SPIDER: Fuzzing for Stateful Performance Issues in the ONOS Software-Defined Network Controller

Ao Li
aoli@cs.cmu.edu
Carnegie Mellon University

Rohan Padhye
rohanpadhye@cmu.edu
Carnegie Mellon University

Vyas Sekar
vsekar@cmu.edu
Carnegie Mellon University

Abstract

Performance issues in software-defined network (SDN) controllers can have serious impacts on the performance and availability of networks. In this paper, we consider a special class of SDN vulnerabilities called stateful performance issues (SPIs), where a sequence of initial input messages drives the controller into a state such that its performance degrades pathologically when processing subsequent messages. Uncovering SPNs in large complex software such as the widely used ONOS SDN controller is challenging because of the large state space of input sequences and the complex software architecture of inter-dependent network services. We present SPIDER, a practical fuzzing framework for identifying SPIs in this setting. The key contribution in our work is to leverage the event-driven modular software architecture of the SDN controller to (a) separately target each network service for SPIs and (b) use static analysis to identify all services whose event handlers can affect the state of the target service directly or indirectly. SPIDER implements this novel dependency-aware modular performance fuzzing approach for 157 network services in ONOS and successfully identifies 10 new performance issues. We present an evaluation of SPIDER against prior work, a sensitivity analysis of design decisions, and case studies of two uncovered SPIs.

1 Introduction

Software-defined networking is increasingly adopted in wide-area, datacenter, and enterprise networks [1]. In contrast to traditional networks where routers and switches run both the routing (i.e., control plane) and forwarding (i.e., data plane), SDN logically decouples of the control and data plane tasks. To this end, SDN introduces a *controller* (e.g., ONOS [2]) that communicates with network devices (routers and switches) through a configuration protocol (e.g., OpenFlow).

The SDN controller performs its tasks based on an internal state which it maintains; this state is updated based on messages received from network hosts and switches, and eventually used to configure the entire network. Given the critical role that the SDN controller plays, vulnerabilities in the controller can lead to undesirable outcomes impacting the overall performance, security, and availability of the network [3–5].

However, finding vulnerabilities in the SDN controller is not trivial. For example, ONOS is a leading open-source SDN controller that is used by many large network providers such as Comcast and AT&T [2]. ONOS is comprised of 150+ network services that communicate with each other asynchronously. Researchers have developed several specialized analyses to identify certain classes of vulnerabilities such as memory-safety issues, protocol race conditions, and configuration issues in SDN controllers such as ONOS [6–9].

In this paper, we consider a new class of vulnerabilities in ONOS which we call *stateful performance issues* (SPIs). First, SPIs are performance issues which lead to excessive resource consumption when processing inputs (i.e., OpenFlow messages). Such issues in an SDN controller can severely compromise the network's availability. Second, SPIs can only be triggered after the SDN controller has reached a specific internal state while processing other messages.

Identifying SPIs is challenging because one must consider a series of messages that first cause the SDN controller to reach a vulnerable state and then finally trigger a very costly state-dependent operation. At a high level, this is a challenging search-space exploration problem due to a combination of algorithmic and system factors. First, we need to consider a large input search space of long *sequences* of OpenFlow messages of interest. The second issue is the large code base and non-trivial software architecture: ONOS has tens of thousands of lines of code comprising hundreds of network services with complex dependencies between them. The third challenge stems from the semantics of SPIs: we need to capture the dependencies between inputs and internal states, and identify which state-input combinations entail high resource consumption.

We present SPIDER, a system for identifying SPIs in the ONOS SDN controller. At its core, SPIDER uses performance fuzzing [10, 11] to automatically generate inputs that maximize execution cost. SPIDER addresses the aforementioned challenges by implementing a novel *dependency-aware modular performance fuzzing* framework.

Our key observation is that ONOS uses an event-based modular software architecture. In this design, network services communicate with each other using asynchronous *events*. Events are first triggered by incoming OpenFlow messages. Network services subscribe to one or more event types; their event handlers can update their internal state and/or fire other events.

SPIDER generates *sequences* of internal events with the goal of triggering an SPI; that is, where the last event in the sequence exacerbates performance in some service S. Our *key insight* in making this scalable is that the only events relevant to such an SPI are those whose processing may directly or indirectly affect the internal state of S. SPIDER leverages this insight in the following way. First, we focus on analyzing one service at a time with the goal of triggering an SPI in just that service. Second, when targeting a service S, we use *static analysis* to identify inter-service dependencies. Finally, our performance fuzzer uses the dependency information to generate event sequences that *only* contain events that may affect the state of S. Such a *dependency-aware modular analysis* allows SPIDER to reduce the search space without sacrificing fidelity.

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For event generation, we borrow the idea from Zest [12], which uses type-specific generator functions for representing and mutating well-formed inputs. For most event types, we can synthesize such generators automatically using their type definitions. For about 10% of event types in the critical path of many services, we use handcrafted input generators to improve fidelity.

We use SPIDER to analyze all 157 services in the ONOS SDN controller. SPIDER flags 11 potential SPIs, of which 10 are true positives and 9 depend on complex state interactions. We classify these issues based on the capabilities/scenarios required for triggering them and on their impact. The most serious identified vulnerabilities include (a) a malicious host can degrade the SDN controller's performance by cumulatively increasing the cost of processing an OpenFlow message without bound, and (b) a vulnerability in the topology service leads to worst-case exponential performance, which can be triggered by a compromised network switch. Our experiment also shows that the SPI enables an attacker to reduce the throughput down to 1Mb/s after sending 4000 spoofed ARP packets at low frequency (10 pkts/s) while only controlling one vulnerable host in the network.

We evaluate our design decisions by comparing SPIDER with three baseline implementations including a monolithic SDN fuzzer (Delta [9]), a variant of SPIDER without dependency information (FULL), and a variant of SPIDER that analyzes services in isolation (SINGLE). We run separate fuzzing campaigns for all three variants of SPIDER for each of the 157 services, fixing the budget in terms of fuzzing time, and with repetitions (~1.6 CPU years). Compared to SPIDER's 10 true positives, FULL identifies only 1 and SINGLE identifies 2; Delta triggers 3 issues, though isolating the inputs is non-trivial. Our results indicate that SPIDER is uniquely effective in identifying stateful performance issues in ONOS.

To summarize, this paper makes the following contributions:

- We identify a new class of SDN vulnerabilities called *stateful performance issues* (SPIs).
- We propose SPIDER, a novel *dependency-aware modular performance fuzzing* technique for identifying SPIs in an event-based software architecture.
- We use SPIDER to implement fuzzers for 157 services in the ONOS SDN controller.
- We identify 10 unique performance issues in ONOS and provide detailed case studies for two of the SPIs.
- We present a thorough evaluation and compare SPIDER with three baseline implementations.

2 Background and Problem Definition

At a high level, the ONOS SDN controller receives one or more messages from the data plane consisting of routers and switches. The controller processes these messages in a stateful manner and generates one or more output messages or actions. Figure 1 shows an abstraction of ONOS, consisting of a list of *services*, an event dispatcher, and an OpenFlow protocol module.

Each service implements certain network functions (e.g., LLDP service implements functions that process LLDP Packets), and can be dynamically loaded and unloaded in a deployment. Each event is delivered to services that have registered corresponding event handlers. Each service maintains a local, and the state changes when the service processes events. When a service state changes, it may

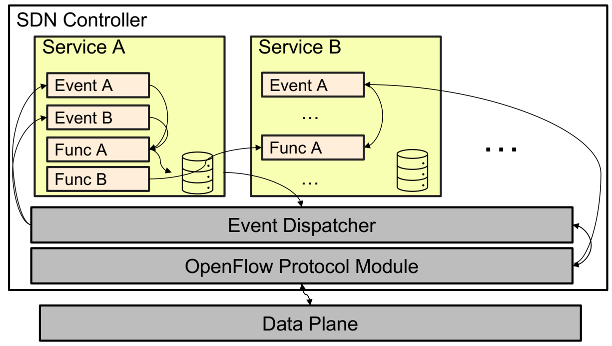

Figure 1: Architecture of ONOS.

```java
public class ARPService {
  private Map<IpAddress,MacAddress> addressMap;
  private Map<IpAddress,MacAddress> getAddressMap(){
    // Generate a shallow copy of addressMap
    // by iterating over each entry in the map.
    Map<IpAddress,MacAddress> copy = new HashMap<>();
    for (Map.Entry entry: addressMap.entrySet()) {
      copy.put(entry.getKey(), entry.getValue());
    }
    return copy;
  }
  public void add(IpAddress ip, MacAddress mac) {
    addressMap.put(ip, mac);
  }
  public MacAddress lookup(IpAddress ip) {
    return getAddressMap().get(ip);
  }
  public void packetHandler(OFPacketIn packetIn) {
    Ethernet payload = packetIn.getPayload();
    if (payload instanceof ARP) {
      ARP arp = (ARP) payload;
      if (arp.opCode == 0x1 || arp.opCode == 0x2) {
        if (lookup(arp.ip) == null) {
          this.add(arp.ip, arp.mac);
} } } } }
```

Figure 2: Simplified view of `ARPService` in ONOS, illustrating a stateful performance issue. The `lookup` function triggered by `OFPacketIn`, performs an $O(n)$ operation w.r.t. the size of `addressMap`.

generate and dispatch events delivered to other subscriber services. For example, the LLDP service processes LLDP packets and dispatches topology events if a device is connected or disconnected. Similarly, the Flow service implements logic related to flow rules and listens to the topology events and updates its internal state.

In this paper, we focus on *stateful performance issues* (SPIs) in these services. Such issues can be a serious concern for critical infrastructures since they risk Denial of Service [13–15] or induce subtle tail latency [16]. Triggering an SPI involves two phases: First, a sequence of inputs drives the system to a vulnerable state. Then, a specific input consumes an excessive amount of compute resources.

SPIs are different from two classical types of potential vulnerabilities explored in the literature. First, in contrast to *stateless* performance issues, where a single input leads to an amplified response (e.g., [17, 18]), stateful issues entail a complex sequence of events. Second, in contrast to stateful security issues related to *protocol state* [19–21], SPIs target the state of internal data structures in the SDN controller. Although SPIs have been studied in other settings (e.g., databases [22]), to the best of our knowledge this has not been explored in the context of SDN controllers.

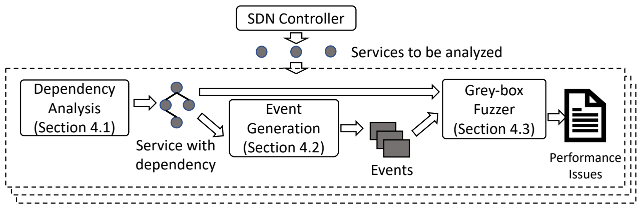

Figure 3: A high-level overview of SPIDER.

SPIs are difficult to catch in pre-deployment software testing or in runtime system profiling. First, the issue may not be revealed in the profiling data from normal runs as the system may not reach a vulnerable state. Second, a misconfiguration (or attack) can slowly build the state over time (e.g., by infrequently adding ARP records in the example above), and remain undetected until the final trigger.

Illustrative example. Figure 2 presents a real issue we discovered in the ONOS `ARPService`. The service processes `OFPacketIn` events with ARP payloads and stores the mapping between IP and MAC addresses. `packetHandler` is an event handler which processes all `OFPacketIn` events corresponding to OpenFlow packets. The `OFPacketIn` event may cause the service to first look up ARP records (Line 23) and add a record to the `addressMap` if the record is missing (Line 24). Unfortunately, the `lookup` function has a subtle performance issue. `lookup` calls `getAddressMap()` to get a shallow copy of the `addressMap` instead of querying `addressMap` directly (Line 7). This leads to $O(n)$ operation with respect to the size of `addressMap` each time `lookup` is called. Note that `OFPacketIn` events can be triggered by data-plane ARP packets; e.g., a misconfigured or malicious host can send spoofed ARP packets to increase the `addressMap` size n and each message will subsequently trigger an $O(n)$ computation in terms of its size.

3 Solution Overview

We adopt a fuzzing-based workflow to address our problem. Fuzz testing [23–26] is a randomized input generation technique that has proven to be effective in finding software bugs and security vulnerabilities in large and complex systems. However, we cannot apply existing fuzzing techniques directly. We start by describing the design space for fuzzing and argue why strawman solutions do not work. Then, we describe our design choices to make this problem tractable. We then present our end-to-end workflow, shown in Figure 3.

Design space and challenges: At a high level, any fuzzing workflow entails the following choices that impose different trade-offs between fidelity, scalability, and manual effort:

- *Granularity of code access:* One extreme is "black-box" fuzzing [23] with access only to input/output of the system under test. At the other extreme, we have "white-box" fuzzing [27] which inspects source code to analyze state and execution paths. Black-box approaches scale well but are imprecise, while white-box approaches are precise but do not scale to a very large complex. A middle ground is "grey-box" [24] fuzzing (e.g., AFL [28] and libFuzzer [29]) which uses lightweight instrumentation to get feedback from the test execution to guide input generation.
- *Granularity of inputs:* Fuzzers can generate inputs in different representations, which entails a trade-off between the quality and the amount of domain knowledge that must be captured. In the simplest case, we send a raw bitstream. At the other extreme, we can directly generate internal data structures for classes. There are also intermediate options; e.g., sending semantic-aware OpenFlow messages.
- *Granularity of system-under-test:* At one end, we can consider a monolithic view of the entire system, but this is also the least scalable. Alternatively, we can analyze individual classes, but we may miss out on vulnerabilities triggered by inter-class dependencies.

A simple workflow is to use black-box SDN fuzzers like Delta [9] to generate OpenFlow message inputs to ONOS and check if some message(s) cause performance issues. However, given the large input space, this approach does not work well and most inputs are not relevant for stateful scenarios. Consider Figure 2; the function `add` is called if and only if an OpenFlow message is received by ONOS and the packet contains an ARP payload with the operation code `0x1` or `0x2` (Lines 19–22). Indeed, we tried using Delta to randomly sample ten thousand OpenFlow messages. Of these, Delta produced 1140 OpenFlow messages with ARP payloads. Only 13/1140 packets trigger the `add` method and increase the size of `addressMap`. To increase the execution cost of the `ARPService`, the fuzzer needs to generate more than 900 OpenFlow messages with valid ARP payloads.

Design choice 1: Performance-oriented grey-box fuzzing. SPIs require us to generate a sequence of relevant messages. The search space of individual messages alone is large, and considering a sequence further increases the search space. Thus, black-box fuzzers are not directly applicable. Grey-box performance fuzzers, such as SlowFuzz [11] or PerfFuzz [10] are a more promising starting point to tame large search spaces by evolving inputs via feedback from program executions. However, the complexity and semantics of ONOS pose key challenges that we need to tackle.

Design Choice 2: Event sequences as inputs. Having chosen a grey-box workflow, we next consider the input granularity. A naive solution is to use a raw bitstream. However, this does not capture any relevant protocol semantics, and thus most inputs would be dismissed as garbage. Another choice would be to use OpenFlow messages and rely on the controller to convert OpenFlow messages to internal states of each service. Again, the space of possible messages is too large to be useful. To avoid these problems, we use the following domain-specific insight. Recall from §2 that ONOS implements an event-based architecture where incoming OpenFlow messages trigger new events. SPIs can arise when some service S reaches an internal state such that handling an *event* becomes costly and the internal state depends on all events that have been handled so far. This enables us to make our problem more tractable by searching for a *sequence of events* instead of a sequence of OpenFlow messages; i.e., we search for a sequence of events $\sigma = e_1, e_2, \ldots, e_N$ so that the processing time of event e_N is greater than a predefined threshold.

Design Choice 3: Dependency-aware modular analysis. Having chosen an event-based fuzzing workflow, we observe an opportunity to improve the scalability of the analysis without sacrificing the fidelity of the analysis. Recall again from §2 that the controller consists of services that each handles one or more event types. If a

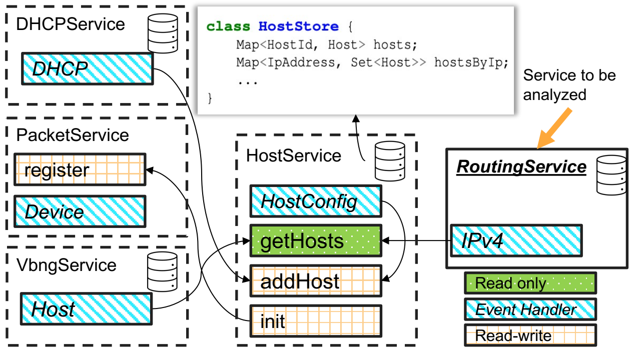

Figure 4: Interactions between different services through function calls in ONOS.

sequence of events $\sigma = e_1, e_2, \ldots, e_N$ triggers a performance vulnerability in some service S that handles the event e_N, then we only need to search over prefixes $e_1, e_2, \ldots, e_{N-1}$ that will directly or indirectly affect the state of S. We can thus reduce the search space by targeting each service S one at a time and searching for SPIs in S by generating only those events that will be handled by S or any other service that communicates with S that affect its state. Note that this modular analysis is possible only because of our decisions to use grey-box and event-driven workflow; black-box analysis or using packet or OpenFlow messages as input would necessarily entail fuzzing the controller in a monolithic fashion. We define a service as *analyzable* if the service registers at least one type of event. Thus, we explicitly reformulate our problem to take as input a list of services to be analyzed, drawn from the set of analyzable services, rather than analyze the entire controller code at once.

Overview: Combining these design choices above, we have the following end-to-end workflow, as depicted in Figure 3. For each service to be analyzed, we first compute its dependency set using static analysis. Then, for each service and its dependency, SPIDER uses event generators and performance fuzzing to generate event sequences of interest that can trigger potential SPIs. Finally, we validate these vulnerabilities by reconstructing OpenFlow message sequences that will trigger the fuzzer-identified event sequences. Note that these design choices naturally dovetail into each other to enable our analysis to be tractable; e.g., the modular decomposition would not be possible without a grey-box event-based workflow. To realize this solution in practice, we still need to address a number of system design and implementation challenges that we address in the following sections.

4 Detailed Design

Next, we describe the detailed design of SPIDER to realize the workflow from the previous section.

4.1 Identifying Service Dependencies

A core benefit of SPIDER's design decision to search over event sequences is that it enables modular analysis instead of a monolithic analysis. Specifically, we can separately analyze each service in ONOS to uncover SPIs in that service.

Analyzing a service S involves searching for sequences $e_1, e_2, \ldots, e_N$ of some fixed length N such that S is an event handler of e_N. Since we are interested in event sequences that trigger a performance issue when S is handling e_N, we only care about events $e_1, e_2, \ldots, e_{N-1}$ that can affect the performance of the handler of e_N. Note that the event sequence includes events handled by some other services S' such that S' affects the internal state of the service S. We call the set of such services S' as the *service dependency set* of S. But how do we determine the service dependency set?

Observe that the state of S may be manipulated by another service S' that calls a function in S. Additionally, S may call a function of some service S'', query the state of S'', and then update its own internal state. Therefore, we would put S' and S'' in the dependency set of S, and then we also have to consider services that affect the states of S' and S'' *transitively*.

One way to compute the service dependency set is to include all services that can reach the analyzed service through function calls or be reached by it. Figure 4 presents a simplified call graph for a subset of services. Each edge represents a function call pointing from callee to the caller. In this example, the service dependency set of `RoutingService` based on this call graph would include `VbngService`, `HostService`, `PacketSerivce`, and `DHCPService`.

However, the call graph approach may include services that do not affect the state of the analyzed service. In our example, `VbngService` does not actually manipulate the state of `HostService`, since it only calls a read-only function `getHosts`; therefore, it cannot indirectly affect the state of `RoutingService`. We want the dependency set to be as small as possible to reduce the search space for analyzing a given service.

To this end, we use a refinement that reduces the search space without sacrificing analysis fidelity. First, for each event handler, we compute a set of *read* and a set of *write* objects accessed by the handler. We use this set to exclude services that do not affect the same state object of the analyzed service *while processing events*. For example, the state of `HostService` is not affected by `VbngService` and `RoutingService` because `getHosts` only reads from the `HostStore`. Additionally, generating events for `PacketService` will not affect the state of `HostService` because the `Device` event handler does not access the `HostStore` state object at all.

Formally, our algorithm for computing the dependency set *Dep* of a service S is as follows:

(1) Initialize a set R of state objects *read* by the event handlers of the analyzed service S and initialize *Dep* to $\{S\}$.
(2) For each service S' that can reach the analyzed service S through function calls or be reached by it:
   a. Compute two sets $R_{S'}$ and $W_{S'}$ which contain state objects *read* and *written* by its event handlers, respectively.
   b. If $W_{S'} \cap R$ is not empty, update $R \leftarrow R \cup R_{S'}$ and $Dep \leftarrow Dep \cup \{S'\}$.
(3) If the dependency set *Dep* is updated, go back to Step 2.

With the optimization algorithm, the service dependency set of `RoutingService` now only includes `HostService` and `DHCPService`. The set excludes `VbngService` and `PacketService` because the event handlers from those services do not write to any state objects read by the `RoutingService`.

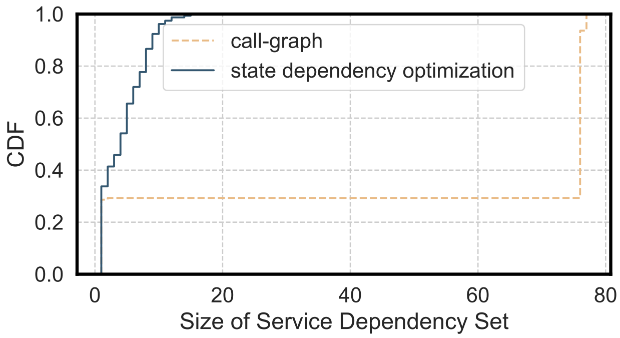

Figure 5: CDF of service dependency set sizes computed by the two algorithms across the 157 services analyzed in ONOS. Smaller size is better: the state-dependency optimization reduces the size of the dependency sets.

```
class HostEvent {
    enum Type {
        HOST_ADDED, HOST_REMOVED
    };
    Host host;
    Type type;
}
class Host {
    String name;
    public Driver(String name{
      this.name = checkNotNull(name);
} }
```

Figure 6: Simplified version of `HostEvent`.

As evidence of this optimization, Figure 5 plots a CDF of the service dependency sets across the 157 services in ONOS. A naive call graph-based approach would have included over 75 dependent services for $\approx 70\%$ of the services. In contrast, our state-dependency optimization results in a median of 4 and a maximum of 15 services in the dependency set.

4.2 Event Generation

Recall that analyzing a service S for stateful performance issues requires searching over *event sequences* corresponding to events handled by any service in the service dependency of S. We decide to use generator functions for randomly sampling event objects. A generator for an event of type T is a function $Random \rightarrow T$, where *Random* is a source of randomness. This approach has been successfully applied by property testing tools such as Quickcheck [30, 31].

In ONOS, each event is a data structure that contains multiple fields. For example, Figure 6 shows a simplified version of `HostEvent`. The `HostEvent` contains two fields `host` with type `Host`, and `type` with type `Type`. `Host` is another data structure that contains one field `name` with type `String`. To randomly sample a `HostEvent`, we must randomly generate its fields recursively. So, we also need a generator for the type (`Type`), `Host`, and the name (`String`). To generate all event types, we need to be able to generate all fields recursively!

By default, SPIDER provides a type-based event generator that generates events purely based on the type of each field [30, 32]. Figure 7 presents the pseudocode of a type-based object generator. The

```
class Generator {
  Object generate(Class type, Random rnd) {
  if (type == Integer.class) {
    return rnd.nextInt(); // random value
  } else if (...) {
    ... /* other primitive types */
  } else { // object type
    Constructor c = type.getConstructor();
    Object o = c.newInstance();
    for (Field field: o.getFields()) {
      Object val = generate(field.getType(), rnd);
      field.set(val); // random value
    }
    return o;
}}}
```

Figure 7: A simple type-based object generator that samples random `Object` instances given any `type`.

```
class HostEventGenerator {
  List<Host> generatedHosts;
  HostEvent generateHostEvent(Random rnd) {
    Type type = rnd.choose(Type.values());
    if (type == Type.HOST_ADDED) {
      Host host = generateHost(rnd);
      generatedHosts.add(host);
      return new HostEvent(host, type);
    } else if (type == Type.HOST_REMOVED) {
      Host host = rnd.choose(generatedHosts);
      generatedHosts.remove(host);
      return new HostEvent(host, type);
    }
  }
  Host generateHost(Random rnd) {
    ... // type-based random sampling
} }
```

Figure 8: Simplified version of `HostEventGenerator`, which maintains inter-event constraints—hosts cannot be removed unless they have been previously added.

generator generates objects recursively based on the type of each field. The automated approach is crucial to be able to quickly generate many types of events, but it has some limitations. In particular, events or other contained objects when generated with unrestricted values for their fields may violate certain constraints that the controller expects to be satisfied. Thus, the type-based event generator may generate events that are *invalid*.

Broadly, we identify two types of validity constraints:

- **Intra-event constraints:** These specify the internal constraints in an event. For example, in Figure 6 the `name` field of a `host` object should not be `null`; the constructor enforces this by calling a helper function `checkNotNull` which will raise an exception if `name` is null.
- **Inter-event constraints:** These are properties that must hold across multiple events. There are two types of `HostEvent`, a `HOST_ADDED` event is posted when a new host is attached to the network, and a `HOST_REMOVED` event is posted when a connected host disconnects from the network. An inter-event constraint is that a `HOST_REMOVED` event is valid if and only if the corresponding `host` has been added to the network and has not been removed.

In general, automatically generating such constraint-aware data structures is hard [33]. While type-based event generators can be

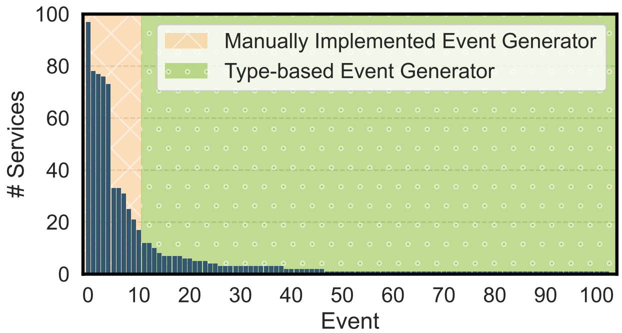

Figure 9: For each event type in ONOS (X-axis), this plot shows the number of services whose states are affected by that event type (Y-axis). SPIDER **uses hand-crafted event generators for the top 10 most critical events, and automates the generation for the rest using type information.**

used to automatically for such events, they run the risk is of generating invalid events. That is, either (a) the service handlers exit with an error message without exercising meaningful behavior, or (b) the search for SPIs may result in false positives.

As a pragmatic compromise between manual effort and coverage, we choose to manually implement generators for only the most *critical* of events, and use automatic type-based generation for the rest. We identify critical events by counting the number of distinct services whose states are affected by the event. Figure 9 shows that a small number of events affect the state of most services. Therefore, we write manual constraint-aware generators (similar to `HostEventGenerator` in Figure 6) for the top 10 events.

4.3 Putting It Together

Next, we describe how we generate event sequences to find SPIs. For each service S to be analyzed, we first identify its service dependency set Dep_S (§4.1). SPIDER then determines a set of event types E_S that cover all event handlers registered in all services in Dep_S. SPIDER will then use event generators (§4.2) to search over sequences of events $e_1, ...e_N$ where $\forall i : type(e_i) \in E_S$; that is, each event is of a type whose handler is registered by at least one service in Dep_S. SPIDER will perform a search, for a time budget of B, for event sequences such that the performance cost of handling event e_N is greater than some pre-defined threshold $t_{\max}$. The parameters N and B are chosen based on available compute resources (§6). We discuss how we pick the threshold for our experiments later.

SPIDER performs the search by combining ideas from PerfFuzz [10], a mutation-based grey-box performance fuzzer, and Zest [12], which enables mutation-based grey-box fuzzing on domain-specific input structures encoded via generator functions. We next provide a brief background on these techniques and then present SPIDER's fuzzing algorithm.

Background on performance fuzzing. Grey-box fuzzing [24], as popularized by tools such as AFL [28] and LibFuzzer [29], works by evolving a set of inputs, usually represented as byte streams, toward maximizing code coverage. Inputs are randomly mutated by flipping random bits; if this results in something desirable (e.g. new coverage), the inputs are retained for further mutation. PerfFuzz [10] extends this idea to find CPU resource exhaustion issues by maximizing not just coverage, but performance cost; we refer to this technique as *performance fuzzing*. However, wall clock time is not a reliable measure of run-time performance, since it depends on external system factors and cannot be deterministically reproduced. Instead, PerfFuzz measures execution cost by tracing conditional branches in the program under test via lightweight instrumentation and attempting to maximize the total execution-path length. The execution-path length is a reliable measurement that strongly correlates with wall clock time. PerfFuzz seeks to maximize execution counts for each branch in the program independently, which allows it to escape local maxima, unlike other tools such as SlowFuzz [11]. As such, PerfFuzz identifies algorithmic performance issues [34], e.g., due to long-running loops, rather than system interactions such as slow I/O operations.

Background on semantic fuzzing. Grey-box fuzzing (including performance fuzzing) depends on the ability to perform random bit-level mutations on program inputs. In SPIDER, the *inputs* are sequences of event objects generated using random sampling functions §4.2. Bit-flipping does not work with strongly typed objects. We therefore need a way to mutate event sequences. The idea of *semantic fuzzing*, first implemented in Zest [12], enables random mutations to be performed on inputs that are produced by generator functions like those described in Section 4.2. Observe that in Figures 7 and 8 a pseudo-random number generator `rnd` is given as a parameter. The key idea behind semantic fuzzing is to record the sequence of pseudo-random choices made during a random sampling process (e.g. the calls to `rnd.nextInt()` in Fig. 7 or `rnd.choose()` in Fig. 8), and replay them again with slight modifications; the resulting event objects are similar to previously generated events but differ in small ways. Essentially, the grey-box fuzzer can mutate a stream of pseudo-random choices represented as a bitstream, which corresponds to mutating event objects that come out of the generator functions without breaking their structure.

Our combined approach. SPIDER's algorithm for fuzzing a service S and its dependencies Dep_S with relevant event types E_S combines performance and semantic fuzzing, as follows:

(1) Initialize a set Q with a randomly generated event sequence $\sigma_0 = e_1, ..., e_N$, where the type of each event e_i is chosen randomly from E_S, and the event is randomly sampled via its corresponding event generator (§4.2).
(2) Initialize a map $maxCounts$, which tracks the maximum execution cost observed at each program branch, by sending the event sequence σ_0 to ONOS and monitoring the execution cost when processing e_N.
(3) Pick a random event sequence σ from Q and mutate it into a new event sequence σ' using the *semantic fuzzing* [12] approach, as described above.
(4) Dispatch σ' to the services in Dep_S and collect its execution instruction trace when processing the last event in σ'.
   a. If the total execution path length is greater than $t_{\max}$, then flag σ' as a potential issue.
   b. Otherwise, cumulate the element-wise execution cost of each program branch when processing the last event, and update the corresponding entry for each branch in $maxCounts$ if the value is greater.

Figure 10: Distribution of the per-event execution cost in a normal-workload emulation environment of ONOS (log-log scale). We set the threshold at three standard deviations higher than the mean normal operation cost.

c. If any item in *maxCounts* was updated, then add σ' to Q. Otherwise, discard σ'.
d. If the time budget B has expired, then stop fuzzing. Otherwise, go back to step 3.

Note that potential issues flagged by this process still need to be validated. First, some event e_i in the flagged event sequence could be invalid as it violates intra-event or inter-event constraints (§4.2). In this case, the flagged issue is *false positive*. Second, just because we have found a sequence of valid events that triggers a high cost of execution, it does not necessarily mean that the same effect can be induced by external OpenFlow messages. We currently use a manual process to translate an event sequence into an OpenFlow message sequence. Automating this step is a natural extension for future work. We flag an issue report as *true positive* if we can (manually) trigger the performance issue in a network emulation.

5 Implementation

Performance fuzzing and state reset. We implement the performance fuzzer in Java and Kotlin [35] on top of the JQF [36] framework, which we extend to support performance fuzzing [10](§4.3). SPIDER uses ASM [37] and ByteBuddy [38] to instrument ONOS to collect the performance costs of events. Performance fuzzing requires that the state of the analyzed system can be reset easily cacross fuzzing executions. A naive option for fuzzing is to launch a new instance for each execution. However, starting an instance of ONOS is slow (e.g. 30 seconds on a laptop with 6-core and 32GB memory). Alternatively, reusing an instance across fuzzing runs is not viable as the state is impacted by previous events. We also considered resetting service state by instantiating a new service object and discarding the old one. However, services such as `Storage-Service` implement distributed persistent storage. This is not only slow to launch but also persists the state to a local file system and does not actually reset state.

Our approach to enable state reset is to leverage *mock services* provided by developers for unit tests. For instance, we can replace the distributed store with a mock in-memory store in the fuzzing harness. Although this prevents us from identifying performance vulnerabilities in the distributed store itself, it allows us to analyze many services that rely on the store instead.

Alerting threshold. Given a sequence of events generated by SPIDER, we need to determine if the events trigger a potential SPI. We use a data-driven threshold selection. First, we build a simple emulation environment using Mininet [39] with 4 hosts and 2 switches. We use `ping` utility to generate data plane packets and monitor execution costs of event handling in ONOS using JVM bytecode instrumentation.[1] As mentioned previously, we replace the distributed storage with an in-memory storage to achieve efficient state reset in ONOS. Note that the implementation of in-memory storage is much simpler than the distributed storage, and to avoid setting an unrealistic high threshold, we disable the instrumentation of the distributed storage in ONOS. We run the emulation environment for 20 minutes, which results in 93,788 events being observed for ONOS. Figure 10 shows the histogram of the per-event execution cost for ONOS. We compare the execution cost of the last event generated by SPIDER with the events generated in the simulation environment using z-score where $Z(x) = \frac{x-\mu}{\sigma}$ (i.e., the number of standard deviations above the mean). We use a threshold z-score of three: if $Z(x) > 3$, then we flag a potential a SPI.[2]

Validation and strategy reconstruction. For each potential vulnerability reported by SPIDER, we want to verify if it actually represents a true vulnerability in ONOS. Our insight here is that events in the SDN controller provide useful information about events in the network. For example, a `DeviceEvent` represents that the status of a device is updated, which contains the type of update and the detailed information of the device. Similarly, a `Link` event represents the link update. Given a sequence of `DeviceEvents` and `LinkEvents`, we can dynamically reconstruct the topology. With this insight, we can provide hints for a reconstruction strategy including network topology information (i.e. hosts, switches, links) and network actions such as OpenFlow messages, (e.g., topology and configuration updates). We replay this sequence in network emulation using Mininet [39]. [3]

6 Evaluation

We evaluate SPIDER on ONOS v2.2.4 [2]. Our evaluation is focused on answering the following research questions.

RQ1. Is SPIDER effective at identifying SPIs in ONOS?

RQ2. How does SPIDER compare to a traditional SDN fuzzer in identifying SPIs?

RQ3. To what extent does the dependency-aware modular fuzzing technique help in identifying SPIs?

For each service to be analyzed, we have two parameters to scope the analysis: (1) *time budget* (B) to run the analysis and (2) *sequence length* (N) of events to explore. With longer time and length, the fuzzer consumes more resources and has a greater chance of identifying SPIs. However, longer sequence lengths also increase the

[1] One potential concern is that the processing time may depend on the specific deployment and topology size; i.e., is threshold based on a small topology relevant. We believe that this baseline is still useful as it indicates potential scalability bottlenecks inside the controller implementation.

[2] The outliers in Figure 10 that have a cost higher than the threshold only appear during initialization; these are not considered as performance issues.

[3] There are still manual steps in setting up the emulation environment to ensure the network is valid; e.g., because of authentication procedures between controller and data plane that we do not yet automate.

ID	Service Name	Description	Source	Smallest N	SPIDER	FULL	SINGLE	SDN-FUZZ
V1	Castor	The execution cost of `CastorArpService` increases linearly with respect to the number of `OFPacketIn` with ARP payload received by the service.	host	2500	✓	✗	✓	✗
V2	Neighbor Resolution	The execution cost of `NeighborResolutionManager` increases linearly with respect to the number of connect points in the network.	switch	50	✓	✗	✗	✓
V3	Port Statistics	The execution cost of `PortStatisticsService` increases linearly with respect to the number of `OFPortStatisticsReply` messages received by the service.	switch	1000	✓	✗	✗	✓
V4	Graph Path Search	The execution cost of `AbstractGraphPathSearch` service increases exponentially with respect to the number of links in the network.	switch	50	✓	✗	✗	✗
V5	My Tunnel App	The execution cost of `MyTunnelApp` increases linearly with respect to the number of hosts in the topology.	switch	50	✓	✗	✗	✓
V6	VPLS	The execution cost of `VplsManager` increases linearly with respect to the number of interfaces configured in the SDN controller.	controller	50	✓	✗	✗	✗
V7	Links Provider	The execution cost of `NetworkConfigLinksProvider` increases linearly with respect to the number of port created for each switch.	switch	50	✓	✗	✗	✗
V8	Rabbit MQ	The `MQEventHandler` performs a costly computation while processing IPv4 packets.	host	1	✓	✓	✓	✗
V9	Router Advertisement	The execution cost of `RouterAdvertisementManager` increases linearly with respect to the number of interfaces created in the network.	switch	50	✓	✗	✗	✗
V10	Link Discovery	The execution cost `LinkDiscoveryProvider` increases linearly with respect to the number of switches in the network.	switch	1000	✓	✗	✗	✗
F1	Control Plane Manager	An invalid `ControlMessageEvent` causes high execution of the `ControlPlaneManager`.	N/A	N/A	✓	✗	✓	✗

Table 1: Summary of performance issues identified by SPIDER and baselines in ONOS. Each row shows the affected class, a description of the issue, the source of OpenFlow messages that can trigger the issue, the smallest empirical sequence length to uncover the issue

search space. We configure SPIDER to find a sequence of events with lengths N=1, 100, 250, 500, 1000, and 2500.

For each N, we allocate a budget B of 1 hour to analyze each service. The fuzzer also uses results from previous sequence length as seeds and the total fuzzing time of each service is 6 hours. We repeated each each experiment 5 times which leads to in total 4740 CPU hours (197 CPU days) per configuration. We conduct all of our experiments on Cloudlab VMs using 4 cores (2.4 GHz) and 4 GB memory for each service. The fuzzer runs and reports the smallest N, where the z-score of the cost of handling the last event is greater than 3, or NULL if no such event was found as described in §5.

6.1 RQ1: SPI Detection with SPIDER

After fuzzing each of the 157 services with the above parameters, SPIDER reported 11 potential issues, summarized in Table 1. We manually analyze these 11 reports and find that 10 are true positives (which we name V1–V10) while one is a false positive (named F1). Out of the 10 true positives, 9 of these are truly *stateful* performance issues; i.e., they require a non-empty of sequence of events to set up a vulnerable state before the issue can be triggered. Only V8 can be triggered with a single event. We manually inspect F1 and identify that it relies on an automatically synthesized type-based event generator for `ControlMessageEvent`, which does not take into account some constraints and produces an invalid event (e.g. the maximum allowable size of a control-message list is exceeded); therefore, the issue cannot be triggered using OpenFlow messages. We also verify that all performance issues can be triggered regardless of the implementation of the storage layer in ONOS.

Validation/Replay. For each reported issue, we use Mininet to manually reconstruct the issue. We successfully replicated 9 issues in the emulated network.[4]

[4]We are not able to replicate V4 due to the another bug triggered by the emulator (Case Study 2).

Responsible disclosure. In the spirit of responsible disclosure, we have notified the ONOS developers and presented them with concrete end-to-end traces to reproduce our reported issues. We are currently in discussions to verify the impact to their deployments.

Classification. We manually classify the 10 performance issues along two dimensions: *source of the triggering event* and *algorithmic complexity*. First, we classify issues based on the types of sources that can generate key events to trigger these issues: *host*, *switch*, *controller*. For example, any host connected to the network can generate `PacketIn` events with IPv4 payloads, so its source is classified to *host*. A `PacketIn` event with LLDP payload can only be sent by switches, so its source is classified as *switch*. Some events can only be triggered by an SDN controller configuration update, and those events will have *controller* as the source. Second, we qualify the algorithmic complexity of the performance issue as a function of the number of events in the sequence. Specifically, we identify *high constant*, *linear*, and *exponential* patterns of *per-event* execution time for the trigger event. Note that per-event linear complexity translates to a cumulative performance cost of $O(n^2)$ for n events.

Table 1 presents a comprehensive listing of all issues discovered by SPIDER and baselines. Out of 10 true positives, 2 issues can be triggered from a malicious host, which is the most serious case; 7 issues can be triggered from compromised switches; 1 issue can only be triggered by ONOS itself. While the latter is not a serious security risk, it may occur due to accidental misconfigurations.

With respect to the temporal footprint, the non-stateful issue V8 causes ONOS to perform a high- constant-cost execution; 8 issues cause ONOS to perform a computation whose per-event cost increases linearly with respect to the number of events generated; 1 issue (V4) causes ONOS to perform a computation whose cost increases exponentially with respect to the events or generated.

Case Study 1: Host-initiated stateful performance issue via spoofed ARP packets (V1). This issue, in class `CastorArpManager`, can be exploited by any malicious hosts in the network. The root cause

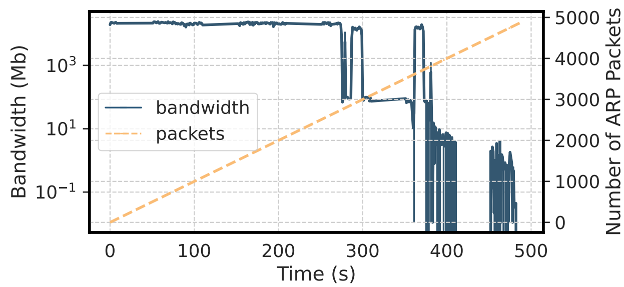

Figure 11: The bandwidth of the network drops significantly 300 seconds after the ARP spoofing attack starts.

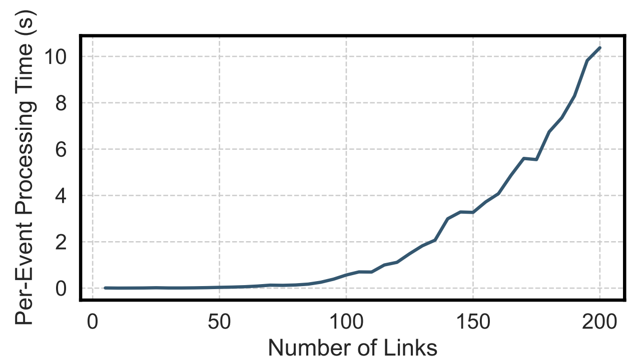

Figure 12: Execution time of `AbstractGraphPathSearch` service increases exponentially with respect to the number of paths created in the network.

of the issue is depicted (highly simplified) in Figure 2. The execution cost of the ARP-related service increases as more ARP records are added to an internal data structure. We were able to manually reconstruct the OpenFlow messages that would trigger this issue. As a proof-of-concept, we use Mininet [39] to create a simple network with three switches. Each switch connects to one host. We use one host to generate spoofed ARP packets and monitor the connectivity between the other two hosts. The malicious host generates 10 spoofed ARP packets every second to avoid the flooding attack.

We use iperf to measure the bandwidth between two benign hosts in the network. Figure 11 shows the result. The bandwidth stays stable at 27 Gbits/s initially and drops significantly after 270 seconds. Note that it only requires the attacker to generate 3000 spoofed ARP packets to bring the entire network down, and the frequency is low. Note that the bandwidth drop is not due to a data plane attack. We further confirmed this by performing the same experiment with `CastorArpManager` disabled and did not observe any bandwidth drop. The SPI reduces the throughput of ONOS by increasing the average processing time of each OpenFlow message. OpenFlow messages with LLDP payload are used to check the liveness of links. ONOS failed to process such messages in time during the attack and marked links as unavailable. Thus, the bandwidth of the entire network is affected.

Moreover, we find that after generating spoofed ARP records, ONOS cannot be recovered easily by disconnecting the malicious host or rebooting. ONOS saves all ARP records in a persistent storage and ONOS does not provide an interface to remove a single field unless the user removes the entire data store. In that case, other configurations will be removed as well.

Case Study 2: Exponential-time stateful performance issue induced by redundant links (V4). SPIDER reports that the execution cost of the `AbstractGraphPathSearch` method increases exponentially with respect to the number of links in the network, in particular when there are *redundant links* between devices. This is incredibly subtle because the link graph is actually a multi-graph, and the path search algorithm degrades in the presence of multiple edges between the same pair of nodes. SPIDER identifies this issue by generating a topology with multiple redundant links.[5]

In order to replay this issue, we used Mininet to generate a simulation network with redundant links. Unfortunately, the simulation environment triggers an unrelated bug in ONOS which hangs up the controller completely and stops processing any OpenFlow messages from data plane (another kind of DoS). We have reported this bug to the developers and are awaiting a fix.

However, we are still able to trigger the issue that SPIDER discovered by implementing a standalone service that can send messages to `TopologyService`. We use this service to generate a topology containing 5 devices, and then slowly add redundant links by sending appropriate messages. Figure 12 shows the performance of the `TopologyService`, which uses `AbstractGraphPathSearch` to compute paths between nodes, with respect to the total number of links created in the network. This subtle case of redundant links in a multi-graph topology demonstrates that SPIDER can identify hard-to-detect stateful performance issues.

6.2 RQ2: Compare to SDN Fuzzer

In order to answer RQ2, we use a packet fuzzer adopted from Delta [9] to fuzz ONOS for 12 hours. Delta is a state-of-the-art black box SDN fuzzer, which generates stateful OpenFlow messages. Since this fuzzer does not use instrumentation, we further instrument ONOS and measure the execution cost of each event triggered by the SDN fuzzer.

Delta triggers over 2 million internal events in total and only 582 events whose z-scores are greater than 3. Except the events generated during the bootstrap stage of ONOS similar to the simulation environment shown in Figure 10, Delta triggers 3 SPIs (V2, V3 and V5), and cannot identify subtle issues that modify the topology. Note that Delta generates OpenFlow messages continuously without resetting; therefore, any state changes are unintentional. It is thus non-trivial to isolate small message sequences that trigger an SPI. From this experiment, we can conclude that SPIDER is more effective than Delta in identifying SPIs.

6.3 RQ3: Sensitivity Analysis

In order to evaluate the efficacy of SPIDER's dependency-aware modular fuzzing design, we compare the worst case input finding ability with two baselines SINGLE and FULL. SINGLE do not use dependency analysis and fuzz each service without any dependency and FULL fuzzes each service with all services as dependencies. We use SINGLE and FULL to analyze all 157 services with the same configuration as SPIDER and repeat the experiment 5 times.

[5] We further validated this issue by manually analyzing the source code.

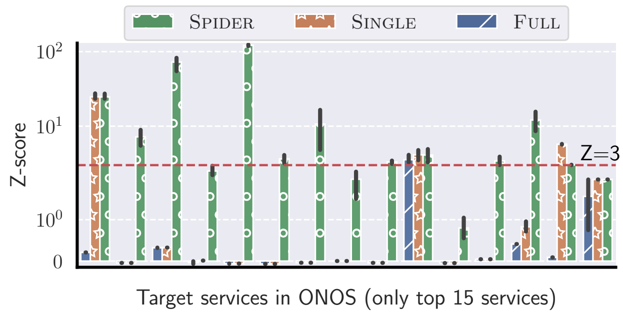

Figure 13: The z-score of the target services reported by different fuzzers.

Bug reports. As shown in Table 1, all SPIs reported by Full and Single are covered by Spider. Full only reports one performance issue (V7), which can be identified easily by generating only one event with an IPv4 payload. Full fails to identify all stateful performance issues that require more than one event to trigger the issue. We found that including all services as the dependency of the analyzed service greatly decreased the performance of the fuzzer because the fuzzer need to not only take more time to initialize each service but also explore a larger state space which is not relevant to the analyzed service. Single reports two performance issues (V1 and V8) and one false positive (F1). All three performance issues can be identified by only exploring the state space of the analyzed service. Other performance issues cannot be identified by Single because the search space has been artificially limited.

Finding the worst case input. Figure 13 shows the z-score of the worst case input of different services identified by different fuzzers across multiple runs. We only show services whose z-score is greater than 1 for at least one fuzzer. Not that there are 12 services whose z-score are greater than 3 because different services trigger the same SPI. In general, Spider outperforms Full in 14 out of 15 services. This confirms that it is important to use modular fuzzing to reduce the search space for the performance fuzzer. Spider out performs Single in 11 out of 15 services. Single reports a higher z-score if the worst case input can be constructed without exploring the state space of other services. However, it failed to construct complex state full input for other services. Our result shows that the dependency-aware modular design is critical in identifying SPIs.

7 Related Work

SDN fuzzers. Existing black-box SDN fuzzers (e.g., Beads [40] and Delta [9]) generate packets based on an existing topology and focus on logic protocol bugs in SDN controllers [9, 40]. Most OpenFlow messages generated by the SDN fuzzer only explore *a small portion of the input space* of the SDN controller, and many performance-sensitive services are left untested using black-box SDN fuzzers.

Other analysis of SDN controllers. Nice [41] uses symbolic execution and model checking to identify property violations. ConGuard [8] and SDNRacer [6] use static analysis to identify race conditions in the SDN controller. EventScope [42] focuses on missing event handlers in SDN applications, and AudiSDN [43] identifies inconsistent policies among different modules. None of these efforts tackle SPIs.

Static code analysis for performance. Static performance analysis techniques (e.g., FindBugs [44], Clarity [45], Torpedo [22]) identify performance issues based on code patterns. Unfortunately, specifying such patterns usually requires domain-specific knowledge and many patterns of SPIs are not described in existing tools.

Symbolic execution for performance analysis. Symbolic execution (e.g., Castan [46] and Wise [47]) can be used to identify states with performance issues. However, the state space of the program increases exponentially with respect to the size of the program. Therefore, such techniques are still limited to analyze small programs and cannot handle the *large state space* of the SDN controllers [48].

Languages for performance analysis. Performance modeling languages such as RAML [49] provide an estimation of the program complexity. However, translating the existing SDN controller implementation into such languages is a challenge. Similar to static performance analysis, performance modeling languages cannot model the existing *complex code base* of the SDN controller such as reflection and runtime code generation.

Trace-driven analysis. Dynamic performance monitors (e.g., Freud [50] and PerfPlotter [51]) collect execution traces and produce an algorithmic complexity estimate [50, 51]. However, if the traces used for modeling (typically of common-case workloads) do not cover the (likely rare) SPI patterns, such tools will not be able to uncover SPIs.

Fuzzing stateful network protocols. Network fuzzers use protocol specifications [19, 52, 53] or try to infer protocols automatically [20, 21, 54]. These focus on protocol bugs or correctness issues, rather than SPIs.

8 Conclusion

In some ways, our effort is a proof-by-construction of the viability of a seemingly intractable program analysis problem: uncovering deep semantic stateful performance issues in large and complex software. We conclude by discussing extensions, limitations, and lessons.

There are three immediate extensions. First, capturing the semantic constraints in the top-10 events manually adds a lot of value. Thus, we can increase coverage by making the type-based generation more semantic aware. Second, we can make the reconstruction and validation process more automated (e.g., via program synthesis) using Spider's hints. Third, we need a way to also find issues in distributed components such as the state store for ONOS.

Finally, our experience sheds light on benefits of domain-specific insights in fuzzing and of design for testability. On a positive note, the presence of mock services and unit tests simplified our implementation. At the same time, the lack of semantic-aware constructors made event generation hard. An interesting direction for future work is to discover such domain-specific invariants and provide hints to developers on how they can support fuzzing workflows.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[2] "Open Network Operating System (ONOS®) is the leading open source sdn controller for building next-generation sdn/nfv solutions." https://opennetworking.org/onos/, accessed: 2021-08-31.
[3] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015.

[4] J. C. Correa Chica, J. C. Imbachi, and J. F. Botero Vega, "Security in SDN: A comprehensive survey," *Journal of Network and Computer Applications*, vol. 159, p. 102595, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804520300692

[5] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, 2013, pp. 1–7.

[6] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency analysis for software-defined networks," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 402–415. [Online]. Available: https://doi.org/10.1145/2908080.2908124

[7] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, "Unexpected data dependency creation and chaining: A new attack to SDN," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1512–1526.

[8] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 451–468.

[9] S. Lee, C. Yoon, C. Lee, S. Seungwon, V. Yegneswaran, and P. Porras, "Delta: A security assessment framework for software-defined networks," 01 2017.

[10] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 254–265. [Online]. Available: https://doi.org/10.1145/3213846.3213874

[11] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," *CoRR*, vol. abs/1708.08437, 2017. [Online]. Available: http://arxiv.org/abs/1708.08437

[12] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[13] R. Kandoi and M. Antikainen, "Denial-of-service attacks in OpenFlow SDN networks," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 1322–1326.

[14] P. Zhang, H. Wang, C. Hu, and C. Lin, "On denial of service attacks in software defined networks," *IEEE Network*, vol. 30, no. 6, pp. 28–33, 2016.

[15] H. Wang, L. Xu, and G. Gu, "Floodguard: A DoS attack prevention extension in software-defined networks," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 239–250.

[16] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[17] L. D. Toffola, M. Pradel, and T. R. Gross, "Synthesizing programs that expose performance bottlenecks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 314–326. [Online]. Available: https://doi.org/10.1145/3168830

[18] C.-A. Staicu and M. Pradel, "Freezing the Web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 361–376. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[19] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: toward a stateful network protocol fuzzer," in *International conference on information security*. Springer, 2006, pp. 343–358.

[20] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[21] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 317–329. [Online]. Available: https://doi.org/10.1145/1315245.1315286

[22] O. Olivo, I. Dillig, and C. Lin, "Detecting and exploiting second order denial-of-service vulnerabilities in web applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 616–628. [Online]. Available: https://doi.org/10.1145/2810103.2813680

[23] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, Dec. 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[24] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[25] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[26] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[27] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[28] "american fuzzy lop," https://github.com/google/AFL, accessed: 2021-08-31.

[29] "libfuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, accessed: 2021-08-31.

[30] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[31] P. Holser, "junit-quickcheck: Property-based testing, junit-style," https://github.com/pholser/junit-quickcheck, accessed: 2021-08-31.

[32] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. [Online]. Available: http://dx.doi.org/10.14722/ndss.2020.24415

[33] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[34] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks." in *USENIX Security Symposium*, 2003, pp. 29–44.

[35] "Kotlin programming language," https://kotlinlang.org/, accessed: 2021-08-31.

[36] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-guided property-based testing in Java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 398–401. [Online]. Available: https://doi.org/10.1145/3293882.3339002

[37] "Asm," https://asm.ow2.io/, accessed: 2021-08-31.

[38] "Bytebuddy, runtime code generation for the java virtual machine," https://bytebuddy.net/#/, accessed: 2021-08-31.

[39] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868447.1868466

[40] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "Beads: Automated attack discovery in openflow-based sdn systems," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Cham: Springer International Publishing, 2017, pp. 311–333.

[41] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USA: USENIX Association, 2012, p. 10.

[42] B. Ujcich, S. Jero, R. Skowyra, S. Gomez, A. Bates, W. Sanders, and H. Okhravi, "Automated discovery of cross-plane event-based vulnerabilities in software-defined networking," 01 2020.

[43] S. Lee, S. Woo, J. Kim, V. Yegneswaran, P. Porras, and S. Shin, "AudiSDN: Automated detection of network policy inconsistencies in software-defined networks," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 1788–1797.

[44] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, p. 92–106, Dec. 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895

[45] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 369–378. [Online]. Available: https://doi.org/10.1145/2737924.2737966

[46] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated synthesis of adversarial workloads for network functions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 372–385. [Online]. Available: https://doi.org/10.1145/3230543.3230573

[47] J. Burnim, S. Juvekar, and K. Sen, "Wise: Automated test generation for worst-case complexity," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 463–473.

[48] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[49] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 3, Nov. 2012. [Online]. Available: https://doi.org/10.1145/2362389.2362393

[50] D. Rogora, A. Carzaniga, A. Diwan, M. Hauswirth, and R. Soulé, "Analyzing system performance with probabilistic performance annotations," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387554

[51] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 49–60. [Online]. Available: https://doi.org/10.1145/2884781.2884794

[52] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in *Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications*, 2007, pp. 47–56.

[53] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

[54] V. T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.