SPIDER: A Practical Fuzzing Framework to Uncover Stateful Performance Issues in SDN Controllers

Ao Li
Carnegie Mellon University

Rohan Padhye
Carnegie Mellon University

Vyas Sekar
Carnegie Mellon University

***Abstract*—Performance issues in software-defined network (SDN) controllers can have serious impacts on the performance and availability of networks. We specifically consider stateful performance issues, where a sequence of initial input messages drives an SDN controller into a state such that its performance degrades pathologically when processing subsequent messages. We identify key challenges in applying canonical program analysis techniques: large input space of messages (e.g., stateful OpenFlow protocol), complex code base and software architecture (e.g., OSGi framework with dynamic launch), and the semantic dependencies between the internal state and external inputs. We design SPIDER, a practical fuzzing workflow that tackles these challenges and automatically uncovers such issues in SDN controllers. SPIDER's design entails a careful synthesis and extension of semantic fuzzing, performance fuzzing, and static analysis, taken together with domain-specific insights to tackle these challenges. We show that our design workflow is robust across two controllers—ONOS and OpenDaylight—with very different internal implementations. Using SPIDER, we were able to identify and confirm multiple stateful performance issues.**

I. Introduction

Software-defined networking is increasingly adopted in wide-area, datacenter, and enterprise networks [1]. In this respect, the performance and availability of the SDN controller are critical, as downtime or latency can affect the overall performance and availability of the network [2].

In this paper, we consider a new class of *stateful performance issues* (or SPIs) in SDN controllers. Specifically, these resource exhaustion events occur over time, by first transitioning the SDN controller into a potentially vulnerable state and then triggering an event that induces high resource consumption. Identifying such performance issues is critical since they could lead to performance and availability issues.

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 and complex code base; e.g., ONOS has tens of thousands of lines of code, hundreds of modules, and over a thousand internal data structures, and complex dependencies between modules. The third challenge stems from the semantics of SPIs; i.e., we need to capture the dependencies between inputs and internal states, and identify which state-input combinations entail high resource consumption. As such, many seemingly natural solutions from the program analysis literature cannot be applied in our context.

Our contribution is a practical synthesis and application of static program analysis, performance-oriented grey-box fuzzing [3], and input-structure-aware semantic fuzzing [4], in conjunction with domain-specific insights to tackle these challenges. The design and implementation of SPIDER is based on three key insights to make our analysis tractable.

- *Practical grey-box performance fuzzing:* We adopt ideas from performance-oriented fuzzing [3], which maximizes execution path lengths—and therefore CPU resource consumption—via lightweight program instrumentation.
- *Event-driven semantic fuzzing:* We adopt an *event-driven* fuzzing workflow based on the insight that *events* are the key entities that impact state computation and resource footprints. We borrow ideas from semantic fuzzing [4], which uses type-specific generator functions for representing and mutating well-formed inputs such as events. As a pragmatic solution, we use handcrafted event generators for the events that are in the critical path for analyzing many services, and use an automated type-based synthesis technique for the remaining events.
- *Modular dependency-aware analysis:* We develop a modular workflow to analyze one service at a time instead of a monolithic analysis over the entire code base. We develop a custom static analysis for decomposing services based on the logical state dependency between services.

SPIDER synthesizes these ideas together in an extensible workflow that iterates through individual SDN services (along with its dependencies) to analyze, runs a semantic performance fuzzing algorithm to generate event sequences, and flags potential SPIs. We show how SPIDER is general and can be applied to both ONOS and OpenDaylight. We implement an extensible core framework for module dependency analysis, type-based event generators, and semantic performance fuzzing. We create custom target-specific adapters for specific controller implementations (e.g., service state operations, constraint-aware event generators, and fuzzing harness). Such target-specific adapters can be implemented manually or automatically synthesized using static or dynamic techniques.

We evaluated SPIDER on ONOS and OpenDaylight. SPIDER identifies 9 unique SPIs (all in ONOS). Additionally, SPIDER identifies one performance issue that does not depend on a specific state, and further reports one potential vulnerability that turns out to be a false positive. 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

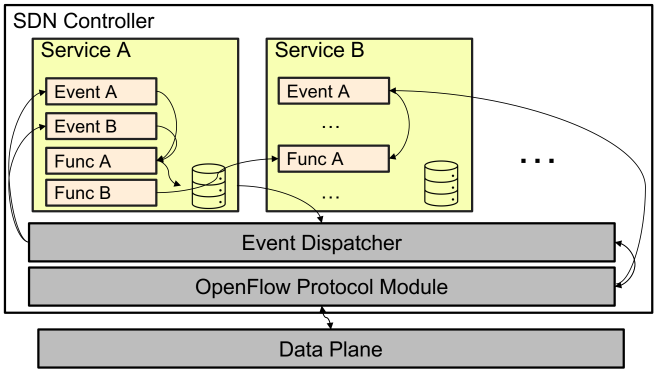


Fig. 1: Architecture of an abstract SDN controller.

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 also show that our design choices provide a significant benefit in achieving these results, by comparing to black-box OpenFlow message fuzzing, fully automated event generation, and pattern-based static analysis.

II. Background and Motivation

We begin by introducing SPIs in SDN controllers. Then, we describe why existing program analysis techniques do not directly apply in this context.

A. Stateful Performance Issues

At a high level, SDN controllers such as ONOS and OpenDaylight receive 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 abstract SDN controller, 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). Note that services 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 state and the state changes when the service processes events. When a service state changes, it may 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; e.g., to launch a Denial of Service [5], [6], [7] or induce subtle tail latency issues [8]. Triggering an SPI involves two phases. In the first phase, a sequence of inputs drives the system to a vulnerable state. Then, a specific input consumes an excessive amount of compute resources.

```
1  public class ARPService {
2    private Map<IpAddress,MacAddress> addressMap;
3    private Map<IpAddress,MacAddress>
      ↪ getAddressMap(){
4      // Generate a shallow copy of addressMap
5      // by iterating over each entry in the map.
6      Map<IpAddress,MacAddress> copy = new
        ↪ HashMap<>();
7      for (Map.Entry entry: addressMap.entrySet())
        ↪ {
8        copy.put(entry.getKey(), entry.getValue());
          ↪
9      }
10     return copy;
11   }
12   public void add(IpAddress ip, MacAddress mac) {
13     addressMap.put(ip, mac);
14   }
15   public MacAddress lookup(IpAddress ip) {
16     return getAddressMap().get(ip);
17   }
18   public void packetHandler(OFPacketIn packetIn)
      ↪ {
19     Ethernet payload = packetIn.getPayload();
20     if (payload instanceof ARP) {
21       ARP arp = (ARP) payload;
22       if (arp.opCode == 0x1 || arp.opCode == 0x2)
          ↪ {
23         if (lookup(arp.ip) == null) {
24           this.add(arp.ip, arp.mac);
25 } } } } }
```

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

SPIs are different from two classical types of potential vulnerabilities explored in the classical literature. First, in contrast to *stateless* performance issues, where a single input leads to an amplified response (e.g., [9], [10]), stateful issues entail a complex sequence of events. Second, in contrast to stateful security issues related to *protocol state* [11], [12], [13], SPIs target the state of internal data structures in the SDN controller. Although SPIs have been studied in other settings (e.g., databases [14]), to the best of our knowledge this has not been explored in the context of SDN controllers.

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 $\mathcal{O}(n)$ 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` and each message will trigger an $\mathcal{O}(n)$ in terms of its size. That is, we can "prime" the system into an undesirable state and then trigger a high-cost computation.

Exploit in ONOS. This performance issue may seem insignificant because the execution time of `CastorService` only increases linearly with respect to the number of ARP packets received. However, in our experiment, we successfully brought the SDN controller into an unstable state by introducing one malicious host to the network. The malicious host sends spoofed ARP packets every 100 ms and the SDN controller becomes unstable and starts to drop network packets from the network after 5 minutes. This causes the SDN controller to mark links as unavailable, affecting the bandwidth of the network. Furthermore, after the SDN controller enters the unstable state, disconnecting the malicious host or restarting the SDN controller will not solve the issue because the state is persistent. Unfortunately, it is also difficult for an automated tool to identify such an attack because the malicious packet is sent at a very low frequency.

B. Threat Model

Broadly, we focus on such resource exhaustion issues in SDN controller implementations; for example, performance bugs or design decisions that are subject to algorithmic complexity attacks [15]. At a high level, SPI affects the availability and performance of the SDN controller. This in turn can be a stepping stone for other kinds of attacks; e.g., inconsistency attacks against the data plane [16].

We consider two different threat scenarios:

- *Compromised network device:*We consider the scenarios where the attacker controls a vulnerable network device and launches the attack by sending legitimate OpenFlow messages.
- *Potential configuration errors vulnerable to attack:*While do not expect that an attacker controls the SDN controller directly (in which case launching SPI is trivial), we consider scenarios where the attacker with knowledge of a potential misconfiguration can trigger an SPI by sending legitimate OpenFlow messages.

Note that we do not require that the attacker can compromise the communication channel between the SDN controller and the network devices.

C. Prior Work and Limitations

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.

Our goal is to automatically uncover SPIs in SDN controllers. While this appears to be a seemingly natural application of program analysis, we observe that there are two key characteristics that make it challenging to apply existing approaches. First, the SDN controller is *stateful* with a large state space spanning multiple per-module and global state stores. Second, the SDN controller is a *complex* system code base, which make the system hard to analyze and test. Table I summarizes why closely related work does not apply in our setting.

Approach	Issue Type	Stateful	SDN
Protocol fuzzers [11], [12], [17], [18]	Memory & Protocol	Y	N
Performance fuzzers [3], [19], [20]	Performance	N	N
Static/dynamic perf. analysis [21], [22], [23], [24], [25]	Performance	Y	N
Performance Modeling [26]	Performance	Y	N
SDN fuzzers [27], [28]	Protocol	Y	Y
SPIDER (this paper)	Performance	Y	Y

TABLE I: Our problem statement and target application domain falls outside the scope of prior work.

SDN fuzzers. Existing black-box SDN fuzzers (e.g., Beads [28] and Delta [27]) generate packets based on an existing topology and focus on logic protocol bugs in SDN controllers [28], [27]. 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.

Performance fuzzers. Performance fuzzers (e.g., PerfFuzz [3] and SlowFuzz [19]) find inputs that maximize computational resource utilization based on execution feedback. Similar to SDN fuzzers, exploring the *large input space* of the SDN controller and obtaining reliable performance measurements in a *complex system* remain challenge problems [29].

Static code analysis for performance. Static performance analysis techniques (e.g., FindBugs [30], Clarity [21], Torpedo [14]) identify performance issues based on code patterns in the program. Unfortunately, specifying such patterns usually requires domain-specific knowledge and many patterns of SPIs are not described in existing tools. Clarity focuses on identifying redundant data traversals, where a program fragment repeatedly iterates over an unmodified data structure. The SPI shown in Figure 2 is not a redundant data traversal because the `addressMap` is only visited once.

Torpedo uses static taint analysis to identify second-order DoS vulnerabilities in web applications, which relies on fine-grained annotations of state updates and identification of user-controllable values. Doing this in SDN controllers is impractical due to the large variety of events/requests and internal state that is represented by arbitrary Java data structures. In addition, the assumptions of the sanitizers in Topedo do not apply to SDN controllers. For example, the branch condition in Line 23 Figure 2 is considered a conditional sanitizer, which prevents Torpedo from identifying the SPI. SPIDER does not require such manual identification of sources/sinks or any other SPI-triggering code patterns, and yet is able to find stateful SPIs.

Symbolic execution for performance analysis. Symbolic execution (e.g., Castan [24] and Wise [22]) can be used to identify program 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

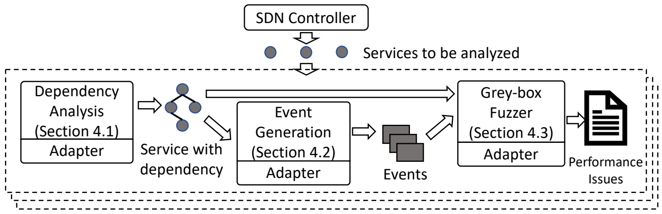


Fig. 3: A high-level overview of SPIDER.

limited to analyze small programs and cannot handle the *large state space* of the SDN controllers [31].

III. SOLUTION OVERVIEW

Given the limitations of static analysis or symbolic execution in tackling this problem, we adopt a fuzzing-based workflow. Fuzz testing or fuzzing [32], [33], [34] has proven to be useful in analyzing large and complex systems. As with any fuzzing approach and many other analysis frameworks, we have to necessarily compromise on completeness [32]. Our pragmatic goal is to build a utility that is helpful for developers to uncover SPIs. That said, we cannot apply existing fuzzing techniques directly, and our overall contribution is a practical application of fuzzing to our problem setting—covering stateful performance issues in SDN controllers. We start by describing the design space for fuzzing and argue why strawman solutions do not work. Then, 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:

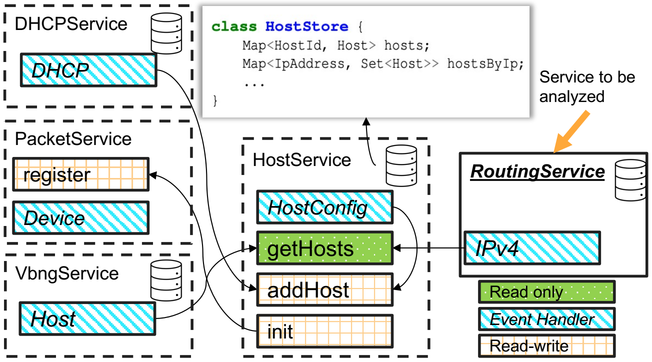
- *Granularity of code access:* One extreme is "black-box" fuzzing [32] with access only to input/output of the system under test. At the other extreme, we have "white-box" fuzzing [35] which inspects source code to analyze state and execution paths. Black-box approaches scale well but are imprecise, while white-box approaches are precise but do not scale to a very large complex. A middle ground is "grey-box" [36] fuzzing (e.g., AFL [37] and libFuzzer [38]) 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 (e.g., [27]) to generate OpenFlow message inputs to the SDN controller 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 the SDN controller and the packet contains an ARP payload with the operation code `0x1` or `0x2` (Lines 19–22). Indeed, we tried using the Delta [27] SDN fuzzer 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 [19] or PerfFuzz [3] 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 SDN controllers 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. 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 §II that both ONOS and OpenDaylight implement an event-based architecture where incoming OpenFlow messages trigger new events. SPIs can arise when some service r 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: Modular dependency-aware 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 §II that the controller consists of services that each handles one or more event types. If a 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.



```
class HostStore {
    Map<HostId, Host> hosts;
    Map<IpAddress, Set<Host>> hostsByIp;
    ...
}
```

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

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.

Design Choice 4: Extensible workflow with target-specific adapters. While the event-driven architecture is usually common among SDN controllers, their internal implementations are drastically different. For example, ONOS uses unified interfaces to construct and initialize all services, whereas services in OpenDaylight use multiple initialization pathways. Implementing a fuzzing framework embodying the above key ideas per target to be analyzed will require significant manual effort! To avoid this, we develop an extensible realization of the above design choices with a common "core". On top of this core framework, we enable target-specific adapters for each component, which can be synthesized automatically or implemented manually.

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 semantic 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.

IV. Detailed Design

Next, we describe the detailed design of SPIDER to realize the workflow from the previous section. We describe the general design using ONOS as a running example. We also discuss how the design can be applied to other controllers via target-specific adapters.

A. *Modular Dependency-Aware Analysis*

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 the SDN controller 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

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

Fig. 5: Simplified version of `HostEvent`.

   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`.

Target-specific adaption. In order to identify state objects, we need to look into the in implementation of storage layer of the SDN controller. Note that different SDN controllers implement the storage layer differently. We need target-specific adapters for identifying state objects and state operations; we discuss these in §V-A.

B. 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 [39], [40].

In the SDN controller, each event is a data structure that contains multiple fields. For example, Figure 5 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 [39], [20]. Figure 6 presents the pseudocode of a type-based object generator. The 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*.

```
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;
}}}
```

Fig. 6: 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
} }
```

Fig. 7: Simplified version of `HostEventGenerator`, which maintains inter-event constraints—hosts cannot be removed unless they have been previously added.

Broadly, we identify two types of validity constraints:

- **Intra-event constraints:** These specify the internal constraints in an event. For example, in Figure 5 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.

We do not yet have an automated way of generating events with inter-event constraints. The risk of using type-based event generators for such events is that they may produce invalid events, and thus 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.

We make a design choice to manually implement generators for only the most *critical* of events, and use automatic type-based generation for the rest.

Target-specific adaptation. Determining which events are *critical* to be manually implemented requires some target-specific insights. For instance, in ONOS there are 103 events in ONOS and manually implementing event generators for all events is impractical. We describe how we determine *critical* events for different SDN controllers in §V-B.

C. Semantic Performance Fuzzing

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 (§IV-A). 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 (§IV-B) 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 (§VI). We discuss how we pick the threshold for our experiments later.

SPIDER performs the search by combining ideas from PerfFuzz [3], a mutation-based grey-box performance fuzzer, and Zest [4], 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. Traditionally, grey-box fuzzing [36], as popularized by tools such as AFL [37] and LibFuzzer [38], 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 [3] 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 [19]. As such, PerfFuzz identifies algorithmic performance issues [15], 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 §IV-B. 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 [4], enables random mutations to be performed on inputs that are produced by generator functions like those described in Section IV-B. Observe that in Figures 6 and 7 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. 6 or `rnd.choose()` in Fig. 7), 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 (§IV-B).
2) Initialize a map $maxCounts$, which tracks the maximum execution cost observed at each program branch, by sending the event sequence σ_0 to the SDN controller 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* [4] 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.
   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 (§IV-B). 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) reconstruct such a message sequence and trigger the performance issue in a network emulation.

Module	Spider Core	ONOS Adapter	ODL Adapter
Dependency Analysis	1080	210	0
Event Generation	1432	1479	848
Performance Fuzzing	5596	485	216

TABLE II: LOC of each component in Spider.

Name	Uniform Constructor	Storage Mocking	Storage Accessing
ONOS	✓	✓	Type-based
ODL	✗	✗	Key-based

TABLE III: Key Differences: ONOS vs. OpenDaylight

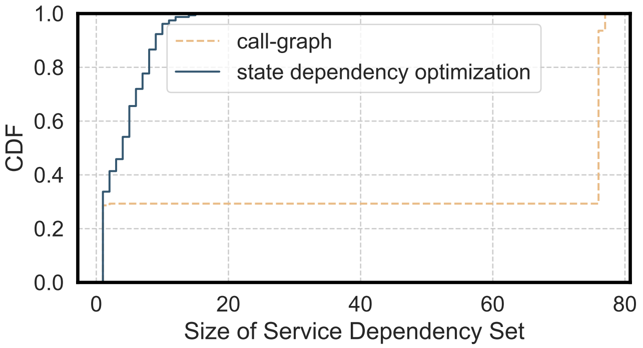


Fig. 8: CDF of service dependency set sizes computed by the two algorithms across the 157 services analyzed in ONOS. Smaller size is better: the state-dependency optimization reduces the size of the dependency sets.

```
// Event key for node connectors.
InstanceIdentifier key1 =
↪  InstanceIdentifier.builder(Nodes.class)
        .child(Node.class, nodeKey)
        .child(NodeConnector.class, connectorKey)
        .build();
// Access node connector data.
Object nodeConnector = storage.read(key1);
// Event key for topology.
InstanceIdentifier key2 =
↪  InstanceIdentifier.create(NetworkTopology.class)
        .child(Topology.class, topoKey)
// Access topology data.
Object topology = storage.read(key2);
```

Fig. 9: The code from OpenDaylight controller that creates different state keys.

Target-specific adaptation. While the fuzzing algorithm is general across controllers, we need target-specific fuzzing harnesses. For example, we need to tackle problems such as how to launch services, how to reset the state, and how to determine the threshold t_{max}. We describe how we solve each problem for ONOS and OpenDaylight in Section V-C.

V. Implementation

We implement Spider in Java and Kotlin [41]. Table II shows the line of code of each module. The dependency analysis framework is built on top of Soot [42] framework and the performance fuzzer is built on top of JQF [43]. For each module in Spider, we first discuss the implementation of the extensible core and then describe the target-specific adapters. Table III summarizes the key differences between ONOS and ODL.

Preprocessing to identify analyzable service Before we begin our analysis, first we need to identify the set of analyzable services in a given controller. An analyzable service is one that registers at least one type of event. In ONOS, all services and event handlers have the same sub-type, which allows us to collect analyzable services automatically. Services in OpenDaylight have different signatures and implement event handlers in different ways. In order to identify analyzable services in OpenDaylight, we had to inspect the source code manually. To scope this manual effort, we focus only on the critical OpenFlow module and identify its attendant services. Together, we have as input 157 services to be analyzed in the ONOS and 10 services to be analyzed in (the OpenFlow module of) OpenDaylight.

A. Dependency Analysis

Core: We extend Soot [42] to build a call graph. However, Soot does not handle interface calls and function calls between threads. To this end, Spider performs an overapproximation and finds all classes that implements the interface class. To build the call graph across threads, we maintain a dictionary of function names that can invoke functions in other threads. We heuristically identify such call relationships based on the method name and callee type. As mentioned in §IV-A, just using the call-graph will include services that do not affect the state of the target service. Identifying the state objects accessed by the event handlers of each service is critical for reducing the search space. Thus, we need target-specific adapters to identify state objects read and written by event handlers.

Adapter for ONOS: For ONOS, state objects can be identified based on their types and the operations of state objects are defined through public interfaces. Thus, we adopt the API specification of read/write operations from EventScope [44] and pass the state operations transitively along the call-graph. Figure 8 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 ≈ 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.

Adapter for OpenDaylight: Unlike ONOS, OpenDaylight accesses state objects with state keys shown in Figure 9. Note that `key1` and `key2` have the same type but are constructed differently to represent different state objects. It is only possible to distinguish them dynamically with the value of the keys. This prevents us from precisely identifying the state objects accessed by each service. In this case, we use an over-approximation and assume that the service accesses the entire state space.

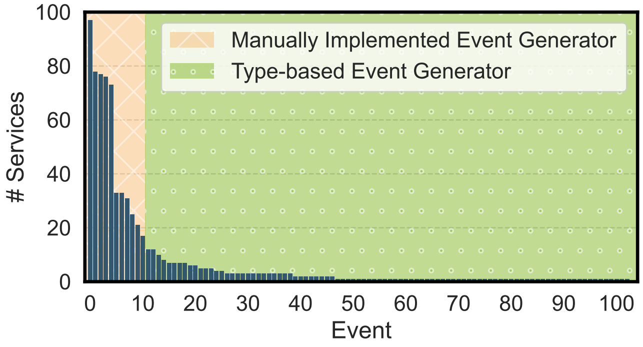


Fig. 10: For each event type in ONOS (X-axis), this plot shows the number of services whose states are affected by that event type (Y-axis). SPIDER uses hand-crafted event generators for the top 10 most critical events, and automates the generation for the rest using type information.

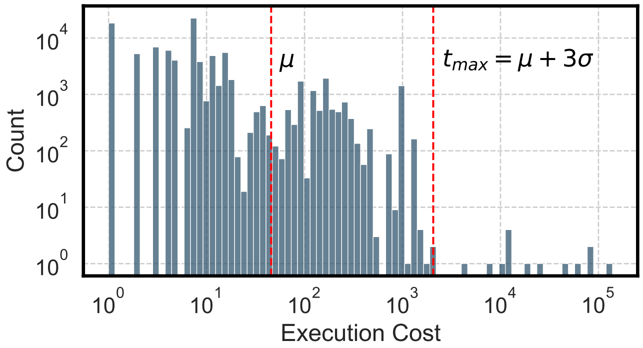


Fig. 11: Distribution of the per-event execution cost in a normal-workload emulation environment of ONOS (log-log scale). **We set the threshold at three standard deviations higher than the mean normal operation cost.**

B. Event Generation

Core system: We implements a type-based event generator using Quickcheck [40] and JQF [43]. Type-based generation may violate constraints, and ideally we need constraint-aware event generators for all events. However, this is impractical for large systems. Thus, we use target-specific adapters to identify and implement constraint-aware event generators for critical events. for all services is not practical.

Adapter for ONOS: In ONOS, all events have the same subtype and can be identified statically. There are 103 types of events, and all of them contain both intra-event and inter-event constraints. We identify critical events by counting the number of distinct services whose states are affected by the event. Figure 10 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 5) for the top 10 events.

Adapter for OpenDaylight: Here, events span different types and services implement event handlers differently. Since we scope our analysis to analyzable services in the OpenFlow module, we identify 7 events registered by these services and manually implement constraint-aware generators.

C. Fuzzing for Performance Issues

Core system: We implement the performance fuzzer in Java and Kotlin [41] on top of the JQF [43] framework, which we extend to support performance fuzzing [3](§IV-C). SPIDER uses ASM [45] and ByteBuddy [46] to instrument ONOS and OpenDaylight to collect the performance costs of events. A naive option for fuzzing is to launch a new instance of the SDN controller for each execution. However, starting an instance of the SDN controller is slow. For example, starting the ONOS system takes 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. Thus, we need efficient target-specific adapters to launch the target services and its dependencies, dispatch the events to the target, and collect the execution cost.

Adapter for ONOS: ONOS services have unified function signatures to construct, initialize, and process events. We exploit this to dynamically construct the target service and its dependencies using dynamic dependency analysis. To enable state reset, we use *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.

Adapter for OpenDaylight: OpenDaylight uses multiple modes for initializing services; e.g., the cluster service is constructed through a class factory, the RPC service has its own class constructor, while OpenFlow services are constructed and initialized using different functions. We launch the entire controller and retrieve a reference to the target service through Java reflection APIs. In terms of state reset, OpenDaylight does not implement an in-memory mock store and reinitializing the storage service will not reset the states. Recall, however, that OpenDaylight uses a key-based storage accessing design (§V-A), and all state objects are accessed by keys through the same interface. We exploit this design, track the data objects created in the storage during each fuzzing execution, and remove them after the execution finishes.

D. Alerting threshold

We need to set a threshold $t_{\max}$ to identify an event sequence where the last event takes longer than $t_{\max}$ to be flagged for further investigation. We use a data-driven threshold selection. First, we build a simple emulation environment using Mininet[47] with 4 hosts and 2 switches. We use `ping` utility to generate data plane packets and monitor execution costs of event handling in the SDN controller using JVM bytecode instrumentation.[1] As mentioned in Section V-C, 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

[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.

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 and 7225 events for the OpenDaylight. Figure 11 shows the histogram of the per-event execution cost for ONOS. We set the threshold $t_{\max}$ to be equal to the mean (μ) plus three standard deviations (3σ); this value is 2,059 conditional branch instructions for ONOS and 31,280 conditional branch instructions for OpenDaylight. The threshold for ODL is higher than that for ONOS because of the difference in instrumenting the distributed storage, as described above. Any event whose execution path is longer than this threshold is flagged as a potential SPI.[2]

E. Validation and Strategy Reconstruction.

For each potential vulnerability reported by SPIDER, we want to verify if it actually represents a true vulnerability in the SDN controller. Our insight here is that events in the SDN controller provide useful information about events in the network. For example, in the ONOS SDN controller, 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 [47]. [3]

VI.EVALUATION

We use SPIDER to analyze ONOS v2.2.4 [48] and OpenDaylight Silicon-SR4 [49]. We analyze each service along with its dependency set (§IV-A). We use Cloudlab VMs with 4 cores (2.4 GHz) and 4 GB memory.

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 search space. We configure SPIDER to find a sequence of events with lengths N=1, 50, 100, 250, 500, 1000, and 2000. For each N, we allocate a budget B of 1 hour to analyze each service. Given these parameters, the fuzzer runs and reports the smallest N, where the cost of handling the last event exceeds the threshold, or NULL if no such event was found. We use offline profiles with normal event sequences to pick the threshold of $t_{\max}^{ONOS}$=2059 instructions for ONOS and $t_{\max}^{ODL}$=31280 for OpenDaylight, as described in §V-D.

A. Analysis Summary

We were able to successfully analyze 148 out of 157 ONOS services and all 10 OpenDaylight services. The remaining 9 services could not be analyzed due to implementation challenges that caused the automated type-based event generator to construct only empty objects or the service to crash while processing input events.

After fuzzing each of the 158 services with the above parameters, SPIDER reported 11 potential issues, summarized in Table IV. All performance issues are identified in the ONOS SDN controller. 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. [4] 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.[5]

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. We have not identified any performance issues in OpenDaylight so we did not contact the OpenDaylight developers.

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 $\mathcal{O}(n^2)$ for n events.

[2] The outliers in Figure 11 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.

[4] We qualitatively confirm that many of these issues manifest in *proactive* SDN, and not merely reactive SDN with a controller setting up rules for every flow arrival. This makes them potentially serious in production.

[5] We are not able to replicate V4 due to the another bug triggered by the emulator §VI-B).

ID	Class Name	Description	Source	Complexity	Smallest N	Time
V1	`CastorArpManager`	The execution cost of `CastorArpService` increases linearly with respect to the number of `OFPacketIn` with ARP payload received by the service.	host	linear	2000	0.9
V2	`NeighborResolutionManager`	The execution cost of `NeighborResolutionManager` increases linearly with respect to the number of connect points in the network.	switch	linear	50	33.8
V3	`PortStatisticsManager`	The execution cost of `PortStatisticsService` increases linearly with respect to the number of `OFPortStatisticsReply` messages received by the service.	switch	linear	2000	46.7
V4	`AbstractGraphPathSearch`	The execution cost of `AbstractGraphPathSearch` service increases exponentially with respect to the number of links in the network.	switch	exponential	50	0.8
V5	`MyTunnelApp`	The execution cost of `MyTunnelApp` increases linearly with respect to the number of hosts in the topology.	switch	linear	50	0.8
V6	`VplsManager`	The execution cost of `VplsService` increases linearly with respect to the number of interfaces configured in the SDN controller.	controller	linear	50	1.1
V7	`NetworkConfigLinksProvider`	The execution cost of `NetworkConfigLinksProvider` increases linearly with respect to the number of port created for each switch.	switch	linear	50	2.6
V8	`MQEventHadler`	The `MQEventHandler` performs a costly computation while processing IPv4 packets.	host	constant	1	3.3
V9	`RouterAdvertisementManager`	The execution cost of `RouterAdvertisementManager` increases linearly with respect to the number of interfaces created in the network.	switch	linear	50	2.8
V10	`LinkDiscoveryProvider`	The execution cost `LinkDiscoveryProvider` increases linearly with respect to the number of switches in the network.	switch	linear	1000	9.8
F1	`ControlPlaneManager`	An invalid `ControlMessageEvent` causes high execution of the `ControlPlaneManager`.	N/A	N/A	1000	N/A

TABLE IV: Summary of performance issues identified by SPIDER in ONOS. Each row shows the affected class, a description of the issue, the source of OpenFlow messages that can trigger the issue, the *per-event* algorithmic complexity when triggering the issue, the smallest empirical sequence length to uncover the issue, and the time in minus for SPIDER to generate a event sequence that triggers the performance issue.

Table IV classifies each along our two dimensions. 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 the SDN controller 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 the SDN controller to perform a high- constant-cost execution; 8 issues cause the SDN controller to perform a computation whose per-event cost increases linearly with respect to the number of events generated; 1 issue (V4) causes the SDN controller to perform a computation whose cost increases exponentially with respect to the events or generated.

B. Case Studies

Next, we take a deeper dive into some of the specific issues to illustrate the subtleties of these and why off-the-shelf fuzzing techniques could not have identified them.

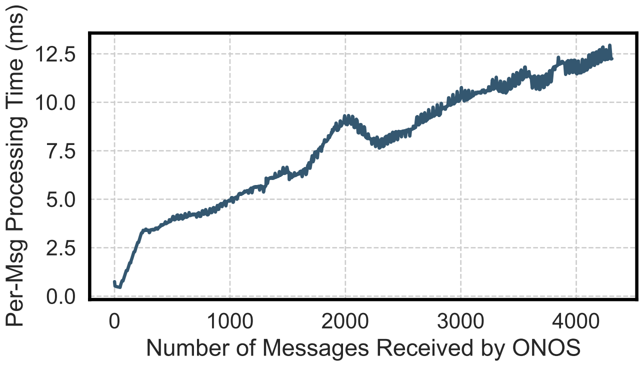


Fig. 12: Proof-of-concept experiment for issues V1 from Table IV simulated in Mininet. A malicious host can degrade the performance of ONOS by sending spoofed ARP packets.

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 of the issue is depicted (highly

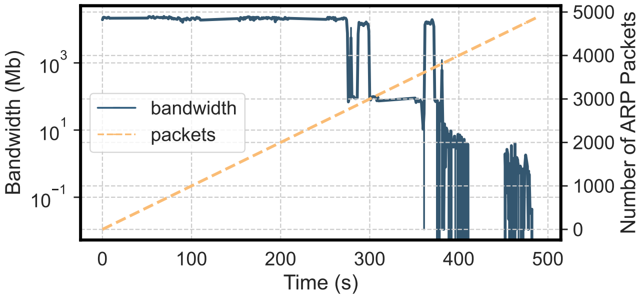


Fig. 13: The bandwidth of the network drops significantly 300 seconds after the ARP spoofing attack starts.

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 [47] 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. Figure 12 presents the moving average of the *per-message* processing time of each OpenFlow message received by the ONOS SDN controller with respect to the number of messages received by ONOS. The plot shows that an attacker can slowly degrade the performance of the SDN controller by generating spoofed ARP packets that take longer and longer to process.

We use iperf to measure the bandwidth between two benign hosts in the network. Figure 13 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 the SDN controller by increasing the average processing time of each OpenFlow message. OpenFlow messages with LLDP payload are used to check the liveness of links. The ONOS SDN controller 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, the SDN controller cannot be recovered easily by disconnecting the malicious host or rebooting. The SDN controller 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.[6]

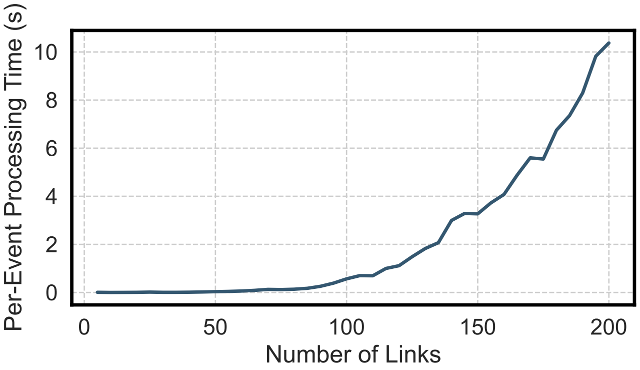


Fig. 14: Execution time of `AbstractGraphPathSearch` service increases exponentially with respect to the number of paths created in the network.

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 14 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.

C. Sensitivity Analysis and Baselines

Next, we validate key design choices against baselines and existing techniques. We first use a black-box SDN fuzzer and a static performance analysis tool to see if they can identify any stateful performance issues. Since there is no prior performance fuzzer for the SDN controllers, as baselines we use: a naive version of SPIDER with only type-based event generators. For these experiments, we focus on ONOS for brevity.

Comparison to black-box fuzzing. Delta [27] is the state-of-the-art black box fuzzer, which can generate stateful OpenFlow messages. We used Delta to analyze the ONOS SDN controller

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

for two hours. Since this fuzzer does not use instrumentation, we can only identify performance issues by measuring the execution cost for every input it generates. The black-box fuzzer is only able to trigger 2 issues (V2 and V5), and cannot identify subtle issues that modify the topology.

Open-sourced static analysis tool. We use FindBugs [30] to analyze the ONOSproject, and it reports 527 performance-related issues. We manually verified that FindBugs does not report any performance issues reported by SPIDER. Most issues reported by Findbugs are linting issues such as the inner class should be static. Other static analysis tools, such as Clarity [21] and Torpedo [14] are not available. We provide a conceptual comparison in Section II-C.

Value of constraint-aware event generation. We disable manual event templates and use the same type-based generation used for the remaining events called SPIDER-generic. We rerun our main experiments for all services with sequence length 2000, which is the sequence length with which SPIDER identifies the most issues, and a time budget of 1 hour. We find that SPIDER-generic is unable to uncover any new issues. This shows that domain knowledge in our workflow was critical to identify complex stateful issues.

VII. Other Related Work

We discussed the most closely relevant work in §II. Here, we discuss other related efforts in fuzzing, performance analysis, and SDN security. To the best of our knowledge, we do not know of prior work that tackles SPIs in SDN controllers.

Languages for performance analysis. Performance modeling languages such as RAML [26] 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 [25] and PerfPlotter [23]) collect execution traces and produce an algorithmic complexity estimate [25], [23]. 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 [11], [50], [18] or try to infer protocols automatically [12], [13], [17]. These focus on protocol bugs or correctness issues, rather than SPIs. AFLNet [17] and RESTler [18] assume a client-server architecture, where a fuzzer generates requests and learns from the server response. This does not translate directly to asynchronous SDN protocols.

Vulnerabilities in SDN controllers: Nice [51] uses symbolic execution and model checking to identify property violations. Beads [28] and Delta [27] identify logical protocol vulnerabilities in SDN controllers that do not rely on a pre-existing state. ConGuard [52] and SDNRacer [53] use static analysis to identify race conditions in the SDN controller. EventScope [44] focuses on missing event handlers in SDN applications, and AudiSDN [54] identifies inconsistent policies among different modules. None of these efforts tackle SPIs.

VIII. Discussion

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 control software. At many points it was not clear that such a combination is even amenable to analysis; e.g., large search space, distributed state stores, complex event semantics, state reset, among other things. 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 automate 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] 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.

[3] 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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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

[9] 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

[10] 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

[11] 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.

[12] 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.

[13] 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

[14] 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

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

[16] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang, "The CrossPath attack: Disrupting the SDN control channel via shared links," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 19–36. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/cao

[17] 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.

[18] 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.

[19] 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

[20] 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

[21] 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

[22] 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.

[23] 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

[24] 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

[25] 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

[26] 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

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

[28] 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.

[29] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, "Industry practice of coverage-guided enterprise-level dbms fuzzing," in *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '21. IEEE Press, 2021, p. 328–337. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[30] 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

[31] 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

[32] 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

[33] 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.

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

[35] 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.

[36] 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.

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

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

[39] 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

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

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

[42] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[43] 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

[44] 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.

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

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

[47] 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

[48] "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.

[49] "OpenDaylight (ODL) is a modular open platform for customizing and automating networks of any size and scale," https://www.opendaylight.org/, accessed: 2021-08-31.

[50] 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.

[51] 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.

[52] 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.

[53] 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

[54] 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.