A SOFT Way for OpenFlow Switch Interoperability Testing

Maciej Kuźniar[†] Peter Perešíni[†] Marco Canini[‡*] Daniele Venzano[†] Dejan Kostić[•*]
[†]EPFL, Switzerland [‡]TU Berlin / T-Labs, Germany [•]Institute IMDEA Networks, Spain
[†]{name.surname}@epfl.ch [‡]m.canini@tu-berlin.de [•]dkostic@imdea.org

ABSTRACT

The increasing adoption of Software Defined Networking, and OpenFlow in particular, brings great hope for increasing extensibility and lowering costs of deploying new network functionality. A key component in these networks is the OpenFlow agent, a piece of software that a switch runs to enable remote programmatic access to its forwarding tables. While testing high-level network functionality, the correct behavior and interoperability of any OpenFlow agent are taken for granted. However, existing tools for testing agents are not exhaustive nor systematic, and only check that the agent's basic functionality works. In addition, the rapidly changing and sometimes vague OpenFlow specifications can result in multiple implementations that behave differently.

This paper presents SOFT, an approach for testing the interoperability of OpenFlow switches. Our key insight is in automatically identifying the testing inputs that cause different OpenFlow agent implementations to behave inconsistently. To this end, we first symbolically execute each agent under test in isolation to derive which set of inputs causes which behavior. We then crosscheck all distinct behaviors across different agent implementations and evaluate whether a common input subset causes inconsistent behaviors. Our evaluation shows that our tool identified several inconsistencies between the publicly available Reference OpenFlow switch and Open vSwitch implementations.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

Keywords

Switches, Bugs, Reliability, OpenFlow, Symbolic execution

*Work done when the author was with EPFL.

CoNEXT'12, December 10–13, 2012, Nice, France.

1. INTRODUCTION

Software defined networking (SDN) holds the promise to lower the barrier for deploying and managing new functionality in networks. For example, Google recently outlined how it uses SDN to solve the problem of scheduling bursty traffic among its datacenters [1]. The main thrust in SDN is currently OpenFlow [21]. In OpenFlow, the software running at a logically centralized controller manages a collection of switches hosting programmable forwarding tables.

It is crucial to have reliable networks, and this requirement does not change with SDN. Unfortunately, with the introduction of greater programmability, the chances of software faults (or bugs) are also on the rise. Debugging application software that runs at the controller has recently started receiving attention. For example, NICE [11] subjects the controller software to a wide range of packet streams to uncover race conditions and other bugs.

However, 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.* In addition, hardware switches run OpenFlow agents in the form of embedded firmware. Firmware is difficult to upgrade due to the impact of downtime, has longer debug cycles than ordinary software, and is notoriously challenging to troubleshoot in the wild [23]. These issues only raise the importance of trying to ensure that the firmware is right during its development stage.

Several issues make it difficult to produce error-free switch software. Consider that just the rule installation command (`Flow Mod`) in the OpenFlow specifications [5] 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. In some cases, vendors do not even follow the specifications [22].

Despite advances in writing provably correct software, testing remains the prime technique for ensuring dependability. We observe that local testing and debugging (*e.g.*, by using OFTest [2]) 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 [3]. 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.

Moreover, since local test suites are unlikely to be exhaustive, the above interoperability testing will not be exhaustive either. For instance, vendors typically test control plane software using manually-composed test cases, refined over time. Evidence shows that critical interoperability bugs survive this process. For example, in two recent episodes, 100% protocol-compliant BGP messages caused significant connectivity problems [6, 7]. These examples strongly confirm that local test cases are incomplete even for well-established router software.

Towards achieving exhaustive testing, we propose 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. Researchers recently applied symbolic execution [18], a systematic program code analysis technique, to test systems software with considerable success [8,9,12]. Symbolic execution effectively asks the code itself to provide the test inputs that are needed to traverse all code paths at least once. The reasoning here is that it is sufficient to test each code path once to exercise all behaviors. Doing this is relatively simple for single-machine code, while trying to find memory-related bugs.

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.[1] 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.*

In this paper, we introduce SOFT (Systematic OpenFlow Testing), a tool that automates interoperability testing of OpenFlow switches. 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.

Unlike normal execution, symbolic execution runs a program with symbolic variables as inputs. A symbolic variable initially has no constraints on its actual value. As the execution progresses, the possible values of symbolic variables become constrained based on how the program uses these variables (*e.g.*, in conditional expressions). At every code branch based on a symbolic variable, symbolic execution logically forks and follows both branches, on each path maintaining a set of constraints, called the path condition, which must hold for the execution of that path.

Using symbolic execution as a base for interoperability testing is conceptually similar to checking functional equivalence on a per-path basis. Assume we have two functions that take a single argument and implement the same algorithm in different ways. We can check functional equivalence by simply feeding them the same symbolic argument and verifying they return the same value. In practice, when applying symbolic execution to OpenFlow agent testing, we must address several challenges. First, the input space is theoretically infinite as an OpenFlow agent is a non-terminating program. Simply limiting the input space (*i.e.*, OpenFlow messages and data packets) to N symbolic bytes is not effective, given that it means feeding completely unstructured inputs. Intuitively, with such inputs, the symbolic execution engine would quickly run into the path explosion problem (number of paths grows exponentially with the number of branches on symbolic inputs). Second, there is no immediate or a standard way to compare the behaviors of different agents. Demanding modifications to the agents' source code for inspecting the state is clearly undesirable. Third, the straightforward equivalence check outlined above requires simultaneous access to both OpenFlow agents' source codes, which is likely to be impossible for commercial implementations.

The contributions of this paper are as follows:

1. We use symbolic execution to systematically identify and collate code paths in OpenFlow agents to determine input subspaces that result in the same outputs. To achieve this, we address the difficult problems of managing the combination of symbolic and concrete inputs, as well as determining internal agent state by observing external actions. This step overcomes the first challenge in applying symbolic execution to interoperability testing.

2. We demonstrate a novel use of a constraint solver to compute an intersection of input subspaces belonging to different agent implementations. By doing so, we quickly determine inputs that cause different behavior in multiple agents (inconsistencies). In addition, we do so without an *a priori* definition of correct behavior and overcome the second challenge (crosschecking implementation behaviors). This phase is separate from symbolic execution and does not require access to source code. By this virtue, it addresses the third aforementioned challenge.

3. 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 finds several inconsistencies between the two. Further, we demonstrate SOFT's effectiveness in finding manually injected differences.

The remainder of this paper is organized as follows. We provide an overview of our approach in Section 2, and follow it with the detailed description in Section 3. Section 4 contains the details of our prototype, and we proceed to evaluate it in Section 5. We place our work in the context of related work in Section 6 and conclude in Section 7.

[1] Although modern symbolic execution engines only require access to the binary code, this still needs to be produced with a particular compiler or be interpreted at runtime, which may incur impractical overhead. Also, it may not be possible to simply use the binary form since the execution environment is generally not known a priori.

2. OVERVIEW

In this section, we first give a brief introduction to symbolic execution, the technique our approach is built on. Next, we describe the kind of implementation inconsistencies we target. We then use an example to guide an overview of our approach and discuss the intended usage of SOFT.

2.1 Symbolic execution

Our approach is inspired by the successful use of symbolic execution [18] in automated testing of systems software [8–10, 12, 15]. The idea behind symbolic execution is to exercise all possible paths in a given program. Therefore, unlike normal execution which runs the program with concrete values, symbolic execution runs program code on symbolic input variables, which are initially allowed to take any value. During symbolic execution, code is executed normally until it reaches a branch instruction where the conditional expression *expr* depends (either directly or indirectly) on a symbolic value. At this point, program execution is logically forked into two executions—one path where the variables involved in *expr* must be constrained to make *expr* true; another path where *expr* must be false. Internally, the symbolic execution engine invokes a constraint solver to verify the feasibility of each path. Then, program execution resumes and continues down all feasible paths. On each path, the symbolic execution engine maintains a set of constraints, called the path condition, which must hold for the execution of that path. For every explored path, symbolic execution passes the path condition to a constraint solver to create a test case with the respective input values that led execution on that path. Since program state is (logically) copied at each branch, the symbolic execution engine can explore multiple paths simultaneously or independently.

Like others [19], we observe that, to deal with loops, symbolic execution would potentially need to explore an unbounded number of paths. As described in Section 3.2, we effectively side-step this problem by exploiting knowledge of the OpenFlow message grammar to construct inputs that ensure we explore a bounded number of paths.

Therefore, symbolic execution is a powerful program analysis technique—rather than having a linear execution where concrete values are used, symbolic execution covers a tree of executions where symbolic values are used. However, the usefulness of symbolic execution is limited by its scalability because the number of paths through a program generally grows exponentially in the number of branches on symbolic inputs. This problem is commonly known as the "path explosion" problem. The path explosion is exacerbated by the fact that the program under test interacts with its environment, e.g., by invoking OS system calls and calls to various library functions. External functions present an additional problem if the symbolic execution engine does not have visibility into their source code. A typical solution to this problem is to abstract away the complexity of the underlying execution environment using models. These models are typically a simplified implementation of a certain subsystem such as file system, network communication, *etc.*. Besides using environment models to "scale" symbolic execution, it is possible and often sufficiently practical to selectively mark as symbolic only the inputs that are relevant for the current analysis. As we show later in Section 3, carefully mixing symbolic and concrete inputs is key to being able to symbolically execute OpenFlow agents.

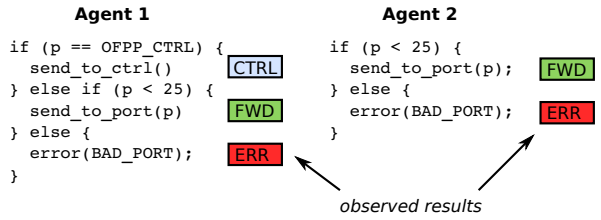

Figure 1: Example OpenFlow agents having different PACKET_OUT message implementations.

2.2 Defining inconsistencies

Switches that are capable of supporting the OpenFlow Switch Specification [5] do so by running an OpenFlow agent. This agent is a piece of software primarily responsible for state management. It receives and processes control messages sent by OpenFlow controllers (*e.g.*, `Flow Mod`, `Packet Out`, *etc.*), and configures the switch forwarding tables accordingly to the given commands. In addition, the OpenFlow agent may take part in packet forwarding itself—in a hardware switch, for packets that are forwarded to the controller; in a pure software implementation, for every packet.

As such, the execution of the OpenFlow agent is mainly driven by external events (*e.g.*, rule installation requests). *We call inputs the data reaching the agent as part of either OpenFlow control messages or packets.*

Intuitively, an *inconsistency* occurs when two (or more) OpenFlow agents which are presented with the same input sequence produce different results. Here, results refer to both externally observable consequences when processing an input (*e.g.*, replying to a request for flow table statistics), and internal state changes (*e.g.*, updating the flow table with a new entry).

To be able to identify inconsistencies, we assume the agents support the OpenFlow interface and we check for inconsistencies in operations at the interface level. To crosscheck behaviors, we rely either on externally observable results or, when necessary, on probe packets to infer the internal state.

Note that, in the case of hardware switches, we are not interested to verify the underlying switching hardware correctness. In fact, such verification is typically already part of the ASIC design process. However, we assume that there is a way to execute the OpenFlow agent without the switching hardware, *e.g.*, through an emulation layer that is commonly readily available for development and testing purposes.

2.3 Example

Our approach to automatically finding inconsistencies among OpenFlow agent implementations is most easily introduced through an example.

Consider an input sequence that only includes one control message of type `Packet Out`. This message instructs the OpenFlow agent to send out a packet on port p, where p is a 16-bit unsigned integer that identifies a specific port or is equal to one of several preset constants (*e.g.*, flood the packet or send to controller). For the sake of presentation, we assume that only p is symbolic (*i.e.*, p is the only part of this input that varies) and we omit the case $p = 0$ (for which an error message would be produced).

We first symbolically execute an OpenFlow agent implementation while feeding it with this input sequence. When executing symbolically, we automatically partition the in-

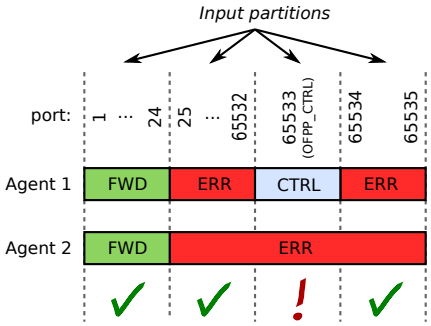

Figure 2: Input space partitions & inconsistency check.

put space of p into several subspaces. Each subspace is an equivalence class of inputs that, in this case, describes which values of p follow the same code path. To make the point more tangible, consider Agent 1 in Figure 1: if $p \in [1, 24]$ the program executes the code path that sends the packet on port p; if $p =$ `OFPP_CTRL` (the predefined controller port) the program executes a different code path that encapsulates the packet in a `Packet In` message and sends it to the controller; and so on. Besides determining the input space partition, we log the output results produced when executing each code path (*e.g.*, we log what packet comes out from which port). Therefore, for each input subspace there exists a corresponding output trace.

Next, we symbolically execute a different OpenFlow agent implementation (Agent 2 in Figure 1) and determine the partitions of input space of p. However, assume that this second OpenFlow agent does not support the special port number `OFPP_CTRL`. Instead, the program sends an error message to the controller when it encounters this case. Likewise, we log the output results produced when executing each code path.

At this point, we have two input space partitions (one for each OpenFlow agent implementation), as depicted in Figure 2. Within each partition, we then group the subspaces by output result (illustrated with different colors in Figure 2). That is, we merge together two subspaces (two code paths), if they produce the same outputs. Such grouping results in two coarse-grained input space partitions–one for each agent. Next, we consider the cross product of the coarse-grained partitions (*i.e.*, all pair-wise combinations of subspaces between the two partitions). From the cross product, we exclude pairs of subspaces that correspond to identical output results. Finally, we intersect the two subspaces in every remaining pair. A *non-empty intersection* defines a subspace of inputs that give different results for different OpenFlow agents: this is an inconsistency. For each inconsistency we discover, we construct a concrete test case that reproduces the observed results. Relative to our current example, we identify that one inconsistency exists and, to reproduce it, we construct the example with input $p =$ `OFPP_CTRL` as illustrated in Figure 2.

2.4 Usage

OpenFlow switch vendors can use SOFT for interoperability testing in two phases. In the first phase, each vendor independently runs SOFT on its OpenFlow agent implementation to produce a set of intermediate results that contain the input space partitions and the relative output results. One benefit of this approach is that a vendor does not require access to the code of other vendors.

In the second phase, SOFT collects and crosschecks these intermediate results to identify inconsistencies. This phase can take place as a part of an inter-vendor agreement (*e.g.*, under an NDA), or during wider interoperability events [3]. Alternatively, a third-party organization such as Open Networking Foundation (ONF) may conduct the tests.

While we focus the presentation of SOFT on interoperability testing, we want to clarify that there exist other applications. For example, SOFT can automate performing regression testing. In addition, it can be used to compare against a well-known set of path conditions that are bootstrapped from unit tests.

We observe that an OpenFlow agent is potentially a software component of a hardware device. As such, some operations can install state directly in the switching hardware (*e.g.*, forwarding rules), seemingly outside of SOFT's reach. We note, however, that vendors typically have a way of running their firmware inside a hardware emulator for testing purposes. We only require that the hardware emulator is integrated with the symbolic execution engine. Previous work (*e.g.*, [12]) demonstrates that it is indeed possible to run complex software systems live, including closed-source device drivers.

3. SYSTEMATIC OPENFLOW TESTING

Our goal is to enable systematic exploration of inconsistencies across multiple OpenFlow agent implementations. In other words, we want to find whether there exists any sequence of inputs under which one OpenFlow agent behaves differently than another agent. To do this, we require a way of (*i*) constructing sequences of test inputs that cover all possible executions for each OpenFlow agent, and (*ii*) comparing the output results that each input produces to identify inconsistencies.

We accomplish the subgoal of finding test inputs by using symbolic execution. The outcome of symbolic execution is twofold: (*i*) a list of path conditions, each of which summarizes the input constraints that must hold during the execution of a given path, and (*ii*) a log of the observed output results for each path executed.

We then identify inconsistencies by grouping the path conditions that share the same output results on a per-agent basis and finding the input subspaces that satisfy the conjunction of the path conditions. Figure 3 provides an illustration of the operation of SOFT as described above. In the remainder of this section, we discuss our approach in detail. After a brief description of a strawman approach for utilizing symbolic execution in functional equivalence testing, we analyze improvements required to apply it to complex software such as OpenFlow agents. Finally, we discuss how we solve the second problem, namely collecting and comparing relevant outputs.

3.1 Automating equivalence testing

Our form of interoperability testing can be viewed as checking the functional equivalence of different OpenFlow agents at the interface level (*i.e.*, the OpenFlow API). To understand how we can use symbolic execution for this purpose, let us first consider a simpler problem.

A strawman approach. Assume we have two functions that implement the same algorithm differently and we want

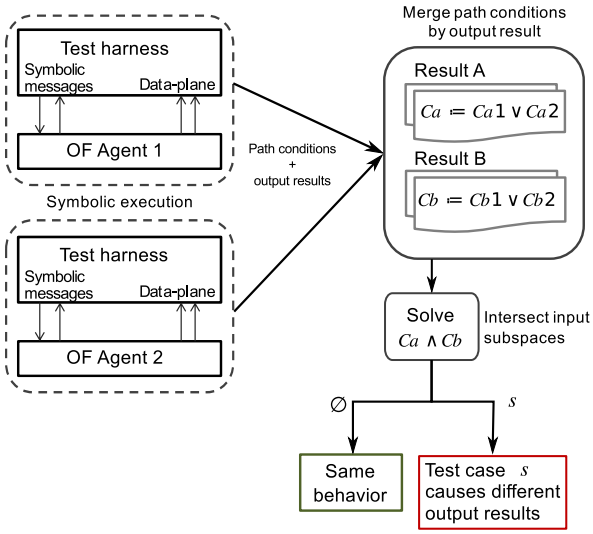

Figure 3: SOFT overview.

to test if they are indeed functionally equivalent. To do this, it is sufficient to symbolically execute both functions by passing identical symbolic inputs to both of them and checking whether they return the same value. If the results differ, the symbolic execution engine can construct a test case to exercise the problematic code path. In essence, symbolic execution enables us to crosscheck the two functions' results through all possible execution paths. This simple approach is sound, *i.e.*, it identifies all cases where results differ, provided that symbolic execution can solve all constraints it encounters. It is also relatively straightforward to extend this approach to crosscheck console utility programs by running with the same symbolic environment and comparing the data printed to `stdout`, as shown in [9].

Challenges and approach. Scaling up this approach to our target system is not an easy task. An OpenFlow agent is a non-terminating, event-driven program that interacts intensely with its environment. In this case, the environment consists of the network data plane, other switch components (*e.g.*, FIB) and the controller.

The first challenge this raises is that the input space is inherently infinite, thus making the problem of comparing OpenFlow agents over unbounded inputs intractable. Instead, to make our problem tractable, we must limit the length of any input sequence used for testing.

Secondly, crosschecking the results of different OpenFlow agents is challenging because there exists no notion such as "switch return value". Furthermore, there does not exist a universal `stdout` format that enables textual comparison unlike console utilities. Instead, we must collect a trace of switch *output results* that enables comparison using detailed information from both the OpenFlow and the data plane interfaces. In other words, we must for example capture packets and OpenFlow messages emitted by the switch, and maintain a non-ambiguous representation of these events.

Third, the approach above works by feeding both functions with the same symbolic input. In turn, this requires that both agents be locally available. However, we cannot assume that SOFT will operate on different OpenFlow agents at the same time. Instead, we make a conscious design choice to decouple the symbolic execution phase from the crosschecking phase.

3.2 Creating symbolic inputs

An OpenFlow agent reacts to OpenFlow messages and data plane packets it receives. Therefore, sequences of such messages can be considered inputs to the agent. In this subsection, we only consider the control channel inputs (the messages sent by the controller) because our goal is to test a switch at the OpenFlow interface (and not the data plane interface).

3.2.1 Structuring inputs

Feeding unstructured inputs is ineffective. As the input space containing sequences of arbitrary numbers of arbitrary messages is infinite, we need to enforce the maximum length of the sequence. A straightforward way to limit the input size would be to use N-byte symbolic inputs, with N bounded. Unfortunately, this approach quickly hits the scalability limits of exhaustive path exploration because these inputs do not contain any information that is of either syntactic or semantic value. As a result, symbolic execution must consider all possible ways in which these symbolic inputs can be interpreted (most of which represent invalid inputs anyway) to exhaust all paths. As an example, consider feeding an agent with the mentioned sequence of N symbolic bytes. Since there exist different types of control messages, some of which have variable lengths, this stream of N bytes can be parsed (depending on its content) as: one message of N bytes, or as any combination of two messages whose lengths add up to N, or as combinations of three messages, *etc.*

Moreover, two messages, `Flow Mod` and `Packet Out`, are variable in length. This is because they both contain the actions field which is a container type for possible combinations of forwarding actions. The major issue arises as each individual action is itself variable in length. As such, we are again in the situation where symbolic execution is left to explore all possible combinations in which it can interpret N symbolic bytes as multiple action items. Although individual lengths must be multiple of 8 bytes to be valid, the combinatorial growth quickly becomes impractical.

Structuring the inputs improves scalability. We overcome the aforementioned problems by using a finite number of finite-size inputs. Most importantly, we construct inputs that adhere to valid format boundaries of OpenFlow control messages rather than leaving symbolic execution to guess the correct sizes. This means that we feed the agent with one symbolic control message at a time and pass the actual message length as a concrete value in the appropriate header field. In practice, we must also make the message type concrete before establishing a valid message length, as the latter is essentially determined by the former. This is not an issue, since every message must be identified by a valid code (at present about 20 codes exist, all described in the protocol specifications [5]). In a similar fashion, for messages that have variable length actions, we predetermine the number of action items and the relative lengths as concrete values.

3.2.2 Choosing the size of inputs

As we choose to limit the size of inputs, the immediate question we face is up to what input size is it practical to symbolically execute an OpenFlow agent, given today's tech-

nology? Indeed, it is known that the scalability of symbolic execution is limited by the path explosion problem: *i.e.*, the number of feasible paths can grow exponentially with the size of the inputs and number of branches. On the other hand, to make testing meaningful, the chosen inputs need to provide satisfactory coverage of agent's code and functionality. In practice, we seek answers through empirical observations. The input size varies along two dimensions: (*i*) number of symbolic control messages, and (*ii*) number of symbolic bytes in each message.

Covering the input space of each message is generally feasible. We first explore to what extent the number of symbolic bytes in each message represents a hurdle to our approach. As we discuss below, we find that the overhead to exhaustively cover the input space of each message is generally acceptable, given the current protocol specifications. We already mentioned that the message length depends in the first place on the message type. It should also be clear that the processing code and especially the processing complexity varies across message types. For example, it is trivial to symbolically execute a message of type `Hello`, which contains no message body. On the other hand, the `Flow Mod` message, which drives modifications to the flow table, carries tens of data fields that need validation and ultimately determine what actions the switch will perform. Indeed, we observe through experimentation that the number of feasible paths varies significantly between different message types (two orders of magnitude between `Flow Mod` and `Packet Out` messages). Most importantly, symbolic execution runs to completion in all cases when testing with the reference OpenFlow implementation.[2]

Achieving good coverage requires just two symbolic messages. However, the question remains about how many symbolic control messages we should inject in practice. Again, the answer depends on what type of messages one considers. We find that for complex messages we can at most use a sequence of three messages. This number may seem small, but it is worth noting that we do not need long message sequences for the type of testing we target. In fact, one symbolic message is already sufficient to cover all feasible code paths involved in message processing. With the subsequent message, we augment the coverage to include additional paths that depend on parts of switch state that are rendered symbolic as a result of running with the first symbolic message. Effectively, the second message enables us to explore the cross-interactions of message pairs. In addition, such interactions exist only for a small fraction of possible message type combinations. For example, two `Flow Mod` messages may affect the same part of the switch state; that is not true for `Echo Request` followed by `Flow Mod`. As such, the increase in instruction coverage due to the second message is a fraction of what the first message covers. A third message does not significantly improve coverage further as shown in Figure 4. Thus, careful consideration of inputs is key to successfully achieving our goals through symbolic execution.

3.2.3 *Defining relevant input sequences.*

Exploiting domain specific knowledge is essential to construct input sequences that target interesting uses of OpenFlow messages to further reduce the testing overhead. First, although the protocol specifications define about 20 messages, some of these are clearly more important than others. For example, the `Hello` and `Echo` messages are simple connection establishment and keep-alive messages, respectively. We focus on complex messages such as `Flow Mod`, `Packet Out`, `Set Config` that require validation and modify the state of an agent. We also note that because these messages are meant to affect different functional aspects of the agent, we find it unnecessarily time-consuming to check all pair-wise combinations of these messages. Section 5 details the actual sequences of messages we use for testing.

[2]Our experimental setup is introduced in Section 5.

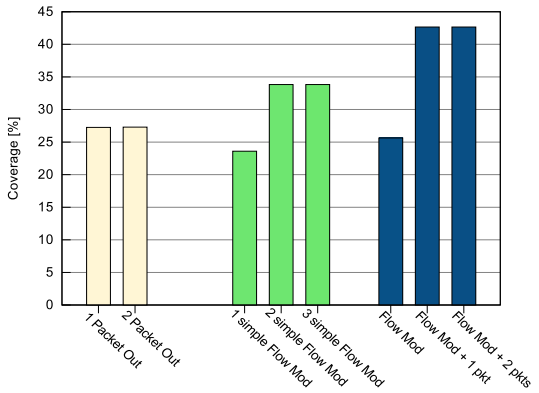

Figure 4: Reference switch code coverage as a function of the number of symbolic messages.

3.3 Collecting output results

So far we have shown how our approach uses exhaustive path exploration to obtain the input space partitions (or equivalence classes of inputs). However, we still need to know what end result each partition produces because only the results enable the comparison across different OpenFlow agents.

As we feed a symbolic message to an OpenFlow agent, its state might be updated. Additionally, there are two possible outcomes: (*i*) the agent outputs some data (*i.e.*, messages back to the controller or data plane packets), or (*ii*) the agent does not produce externally observable data. In this work we treat only data explicitly returned by an agent (OpenFlow messages and data plane packets) as an output. Instead of directly fetching the internal state, we use additional packets and messages to infer the impact of the state on agent's behavior.

Capturing output data. To collect the outputs, we make use of the OpenFlow and data plane interfaces to capture data. Specifically, we log all OpenFlow messages and packets emitted by the agent. Note that the entire analysis runs in software (the output data may even contain symbolic inputs); therefore, with data plane interface we simply mean the socket API (or equivalent) that the agent uses to send packets.

Using concrete packets for probing state. Regardless of whether the agent does or does not output data, we cannot immediately determine if the symbolic message caused any internal state change (*e.g.*, the `Flow Mod` message installs a new rule in the flow table). Differences in internal state not necessarily result in differences in observed behavior. Moreover, we want to avoid directly fetching an agent's internal state as this would require a dependency on the specific implementation. As a solution, following any poten-

tially state changing symbolic message, we inject a concrete packet through the data plane interface as a simple state probe. The effect of this probe is that it enables symbolic execution to exercise the code that matches incoming packets and the code that applies the forwarding actions. The probe packet is then either forwarded (to a port or controller), in which case we log it, or it is dropped, in which case we log an empty probe response.

Normalizing results. Rather than saving the logs verbatim, we normalize the output results to remove certain data from the results for which spurious differences are expected. For example, the buffer identifiers used by different agents may differ and such a difference should not be considered an inconsistency.

3.4 Finding inconsistencies

In this phase, we seek to find inconsistencies between two OpenFlow agents denoted A and B.

With respect to agent A, we denote with PC_A the set of path conditions (outcome of symbolic execution). For each $pc \in PC_A$, let $\text{res}_A(pc)$ be the normalized output result when executing the path represented by pc. We denote the set of distinct output results as RES_A.

Grouping paths by output results. Our first step is to group all different path conditions that produce the same output result. Formally, $\forall r \in RES_A$ we set $C_A(r) = \bigvee\{pc \mid pc \in PC_A;\ \text{res}_A(pc) = r\}$ to be the disjunction of all path conditions that share the same output result. PC_B, RES_B, and C_B are similarly defined.

Intersecting input subspaces. In our second and last step, for each pair of different outputs of agents A and B, we check if there exists at least one common input that leads to these inconsistent outputs. For each pair (i, j) of results $i \in RES_A$, $j \in RES_B$ s.t. $i \neq j$, we query a satisfiability solver (STP [14]) to obtain an example test case that satisfies the condition $C_A(i) \bigwedge C_B(j)$. If the solver can satisfy this conjunction, then we have an inconsistency.

Discussion. It is easy to note that an upper bound of the number of queries to the solver for our approach is $|RES_A| \cdot |RES_B|$. In addition, note that our approach produces only one inconsistency example per pair of different output results. In other words, we do not provide one example for each path that produces the inconsistency. If this is desired, one can omit grouping all paths that share the same output. However, doing so has an inherent overhead cost because it increases the number of STP queries. Instead, our approach amortizes the start-up costs of a multitude of solver invocations by using fewer larger queries and enables the solver to apply built-in optimizations to handle such larger queries.

As with any bug finding tool, it is important to know whether our approach incurs in false positives/negatives. We observe that SOFT does not produce false positives: each identified inconsistency is evidence of divergent behavior. Note this does not necessarily mean that one agent does something in violation to the specifications. According to our previous definition, an inconsistency is reported if the tested agents perform different actions when exposed to the same input. However, the tool might have false negatives for two reasons. The first is that our path coverage may not be complete. For instance, symbolic execution might not cover all feasible paths due to path explosion. The second is that all agent implementations under test might contain the same bug, and therefore produce the same output.

4. SOFT PROTOTYPE

We built our SOFT prototype on top of the Cloud9 [8] symbolic execution engine. SOFT consists of three major components: (*i*) a test harness, which drives the testing of OpenFlow agents, (*ii*) a grouping tool to group path conditions that share output results, and (*iii*) a tool for finding inconsistencies.

4.1 Test harness and Cloud9

To provide the necessary execution environment for the OpenFlow agent, we build a test harness that emulates both a remote controller and the underlying network. The emulated controller is capable of injecting symbolic inputs.

As a symbolic execution engine, Cloud9 can symbolically execute only a single binary. We therefore create a test "driver" by linking the OpenFlow agent and our test harness controller together. Upon startup, the test driver forks into two processes, one of which runs the OpenFlow agent while the second runs the test harness itself. The two processes are connected via standard UNIX sockets. Upon startup, the OpenFlow agent connects to the test harness. After the connection setup and exchange of the initial `Hello` messages, the test harness injects a sequence of several symbolic OpenFlow messages and/or probes, one at a time (we discuss the input sequences in more details in Section 5). Upon confirming that the switch processed all messages and probes, we kill the execution.

To use Cloud9 for our goal, we had to improve its environment model. Cloud9 provides a symbolic model of the POSIX environment. Such a model, most importantly, allows us to efficiently use the socket API without accessing the entire networking stack. As a result, all symbolic variables remain symbolic after being transferred as data in a packet. However, such a model needs to provide all functions used by the tested application. Notably, we needed to implement the RAW socket API which was missing in Cloud9 but is used by the OpenFlow agents in our tests. Moreover, we replace or simplify some library functions as described next.

We assume that the agents correctly use network versus host byte ordering, and we change functions `ntoh` and `hton` to simply return their argument unchanged. This simplifies constraints by removing double-shuffling (first when the test harness creates a message, second when the OpenFlow agent parses the message). We also simplify checksum and hash functions to return constants or identities, because they cannot be reversed or it is computationally very expensive to do so (this is a well-known issue in using a constraint solver). The aforementioned modifications reduce complexity and improve symbolic execution efficiency.

Finally, the symbolic execution engine may use several search strategies that prioritize different goals while exploring the program. We choose to use the default Cloud9 strategy that is an interleaving of a random path choice and a strategy that aims to improve coverage. However, the choice of the search strategy has small impact on our tool. By controlling the inputs we tend to exhaustively cover all possible execution paths, which in turn diminishes the impact of choosing a particular search strategy. Moreover, SOFT is

capable of working with traces that are only partially covering agents' code.

4.2 Tools

Apart from the test harness, we provide two tools for manipulating Cloud9 results. Both of these tools are written in C++ and heavily reuse existing Cloud9 code for reading, writing and manipulating path conditions. The tools contain less than 200 lines of new code in total.

The group tool reads multiple files (results of Cloud9 execution), identifies different output results and groups the path conditions by result. To improve performance of further constraint parsing, we group path conditions by building a balanced binary tree minimizing the depth of nested expressions. The inconsistency finder tool expects two directories holding grouped results as its arguments. The tool iterates over all combinations of different results and queries the STP solver to check for inconsistencies. If there is an inconsistency (the condition is satisfiable), the STP solver provides an example set of variables that satisfy the condition. This is a test case that can be used to understand and trace the root cause of the inconsistency and verify if a behavior is erroneous.

5. EVALUATION

We evaluate SOFT using two publicly available OpenFlow agents compatible with the specifications in version 1.0. The first one is a reference OpenFlow switch implementation written in C released with version 1.0 of the specifications. Its main purpose is to clarify the specifications and present available features. Although the reference implementation is not designed for high performance, it is expected to be correct as others will build upon and test against it. We are referring to this version as Reference Switch (55K LoC). The second is Open vSwitch 1.0.0 [4] (80K LoC). It is a production quality virtual switch written in C and used in several commercial switches.[3] OpenFlow is just one the supported protocols. We also created a third OpenFlow agent by modifying the Reference Switch and introducing different corner case behaviors (Modified Switch). This way we can tell how efficiently SOFT finds the injected differences and which of them remain unnoticed.

To evaluate SOFT we use the set of tests summarized in Table 1. We run our experiments using a machine with Linux 3.2.0 x86_64 that has 128 GB of RAM and a clock speed of 2.4 GHz. Our implementation does not use multiple cores for a single experiment.

5.1 Can SOFT identify inconsistencies?

In this section, we report and analyze the inconsistencies SOFT detects. We apply a set of tests to all three OpenFlow agents and compare Reference Switch with both Modified Switch and Open vSwitch.

5.1.1 Modified Switch vs. Reference Switch

First, we look for differences between Reference Switch and Modified Switch. Two team members who did not take part in the tool's implementation and test preparation were designated to introduce a few modifications to the Reference Switch. The modifications were meant to affect the externally visible behavior of the OpenFlow agent. Having purposefully injected changes, we set out to check how many can be detected by SOFT.

[3]For example, in Pica8 products: `http://www.pica8.org/`.

Test	Description
Packet Out	A single `Packet Out` message containing a symbolic action and a symbolic output action.
Stats Request	A single symbolic `Stats Req`. It covers all possible statistics requests.
Set Config	A symbolic `Set Config` message followed by a probing TCP packet.
FlowMod	A symbolic `Flow Mod` with 1 symbolic action and a symbolic output action followed by a probing TCP packet.
Eth FlowMod	Symbolic `Flow Mod` with 1 symbolic action and a symbolic output action. Fields not related to Ethernet are concretized. The message is followed by a probing Ethernet packet.
CS FlowMods	2 `Flow Mod`. The fist one is concrete, the second is symbolic.
Concrete	4 concrete 8-byte messages. These are the messages that do not have variable fields.
Short Symb	A 10-byte symbolic message. Only the OpenFlow version field is concrete.

Table 1: Tests used in the evaluation.

SOFT is able to correctly pinpoint 5 out of 7 injected modifications. We further investigate the cases in which SOFT failed to flag the effect of the differences. It turns out that one of them concerns the `Hello` message received while establishing a connection to the controller. SOFT does not recognize this problem because it establishes a correct connection first and then performs the tests. The second missed modification manifests itself only when a rule is deleted because of a timeout. This occurs because the symbolic execution engine is not able to trigger timers. As part of our future work, we plan to extend our approach to deal with time, *e.g.*, similarly to MODIST [24].

5.1.2 Open vSwitch vs. Reference Switch

Knowing that SOFT is capable of finding inconsistencies, we compare the Reference Switch with Open vSwitch to verify how useful SOFT is when applied to a production quality OpenFlow agent. The list of differences between the two major software agents contains a few significant ones. In the following, we present the observed inconsistencies and analyze their root causes.

Packet dropped when action is invalid. This case describes a `Packet Out` message containing a packet that is silently dropped by Open vSwitch while the Reference Switch forwards it. The inconsistency appears when the `Packet Out` control message satisfies the following conditions: (*i*) it contains the packet that the agent should forward and (*ii*) one of the actions is setting the value of VLAN or IP Type of Service field. Further investigation leads us to the conclusion that Open vSwitch validates whether a new VLAN value set by the action fits in 12 bits and similarly whether the last two bits of the TOS value are equal to 0. When an action specified in the message does not pass this strict validation, Open vSwitch silently ignores the whole message. Additional tests with `Flow Mod` messages reveal a similar issue. These tests also show that the `vlan_pcp` field undergoes additional validation in Open vSwitch. Reference Switch does not validate values of the aforementioned fields, but it automatically modifies them to fit the expected format.

The specifications do not state that the OpenFlow agent should perform such a precise validation of any of the men-

Test	Message count	Reference Switch				Modified Switch				Open vSwitch			
		CPU time	Path count	Constraints avg size	Constraints max size	CPU time	Path count	Constraints avg size	Constraints max size	CPU time	Path count	Constraints avg size	Constraints max size
Packet Out	1	14s	49	71.57	96	24s	117	74.09	91	44s	241	63.48	79
Stats Request	1	44s	218	53.10	65	46s	218	53.10	65	186s	136	52.63	67
Set Config	2	446s	207	76.89	112	451s	207	76.89	112	569s	207	80.97	116
Eth FlowMod	2	40m	7680	101.12	132	83m	14280	106.62	136	198m	27682	98.56	127
FlowMod	2	373m	87828	109.77	161	>140h	>356753	123.09	164	60h	181620	123.28	159
CS FlowMods	2	69m	29179	96.63	136	121m	51419	99.74	139	36h	462488	115.22	173
Concrete	4	6s	1	0	0	6s	1	0	0	8s	1	0	0
Short Symb	1	50s	31	27.9	69	50s	31	27.9	69	12s	14	27.5	50

Table 2: Symbolic execution statistics for selected tests for all 3 OpenFlow agents. We report time, number of explored paths (input equivalence classes) and constraint size (average and maximum size).

tioned fields. Therefore, both implementations might be considered correct. However, such a difference in behavior might cause unexpected packet drops if the controller developers test their applications with switches that are different from those deployed in the network.

Forwarding a packet to an invalid port. Here we describe a case in which the tested OpenFlow agents return error messages concerning incorrect output ports in an inconsistent fashion. According to the specifications, the agent has to return an error message if the output port will never be valid. However, if the port may become valid in the future, the message might either be rejected with an error, or the agent may drop packets intended for this port while it is not valid. The differences in interpretation when the port will be invalid forever lead to a few differences between OpenFlow agents. First, when the ingress port in the match is equal to the output port, the Reference Switch returns an error, as no packets will ever be forwarded to this port.[4] Open vSwitch accepts such a rule and drops all matching packets. On the other hand, Open vSwitch immediately returns an error when the action defines an output port greater than a configurable maximum value. Reference Switch does not validate ports this way.

Thus, if the controller application relies on error messages received, it may misbehave when deployed with a different agent than it was tested with. If the agent used in testing considered a port valid but the other agent did not, the controller would fail to install rules it was designed to install. The opposite situation is equally unsafe. The rule installation that used to return an error succeeds, but all matching packets get dropped. As a result, some packets will not to be sent to the controller, although they were expected to be. Moreover, such a rule may cover another, lower priority one.

Lack of error messages. We have already presented a few cases when one of the agents silently drops the incorrect message without returning an error. SOFT detects another instance of such a problem in the Reference Switch while testing with `Packet Out` and `Flow Mod` messages. When the `buffer_id` field refers to a non-existent buffer, the Reference Switch handles the message but does not apply actions to any packet and does not report any error. Open vSwitch replies with an error message, but installs the flow as well. We analyzed the Reference Switch source code and discovered that although the error is returned by the message handler, it is not propagated further as an OpenFlow message.

OpenFlow agent terminates with an error. There are three independent cases when the Reference Switch crashes. First, when the OpenFlow agent receives a `Packet Out` message with output port set to `OFPP_CTRL`. This may be a rare case (*e.g.*, when the developer demands such behavior) but it is not forbidden by the specifications. Second, when the agent executes an action setting the `vlan` field in a `Packet Out` message the same error appears and the agent crashes. Finally, when the agent receives a queue configuration request for port number 0, it encounters a memory error. All the aforementioned problems are not only inconsistencies, but also major reliability problems in the OpenFlow agent.

Different order of message validation. In this case, the order in which message fields should be validated is not made explicit in the specifications. This vagueness results in externally visible differences in agents' behavior. The same incorrect message may induce two different error messages, or an error message and a lack of response in case of the mentioned problem. We encountered such a situation for a `Packet Out` message with an incorrect buffer id and output port.

Statistics requests silently ignored. The Reference Switch silently ignores requests for statistics to which it is not able to respond. This behavior is a specific case of the "Lack of error messages" problem. Even though the handler returns an error it is not converted to an OpenFlow message. The problem was detected because Open vSwitch sends an error in response to an invalid or unknown request.

Missing features. SOFT is able to detect features that are missing in one OpenFlow agent, but are present in the other. We were able to automatically infer that Open vSwitch does not support emergency flow entries that are defined in the specifications. Secondly, Reference Switch being purely an OpenFlow switch, does not support the traditional forwarding paths (`OFPP_NORMAL`).

5.2 What is the overhead of using SOFT?

In this section, we present the performance evaluation of the two key stages of SOFT's execution.

Symbolic execution. In the first stage, the OpenFlow agent is symbolically executed with an input sequence and SOFT gathers path constraints and corresponding outputs. For all three OpenFlow agents we report the running time, as well as the number and size (number of boolean operations in a path condition) of paths (equivalence classes of inputs) in Table 2. These metrics are strongly variable and depend not only on the input length but also on the message type. Moreover, adding a second message or a probe packet significantly increases complexity by orders of magnitude. Additionally, Open vSwitch–the most complex of the tested agents–is noticeably more challenging for symbolic execu-

[4]A special `OFPP_IN_PORT` port must be explicitly used to forward packets back to the port they came from [5].

Test	Grouping results				Inconsist. checking	
	Reference Switch		Open vSwitch			
	time	#res	time	#res	time	#
Packet Out	0.038s	6	0.090s	10	26s	14
Stats Request	0.116s	8	0.061s	9	10s	7
Set Config	0.141s	69	0.43s	69	236s	0
Eth FlowMod	8s	12	23s	31	23m	58
CS FlowMods	79s	4	344m	6	>28h	≥8
Short Symb	0.039s	9	0.01s	7	6s	4

Table 3: Time needed to find overlapping input subspaces and number of created test cases. Each test case represents one intersection of overlapping input subspaces. Additionally, time needed to group constraints by the output and a number of distinct outputs for Reference Switch and Open vSwitch.

tion (we note that it is possible to use even partial results of symbolic execution to look for inconsistencies). As a result of multiple additional validations, the test input space for Open vSwitch is partitioned into 3-15 times more subspaces than for the Reference Switch.

Subspaces intersections. We distinguish between two sub-stages of the second stage: (*i*) grouping input subspaces by the same output, (*ii*) intersecting subspaces corresponding to potential inconsistencies.

For the first sub-stage we report the time required to group and the number of distinct outputs. As presented in Table 3, this part requires orders of magnitude less time than symbolic execution. Grouping constraints dramatically reduces the number of expressions that need to be checked for satisfiability, as there are only up to 30 distinct outputs (a 1-5 orders of magnitude reduction compared to the initial number of equivalence classes).

The search for overlapping subspaces depends on the complexity of constraints and usually finishes within a couple of minutes. There is one exceptional case in which the STP solver is unable to solve the merged constraints in one day. In the future we plan to investigate grouping constraints into smaller groups for such cases.

The achieved results in finding inconsistencies confirm our expectations. Usually one difference manifests itself multiple times and affects many subspaces of inputs. In the extreme example, although there are 58 reported inconsistencies, manual analysis reveals only 6 distinct root causes of differences.

5.3 How relevant is input sequence selection?

To quantify the relevance of chosen tests, we measure the instruction and branch coverage provided by Cloud9. The instruction/branch reached at least once in the execution is considered covered, regardless of its arguments. We consider only the sections of OpenFlow agent's code relevant to OpenFlow processing. The initialization that is repeated for each test covers 12% of instructions and 8% of branches. The test specific results, shown in Table 4, are spread between 20 and 40%. To verify that the low reported coverage is a result of the fact that each test targets a few specific message handlers, we manually analyze cumulative coverage of all tests. We observe that SOFT covers approximately 75% of the code and that the remaining instructions belong mostly to code that is not accessible in standard execution (*e.g.*, command line configuration, dead code, cleanup functions, logging functions).

The importance of concretizing inputs. Due to time

Test	Reference Switch		Open vSwitch	
	Inst.(%)	Branch(%)	Inst.(%)	Branch(%)
No Message	12.21	8.27	19.03	13.34
Packet Out	26.23	19.31	25.68	17.28
Stats Request	30.27	24.15	24.31	16.75
Set Config	26.23	19.31	23.98	16.16
Eth FlowMod	41.74	34.65	38.15	25.49
FlowMod	42.65	34.25	38.24	26.27
Concrete	17.13	11.42	20.16	13.62
Short Symb	19.92	13.39	21.60	14.34

Table 4: Instruction and branch coverage for selected tests for Reference Switch and Open vSwitch.

Test	Time	Paths	Coverage
Fully Symbolic	31h	226224	42.93%
Concrete Match	12m	2634	40.60%
Concrete Action	193m	30396	37.32%
Concrete Probe	48m	9216	41.6%
Symbolic Probe	172m	33168	43.9%

Table 5: Effects of concretizing on execution time, generated paths and instruction coverage.

and memory constraints it is often convenient to concretize selected fields in the message. We evaluate the benefits and drawbacks of using the domain knowledge to reduce the input space. As a baseline, we choose a test where a single symbolic `Flow Mod` message containing 2 symbolic actions and 2 symbolic output actions is followed by a TCP probe packet. We then compare the results of: (*i*) the baseline, (*ii*) a version of the baseline with a concrete match (wildcard), and (*iii*) a version of the baseline with a single concrete action instead of 4 symbolic ones. All values are summarized in the upper part of Table 5. While the drop in the coverage percentage is only 2-5% in comparison to the baseline test, the difference in time and path count is noticeable. Specifically, the tests finish 10 to 50 times quicker, while generating 1 to 2 orders of magnitude less paths.

To verify how much coverage we lose by not using symbolic probes, we create a separate test. This test first installs a partially symbolic `Flow Mod` that applies actions to Ethernet packets. It then sends a short probe packet that is concrete or symbolic depending on the test version. Results in the lower part of Table 5 show that a symbolic probe adds just 2% to the coverage. The cost is 3.5 times longer running time and 3.5 times more paths.

To summarize, concretizing parts of the inputs significantly reduces the time needed to conduct the test at the cost of leaving small portion of additional instructions uncovered. Therefore, it is possible to use the concretized inputs to conduct regular tests more often. When combined with careful choice of concrete fields, the coverage is marginally affected. The fully symbolic messages can be used just for the final checks before a major release when the best coverage possible is required, and testing time is less of an issue.

6. RELATED WORK

We present in the following generic techniques that can be applied to testing OpenFlow switches, as well controller applications and networks in general.

6.1 Testing OpenFlow switches

There are multiple testing approaches applicable to OpenFlow switches. The approaches differ in terms of the scope

and type of problems they aim for, as well as the process in which the test cases are created. As an exhaustive review is beyond the scope of this paper, we briefly describe a few relevant to our discussion.

System testing. System-level testing is concerned with an integrated system such as an OpenFlow switch. This approach treats the device under test as a black box. To ensure that the tested device does not depend on external factors, the interactions with the controller and other network elements are commonly emulated by the testing framework. This approach typically requires a large number of test cases to achieve high coverage. Each test case is carefully designed to target a specific feature and checks the correctness of simple functionalities. A developer, using tools such as OFTest [2] or the default OpenFlow Perl testing framework, has to manually provide a step by step execution scenario containing the inputs and expected outputs. This process is time-consuming and additionally makes designing non-trivial and complex test cases complicated.

Symbolic and concolic execution. Others have successfully applied symbolic execution [9] and selective symbolic execution [12] to testing of systems code. As we already mentioned, blindly applying symbolic execution results in an exponential explosion of code paths. It also requires excessive human effort to specify correct behavior. SOFT effectively overcomes these issues. With these issues resolved, one could use symbolic execution for crosschecking switch behaviors, but it results in excessive, time-consuming overhead. SOFT goes one step further in that eliminates this step by coalescing constraints that result in the same output, and using the constraint solver to identify inconsistent behaviors.

Others have considered the problem of manipulating inputs to conform to an input grammar, in the form of white-box "fuzz" testing [16]. By doing so, the symbolic execution engine can quickly pass over the validation checks to try to reach deeper in the code. The problem that SOFT addresses is even harder, as we need to be careful about the number and type of messages, as well as the nature of individual fields. In addition, we address the problem of observing internal state.

Canini *et al.* [10] use a variant of symbolic execution called concolic execution to identify faults in federated, heterogeneous distributed systems. Their system, called DiCE, tests the impact of feeding various inputs to participating nodes (*e.g.*, BGP routers) in isolation. DiCE and SOFT differ in several ways. DiCE is an online technique, whereas SOFT is used for interoperability testing prior to deployment. Moreover, SOFT's goal is crosschecking of different implementations. Finally, SOFT does not require the definition of correct behavior to be specified. Complementary to SOFT, Kothari *et al.* [19] use symbolic execution to identify protocol manipulation attacks. The goal here is for a node to try to determine harmful behavior induced upon itself by received messages from other participants. In contrast, SOFT systematically determines and compares the input subspaces of multiple implementations to find inconsistencies, without prior knowledge of correct behavior.

Performance testing. Performance tests are a subset of system tests used to determine device's capability under high load. Not only is this type of tests able to detect performance problems such as slow packet forwarding or control plane communication, but can also be successfully applied to find correctness problems that may not appear in other scenarios. As shown in OFLOPS [22], a continuous packet stream may be used to check the consistency between data plane and control plane in case of the OpenFlow barrier commands. Moreover, this method is able to discover timing issues that require multiple events occurring with a specific time correlation. On the other hand, it may potentially miss functional errors and classify them as correct behavior in some circumstances. Finally, performance testing requires that, while under test, the system works under realistic conditions. Packets, for example, need to be injected to the device at the rate and times defined in the scenario. Consequently, in addition to the usual setup time, the tests need to be run in real time and cannot be sped up.

6.2 Testing OpenFlow controllers

NICE [11] is a tool for testing unmodified OpenFlow controller applications. It combines model checking and concolic execution in order to systematically explore the behavior of the network under a variety of possible event orderings. The tool starts with the network topology model containing the controller, switches and end hosts, and exercises sequences of state transitions on these network elements. NICE and SOFT target fundamentally different parts of the network: controller vs. switches. In NICE, only the controller is running the unmodified application, while other elements (switches, end hosts) are replaced with simplified models. In contrast, SOFT finds inconsistencies among the implementations of OpenFlow agents that run in the switches.

Further, SOFT does not require the specification of correct behavior for the tested software, while NICE uses correctness properties provided by developers or testers.

6.3 Trace-based debugging of a network

OFRewind [23] is a tool that enables temporary consistent network event trace recording in a running system, as well as replaying it later. Despite available mechanisms allowing operators to filter recorded events, the debugging process is still manual. Neither problem detection, nor its root cause localization and analysis is automatic and needs supervision. The operator has to first realize that there is an issue in the network, and then find the root cause by replaying subsets of the recorded trace. Although the tool is directed toward debugging, it should be possible to use a similar technique to create test inputs based on previously recorded traces. The efficiency of using pre-recorded traces for future network testing is limited, as the traces explore only one specific execution path (set of network events) and might miss important corner cases.

6.4 Other approaches

Approaches exist for statically analyzing network configurations. For example, RCC [13] identifies misconfigurations in intradomain BGP routers. Anteater [20] uncovers problems in the data plane due to forwarding misconfigurations. Header Space Analysis (HSA) [17] checks the network configurations to identify network configuration problems. Automatic Test Packet Generation (ATPG) [25] is a solution based on HSA that creates a minimum set of test packets required to cover all links or rules in the network. Then, ATPG uses these packets to detect and localize failures. We consider these approaches orthogonal to ours as they fo-

cus on testing the network from the data plane perspective whereas we test switch behaviors as driven by the OpenFlow interface.

7. CONCLUSIONS

Software Defined Networking, and its OpenFlow incarnation in particular, stands a real chance of enabling cheap and easy extensibility in networks. OpenFlow owes its increasing adoption to the relatively simple changes that enable control over the way packets are forwarded by the OpenFlow switches. With little attention on ensuring reliability of OpenFlow switches, danger exists that failures in production networks could erode trust in this new technology. In particular, the OpenFlow specification changes rapidly, and allows for different interpretations. As a result, switches from multiple vendors can behave differently and cause an inconsistency in the network.

In this paper, we have described a tool that automates the task of identifying such deviations in behavior among different switches. We demonstrate the effectiveness of our tool by using it to identify several inconsistencies involving the Reference OpenFlow switch and the Open vSwitch. While the work centered around the specific details of OpenFlow, we posit our approach could find more general application with other router software and heterogeneous networked systems.

8. ACKNOWLEDGMENTS

We thank our shepherd Christos Gkantsidis and the anonymous reviewers who provided excellent feedback. We are grateful to Stefan Bucur for supporting us with using Cloud9 and Jennifer Rexford 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.

9. REFERENCES

[1] Going With the Flow: Google's Secret Switch to the Next Wave of Networking. `http://www.wired.com/wiredenterprise/2012/04/going-with-the-flow-google/all/1`.

[2] OFTest. `http://oftest.openflowhub.org`.

[3] ONF Holds Its First Test Event. `https://www.opennetworking.org/?p=249&option=com_wordpress&Itemid=72`.

[4] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org`.

[5] OpenFlow Switch Specification. `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`.

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

[7] Staring Into The Gorge: Router Exploits. `http://www.renesys.com/blog/2009/08/staring-into-the-gorge.shtml`.

[8] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

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

[10] 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.

[11] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[12] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 30(1):1–49, 2012.

[13] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*, 2005.

[14] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[17] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[18] J. C. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, 1975.

[19] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, 2011.

[20] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

[21] 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.*, 38:69–74, March 2008.

[22] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. Oflops: An open framework for openflow switch evaluation. In *PAM*, 2012.

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

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

[25] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.