Paper 6: A NICE Way to Test OpenFlow Applications

OpenFlow-capable switches enables exciting new network functionality and the centralized programming model, where a single controller program manages the network, seems to reduce the likelihood of bugs. The problem is that the system is inherently distributed and asynchronous, with events happening at multiple switches, and inevitable delays affecting communication with the controller.
Example of problems that can happen: 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. 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.
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.
The paper presents NICE an efficient, systematic techniques for testing unmodified controller programs. NICE tool applies **model checking** to explore the state space of the entire system—the controller, the switches, and the hosts. It efficiently uncovers bugs in OpenFlow programs, through a combination of model checking and symbolic execution.

Challenges of testing OpenFlow Apps:
Testing OpenFlow applications is challenging because the behavior of a program depends on the larger environment. 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. 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.
- Large space of input packets: Applications are dataplane driven, i.e., programs must react to a huge space of possible packets. The OpenFlow specification allows switches to match on source and destination MAC addresses, IP addresses, and TCP/UDP port numbers, as well as the switch input port; future generations of switches will match on even more fields
- 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.

To simplify the problem, we could require programmers to use domain-specific languages that prevent certain classes of bugs but it is too difficult. 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 [12] and Java PathFinder (JPF) 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. The paper argues that testing tools should operate directly on unmodified OpenFlow applications, and leverage domain-specific knowledge to improve scalability.

How does NICE work
To address these scalability challenges NICE (No bugs In Controller Execution)is created. It is tool that tests unmodified controller programs 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 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 deterministically reproduce them. The programmer can also use NICE as a simulator to perform manually-driven, step-by-step system executions or random walks on system states.

The design uses explicit state, software model checking to explore the state space of the entire system—the controller program, the OpenFlow switches, and the end hosts. 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** the handlers. 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.

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 operate on system state, or select from a library of common properties. NICE prototype tests unmodified applications written in Python for the popular NOX platform. The performance evaluation shows that: (i) even on small examples, NICE is five times faster than 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 7-fold reduction on its own. NICE was applied in three real OpenFlow applications and uncover 11 bugs.

Model Checking

How it works(background): 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 that 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.

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— 1. the controller program, 2. the OpenFlow switches, and 3. 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

1. Controller Program: Modeling the controller as a transition system seems relatively straightforward. To execute a transition, the model checker can simply invoke the associated event handler. However, the behavior of event handlers is often data dependent. Unfortunately, model checking does not cope well with data-dependent applications. Since enumerating all possible inputs is intractable. NICE overcomes this limitation by using symbolic execution to automatically identify the relevant inputs. (***Why Symbolic execution)*** Symbolic execution examines the code paths to identify equivalence classes of packets (relevant input packets) for testing.

2. OpenFlow Switches. To test the controller program, the system model must include the underlying switches. The reference switch software has a large amount of state. Importantly, such a large program has many sources of nondeterminism and it is difficult to identify them automatically. We create a switch model that omits inessential details. We view a switch as a set of communication channels, transitions that handle data packets and OpenFlow messages, and a flow table. **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. **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. As often done in model checking, we construct a canonical representation of the flow table that derives a unique order of rules with overlapping patterns

3. End hosts. 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

Symbolic Execution.

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.

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. ☺ NICE uses model checking to explore system execution paths (and detect repeated visits to the same state), and symbolic execution to determine which inputs would exercise a particular state transition.

Two exceptions of why applying it is not that straightforward. 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.

1. **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.

2. **Concrete controller state.** The execution of the event handlers also depends on the controller state. 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.

Combining SE with Model Checking.

Symbolic execution of the handler starts from the initial state defined by (i) the concrete controller state 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). There is an inizialization and a checking process. By symbolically executing the controller event handlers, NICE can automatically infer the test inputs for enabling model checking without developer input, at the expense of some limitations in coverage of the system state space.

OpenFlow-Specific Search Strategies.
Even with our optimizations from the last two sections, the model checker cannot typically explore the entire state space, since this may be prohibitively large or even infinite. Thus, 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 PKTSEQ which reduces the state-space explosion due to the transitions uncovered by symbolic execution. 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. PKT-SEQ is enabled by default and used in our experiments (unless otherwise noted). The other heuristics can be selectively enabled.

Specifying Application Correctness> NICE allows programmers to specify correctness properties as Python code snippets, and provides a library of common properties

-- 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.---

Limitations

1. Concrete execution on the switch: In identifying the equivalence classes of packets, the algorithm 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 would 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 leads to some loss in both efficiency and coverage.
2. 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.
3. Infinite execution trees in symbolic execution: Despite its many advantages, symbolic execution can lead to infinite execution trees. 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.