# Automating the Testing of OpenFlow Applications

Marco Canini*, Dejan Kostić*, Jennifer Rexford†, Daniele Venzano*
*EPFL, {marco.canini,dejan.kostic,daniele.venzano}@epfl.ch
†Princeton University, jrex@cs.princeton.edu

***Abstract*—Software-defined networking, and the emergence of OpenFlow-capable switches, enables a wide range of new network functionality. However, enhanced programmability inevitably leads to more software faults (or *bugs*). We believe that tools for testing OpenFlow programs are critical to the success of the new technology. However, the way OpenFlow applications interact with the *data plane* raises several challenges. First, the space of possible inputs (*e.g.*, packet headers and inter-packet timings) is huge. Second, the centralized controller has a indirect view of the traffic and experiences unavoidable delays in installing rules in the switches. Third, external factors like user behavior (*e.g.*, mobility) and higher-layer protocols (*e.g.*, the TCP state machine) affect the correctness of OpenFlow programs.**

**In this work-in-progress paper, we extend techniques for *symbolic execution* to generate inputs that systematically explore the space of system executions. First, we analyze controller applications to identify equivalence classes of packets that exercise different parts of the code. Second, we propose several network models with increasing precision, ranging from simple traffic models to live testing on the target network. Initial experiences with our prototype, which symbolically executes OpenFlow applications written in Python, suggest that our techniques can help programmers identify bugs in their OpenFlow programs.**

## I. Introduction

After more than a decade of arguing for programmable networks, this vision is finally materializing. In particular, the OpenFlow protocol allows a logically-centralized controller to programmatically install packet-handling rules in the underlying switches [1]. Controller applications can dynamically install new rules, read traffic statistics for existing rules, learn about switch and link failures, and handle data packets on behalf of the switches. Several commercial switch vendors support OpenFlow, a number of campus and backbone deployments are underway, and a growing collection of controller applications implement new network functionality [2]–[7]. However, along with easier extensibility, programmable networks increase the potential for software faults (or *bugs*).

Conventional networks are largely driven by *configurations* that can be statically checked. Even though today's routers and switches run complex software, the code is written, tested, and debugged by the equipment vendors, and the functionality is (somewhat) constrained by the protocol standardization process. Still, even carefully-debugged, closed-source router software can have bugs that trigger Internet-wide failures [8]–[10]. In contrast, OpenFlow networks are driven by *dynamic programs*, written by network operators and third-party developers. As more people start programming the network, the problems with buggy software will only get worse.

Our goal is to help these programmers produce reliable new OpenFlow applications. Designing domain-specific programming languages is one approach to prevent common coding mistakes. However, the adoption of new languages is difficult in practice. Not surprisingly, existing OpenFlow applications are written in well-known, general-purpose languages, like Python and C++. Rather than design a new language, we are creating tools and techniques for testing OpenFlow applications as extensively as possible, to detect and eliminate bugs. In particular, we identify ways to explore the large space of possible system executions, using carefully crafted inputs.

Two techniques from the verification community—model checking [11], [12] and symbolic execution [13]—have proven quite effective in detecting bugs in distributed systems software. However, we cannot simply apply these techniques "out of the box," because programming OpenFlow networks raises several additional challenges, such as:

**Much larger space of inputs:** An OpenFlow application is *data-plane* driven, *i.e.*, the program must react to a very large space of possible packet headers and inter-packet timings. Thus, the problem at hand is much harder than that of testing control-plane software (*e.g.*, BGP implementations).

**Distributed collection of switches:** While OpenFlow programs run on a centralized controller, the *system* is still distributed. The packets seen at the controller can be a subset or superset of those expected by the programmer. The abstraction of a central control plane hides the network latencies and can give rise to race conditions, where rules are installed while packets in flight are processed based on previous settings.

**End-host devices, protocols, and applications:** The traffic and events seen in an OpenFlow network depends on outside factors, such as user behavior (*e.g.*, device mobility), transport protocols (*e.g.*, the TCP state machine), and end-host applications (*e.g.*, Web servers). Hence, testing cannot focus on the network topology and controller application in isolation.

To make testing tractable, we identify: $(i)$ distinct packet header patterns and $(ii)$ packet timings, orderings, and other network events that can exercise different paths through the code. To accomplish the first goal, we perform dynamic analysis of the OpenFlow application code to determine *equivalence classes of packets* that exercise different parts of the code. For the second task, we automatically infer and use *network models* of increasing coverage and precision to inject the relevant streams of packets into the application code. Each execution of the application is checked for correctness by verifying the universally applicable invariants (*e.g.*, avoiding black holes and loops), as well as those added by developers.

The main novelty of our work lies in automatically determining the complex network model that drives application behavior. Existing model checking and symbolic-execution tools for distributed systems require this model to be manually specified. Moreover, we go one step further by checking the network itself, in contrast to existing approaches that take basic network connectivity for granted.

In the remainder of this paper, we present a brief overview of OpenFlow in Section II and a set of motivating OpenFlow examples of software bugs in Section III. We then describe the challenges of testing OpenFlow programs and we address these challenges in Section IV. Section V presents our initial implementation of a symbolic-execution engine capable of running applications written in Python for the popular NOX OpenFlow controller [14]. Section VI discusses related work, and Section VII concludes the paper with a discussion of future research directions.

## II. OpenFlow Background

The OpenFlow protocol allows programs running on a logically-centralized controller to coordinate a distributed collection of switches.

**OpenFlow switches:** An OpenFlow switch has a flow table that stores an ordered list of rules for processing packets. Each rule consists of a pattern (matching on packet header fields), actions (such as forwarding, dropping, flooding, or modifying the packets, or sending them to the controller), a priority (to distinguish between rules with overlapping patterns), and a timeout (indicating whether/when the rule expires). 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 number of bytes and packets processed so far. When a packet arrives, a switch selects the highest-priority matching rule, updates the traffic counters, and performs the specified action(s). Switches also generate events, such as a "join" event upon joining the network, or "port change" events when links go up or down.

**Centralized controller:** An OpenFlow network has a centralized programming model, where one (or a few) software controllers manages the underlying switches. The controller (un)installs rules in the switches, reads traffic statistics collected by the switches, and responds to network events. A controller application defines a handler for each event (*e.g.*, packet arrival, rule timeout, and switch join), which may install new rules or issue new requests for traffic statistics. A common idiom for controller applications is to respond to a packet arrival by installing a rule for handling subsequent packets directly in the data plane. Sending packets to the controller introduces overhead and delay, so most applications try to minimize the fraction of traffic that must go to the controller. Most OpenFlow applications are written on the NOX controller platform [14], which offers OpenFlow API for applications written in Python or C++. These controller applications are general-purpose programs that can perform arbitrary computation and maintain arbitrary state.

## III. Example Bugs in Controller Programs

Testing and debugging controller programs is challenging, since small differences in packet header fields or packet timing can affect the state of the network and "tickle" subtle bugs.

**Multiple packets of a flow reaching the controller:** A common idiom in programming OpenFlow networks is to direct a packet to the controller, and then install a rule for the switches to handle the remaining packets of a flow in the data plane. Yet, a race condition can easily arise if additional packets arrive while the controller is in the middle of installing the rule. These packets arrive at the controller as well. A program that implicitly expects to see just one packet may behave incorrectly when multiple packets arrive. For example, imagine a program that intends to directs all packets in a flow to the same randomly-selected server. The arrival of a second packet may trigger the application to install a second rule that directs packets to a different server replica. The program would behave correctly in the common case where the subsequent packets enter the network only after the rule is installed, but break if a burst of packets arrives at the controller.

**No atomic update across multiple switches:** Many applications need to install rules at multiple switches (*e.g.*, to direct the packets over a particular path through the network). These rules are not installed atomically, so some switches may start applying new rules before other switches have installed their rules. This can lead to unexpected behavior, where an intermediate switch may encounter a packet that must go to the controller for handling. Implicitly assuming that rules are installed atomically can lead to subtle bugs that only manifest themselves under certain packet timings and rule-installation delays. Installing rules from "back to front" (from the end of the path to the beginning) can prevent this mistake, but the programmer may not choose to install the rules this way.

**Previously-installed rules limit the controller's visibility:** The controller program is really just one part of a distributed system that includes the processing performed by the underlying switches. Installing a rule (*e.g.*, that forwards or drops all matching packets) not only dictates what processing a *switch* performs, but also what packets the *controller* sees in the future! For example, imagine a program implementing a learning switch. Installing a wildcard rule to forward traffic based only on the destination MAC address would keep the controller from seeing some packets sent by new source MAC addresses—preventing the network from "learning" how to reach these addresses. While still successfully delivering traffic, this program would lead to inefficient delivery (*e.g.*, via unnecessary flooding) in some corner cases.

**Composing functions that affect the same packets:** Networks often perform multiple tasks that affect the handling of the same packets. For example, routing determines which path carries each packet (*e.g.*, based on destination IP address), and monitoring determines which packets should be grouped together for accumulating statistics (*e.g.*, based on TCP port number). Combining functionality is complicated, potentially involving the "cross product" of the rules needed for each function independently. OpenFlow switches rely on rule priority to disambiguate between overlapping groups of packets (*e.g.*, to ensure a rule with destination address 1.2.3.4 and port 80 gets precedence over another rule that matches all traffic to destination 1.2.3.4). Subtle mistakes in setting the priorities can lead to a program that operates correctly except for certain packets, or packets arriving in a particular order.

**Interaction with end-host software:** Some controller applications rely on implicit assumptions about the software running on the end host (*e.g.*, new TCP connections start with a SYN packet, or a Web download idle for more than 60 seconds has completed). These applications may have subtle bugs that only arise when hosts generate traffic that violates these assumptions. For example, imagine a server load-balancing application that directs client traffic to different Web server replicas (*e.g.*, sending traffic from source IP addresses starting with 0 to one replica and starting with 1 to another) [5]. Any changes to the load-balancing policy should ensure any *ongoing* TCP connection completes on the same server. By installing a rule that temporarily directs traffic to the controller, the application could inspect the *next* packet of each flow to install a microflow rule directing new flows (*i.e.*, if the packet is a SYN) to a new server and ongoing flows (*i.e.*, if the next packet is not a SYN) to the old server. However, the TCP state machine allows a host to retransmit SYN packets, raising the possibility of duplicate SYN packets which could lead the application to wrongly classify an ongoing connection as new.

## IV. Testing OpenFlow Programs

The lesson we draw from the example bugs is that controller programs may cause the network to misbehave when exposed to corner cases unforeseen by the programmer. Intuitively, testing the correctness of OpenFlow programs needs to account for all possible corner cases. As these are clearly not known *a priori*, we then want to focus our effort on subjecting the application to a variety of carefully chosen inputs (*i.e.*, sequences of packets and network events). To choose relevant inputs, we want to identify: (1) what values of packet header fields and (2) what packet timings, orderings, and other network events cause the execution of a certain path through the controller program's code. Fig. 1 illustrates how we want to proceed:

For (1), we note that the code itself, in its branching predicates, can reveal the relationships between packet header fields and code paths (*e.g.*, an `if` statement checking for a broadcast MAC address signals different behaviors depending on whether the check succeeds). Therefore, we use code analysis to determine "equivalence classes" of packets.

For (2), we must first look at what constitutes system state. An OpenFlow network should be seen as a distributed program composed of a controller program and several "switch programs". A switch program essentially implements a large switch-case structure that performs different operations depending on which rule matches the incoming packet. To reason about the correctness of an OpenFlow program, we need to observe the state of the entire network. Therefore, we specify several models of network behavior that cover, with increasing precision, the possible network events. With these models, we can strike trade-offs between testing time/effort and fidelity.

Then, the process of testing OpenFlow applications starts from combining (1) and (2) into a single model that succinctly describes the space of many possible system executions. In other words, this model describes what events are possible at any given state. We use this model to decide what events to inject into the application, similar to applying model checking based on explicit state enumeration. We extensively explore the space of possible states and test each state against a collection of universal and application-specific invariants.

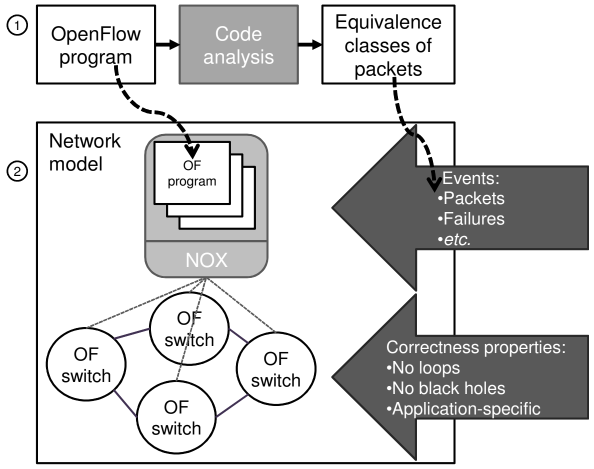

Fig. 1. Identifying (1) relevant packet header fields and (2) packet sequences and network events for testing OpenFlow applications.

### A. Huge Space of Inputs: Equivalence Classes of Packets

The space of possible inputs that could be fed to the application under test is huge. Each individual input is a packet header from the data plane (*e.g.*, a connection establishment packet). Consider that there exists $2^{32}$ different IPv4 addresses, and $2^{16}$ different ports, and each packet header has a source and destination (address, port) pair. Further, it takes more than one packet to uncover even the faults in our simple examples.

To deal with large input sizes, we use code analysis of the OpenFlow application to determine *equivalence classes* of packets. That is, we identify what (ranges of) values of header fields determine the path through the application code. Within each equivalence class, we pick a representative packet to feed the application to exercise a particular code path.

We analyze the code using symbolic execution, which automatically determines what input values can exercise each path through the code. To do so, a symbolic execution engine runs the program with symbolic inputs. The engine tracks the use of the symbolic inputs and records the constraints involving the possible input values. When the execution encounters a branching point, the engine queries a satisfiability solver to determine which paths are feasible, and logically forks the execution to follow all feasible paths.

OpenFlow programs are typically structured as event-driven code where each event is associated with an handler. For example, Fig. 2 shows the pseudocode for a simple MAC-learning switch application: it defines a packet handler named `packet_in` (line 1). Briefly, this code maintains a per-switch address table (line 2) that maps a host's MAC address to its associated switch port. If the source (line 3) and/or destination address (line 10) is a broadcast address, it simply floods the network (line 17). Otherwise, it first checks if the source is unknown (line 4), in which case it updates the address table (line 5). When the source is already known, it ensures the packet came in from the expected port (lines 7-8) or it updates the registered port (line 9). Then, if the destination is known

```
 1 def packet_in(switch_id, inport, pkt):
 2   mactable = app_state[switch_id]
 3   if not is_broadcast(pkt.src):
 4     if not mactable.has_key(pkt.src):
 5       mactable[pkt.src] = (inport, time(), pkt)
 6     else:
 7       e = mactable[pkt.src]
 8       if e[0] != inport:
 9         mactable[pkt.src] = (inport, time(), pkt)
10   if not is_broadcast(pkt.dst) and              ↩
       mactable.has_key(pkt.dst):
11     outport = mactable[pkt.dst][0]
12     if outport != inport:
13       rule = extract_rule(pkt, inport)
14       rule.actions.output = outport
15       install_rule(switch_id, rule)
16       return
17   flood_packet(switch_id, pkt)
```

Fig. 2. Pseudocode of a simple MAC-learning switch application, loosely based on the `pyswitch` NOX application.

and it is not a broadcast address (line 10), it makes a final check to verify that the destination and source ports are different (line 12). If this sanity check succeeds, it installs a new rule in the switch to forward the packet to the destination port (lines 13-15). Instead, when input and output ports are the same, it simply floods the network (line 17).

To learn classes of packets, we apply symbolic execution to `packet_in` event handlers. We expect symbolic execution to be effective in covering all code paths in these handlers because, to quickly react to many such events, handlers do not generally perform complex, compute-intensive operations. This alleviates the problem that symbolic execution usually faces: an exponential number of paths in the program size.

However, plugging a packet handler into a symbolic execution engine is not sufficient for deriving the equivalence classes of packets. In practice, the application state also plays a role in determining what code is executed. For example, it is easy to note that a symbolic execution engine would not be able to find input values for `packet_in` that execute the branch at lines 6-9. This is because, starting from the initial state (*i.e.*, an empty address table), the branch predicate at line 4 is always false. Therefore, we note that certain application behaviors are the result of *sequences* of packets. Our use of network models of varying precision explained below is a first step in addressing this issue. Secondarily, we want to infer from the code the relationship between consecutive packets. The idea is to track changes to the application state and relate them to code paths that depend on state variables having certain values. With reference to the previous example, we can easily see that the statement at line 5 changes the application state and that a consecutive call of `packet_in` with the same source address would finally execute the branch at lines 6-9. We want to automatically identify these state transitions and use them to determine the initial state needed for expanding the coverage of symbolic execution.

## *B. Complex Network Behavior: Progressively Detailed Network Models*

Testing typically requires a model of system behavior. While effective in capturing the behavior expressed by the underlying code, symbolic execution must be complemented by a model of the environment. In our case, this model typically includes a series of possible network-related events such as packet drops, broken TCP connections, packet reorderings, node or component failures, *etc.* [11], [15].

One could argue that this *network model*, which describes interactions of packets and events within the network, is common knowledge. However, we face the challenge that we are dealing with a distributed system that controls basic network connectivity—in contrast to previous work on checking distributed systems, which takes network connectivity for granted. Therefore, it is difficult to know beforehand how to specify a model that covers relevant sets of packet reorderings, inter-packet timings, and other network events.

Further, it is not possible to completely specify such a model because the network behavior is in part determined by the OpenFlow application, as we discussed earlier. Even if it were possible to automatically infer the network model, there exist a form of circular dependency in that the behavior of the OpenFlow application depends on the network that it is trying to control. For example, if the underlying network does not form a spanning tree, a simple OpenFlow application that floods packets can result in infinite packet loops.

Finally, we want to be able to detect problems due to interoperability issues that may arise when the controller program uses specific implementations of OpenFlow. Therefore, we cannot simply focus on checking the correctness of a controller program in isolation from the switches.

We take a four-prong approach for building progressively more detailed network models:

1) We start with a simple model based on the commonly available knowledge. This model accounts for events such as reordered packets, dropped packets, switch failures, link failures, topology changes, and user mobility.
2) We want to automatically augment the network model by examining the OpenFlow application code. To illustrate, consider the case of the TCP state machine. If we can automatically process the state machine specification, we automatically realize the need to subject the application to repeated SYN packets and other corner cases.
3) We want to re-create the target network topology in our testing environment by using virtual instances of actual OpenFlow switch implementations (*e.g.*, Open vSwitch [16]). That is, rather than modeling switch behavior at a high level, we wish to base our model on the code that runs an OpenFlow switch.
4) To account for the unpredictability of the target network and its behavior, we want to integrate our approach with the network itself to enable testing the OpenFlow application on the target network, but in isolation from production traffic (similar to [13]).

These approaches are illustrated in Fig. 3 as concentric circles where the captured network behavior monotonically increases from approach one to four. This benefit of higher precision, however, comes at the increase in complexity.

For example, the first approach already enables developers to test their code under different packet-arrival patterns just by using their workstation. The second approach introduces relevant sequences of packets based on the analysis of the

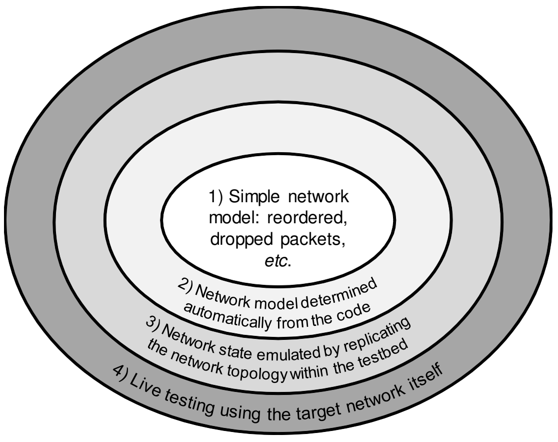
1) Simple network model: reordered, dropped packets, *etc.*
2) Network model determined automatically from the code
3) Network state emulated by replicating the network topology within the testbed
4) Live testing using the target network itself

Fig. 3. Overview of our approach to building a network model that covers progressively more network behaviors.

OpenFlow program code. The third step allows the behavior of the system comprising the controller program and the network of OpenFlow switches to be checked (still in a local setting). Finally, the fourth step allows the programmer to gain more confidence that the target network performs as expected by testing on the network itself.

For testing, we use the network model to decide what events can happen at any given state and to subject the OpenFlow network to these events in a systematic way. Effectively, we inject the application with a number of relevant inputs that explore a variety of expected behaviors, as well as difficult-to-produce corner cases.

### C. Specifying Correctness is Hard: Testing Invariants

We have discussed how different application behavior can be triggered by carefully crafted and timed inputs. However, the question remains of knowing what constitutes a fault. We will uncover software faults by detecting violations of desired behavior, *e.g.*, incorrect behavior. We are not interested in finding straightforward memory safety violations, memory leaks, *etc.*, as we believe that the existing tools are sufficient for this task.

Specifying correct behavior is a challenging problem even for single machine applications. Simply put, most software is written without considering the safety and liveness properties typically used to describe desired behavior. In our case, the addition of the distributed collection of switches presents additional challenges, as it further complicates reasoning about correctness.

Our approach to dealing with correctness is to: (i) incorporate properties that are widely applicable to all networking applications (*e.g.*, the forwarding rules should not induce loops or black holes), and (ii) provide an API that the developer can use to specify additional safety invariants or liveness properties (*e.g.*, all packets of the same microflow go to the same Web server). As we inject inputs to the system, we test these invariants after each system transition.

However, programmers often have difficulty stating meaningful invariants for their code. OpenFlow programming raises an alternative way to define correctness: the programmer could write a simpler controller application where *all* packets are handled by the controller, with *no* rules installed in the underlying switches. These programs, while clearly inefficient, are much easier to write because they side-step the challenges of distributed state and race conditions that arise in delegating work to the switches. In fact, writing the simple centralized program is a natural first step toward writing a more complex version that offloads packet-handling work to the switches. By injecting the same inputs to both programs, we can check whether the two programs treat all traffic the same way.

## V. Preliminary Prototype

We developed a symbolic execution engine capable of running NOX applications implemented in Python. Within it, the application is executed in a controlled environment that provides a subset of the NOX API and is fed with symbolic packets. The application execution is then traced through the Python interpreter and this process is re-iterated until all constraints have been recorded and negated, by using a constraint solver, to explore all possible execution paths.

**Python specifics**. A key step in symbolic execution is tracking the constraints during code execution. A notable difference from symbolic execution of C code [13] is that we had the option of changing the data types to be symbolic, instead of having to instrument C code. For example, we implemented a "symbolic integer" type to be able to follow a symbolic variable through the program execution. This way the engine knows immediately whenever a change/assignment/comparison to a symbolic variable is made. We also implemented arrays of these symbolic integers.

**OpenFlow specifics**. The basic unit of symbolic input is a packet, and the engine feeds a symbolic packet at a time to the application and records all constraints that are applied to it. Our symbolic integer is the base for a real packet symbolic type that is substituted to the packet type provided by NOX. For example, we currently mark two MAC addresses as symbolic six-byte arrays, as well as a symbolic type to inject the symbolic packet into the `pyswitch` MAC-learning switch application.

## VI. Related Work

The CMC [17] model checker was successfully used to check network protocol implementations (*e.g.*, the Linux implementation of TCP/IP and the AODV ad-hoc routing protocol). The MaceMC distributed systems model checker [11] can detect liveness property violations in distributed systems code, while CrystalBall [12] can guard against safety property violations due to unknown bugs. All of these require manual creation of the network model used to trigger state changes.

Symbolic execution has proven useful in automatically creating test cases that attempt to exhaustively exercise all code paths in a given piece of code [18]. Recently, Canini *et al.* [13] have shown that it is possible to use symbolic execution in the live setting to detect BGP misconfiguration. However, both Klee [18] and DiCE [13] use small inputs to overcome the path explosion problem that arises when large inputs (as those needed for testing of OpenFlow applications) are used. Moreover, symbolic execution requires manual effort to create branches in the code that the path exploration engine can subsequently negate and trigger desired changes in the network. For example, this is the approach taken in KleeNet [15].

FlowChecker [19] can detect misconfiguration in one or more OpenFlow forwarding tables. The main motivation for this work arises in federated environments, potentially with different OpenFlow controllers. FlowChecker builds upon the existing ConfigChecker [20] tool, which requires manual construction of a network model using binary decision diagrams. This model is then checked using symbolic model checking techniques for reachability and security. We view this work as orthogonal to ours since it aims to check the OpenFlow application behavior at runtime.

Frenetic [21] is a domain-specific language for OpenFlow networks that introduces an abstraction that the OpenFlow program examines every packet. By doing so, Frenetic aims to eradicate a large class of programming faults that arise due to a programmer's confusion in handling individual packets vs. installing wildcard rules on the switches (*i.e.*, reactive vs. proactive OpenFlow programming model). Using Frenetic requires the network programmer to learn another language. That said, the research on Frenetic inspired several of our example bugs in Section III, and the idea of defining correctness through a simple controller program that "sees every packet."

OFRewind [22] is a system that enables recording and replay of network events with the aim of facilitating troubleshooting problems in production networks due to closed-source components such as commercial routers and switches. While OFRewind could be used to investigate misconfiguration in OpenFlow networks, it does not automate the testing of OpenFlow controller programs.

## VII. Conclusion and Future Work

In this paper, we argued for automating the testing of OpenFlow applications. As the network programmability enhances, risks arise that even a single bug in the centralized controller can disrupt the entire network. Through a series of examples, we showed several pitfalls where small differences in packet headers or timing can result in unanticipated corner cases.

We identified the huge space of inputs, the complexity of the distributed system, and the dependency of correctness on external events as some of the challenges in testing OpenFlow programs. Our goal is to automate the testing by subjecting the application to a variety of carefully chosen sequences of packets and network events. To do so, we propose to dynamically analyze controller programs to identify equivalence classes of packet headers and to use several network models with increasing coverage and precision, ranging from simple traffic models to live testing on the target network. Initial experiences with our prototype, which symbolically executes NOX applications written in Python, suggest that our techniques can help programmers identify bugs in their OpenFlow programs.

Our work is on-going. We plan to complete the implementation of the symbolic execution engine for Python to include support for more data types and to automatically detect relevant initial states. Further, we will build the network models, analyze their coverage, and find techniques grounded on formal methods that can help to cope with the large space of system executions.

*Acknowledgments.:* 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] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008.

[2] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Transactions on Networking*, vol. 17, Aug. 2009.

[3] A. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic Access Control for Enterprise Networks," in *ACM Workshop on Research on Enterprise Networks (WREN)*, Aug. 2009.

[4] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," Aug. 2009. Demo at *ACM SIGCOMM*.

[5] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-Based Server Load Balancing Gone Wild," in *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, Mar. 2011.

[6] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks," in *Proc. Networked Systems Design and Implementation*, Apr. 2010.

[7] D. Erickson *et al.*, "A demonstration of virtual machine mobility in an OpenFlow network," Aug. 2008. Demo at *ACM SIGCOMM*.

[8] "Research experiment disrupts Internet, for some," http://www.computerworld.com/s/article/9182558/Research_experiment_disrupts_Internet_for_some.

[9] "AfNOG Takes Byte Out of Internet," http://www.renesys.com/blog/2009/05/byte-me.shtml.

[10] "Reckless Driving on the Internet," http://www.renesys.com/blog/2009/02/the-flap-heard-around-the-worl.shtml.

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

[12] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak, "CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems," in *NSDI*, 2009.

[13] M. Canini, V. Jovanović, D. Venzano, B. Spasojević, O. Crameri, and D. Kostić, "Toward Online Testing of Federated and Heterogeneous Distributed Systems," in *USENIX Annual Technical Conference*, 2011.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, July 2008.

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

[16] "Open vswitch: An open virtual switch." http://openvswitch.org.

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

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

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

[20] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *ICNP*, 2009.

[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *ACM International Conference on Functional Programming*, Sept. 2011.

[22] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *USENIX Annual Technical Conference*, 2011.