Systematic Software Testing Meets Networking

Marco Canini† and Dejan Kostić‡
† TU Berlin / T-Labs, ‡ Institute IMDEA Networks

Nowadays users expect and demand highly dependable network connectivity and services. However, several recent episodes demonstrate that software errors and operator mistakes continue to cause undesired disruptions and outages. It is crucial to have reliable networks, and this requirement does not change with Software Defined Networking (SDN). Unfortunately, as the network programmability enhances and software plays a greater role in it, risks that buggy software may disrupt an entire network also increase. 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.

This extended abstract presents an overview of efficient, systematic techniques for testing the SDN software stack at both its highest and lowest layer. That is, our testing techniques target at the top layer, the OpenFlow controller programs (Section 1) and, at the bottom layer, the OpenFlow agents (Section 2)—the software that each switch runs to enable remote programmatic access to its forwarding tables. The papers describing these tools have been published in [2] and [3]. Our goal here is to increase the awareness of the ever-increasing number of SDN adopters to our tools. In doing so, we hope to: (1) enable faster adoption of OpenFlow/SDN due to accelerated switch interoperability testing, and (2) decrease the chance of encountering bugs in the deployment of OpenFlow controller applications. Combined, our tools should increase the confidence in SDN as a whole.

1 NICE: Testing Application-Level Controller Logic

Our NICE (*No bugs In Controller Execution*) [2] tool tests unmodified controller programs by subjecting them to automatically generated carefully-crafted streams of packets under many possible event orderings. NICE usage is illustrated in Fig. 1: 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 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.

Our 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 because the need to consider the entire network state leads to an extremely large state space, which "explodes" along three dimensions: (1) switch state, (2) input packets, and (3) event orderings. 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 develop several domain-specific search strategies that generate event orderings that are likely to uncover bugs in the controller program.

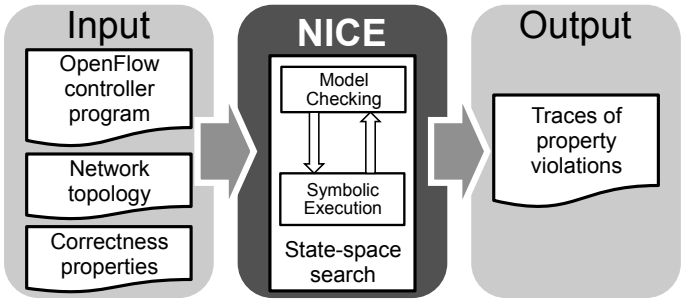

Fig. 1: Given an OpenFlow program, a network topology, and correctness properties, NICE performs a state-space search and outputs traces of property violations.

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. Our NICE prototype tests unmodified applications written in Python for the popular NOX platform. We apply NICE to three real OpenFlow applications and uncover *11* bugs. Most of the bugs we found are design flaws, which are inherently less numerous than simple implementation bugs. A release of NICE is publicly available at `http://code.google.com/p/nice-of/`.

2 SOFT: Testing the Interoperability of OpenFlow Switches

An aspect that is mostly going unnoticed is that OpenFlow switches also run software, which must behave correctly. This software takes the name of *OpenFlow agent*, and its role is to expose a standardized programmatic interface to the switch forwarding tables and to handle the communication with the controller. However, while testing high-level network functionality, the interoperability and correct behavior of any OpenFlow agent are taken for granted. In practice, a real OpenFlow deployment likely has switches from multiple vendors managed by one or more controllers. To ensure correct network operation, *all* switches must work properly. In other words, it may take just one buggy switch to cause problems in the form of lost connectivity, unauthorized accesses, traffic overload, and so on. If failures start occurring in OpenFlow deployments, *the hard-earned ability to innovate in the networking space will be severely hampered by mistrust.*

Several issues make it difficult to produce error-free switch software. Consider that just the rule installation command (`Flow Mod`) in the OpenFlow specifications is two and a half pages long. Moreover, the specifications are in rapid flux (going through three revisions in slightly over one year). Further, even given specifications have interpretation ambiguities or gives explicit implementation freedom.

Despite advances in writing provably correct software, testing remains the prime technique for ensuring dependability. We observe that local testing and debugging can get the basic functionality working. Beyond this, the only way of gaining confidence in the behavior of multiple different switches currently is interoperability testing. One way of doing this involves placing personnel and switches at a third-party location for several days, and running OFTest and similar test suites [1]. Besides being expensive, this task is complex, in part because the number of new OpenFlow switch implementations is quickly growing. Of course, any new version of the specifications require a new round of interoperability testing.

Towards achieving exhaustive testing, we recently proposed SOFT (*Systematic OpenFlow Testing*) [3], an approach to interoperability testing that leverages the multiple, existing OpenFlow implementations and herein identifies potential interoperability problems by crosschecking their behaviors.

Exploring code behaviors in a systematic way is key to observe behavioral inconsistencies. Symbolic execution effectively asks the code itself to provide the test inputs that are needed to traverse all code paths at least once. While appealing, the use of symbolic execution is generally met with the scalability challenges of exhaustive path coverage, which we must face. In addition, it would not be practical to assume that a tool for interoperability testing would have access to the source code of commercial OpenFlow implementations from all vendors. It is then our goal to make symbolic execution scale to crosscheck different OpenFlow implementations and find interoperability issues *without having simultaneous access to all source codes*.

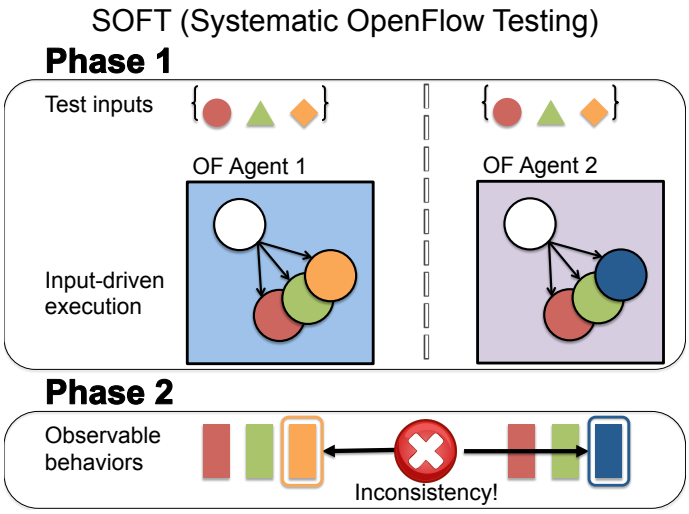

Fig. 2: SOFT looks for interoperability problems that manifest as input ranges that cause two OpenFlow agents to exhibit different behaviors.

Operating in two phases, SOFT uses symbolic execution and constraint solving. In the first testing phase, symbolic execution runs locally on each vendor's source code. Then, using the outputs of symbolic execution (not the source codes), SOFT determines the input ranges (*e.g.*, fields in OpenFlow messages) that cause two OpenFlow agent implementations to exhibit different behaviors. We demonstrate the effectiveness of our approach by applying it to the Reference Switch (55K LoC) and Open vSwitch (80K LoC), the two publicly available OpenFlow agent implementations. SOFT quickly found seven classes of inconsistencies between the two.

Acknowledgments. NICE and SOFT stem from our collaboration with a team of very passionate contributors: Maciej Kuźniar, Peter Perešíni, and Daniele Venzano. Jennifer Rexford collaborated with us on NICE. 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. ONF Holds Its First Test Event. `https://www.opennetworking.org/?p=249&option=com_wordpress&Itemid=72`.
2. M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.
3. M. Kuźniar, P. Perešíni, M. Canini, D. Venzano, and D. Kostić. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT*, 2012.