section 11 discusses related work, and Section 12 concludes the paper.

2. Model Checking OpenFlow Applications

The execution of a controller program depends on the underlying switches and end hosts; the controller, in turn, affects the behavior of these components. As such, testing is not just a matter of exercising every path through the controller program—we must consider the state of the larger system. The requirements for systematically exploring the space of system states, and checking correctness in each state, naturally lead us to consider *model checking* techniques. To apply model checking, we need to identify the system states and the transitions between states. After a brief review of model checking, we present a strawman approach for applying model checking to OpenFlow applications, and proceed by describing changes that make it more tractable.

2.1. Background on Model Checking

Modeling the state space. A distributed system consists of multiple *components* that communicate asynchronously over message *channels*, *i.e.*, first-in, first-out buffers (*e.g.*, see Chapter 2 of [22]). Each component has a set of variables, and the *component state* is an assignment of values to these variables. The *system state* is the composition of the component states. To capture in-flight messages, the system state also includes the contents of the channels. A *transition* represents a change from one state to another (*e.g.*, due to sending a message). At any given state, each component maintains a set of enabled transitions, *i.e.*, the state's possible transitions. For each state, the enabled system transitions are the union of enabled transitions at all components. A *system execution* corresponds to a sequence of these transitions, and thus specifies a possible behavior of the system.

Model-checking process. Given a model of the state space, performing a search is conceptually straightforward. Figure 5 (non boxed-in text) shows the pseudo-code of the model-checking loop. First, the model checker initializes a stack of states with the initial state of the system. At each step, the checker chooses one state from the stack and one of its enabled transitions. After executing this transition, the checker tests the correctness properties on the newly reached state. If the new state violates a correctness property, the checker saves the error and the execution trace. Otherwise, the checker adds the new state to the set of explored states (unless the state was added earlier) and schedules the execution of all transitions enabled in this state (if any). The model checker can run until the stack of states is empty, or until detecting the first error.

2.2. Transition Model for OpenFlow Apps

Model checking relies on having a model of the system, *i.e.*, a description of the state space. This requires us to identify the states and transitions for each component—the controller program, the OpenFlow switches, and the end hosts. However, we argue that applying existing model-checking techniques imposes too much work on the developer and leads to an explosion in the state space.

2.2.1. Controller Program

Modeling the controller as a transition system seems straightforward. A controller program is structured as a set of event handlers (*e.g.*, packet arrival and switch join/leave for the MAC-learning application in Figure 3), that interact with the switches using a standard interface, and these handlers execute atomically. As such, we can model the state of the program as the values of its global variables (*e.g.*, `ctrl_state` in Figure 3), and treat event handlers as a transitions. To execute a transition, the model checker can simply invoke the associated event handler. For example, receiving a packet-in message from a switch enables the `packet_in` transition, and the model checker can execute it by invoking the corresponding event handler.

However, the behavior of event handlers is often data-dependent. In line 7 of Figure 3, for instance, the `packet_in` handler assigns `mactable` only for unicast source MAC addresses, and either installs a forwarding rule or floods a packet depending on whether or not the destination MAC is known. This leads to different system executions. Unfortunately, model checking does not cope well with

```
1 ctrl_state = {} # State of the controller is a global variable
2 def packet_in(sw_id, inport, pkt, bufid): # Handle new packets
3   mactable = ctrl_state[sw_id]
4   is_bcast_src = pkt.src[0] & 1
5   is_bcast_dst = pkt.dst[0] & 1
6   if not is_bcast_src:
7     mactable[pkt.src] = inport
8   if (not is_bcast_dst) and (mactable.has_key(pkt.dst)):
9     outport = mactable[pkt.dst]
10    if outport != inport:
11      match = {DL_SRC: pkt.src, DL_DST: pkt.dst,
            DL_TYPE: pkt.type, IN_PORT: inport}
12      actions = [OUTPUT, outport]
13      install_rule(sw_id, match, actions, soft_timer=5,
            hard_timer=PERMANENT) # 2 lines optionally
14      send_packet_out(sw_id, pkt, bufid) # combined in 1 API
15      return
16  flood_packet(sw_id, pkt, bufid)
17 def switch_join(sw_id, stats): # Handles when a switch joins
18   if not ctrl_state.has_key(sw_id):
19     ctrl_state[sw_id] = {}
20 def switch_leave(sw_id): # Handles when a switch leaves
21   if ctrl_state.has_key(sw_id):
22     del ctrl_state[sw_id]
```

Figure 3: Pseudo-code of a MAC-learning switch, based on the `pyswitch` application. The `packet_in` handler learns the input port associated with non-broadcast source MAC addresses; if the destination MAC is known, the handler installs a forwarding rule and instructs the switch to send the packet according to that rule. Otherwise it floods the packet.

data-dependent applications (see Chapter 1 of [22]). Since enumerating all possible inputs is intractable, a brute-force solution would require developers to specify "relevant" inputs based on their knowledge of the application. Hence, a controller transition would be modeled as a pair consisting of an event handler and a concrete input. This is clearly undesirable. NICE overcomes this limitation by using *symbolic execution* to automatically identify the relevant inputs, as discussed in Section 3.

2.2.2. OpenFlow Switches

To test the controller program, the system model must include the underlying switches. Yet, switches run complex software, and this is not the code we intend to test. A strawman approach for modeling the switch is to start with an existing reference OpenFlow switch implementation (*e.g.*, [23]), define its state as the values of all variables, and identify transitions as the portions of the code that process packets or exchange messages with the controller. However, the reference switch software has a large state (*e.g.*, several hundred KB), not including the buffers containing packets and OpenFlow messages awaiting service; this aggravates the state-space explosion problem. Importantly, such a large program has many sources of nondeterminism and it is difficult to identify them automatically [19].

Instead, we create a switch model that omits inessential details. Indeed, creating models of some parts of the system is common to many standard approaches for applying model checking. Further, in our case, this is a one-time effort that does not add burden on the user. Following the OpenFlow specification [24], we view a switch as a set of communication channels, transitions that handle data packets and OpenFlow messages, and a flow table.

Simple communication channels: Each channel is a first-in, first-out buffer. Packet channels have an optionally-enabled fault model that can drop, duplicate, or reorder packets, or fail the link. The channel with the controller offers reliable, in-order delivery of OpenFlow messages, except for optional switch failures. We do not run the OpenFlow protocol over SSL on top of TCP/IP, allowing us to avoid intermediate protocol encoding/decoding and the substantial state in the network stack.

Two simple transitions: The switch model supports `process_pkt` and `process_of` transitions—for processing data packets and OpenFlow messages, respectively. We enable these transitions if at least one packet channel or the OpenFlow channel is non empty, respectively. A final simplification we make is in the `process_pkt` transition. Here, the switch dequeues the first packet from each packet channel, and processes *all* these packets according to the flow table. So, multiple packets at different channels are processed as a single transition. This optimization is safe because the model checker already systematically explores the possible orderings of packet arrivals at the switch.

Merging equivalent flow tables: A flow table can easily have two states that appear different but are semantically equivalent, leading to a larger search space than necessary. For example, consider a switch with two microflow rules. These rules do not overlap—no packet would ever match both rules. As such, the order of these two rules is not important. Yet, simply storing the rules as a list would cause the model checker to treat two different orderings of the rules as two distinct states. Instead, as often done in model checking, we construct a canonical representation of the flow table that derives a unique order of rules.

2.2.3. End Hosts

Modeling the end hosts is tricky, because hosts run arbitrary applications and protocols, have large state, and have behavior that depends on incoming packets. We could require the developer to provide the host programs, with a clear indication of the

transitions between states. Instead, NICE provides simple programs that act as clients or servers for a variety of protocols including Ethernet, ARP, IP, and TCP. These models have explicit transitions and relatively little state. For instance, the default client has two basic transitions—`send` (initially enabled; can execute C times, where C is configurable) and `receive`—and a counter of sent packets. The default server has the `receive` and the `send_reply` transitions; the latter is enabled by the former. A more realistic refinement of this model is the mobile host that includes the `move` transition that moves the host to a new <switch, port> location. The programmer can also customize the models we provide, or create new models.

3. Symbolic Execution of Event Handlers

To systematically test the controller program, we must explore all of its possible transitions. Yet, the behavior of an event handler depends on the inputs (*e.g.*, the MAC addresses of packets in Figure 3). Rather than explore all possible inputs, NICE identifies which inputs would exercise different code paths through an event handler. Systematically exploring all code paths naturally leads us to consider *symbolic execution* (SE) techniques. After a brief review of symbolic execution, we describe how we apply symbolic execution to controller programs. Then, we explain how NICE combines model checking and symbolic execution to explore the state space effectively.

3.1. Background on Symbolic Execution

Symbolic execution runs a program with symbolic variables as inputs (*i.e.*, any values). The symbolic-execution engine tracks the use of symbolic variables and records the constraints on their possible values. For example, in line 4 of Figure 3, the engine learns that `is_bcast_src` is "`pkt.src[0] & 1`". At any branch, the engine queries a constraint solver for two assignments of symbolic inputs—one that satisfies the branch predicate and one that satisfies its negation (*i.e.*, takes the "else" branch)— and logically forks the execution to follow the feasible paths. For example, the engine determines that to reach line 7 of Figure 3, the source MAC address must have its eighth bit set to zero.

Unfortunately, symbolic execution does not scale well because the number of code paths can grow exponentially with the number of branches and the size of the inputs. Also, symbolic execution does not explicitly model the state space, which can cause repeated exploration of the same system state unless expensive and possibly undecidable state-equivalence checks are performed. In addition, despite exploring all *code paths*, symbolic execution does not explore all *system execution paths*, such as different event interleavings. Techniques exist that can add artificial branching points to a program to inject faults or explore different event orderings [21, 25], but at the expense of extra complexity. As such, symbolic execution is insufficient for testing OpenFlow applications. Instead, NICE uses model checking to explore system execution paths (and detect repeated visits to the same state [26]), and symbolic execution to determine which inputs would exercise a particular state transition.

3.2. Symbolic Execution of OpenFlow Apps

Applying symbolic execution to the controller event handlers is relatively straightforward, with two exceptions. First, to handle the diverse inputs to the `packet_in` handler, we construct *symbolic packets*. Second, to minimize the size of the state space, we choose a *concrete* (rather than symbolic) representation of controller state.

Symbolic packets. The main input to the `packet_in` handler is the incoming packet. To perform symbolic execution, NICE must identify which (ranges of) packet header fields determine the path through the handler. Rather than view a packet as a generic array of symbolic bytes, we introduce *symbolic packets* as our symbolic data type. A symbolic packet is a group of symbolic integer variables that each represents a header field. To reduce the overhead for the constraint solver, we maintain each header field as a lazily-initialized, individual symbolic variable (*e.g.*, a MAC address is a 6-byte variable), which reduces the number of variables. Yet, we still allow byte- and bit-level accesses to the fields. We also apply domain knowledge to further constrain the possible values of header fields (*e.g.*, the MAC and IP addresses used by the hosts and switches in the system model, as specified by the input topology).

Concrete controller state. The execution of the event handlers also depends on the controller state. For example, the code in Figure 3 reaches line 9 only for unicast destination MAC addresses stored in `mactable`. Starting with an empty `mactable`, symbolic execution cannot find an input packet that

forces the execution of line 9; yet, with a non-empty table, certain packets could trigger line 9 to run, while others would not. As such, we must incorporate the global variables into the symbolic execution. We choose to represent the global variables in a concrete form. We apply symbolic execution by using these concrete variables as the initial state and by marking as symbolic the packets and statistics arguments to the handlers. The alternative of treating the controller state as symbolic would require a sophisticated type-sensitive analysis of complex data structures (*e.g.*, [26]), which is computationally expensive and difficult for a dynamically typed language like Python.

3.3. Combining SE with Model Checking

With all of NICE's parts in place, we now describe how we combine model checking (to explore system execution paths) and symbolic execution (to reduce the space of inputs). At any given controller state, we want to identify packets that each client should send—specifically, the set of packets that exercise all feasible code paths on the controller in that state. To do so, we create a special client transition called `discover_packets` that symbolically executes the `packet_in` handler. Figure 4 shows the unfolding of controller's state-space graph.

Symbolic execution of the handler starts from the initial state defined by (*i*) the concrete controller state (*e.g.*, State 0 in Figure 4) and (*ii*) a concrete "context" (*i.e.*, the switch and input port that identify the client's location). For every feasible code path in the handler, the symbolic-execution engine finds an equivalence class of packets that exercise it. For each equivalence class, we instantiate one *concrete packet* (referred to as the relevant packet) and enable a corresponding `send` transition for the client. While this example focuses on the `packet_in` handler, we apply similar techniques to deal with traffic statistics, by introducing a special `discover_stats` transition that symbolically executes the statistics handler with symbolic integers as arguments. Other handlers, related to topology changes, operate on concrete inputs (*e.g.*, the switch and port ids).

Figure 5 shows the pseudo-code of our search-space algorithm, which extends the basic model-checking loop in two main ways.

Initialization (lines 3-5): For each client, the algorithm (*i*) creates an empty map for storing the relevant packets for a given controller state and (*ii*) enables the `discover_packets` transition.

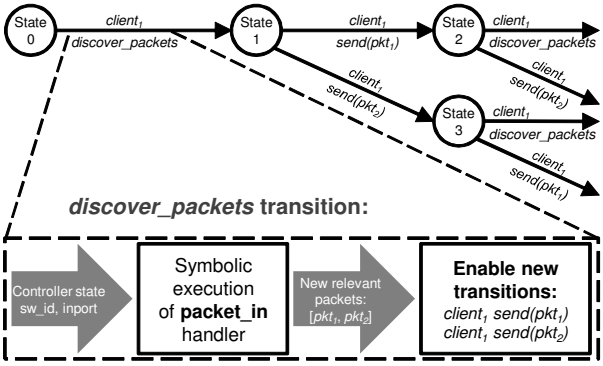


Figure 4: Example of how NICE identifies relevant packets and uses them as new enabled send packet transitions of $client_1$. For clarity, the circled states refer to the controller state only.

Checking process (lines 12-18): Upon reaching a new state, the algorithm checks for each client (line 15) whether a set of relevant packets already exists. If not, it enables the `discover_packets` transition. In addition, it checks (line 17) if a `process_stats` transition is enabled in the newly-reached state, meaning that the controller is awaiting a response to a previous query for statistics. If so, the algorithm enables the `discover_stats` transition.

Invoking the `discover_packets` (lines 26-31) and `discover_stats` (lines 32-35) transitions allows the system to evolve to a state where new transitions become possible—one for each path in the packet-arrival or statistics handler. This allows the model checker to reach new controller states, allowing symbolic execution to again uncover new classes of inputs that enable additional transitions, and so on.

By symbolically executing the controller event handlers, NICE automatically infers the test inputs for enabling model checking without developer input, at the expense of some limitations in coverage of the state space which we discuss in Section 10.

4. General-Purpose Search Optimizations

Even with our optimizations from the last two sections, the model checker cannot typically explore the entire state space, since it may be prohibitively large or even infinite. In this and the next section, we describe both general-purpose and domain-specific techniques for reducing the search space.

Partial-Order Reduction (POR) is a technique that reduces the number of possible orderings to be analyzed without sacrificing search completeness (*e.g.*, see Chapter 8 of [22]). POR takes advantage

```
1 state_stack = []; explored_states = []; errors = []
2 initial_state = create_initial_state()
3 for client in initial_state.clients
4   client.packets = {}
5   client.enable_transition(discover_packets)
6 for t in initial_state.enabled_transitions:
7   state_stack.push([initial_state, t])
8 while len(state_stack) > 0:
9   state, transition = choose(state_stack)
10  try:
11    next_state = run(state, transition)
12    ctrl = next_state.ctrl # Reference to controller
13    ctrl_state = ctrl.get_state() # Stringified controller state
14    for client in state.clients:
15      if not client.packets.has_key(ctrl_state):
16        client.enable_transition(discover_packets, ctrl)
17    if process_stats in ctrl.enabled_transitions:
18      ctrl.enable_transition(discover_stats, state, sw_id)
19    check_properties(next_state)
20    if next_state not in explored_states:
21      explored_states.add(next_state)
22      for t in next_state.enabled_transitions:
23        state_stack.push([next_state, t])
24  except PropertyViolation as e:
25    errors.append([e, trace])
26 def discover_packets_transition(client, ctrl):
27   sw_id, inport = switch_location_of(client)
28   new_packets = SymbolicExecution(ctrl, packet_in,
         context=[sw_id, inport])
29   client.packets[state(ctrl)] = new_packets
30   for packet in client.packets[state(ctrl)]:
31     client.enable_transition(send, packet)
32 def discover_stats_transition(ctrl, state, sw_id):
33   new_stats = SymbolicExecution(ctrl, process_stats,
         context=[sw_id])
34   for stats in new_stats:
35     ctrl.enable_transition(process_stats, stats)
```

Figure 5: The state-space search algorithm used in NICE for finding errors. The highlighted parts, including the special "*discover*" transitions, are our additions to the basic model-checking loop.

of the fact that many events that are happening in the system are independent and can be explored following just one arbitrary order. Formally, let T be a set of transitions; we say transitions $t_1 \in T$ and $t_2 \in T$ are independent (adapted from [27]) if for all states s:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'.
2. if t_1 and t_2 are enabled in s, then there is a state s' that $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$.

In other words, independent transitions do not enable or disable each other and if they are both enabled they commute. If two transitions are not independent, they are dependent. Further, transitions that are dependent are *worth reordering* because they lead to different system behaviors.

In practice, to check if transitions are independent, POR algorithms use the notion of *shared objects*. Transitions that use the same shared object are dependent and the ones that use disjoint sets of shared objects are independent. Identifying the right granularity for shared objects is crucial because if the choice is too conservative, transitions that are not worth reordering would be considered dependent and make POR less effective. On the other hand, missing a shared object generally means that the search is no longer complete.

To apply POR in our context, we identify the following shared objects:
(1) input buffers of switches and hosts: All transitions that read from or write to a buffer of a given node (send or receive a packet) are dependent.
(2) switch flow tables: The order of switch flow table modifications is important because it matters if a rule is installed before or after a packet matching that rule arrives at the switch. However, we exploit the semantics of flow tables to identify more fine-grained shared objects and hence make POR more effective. In particular, a pair of transitions that only read from the flow table are independent. Also, transitions that operate on rules with non-overlapping matches are independent. All remaining transition combinations are dependent.
(3) the controller application: Because NICE treats the controller state as a single entity all transitions related to the controller are dependent. No further specialization is possible without relying on more fine-grained analysis of controller state.

Rather than relying on a-priori knowledge about what shared objects are used by which transition, we use a dynamic variant of POR, namely Dynamic Partial-Order Reduction (DPOR) [27], which gathers the dependency information at run-time while executing transitions, and modifies the set of enabled transitions at each visited state accordingly.

We base our DPOR algorithm on the one presented in [27] and extend it to work for our context. The base DPOR algorithm relies on the fact that a single execution path ends when there are no enabled transitions. NICE typically bounds the search depth and uses state matching to avoid exploring the same state multiple times. Therefore, some search paths end before all possible transitions that could be worth reordering are considered by DPOR. To avoid this problem, we conservatively consider all enabled transitions as worth exploring unless specifically marked otherwise.

We first introduce some notation: Let E_i be the set of transitions enabled in state s_i; let $SE_i \subseteq E_i$ be the set of transitions strongly enabled in s_i (*i.e.*, the transitions that cannot be ignored); let $D_i \subseteq$

```
1  def DPOR(current_path):
2    for t_i in current_path[1 : n−1]:
3      state = get_start_state(t_i)
4      for t_j in current_path[idx(t_i) + 1 : n]:
5        if is_worth_reordering(t_i, t_j):
6          if t_j in state.enabled:
7            t_k = t_j
8          else:
9            t_k = get_predecessor(state, t_j)
10         state.strongly_enabled.add(t_k)
11         state.force_order(t_k, (t_j, t_i))
12       elif t_j in state.enabled:
13         state.unnecessary.add(t_j)
14 def get_next_transition(state):
15   for t in state.enabled:
16     if (t in state.unnecessary) and (t not in
           state.strongly_enabled):
17       state.enabled.remove(t)
18   return state.enabled.pop()
```

Figure 6: Pseudo-code of the DPOR algorithm. The get_next_transition method prunes the enabled actions and returns the next transition worth executing. The DPOR method runs when the search ends in a state with no enabled transitions.

E_i be the set of transitions enabled in s_i that are temporarily disabled by DPOR. The search path is represented by a transition sequence: $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_{n+1}$. For each transition t_j in this sequence, we keep track of the transition t_i that enabled t_j. In other words, we look for i such that $t_j \notin E_i$ and $t_j \in E_{i+1}$. Let $P(j) = i$, we call $t_{P(j)}$ a first level predecessor of t_j, $t_{P(P(j))}$ a second level predecessor of t_j, and so on.

Figure 6 presents the pseudo-code of our DPOR algorithm. The function `DPOR` is invoked for the current execution path once the model checker reaches a state s_n with no enabled transitions. For every transition t_i with $i \in [1, n]$, we identify all transitions t_j where $j \in [i+1, n]$ that t_i is worth reordering with (line 5). In this case, we want to enforce a new search path that, starting from s_i, would first execute t_j and then t_i (lines 6-11). If $t_j \in E_i$, we simply add t_j to SE_i. If $t_j \notin E_i$, we find a transition $t_k \in E_i$ that is a predecessor of t_j (line 9) and add it to SE_i (line 10). If t_i is not worth reordering with $t_j \in E_i$, we add t_j to D_i (line 13). Marking t_k for exploration is necessary to ensure search completeness, but it is not sufficient to efficiently prune states. After executing t_k, the model checker would again have two possible orderings of t_i and t_j. To achieve the desired state space reduction, we enforce that t_j is executed before t_i on this particular search path (line 11). Before selecting the next transition to explore in a given state, the `get_next_transition` function disables all transitions enabled in this state, that were considered not worth reordering while exploring other paths (line 16).

5. OpenFlow-Specific Search Strategies

We propose domain-specific heuristics that substantially reduce the space of event orderings while focusing on scenarios that are likely to uncover bugs. Most of the strategies operate on the event interleavings produced by model checking, except for PKT-SEQ which reduces the state-space explosion due to the transitions uncovered by symbolic execution.

PKT-SEQ: Relevant packet sequences. The effect of discovering new relevant packets and using them as new enabled `send` transitions is that each end-host generates a potentially-unbounded tree of packet sequences. To make the state space finite and smaller, this heuristic reduces the search space by bounding the possible end host transitions (indirectly, bounding the tree) along two dimensions, each of which can be fine-tuned by the user. The first is merely *the maximum length of the sequence*, or in other words, the depth of the tree. Effectively, this also places a hard limit to the issue of infinite execution trees due to symbolic execution. The second is the *maximum number of outstanding packets*, or in other words, the length of a packet burst. For example, if $client_1$ in Figure 4 is allowed only a 1-packet burst, this heuristic would disallow both `send(`pkt_2`)` in State 2 and `send(`pkt_1`)` in State 3. Effectively, this limits the level of "packet concurrency" within the state space. To introduce this limit, we assign each end host with a counter c; when $c = 0$, the end host cannot send any more packets until the counter is replenished. As we are dealing with communicating end hosts, we adopt as default behavior to increase c by one unit for every received packet. However, this behavior can be modified in more complex end host models, *e.g.*, to mimic the TCP flow and congestion controls.

NO-DELAY: Instantaneous rule updates. When using this simple heuristic, NICE treats each communication between a switch and the controller as a single atomic action (*i.e.*, not interleaved with any other transitions). In other words, the global system runs in "lock step". This heuristic is useful during the early stages of development to find basic design errors, rather than race conditions or other concurrency-related issues. For instance, it would allow the developer to realize that installing a rule prevents the controller from seeing other

packets that are important for program correctness. For example, a MAC-learning application that installs forwarding rules based only on the destination MAC address would prevent the controller from seeing some packets with new source MAC addresses.

UNUSUAL: Unusual delays and reorderings. With this heuristic, NICE only explores event orderings with uncommon and unexpected delays aiming to uncover race conditions. For example, if an event handler in the controller installs rules in switches 1, 2, and 3, the heuristic explores transitions that reverse the order by allowing switch 3 to install its rule first, followed by switch 2 and then switch 1. This heuristic uncovers bugs like the example in Figure 1.

FLOW-IR: Flow independence reduction. Many OpenFlow applications treat different groups of packets independently; that is, the handling of one group is not affected by the presence or absence of another. In this case, NICE can reduce the search space by exploring only one relative ordering between the events affecting each group. To use this heuristic, the programmer provides `isSameFlow`, a Python function that takes two packets as arguments and returns whether the packets belong to the same group. For example, in some scenarios different microflows are independent, whereas other programs may treat packets with different destination MAC addresses independently.

Summary. PKT-SEQ is complementary to other strategies in that it only reduces the number of `send` transitions rather than the possible kind of event orderings. It is enabled by default and used in our experiments (unless otherwise noted). The other heuristics can be selectively enabled.

6. Specifying Application Correctness

Correctness is not an intrinsic property of a system—a specification of correctness states what the system should do, whereas the implementation determines what it actually does. NICE allows programmers to define correctness properties as Python code snippets, and provides a library of common properties (*e.g.*, no loops or black holes).

6.1. Customizable Correctness Properties

Testing correctness involves asserting safety properties (*"something bad never happens"*) and liveness properties (*"eventually something good happens"*), defined more formally in Chapter 3 of [22]. Checking for safety properties is relatively easy, though sometimes writing an appropriate predicate over all state variables is tedious. As a simple example, a predicate could check that the collection of flow rules does not form a forwarding loop or a black hole. Checking for liveness properties is typically harder because of the need to consider a possibly infinite system execution. In NICE, we make the inputs finite (*e.g.*, a finite number of packets, each with a finite set of possible header values), allowing us to check some liveness properties. For example, NICE could check that, once two hosts exchange at least one packet in each direction, no further packets go to the controller (a property we call "StrictDirectPaths"). Checking this liveness property requires knowledge not only of the system state, but also which transitions were executed.

To check safety and liveness properties, NICE allows correctness properties to (*i*) access the system state, (*ii*) register callbacks invoked by NICE to observe important transitions in system execution, and (*iii*) maintain local state. In our experience, these features offer enough expressiveness. For ease of implementation, the properties are represented as snippets of Python code that make assertions about system state. NICE invokes these snippets after each transition. For example, to check the StrictDirectPaths property, the code snippet defines local state variables that keep track of whether a pair of hosts has exchanged at least one packet in each direction, and would flag a violation if a subsequent packet triggers a `packet_in` event. When a correctness check signals a violation, NICE records the execution trace that recreates the problem.

6.2. Library of Correctness Properties

NICE provides a library of correctness properties applicable to a wide range of OpenFlow applications. A programmer can select properties from a list, as appropriate for the application. Writing these correctness modules can be challenging because the definitions must be robust to communication delays between the switches and the controller. Many of the definitions must intentionally wait until a "safe" time to test the property to prevent natural delays from erroneously triggering a violation. Providing these modules as part of NICE can relieve the developers from the challenges of specifying correctness properties precisely, though creating any custom modules would require similar care.

• *NoForwardingLoops*: This property asserts that packets do not encounter forwarding loops. It is im-

plemented by checking if each packet goes through any <switch, input port> pair at most once.

• *NoBlackHoles*: This property states that no packets should be dropped in the network, and is implemented by checking that every packet that enters the network and is destined to an existing host, ultimately leaves the network (for simplicity, we disable optional packet drops and duplication on the channels). In case of host mobility, an extended version of this property: *NoBlackHolesMobile* ignores the inevitably dropped packets in the period from when the host moves to when the controller can realize that the host moved.

• *DirectPaths*: This property checks that, once a packet has successfully reached its destination, future packets of the same flow do not go to the controller. Effectively, this checks that the controller successfully establishes a direct path to the destination as part of handling the first packet of a flow. This property is useful for many OpenFlow applications, though it does not apply to the MAC-learning switch, which requires the controller to learn how to reach both hosts before it can construct unicast forwarding paths in either direction.

• *StrictDirectPaths*: This property checks that, after two hosts have successfully delivered at least one packet of a flow in each direction, no successive packets reach the controller. This checks that the controller has established a direct path in *both* directions between the two hosts.

• *NoForgottenPackets*: This property checks that all switch buffers are empty at the end of system execution. A program can easily violate this property by forgetting to tell the switch how to handle a packet. This can eventually consume all the available buffer space for packets awaiting controller instruction; after a timeout, the switch may discard these buffered packets[3]. A short-running program may not run long enough for the queue of awaiting-controller-response packets to fill, but the *NoForgottenPackets* property easily detects these bugs. Note that a violation of this property often leads also to a violation of *NoBlackHoles*.

[3]In our tests of the ProCurve 5406zl OpenFlow switch, we see that, once the buffer becomes full, the switch starts sending the *entire contents* of new incoming packets to the controller, rather than buffering them. After a ten-second timeout, the switch deletes the packets that are buffered awaiting instructions from the controller.

7. Implementation Highlights

We have built a prototype implementation of NICE written in Python so as to seamlessly support OpenFlow controller programs for the popular NOX platform (which provides an API for Python).

As a result of using Python, we face the challenge of doing symbolic execution for a dynamically typed language. This task turned out to be quite challenging from an implementation perspective. To avoid modifying the Python interpreter, we implement a derivative technique of symbolic execution called *concolic execution* [28][4], which executes the code with concrete instead of symbolic inputs. Like symbolic execution, it collects constraints along code paths and tries to explore all feasible paths. Another consequence of using Python is that we incur a significant performance overhead, which is the price for favoring usability. We plan to improve performance in a future release of the tool.

NICE consists of three parts: (*i*) a model checker, (*ii*) a concolic-execution engine, and (*iii*) a collection of models including the simplified switch and several end hosts. We now briefly highlight some of the implementation details of the first two parts: the model checker and concolic engine, which run as different processes.

Model checker details. To checkpoint and restore system state, NICE in the basic version takes the approach of remembering the sequence of transitions that created the state and restores it by replaying such sequence, while leveraging the fact that the system components execute deterministically. State-matching is done by comparing and storing hashes of the explored states. The main benefit of this approach is that it reduces memory consumption and, secondarily, it is simpler to implement. Trading computation for memory is a common approach for other model-checking tools (*e.g.*, [18, 19]). To create state hashes, NICE serializes the state via the json module and applies the built-in hash function to the resulting string.

In the extended version, NICE stores the serialized states themselves, at the cost of higher memory usage. This approach has a potential to reduce NICE running time, but the exact benefits depend mostly on the time required to save and restore the controller state. Moreover, saving the state required for DPOR is challenging and in the current prototype we do not support it.

[4]Concolic stands for concrete + symbolic.

	NICE-MC			NO-SWITCH-REDUCTION			
Pings	**Transitions**	**Unique states**	**CPU time**	**Transitions**	**Unique states**	**CPU time**	ρ
2	530	315	1.1 [s]	706	432	1.72 [s]	0.27
3	14,762	6,317	37.24 [s]	31,345	13,940	92.9 [s]	0.54
4	356,469	121,320	17 [m]	1,134,515	399,919	59 [m]	0.69
5	7,816,517	2,245,345	8 [h]	33,134,573	9,799,538	57 [h]	0.77

Table 1: Dimensions of exhaustive search in NICE-MC vs. model-checking without a canonical representation of the switch state, which prevents recognizing equivalent states. Symbolic execution is turned off in both cases.

Concolic execution details. A key step in concolic execution is tracking the constraints on symbolic variables during code execution. To achieve this, we first implement a new "symbolic integer" data type that tracks assignments, changes and comparisons to its value while behaving like a normal integer from the program point of view. We also implement arrays (tuples in Python terminology) of these symbolic integers. Second, we reuse the Python modules that naturally serve for debugging and disassembling the byte-code to trace the program execution through the Python interpreter.

Further, before running the code symbolically, we normalize and instrument it since, in Python, the execution can be traced at best with single code-line granularity. Specifically, we convert the source code into its abstract syntax tree (AST) representation and then manipulate this tree through several recursive passes that perform the following transformations: (*i*) we split composite branch predicates into nested if statements to work around shortcut evaluation, (*ii*) we move function calls before conditional expressions to ease the job for the STP constraint solver [29], (*iii*) we instrument branches to inform the concolic engine on which branch is taken, (*iv*) we substitute the built-in dictionary with a special stub to track uses of symbolic variables, and (*v*) we intercept and remove sources of nondeterminism (*e.g.*, seeding the pseudo-random number generator). The AST tree is then converted back to source code for execution.

8. Performance Evaluation

Here we present an evaluation of how effectively NICE copes with the large state space in OpenFlow.

Experimental setup. We run the experiments on the simple topology of Figure 1, where the end hosts behave as follows: host A sends a "*layer-2 ping*" packet to host B which replies with a packet to A. The controller runs the MAC-learning switch program of Figure 3. We report the numbers of transitions and unique states, and the execution time as we increase the number of concurrent pings (a pair of packets). We run all our experiments on a machine with Linux 3.8.0 x86_64 that has 64 GB of RAM and a clock speed of 2.6 GHz. Our prototype does not yet make use of multiple cores.

Benefits of simplified switch model. We first perform a full search of the state space using NICE as a depth-first search model checker (NICE-MC, without symbolic execution) and compare to NO-SWITCH-REDUCTION: doing model-checking without a canonical representation of the switch state. Effectively, this prevents the model checker from recognizing that it is exploring semantically equivalent states. These results, shown in Table 1, are obtained without using any of our search strategies. We compute ρ, a metric of state-space reduction due to using the simplified switch model, as $\frac{Unique(\text{NO-SWITCH-REDUCTION}) - Unique(\text{NICE-MC})}{Unique(\text{NO-SWITCH-REDUCTION})}$. We observe the following:

- In both samples, the number of transitions and of unique states grow roughly exponentially (as expected). However, using the simplified switch model, the unique states explored in NICE-MC only grow with a rate that is lower than the one observed for NO-SWITCH-REDUCTION.
- The efficiency in state-space reduction ρ scales with the problem size (number of pings), and is substantial (factor of 3.5 for four pings).
- The time taken to complete a full state-space search in this small-scale example grows exponentially with the number of packets. However, thanks to symbolic execution, we expect NICE to explore most of the system states using a modest number of symbolic packets and even small network models.

Heuristic-based search strategies.

Figure 7 illustrates the contribution of NO-DELAY and UNUSUAL in reducing the search space relative to the metrics reported for the full search (NICE-MC). The state space reduction is again significant; about factor of five and factor of ten for over two pings with UNUSUAL and NO-DELAY respectively. In summary, our switch

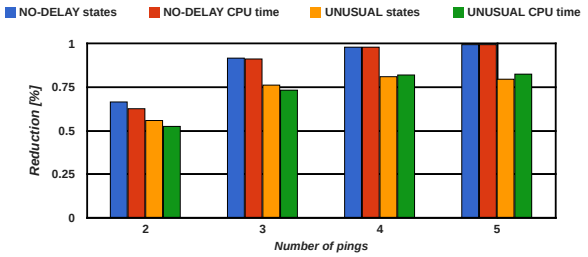
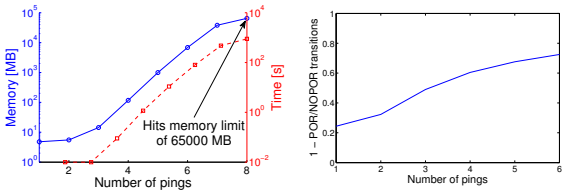


Figure 7: Relative state-space search reduction of our heuristic-based search strategies vs. NICE-MC.



(a) Memory usage and elapsed time (log y-scales).

(b) Efficiency of POR.

Figure 8: SPIN: Exponential increase in computational resources partially mitigated by POR.

model and these heuristics result in over 20-fold state space reduction for four and more pings. FLOW-IR is unable to reduce the state space in a network with two end hosts. Because of the way the PySwitch works, all flows are dependent. However, in Table 5 we show the effectiveness of this strategy in other scenarios (even 10 times reduction).

Comparison to other model checkers. Next, we contrast NICE-MC with two state-of-the-art model checkers, SPIN [15] and JPF [16]. We create system models in these tools that replicate as closely as possible the system tested in NICE, using the same experimental setup as with our heuristics.

SPIN is one of the most popular tools for verifying the correctness of software models, which are written in a high-level modeling language called PROMELA. This language exposes nondeterminism as a first-class concept, making it easier to model the concurrency in OpenFlow. However, using this language proficiently is non-trivial and it took several person-days to implement the model of the simple OpenFlow system (Figure 1). To capture the system concurrency at the right level of granularity, we use the `atomic` language feature to model each transition as a single atomic computation that cannot be interleaved to any other transition. In practice, this behavior cannot be faithfully modeled due to the blocking nature of `channels` in PROMELA. To enable SPIN's POR to be most effective, we assign exclusive rights to the processes involved in each communication channel.

Figure 8a shows the memory usage and elapsed time for the exhaustive search with POR as we increase the number of packets sent by host A. As expected, we observe an exponential increase in computational resources until SPIN reaches the memory limit when checking the model with 8 pings (*i.e.*, 16 packets).

To see how effective POR is, we compare in Figure 8b the number of transitions explored with POR vs. without POR (NOPOR) while we vary the number of pings. In relative terms, POR's efficiency increases, although with diminishing returns, from 24% to 73% as we inject more packets that are identical to each other. The benefits due to POR on elapsed time follow a similar trend and POR can finish 6 pings in 28% of time used by NOPOR. However, NOPOR hits the memory limit at 7 pings, so POR only adds one extra ping.

Finally, we test if POR can reduce the search space by taking advantage of the simple independence property as in FLOW-IR. Unfortunately, we observe that there is no reduction when we inject two packets with distinct address pairs compared to the case with identical packets. This is because SPIN uses the accesses to communication channels to derive the independence of events. Our DPOR algorithm instead considers a more fine-grained definition of shared objects and achieves better state space reduction.

Java PathFinder (JPF) is one among the first modern model checkers which use the implementation in place of the model. We follow two approaches to model the system by porting our Python code to Java.

In the first approach, we naively use threads to capture nondeterminism, hoping that JPF's automatic state-space reduction techniques would cope with different thread creation orders of independent transitions. However, in our case, the built-in POR is not very efficient in removing unnecessary network event interleavings because thread interleaving happens at finer granularity than event interleavings. To solve this problem, we tune this model by using the `beginAtomic()` and `endAtomic()` functions provided by JPF. As this still produces too many possible interleavings, we further introduced a global lock.

In a second approach to further refine the model,

Pings	Time [s]	Unique states	End states	Mem. [MB]
1	0	55	2	17
2	9	20638	134	140
3	13689	25470986	2094	1021

Table 2: JPF: Exhaustive search on thread-based model.

Pings	Time [s]	Unique states	End states	Mem. [MB]
1	0	1	1	17
2	1	691	194	33
3	16	29930	6066	108
4	11867	16392965	295756	576

Table 3: JPF: Exhaustive search on choice-based model.

we capture nondeterminism via JPF's choice generator: `Verify.getInt()`. This gives a significant improvement over threads, mainly because we are able to specify precisely the granularity of interleavings. However, this second modeling effort is non trivial since we are manually enumerating the state space and there are several caveats in this case too. For example, explicit choice values should not be saved on the stack as the choice value may became a part of the global state, thus preventing reduction. The vector of possible transitions must also be sorted[5].

Table 2 illustrates the state space explosion when using the thread-based model. Unfortunately, as show in Table 3, the choice-based model improves only by 1 ping the size of the model that we can explore within a comparable time period ($\approx$ 4 hours).

These results suggest that NICE, in comparison with the other model-checkers, strikes a good balance between (*i*) capturing system concurrency at the right level of granularity, (*ii*) simplifying the state space and (*iii*) allowing testing of unmodified controller programs.

9. Experiences with Real Applications

In this section, we report our experience with applying NICE to three real applications—a MAC-learning switch, a server load-balancer, and energy-aware traffic engineering—and uncovering *13* bugs. In all experiments, it was sufficient to use a network model with at most three switches.

9.1. MAC-learning Switch (PySwitch)

Our first application is the `pyswitch` software included in the NOX distribution (98 LoC). The application implements MAC learning, coupled with flooding to unknown destinations, common in Ethernet switches. Realizing this functionality seems straightforward (*e.g.*, the pseudo-code in Figure 3), yet NICE automatically detects three violations of correctness properties.

BUG-I: Host unreachable after moving. This fairly subtle bug is triggered when a host B moves from one location to another. Before B moves, host A starts streaming to B, which causes the controller to install a forwarding rule. When B moves, the rule stays in the switch as long as A keeps sending traffic, because the soft timeout does not expire. As such, the packets do not reach B's new location. This serious correctness bug violates the *NoBlackHoles* and *NoBlackHolesMobile* properties. If the rule had a *hard* timeout, the application would eventually flood packets and reach B at its new location; then, B would send return traffic that would trigger MAC learning, allowing future packets to follow a direct path to B. While this "bug fix" prevents persistent packet loss, the network still experiences *transient* loss until the hard timeout expires.

BUG-II: Delayed direct path. The `pyswitch` also violates the *StrictDirectPaths* property, leading to suboptimal performance. The violation arises after a host A sends a packet to host B, and B sends a response packet to A. This is because `pyswitch` installs a forwarding rule in one direction—from the sender (B) to the destination (A), in line 13 of Figure 3. The controller does *not* install a forwarding rule for the other direction until seeing a subsequent packet from A to B. For a three-way packet exchange (*e.g.*, a TCP handshake), this performance bug directs 50% more traffic than necessary to the controller. Anecdotally, fixing this bug can easily introduce another one. The naïve fix is to add another `install_rule` call, with the addresses and ports reversed, after line 14, for forwarding packets from A to B. However, since the two rules are not installed atomically, installing the rules in this order can allow the packet from B to reach A before the switch installs the second rule. This can cause a subsequent packet from A to reach the controller unnecessarily. A correct fix installs the rule for traffic from A first, before allowing the packet from B to A to traverse the switch. With this "fix", the program satisfies the *StrictDirectPaths* property.

BUG-III: Excess flooding. When we test `pyswitch` on a topology that contains a cycle, the program violates the *NoForwardingLoops* property. This is not surprising, since `pyswitch` does not con-

[5] We order events by their states' hash values.

struct a spanning tree.

9.2. Web Server Load Balancer

Data centers rely on load balancers to spread incoming requests over service replicas. Previous work created a load-balancer application that uses wildcard rules to divide traffic based on the client IP addresses to achieve a target load distribution [8]. The application dynamically adjusts the load distribution by installing new wildcard rules; during the transition, old transfers complete at their existing servers while new requests are handled according to the new distribution. We test this application with one client and two servers connected to a single switch. The client opens a TCP connection to a virtual IP address corresponding to the two replicas. In addition to the default correctness properties, we create an application-specific property *FlowAffinity* that verifies that all packets of a single TCP connection go to the same replica. Here we report on the bugs NICE found in the original code (1209 LoC), which had already been unit tested to some extent.

BUG-IV: ARP packets forgotten during address resolution. Having observed a violation of the *NoForgottenPackets* property for ARP packets, we identified two bugs. The controller program handles client ARP requests on behalf of the server replicas. Despite sending the correct reply, the program neglects to discard the ARP request packets from the switch buffer. A similar problem occurs for server-generated ARP messages.

BUG-V: TCP packets always dropped before the first reconfiguration. A violation of the *NoForgottenPackets* property for TCP packets allowed us to detect a problem where the controller ignores all `packet_in` messages for TCP packets caused by no matching rule at the switch. As before the first reconfiguration there are no rules installed, all flows that start during this period are ignored. Dropping such packets is understandable as the controller may have insufficient information about the replicas. However, ignoring them completely occupies space in switch buffers.

BUG-VI: Next TCP packet always dropped after reconfiguration. Having observed another violation of the *NoForgottenPackets* property, we identified a bug where the application neglects to handle the "next" packet of each flow—for both ongoing transfers and new requests—after any change in the load-balancing policy. Despite correctly installing the forwarding rule for each flow, the application does *not* instruct the switch to forward the packet that triggered the `packet_in` handler. Since the TCP sender ultimately retransmits the lost packet, the program does successfully handle each Web request, making it hard to notice this bug that degrades performance and, for a long execution, would ultimately exhaust the switch's space for buffering packets awaiting controller action.

BUG-VII: Some TCP packets dropped during reconfiguration. After fixing previously described bugs, NICE detected another *NoForgottenPackets* violation due to a race condition. In switching from one load-balancing policy to another, the application sends multiple updates to the switch for each existing rule: (i) a command to remove the existing forwarding rule followed by (ii) commands to install one or more rules (one for each group of affected client IP addresses) that direct packets to the controller. Since these commands are not executed atomically, packets arriving between the first and second step do not match either rule. The OpenFlow specification prescribes that packets that do not match any rule should go to the controller. Although the packets go to the controller either way, these packets arrive with a different "reason code" (*i.e.*, `NO_MATCH`). As written, the `packet_in` handler ignores such (unexpected) packets, causing the switch to hold them until the buffer fills. This appears as packet loss to the end hosts[6]. To fix this bug, the program should reverse the two steps, installing the new rules (perhaps at a lower priority) before deleting the existing ones. Finding this bug poses another challenge: only a detailed analysis of event sequences allows us to distinguish it from **BUG-V**. Moreover, a proper fix of **BUG-V** hides all symptoms of **BUG-VII**.

BUG-VIII: Incomplete packets encapsulated in `packet_in` messages. The final *NoForgottenPackets* property violation was caused by a change in the OpenFlow specification before version 1.0. If a rule's action is to send packets to the controller, the action needs to define the maximum number of packet bytes that should be encapsulated in an `packet_in` message. The controller uses value 0, which in the initial versions of the specification means encapsulate the entire packet. However, in OpenFlow 1.0, 0 is no longer a special value. As a result, the controller does not receive any bytes of

[6] To understand the impact, consider a switch with 1 Gb/s links, 850-byte frames, and a flow-table update rate of 257 rules/s (as widely reported for the HP 5406zl). That would lead to 150 dropped packets per switch port.

the packet header and is unable to analyze it.

BUG-IX: Duplicate SYN packets during transitions. A *FlowAffinity* violation detected a subtle bug that arises only when a connection experiences a duplicate (*e.g.*, retransmitted) SYN packet while the controller changes from one load-balancing policy to another. During the transition, the controller inspects the "next" packet of each flow, and assumes a SYN packet implies the flow is new and should follow the new load-balancing policy. Under duplicate SYN packets, some packets of a connection (arriving before the duplicate SYN) may go to one server, and the remaining packets to another, leading to a broken connection. The authors of [8] acknowledged this possibility (see footnote #2 in their paper), but only realized this was a problem after careful consideration.

9.3. Energy-Efficient Traffic Engineering

OpenFlow enables a network to reduce energy consumption [9, 30] by selectively powering down links and redirecting traffic to alternate paths during periods of lighter load. REsPoNse [30] precomputes several routing tables (the default is two), and makes an online selection for each flow. The NOX implementation (374 LoC) has an *always-on* routing table (that can carry all traffic under low demand) and an *on-demand* table (that serves additional traffic under higher demand). Under high load, the flows should probabilistically split evenly over the two classes of paths. The application learns the link utilizations by querying the switches for port statistics. Upon receiving a packet of a new flow, the `packet_in` handler chooses the routing table, looks up the list of switches in the path, and installs a rule at each hop.

For testing with NICE, we install a network topology with three switches in a triangle, one sender host at one switch and two receivers at another switch. The third switch lies on the on-demand path. We define the following application-specific correctness property:

• *UseCorrectRoutingTable*: This property checks that the controller program, upon receiving a packet from an ingress switch, issues the installation of rules to all and just the switches on the appropriate path for that packet, as determined by the network load. Enforcing this property is important, because if it is violated, the network might be configured to carry more traffic than it physically can, degrading the performance of end-host applications running on top of the network.

NICE found several bugs in this application:

BUG-X: The first packet of a new flow is dropped. A violation of *NoForgottenPackets* and *NoBlackHoles* revealed a bug that is almost identical to **BUG-VI**. The `packet_in` handler installed a rule but neglected to instruct the switch to forward the packet that triggered the event.

BUG-XI: The first few packets of a new flow can be dropped. After fixing **BUG-X**, NICE detected another *NoForgottenPackets* violation at the second switch in the path. Since the `packet_in` handler installs an end-to-end path when the first packet of a flow enters the network, the program implicitly assumes that intermediate switches would never direct packets to the controller. However, with communication delays in installing the rules, the packet could reach the second switch before the rule is installed. Although these packets trigger `packet_in` events, the handler implicitly ignores them, causing the packets to buffer at the intermediate switch. This bug is hard to detect because the problem only arises under certain event orderings. Simply installing the rules in the reverse order, from the last switch to the first, is not sufficient—differences in the delays for installing the rules could still cause a packet to encounter a switch that has not (yet) installed the rule. A correct "fix" should either handle packets arriving at intermediate switches, or use "barriers" (where available) to ensure that rule installation completes at all intermediate hops before allowing the packet to depart the ingress switch.

BUG-XII: Only on-demand routes used under high load. NICE detects a *CorrectRoutingTableUsed* violation that prevents on-demand routes from being used properly. The program updates an extra routing table in the port-statistic handler (when the network's perceived energy state changes) to either always-on or on-demand, in an effort to let the remainder of the code simply reference this extra table when deciding where to route a flow. Unfortunately, this made it impossible to split flows equally between always-on and on-demand routes, and the code directed all new flows over on-demand routes under high load. A "fix" was to abandon the extra table and choose the routing table on a per-flow basis.

BUG-XIII: Packets can be dropped when the load reduces. After fixing **BUG-XI**, NICE detected another violation of the *NoForgottenPackets*. When the load reduces, the program recomputes the list of switches in each always-on path. Un-

der delays in installing rules, a switch not on these paths may send a packet to the controller, which ignores the packet because it fails to find this switch in any of those lists.

9.4. Overhead of Running NICE

In Table 4, we summarize how many seconds NICE took (and how many state transitions were explored) to discover the *first property violation* that uncovered each bug, under four different search strategies with and without DPOR. Note the numbers are generally small because NICE quickly produces simple test cases that trigger the bugs. One exception, **BUG-IX**, is found in 1 hour by doing a PKT-SEQ-only search but NO-DELAY and UNUSUAL can detect it in just 3-8 minutes. Our search strategies are also generally faster than PKT-SEQ-only to trigger property violations, except in one case (**BUG-VI**). Adding DPOR improves all strategies unless the bug is found on one of the first explored paths. Also, note that there are no false positives in our case studies—every property violation is due to the manifestation of a bug—and only in few cases (**BUG-VII**, **BUG-IX**, **BUG-XI** and **BUG-XIII**) the heuristic-based strategies experience false negatives. Expectedly, NO-DELAY, which does not consider rule installation delays, misses race condition bugs (23% missed bugs). **BUG-IX** is missed by FLOW-IR because the duplicate SYN is treated as a new independent flow (8% missed bugs).

Finally, the reader may find that some of the bugs we found—like persistently leaving some packets in the switch buffer—are relatively simple and their manifestations could be detected with run-time checks performed by the controller platform. However, the programmer would not know what caused them. For example, a run-time check that flags a "no forgotten packets" error due to **BUG-VI** or **BUG-VII** would not tell the programmer what was special about this particular execution that triggered the error. Subtle race conditions are hard to diagnose, so having a (preferably small) example trace—like NICE produces—is crucial.

9.5. Effectiveness of Optimizations

Until now we reported only times to find the first invariant violation, which is critical when looking for bugs. However, to fully evaluate various optimizations described earlier, we disable all invariants and in Table 5 present a total number of transitions and running time for three configurations: MAC-learning controller like in Section 8 with 4 pings, Load Balancer with one connection, and Energy-Efficient Traffic Engineering with two connections (like for **BUG-XIII**).

First, state serialization improves the performance and the improvement depends on the complexity of serializing the controller state. Load Balancer has a more complex state than the other two. For the controllers with a simpler state, the state serialization allows to finish the state space exploration even in less than 60% of the original time.

DPOR reduces the number of transitions and states that the model checker needs to explore, however, it comes with two sources of overhead: (i) it performs additional computations, and (ii) in our implementation DPOR works does not work with state serialization. That is the reason why in a network where many transitions are dependent and where serializing the controller is simple (learning switch) the benefits of using DPOR are smaller than the costs. On the other hand, with the Load Balancer, DPOR reduces the number of explored transitions even 9 times which leads to even 10 times shorter exploration time. For REsPoNse the difference is smaller, but still meaningful: over 4 times.

10. Coverage vs. Overhead Trade-Offs

Testing is inherently incomplete, walking a fine line between good coverage and low overhead. Here we discuss some limitations of our approach.

Concrete execution on the switch: In identifying the equivalence classes of packets, the algorithm in Figure 5 implicitly assumes the packets reach the controller. However, depending on the rules already installed in the switch, some packets in a class may reach the controller while others do not. This leads to two limitations. First, if *no* packets in an equivalence class go to the controller, generating a representative packet from this class was unnecessary. This leads to some loss in efficiency. Second, if *some* (but not all) packets go to the controller, we may miss an opportunity to test a code path through the handler by inadvertently generating a packet that stays in the "fast path" through the switches. This causes some loss in both efficiency and coverage. We could overcome these limitations by extending symbolic execution to include our simplified switch model and performing "symbolic packet forwarding" across multiple

Bug	only PKT-SEQ		NO-DELAY		FLOW-IR		UNUSUAL	
	no DPOR	DPOR	no DPOR	DPOR	no DPOR	DPOR	no DPOR	DPOR
I	2149/8.3	1631/7.14	418/1.6	355/1.34	2149/8.55	1631/7.61	994/3.88	819/3.17
II	540/1.65	415/1.18	187/0.62	156/0.4	540/1.70	415/1.21	241/0.8	179/0.45
III	23/0.22	23/0.16	17/0.2	17/0.16	23/0.21	23/0.16	24/0.2	24/0.18
IV	49/0.61	49/0.36	49/0.58	49/0.32	49/0.53	49/0.47	30/0.36	30/0.26
V	52/0.59	52/0.33	52/0.53	52/0.37	52/0.59	52/0.38	33/0.5	33/0.33
VI	738/6.65	312/2.25	1977/15.35	361/1.8	778/10.0	306/2.31	139/1.47	208/1.24
VII	12k/93.14	1782/13.26	Missed	Missed	481/4.0	385/2.28	112/1.5	49/0.38
VIII	1274/12.12	245/3.74	1237/10.31	265/1.33	1134/11.22	278/2.15	709/6.81	200/1.54
IX	432.7k/66m	33.7k/324	33.5k/213	15.6k/111	Missed	Missed	53.5k/478	13.4k/138
X	22/0.31	22/0.19	22/0.24	22/0.21	22/0.24	22/0.21	22/0.24	22/0.19
XI	97/0.97	53/0.63	Missed	Missed	97/0.98	53/0.61	24/0.25	24/0.2
XII	19/0.27	19/0.21	17/0.22	18/0.17	19/0.23	19/0.21	19/0.25	19/0.19
XIII	3126/24.86	697/6.59	Missed	Missed	2287/18.54	617/5.59	1225/11.24	589/4.86

Table 4: Comparison of the number of transitions / running time to the first violation that uncovered each bug. Time is in seconds unless otherwise noted. For **BUG-VII** transitions are explored in the reverse order, so that this bug gets detected before **BUG-V**.

	PySwitch		Load Balancer		REsPoNse	
Strategy	Transitions	Time [s]	Transitions	Time [s]	Transitions	Time [s]
PKT-SEQ	356,469	1,739.42	84,831	486.88	62,152	541.32
PKT-SEQ + serialize	356,469	1,051.89	84,831	428.86	62,152	316.38
PKT-SEQ + DPOR	253,511	1,604.62	9,206	47.59	14,549	116.70
NO-DELAY	6,385	19.87	7,663	22.76	2,012	12.17
NO-DELAY + serialize	6,385	15.31	7,663	34.53	2,012	8.37
NO-DELAY + DPOR	4,962	20.22	2,841	9.15	834	5.78
FLOW-IR	356,469	1,587.34	8,636	41.02	32,352	251.76
FLOW-IR + serialize	356,469	1,058.00	8,636	40.35	32,352	167.88
FLOW-IR + DPOR	253,511	1,468.23	1,420	6.24	8,447	62.28
UNUSUAL	67,816	258.83	4,716	21.62	3,544	21.79
UNUSUAL + serialize	67,816	186.69	4,716	22.20	3,544	17.22
UNUSUAL + DPOR	21,659	111.93	1,406	8.11	1,072	6.12

Table 5: Comparison of number of transitions and running time when using basic NICE, NICE with state serialization and NICE with DPOR. Except for the Load Balancer application for which serializing the controller state takes a lot of time, state serialization significantly reduces running time. DPOR gives additional except for the PySwitch application where many transitions are dependent and a small reduction in explored states is insufficient to balance the cost of running DPOR and the time advantage of serialization.

switches. We chose not to pursue this approach because (*i*) symbolic execution of the flow-table code would lead to a path-explosion problem, (*ii*) including these variables would increase the overhead of the constraint solver, and (*iii*) rules that modify packet headers would further complicate the symbolic analysis.

Concrete global controller variables: In symbolically executing each event handler, NICE could miss complex dependencies *between* handler invocations. This is a byproduct of our decision to represent controller variables in a concrete form. In some cases, one call to a handler could update the variables in a way that affects the symbolic execution of a second call (to the same handler, or a different one). Symbolic execution of the second handler would start from the *concrete* global variables, and may miss an opportunity to recognize additional constraints on packet header fields. We could overcome this limitation by running symbolic execution across multiple handler invocations, at the expense of a significant explosion in the number of code paths. Or, we could revisit our decision to represent controller variables in a concrete form.

Infinite execution trees in symbolic execution: Despite its many advantages, symbolic execution can lead to infinite execution trees [26]. In our context, an infinite state space arises if each state has at least one input that modifies the controller state. This is an inherent limitation of symbolic execution, whether applied independently or in conjunction with model checking. To address this limitation, we explicitly bound the state space by limiting the size of the input (*e.g.*, a limit on the number of packets) and devise OpenFlow-specific search strategies that explore the system state space efficiently. These heuristics offer a tremendous improvement in efficiency, at the expense of some loss

in coverage.

Finally, there are two main sources of coverage incompleteness: (*i*) heuristic-driven and bounded-depth model checking, and (*ii*) incomplete symbolic execution of the controller code. We showed in Section 9 that at least one heuristic was always able to detect each bug. We do not report the code coverage of the controller because symbolic execution applies to event handlers that are a subset of the actual application logic, making it is difficult to distinguish between this logic and the rest of the system. Second, there are many hidden branches (*e.g.*, in dictionaries) that are not visible with code coverage statistics.

11. Related Work

Bug finding. While model checking [18, 17, 15, 16, 19] and symbolic execution [28, 20, 21] are automatic techniques, a drawback is that they typically require a closed system, *i.e.*, a system (model) together with its environment. Typically, the creation of such an environment is a manual process (*e.g.*, [25]). NICE re-uses the idea of model checking—systematic state-space exploration—and combines it with the idea of symbolic execution—exhaustive path coverage—to avoid pushing the burden of modeling the environment on the user. Also, NICE is the first to demonstrate the applicability of these techniques for testing the dynamic behavior of OpenFlow networks. Finally, NICE makes a contribution in managing state-space explosion for this specific domain.

Khurshid *et al.* [26] enable a model checker to perform symbolic execution. Both our and their work share the spirit of using symbolic variables to represent data from very large domains. Our approach differs in that it uses symbolic execution in a selective way for uncovering possible transitions given a certain controller state. As a result, we (*i*) reduce state-space explosion due to feasible code paths because not all code is symbolically executed, and (*ii*) enable matching of concrete system states to further reduce the search of the state space.

Software-defined networking. Frenetic [14], Maple [12], and FlowLog [13] are domain-specific languages for SDN. The higher-level abstractions in these works make it possible to eradicate certain classes of programming faults and to some extent enable controller verification. However, these languages have not yet seen wide adoption.

OFRewind [31] enables recording and replay of events for troubleshooting problems in production networks due to closed-source network devices. However, it does not automate the testing of OpenFlow controller programs.

Mai *et al.* [32] use static analysis of network devices' forwarding information bases to uncover problems in the data plane. FlowChecker [33] applies symbolic model checking techniques on a manually-constructed network model based on binary decision diagrams to detect misconfigurations in OpenFlow forwarding tables. We view these works as orthogonal to ours since they both aim to analyze a snapshot of the data plane.

Bishop *et al.* [34] examine the problem of testing the specification of end host protocols. NICE tests the network itself, in a new domain of software defined networks. Kothari *et al.* [35] use symbolic execution and developer input to identify protocol manipulation attacks for network protocols. In contrast, NICE combines model checking with symbolic execution to identify relevant test inputs for injection into the model checker.

Several recent works have considered the problem of checking the correctness in SDN under dynamic controller behavior. Sethi *et al.* [36] present data- and network-state abstractions for model checking SDN controllers. Their approach extends verification to an arbitrarily number of packets by considering only one concrete packet for the verification task. This reduces the state space but it also limits the invariants that can be checked to just per-packet safety properties. Kuai [37] introduces a set of partial order reduction techniques to reduce the state space. VeriCon [38] extends verification of SDN programs to check their correctness on all admissible topologies and for all possible sequences of network events. These approaches need to manually port the controller application to a different programming language and do not use symbolic execution to reduce the space of input packets.

12. Conclusion

We built NICE, a tool for automating the testing of OpenFlow applications that combines model checking and concolic execution in a novel way to quickly explore the state space of unmodified controller programs written for the popular NOX platform. Further, we devised a number of new, domain-specific techniques for mitigating the state-space explosion that plagues approaches such as

ours. We contrast NICE with an approach that applies off-the-shelf model checkers to the OpenFlow domain, and demonstrate that NICE is five times faster even on small examples. We applied NICE to implementations of three important applications, and found 13 bugs. A release of NICE is publicly available at https://github.com/mcanini/nice.

Acknowledgments. We are grateful to Stefan Bucur, Olivier Crameri, Johannes Kinder, Viktor Kuncak, Sharad Malik, and Darko Marinov for useful discussions and comments on earlier drafts of this work. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

References

[1] Dyn Research (formerly Renesys), AfNOG Takes Byte Out of Internet, http://research.dyn.com/2009/05/byte-me/ (2009).

[2] Dyn Research (formerly Renesys), Reckless Driving on the Internet, http://research.dyn.com/2009/02/the-flap-heard-around-the-world/ (2009).

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling Innovation in Campus Networks, SIGCOMM Comput. Commun. Rev. 38 (2008) 69–74.

[4] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, NOX: Towards an Operating System for Networks, SIGCOMM Comput. Commun. Rev. 38 (2008) 105–110.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, S. Shenker, Rethinking Enterprise Network Control, IEEE/ACM Transactions on Networking 17 (4).

[6] A. Nayak, A. Reimers, N. Feamster, R. Clark, Resonance: Dynamic Access Control for Enterprise Networks, in: WREN, 2009.

[7] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, R. Johari, Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow, SIGCOMM Demo (Aug. 2009).

[8] R. Wang, D. Butnariu, J. Rexford, OpenFlow-Based Server Load Balancing Gone Wild, in: Hot-ICE, 2011.

[9] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, N. McKeown, ElasticTree: Saving Energy in Data Center Networks, in: NSDI, 2010.

[10] D. Erickson, G. Gibb, B. Heller, D. Underhill, J. Naous, G. Appenzeller, G. Parulkar, N. McKeown, M. Rosenblum, M. Lam, S. Kumar, V. Alaria, P. Monclus, F. Bonomi, J. Tourrilhes, P. Yalagandula, S. Banerjee, C. Clark, R. McGeer, A Demonstration of Virtual Machine Mobility in an OpenFlow Network, SIGCOMM Demo (Aug. 2008).

[11] M. Canini, D. Kostić, J. Rexford, D. Venzano, Automating the Testing of OpenFlow Applications, in: WRiPE, 2011.

[12] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, P. Hudak, Maple: Simplifying SDN Programming Using Algorithmic Policies, in: SIGCOMM, 2013.

[13] T. Nelson, A. D. Ferguson, M. J. Scheer, S. Krishnamurthi, Tierless Programming and Reasoning for Software-Defined Networks, in: NSDI, 2014.

[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, D. Walker, Frenetic: A Network Programming Language, in: ICFP, 2011.

[15] G. Holzmann, The Spin Model Checker - Primer and Reference Manual, Addison-Wesley, Reading Massachusetts, 2004.

[16] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model Checking Programs, Automated Software Engineering 10 (2) (2003) 203–232.

[17] M. Musuvathi, D. R. Engler, Model Checking Large Network Protocol Implementations, in: NSDI, 2004.

[18] C. E. Killian, J. W. Anderson, R. Jhala, A. Vahdat, Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code, in: NSDI, 2007.

[19] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, L. Zhou, MODIST: Transparent Model Checking of Unmodified Distributed Systems, in: NSDI, 2009.

[20] C. Cadar, D. Dunbar, D. R. Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, in: OSDI, 2008.

[21] S. Bucur, V. Ureche, C. Zamfir, G. Candea, Parallel Symbolic Execution for Automated Real-World Software Testing, in: EuroSys, 2011.

[22] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.

[23] Open vSwitch, Open vSwitch: An Open Virtual Switch, http://openvswitch.org (2012).

[24] Open Networking Foundation, OpenFlow Switch Specification 1.1.0, http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf (2011).

[25] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, K. Wehrle, KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment, in: IPSN, 2010.

[26] S. Khurshid, C. S. Păsăreanu, W. Visser, Generalized Symbolic Execution for Model Checking and Testing, in: TACAS, 2003.

[27] C. Flanagan, P. Godefroid, Dynamic Partial-Order Reduction for Model Checking Software, in: POPL, 2005.

[28] P. Godefroid, N. Klarlund, K. Sen, DART: Directed Automated Random Testing, in: PLDI, 2005.

[29] V. Ganesh, D. L. Dill, A Decision Procedure for Bit-Vectors and Arrays, in: CAV, 2007.

[30] N. Vasić, D. Novaković, S. Shekhar, P. Bhurat, M. Canini, D. Kostić, Identifying and Using Energy-Critical Paths, in: CoNEXT, 2011.

[31] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann, OFRewind: Enabling Record and Replay Troubleshooting for Networks, in: USENIX ATC, 2011.

[32] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, S. T. King, Debugging the Data Plane with Anteater, in: SIGCOMM, 2011.

[33] E. Al-Shaer, S. Al-Haj, FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures, in: SafeConfig, 2010.

[34] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, K. Wansbrough, Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets, in: SIGCOMM, 2005.

[35] N. Kothari, R. Mahajan, T. Millstein, R. Govindan,

M. Musuvathi, Finding Protocol Manipulation Attacks, in: SIGCOMM, 2011.
[36] D. Sethi, S. Narayana, S. Malik, Abstractions for Model Checking SDN Controllers, in: Formal Methods in Computer-Aided Design (FMCAD), 2013.
[37] R. Majumdar, S. Tetali, Z. Wang, Kuai: A Model Checker for Software-defined Networks, in: Formal Methods in Computer-Aided Design (FMCAD), 2014.
[38] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, A. Valadarsky, VeriCon: Towards Verifying Controller Programs in Software-Defined Networks, in: PLDI, 2014.