What, Where, and When: Software Fault Localization for SDN

Abstract

In this paper we leverage the structure of the SDN software stack to automate the process of troubleshooting networks. We present two techniques for programmatically localizing the root cause of network problems: cross-layer correspondence checking infers what problems exist in the network, and where in the control software the problem first developed; and simulation-based causal inference infers when the triggering event(s) occurred. We evaluated our tools on three popular SDN platforms—Frenetic, Floodlight and POX—and found or reproduced three bugs: isolation breaches and faulty failover logic between replicated controllers.

1 Introduction

Given their critical role in enterprises, one might expect that networks would come with a well-developed suite of troubleshooting tools. However, the unfortunate truth is that `traceroute`, developed in 1987 [15], remains the network administrator's most sophisticated diagnostic mechanism. This reflects the lack of structure in network control planes, which are an ad hoc mixture of distributed protocols and manual configuration that directly manipulate forwarding tables. It is hard to tell what is broken when desired behavior is only implicitly expressed in the routing entries themselves, and that is the case with today's networks.

In this respect, the emergence of Software-Defined Networks (SDN) provides both an opportunity and a challenge. Moving control logic out of hardware and into software enables enables concise policy specifications and significantly more sophisticated testing and troubleshooting tools. Moreover, since SDN is still in its infancy (compared to traditional networking approaches), we as a community have an opportunity to make diagnostic tools a more integral part of the overall design process. Although SDN's goal is to simplify the *management* of networks, the challenge is that the SDN software stack itself is a complex distributed system, operating in asynchronous, heterogeneous, and failure-prone environments. In this paper we present our approach to troubleshooting bugs in SDN control software, a tool called W^3.

W^3 simplifies the troubleshooting process by programmatically localizing the root cause of network problems along three dimensions: *what* network problems exist at a given point in time, *where* in the control software a problem first developed, and *when* the triggering event occurred. To accomplish this W^3 employs two techniques:

Correspondence Checking. We observe that the structure of the SDN software stack (as we discuss in the next section) enables a straightforward algorithm for checking that control applications' policies are implemented correctly in the physical network. Our algorithm enumerates all policy-violations (*i.e.*, any instance where the high-level policies are not properly implemented by the SDN control software) present in the network at a given point in time, providing a crisp determination of the range of inputs and the system component(s) responsible for the fault.

Simulation-based causal inference. W^3 can replay the execution of the system against a stream of network events (e.g., link failures). This allows us to (i) distinguish between policy-violations that are harmless and quickly heal and those that are persistent, and (ii) identify the minimal set of events that triggered the policy-violation.

In combination, correspondence checking and simulation-based causal inference programmatically localize software-faults in the SDN software stack. With W^3, operators and developers are free to focus their efforts on debugging the code itself, without needing to diagnose the symptoms in the first place.

It is important to place this work in context and scope the problem we are attacking. First, W^3 is a troubleshooting tool, not a debugger; by this we mean that W^3 helps identify and localize network problems (what, where, and when?), but it does not help identify exactly which line of code causes the error (why?). Second, W^3 is focused on the system software of the SDN stack (described in the next section). While progress has been made in troubleshooting control applications that run on top of the

SDN platform [6] and in troubleshooting the forwarding tables in the physical switches [20, 16], we are not aware of previous troubleshooting work that focuses on the SDN platform itself; to the best of our knowledge, painstaking analysis of detailed logs is the current state-of-the-art in SDN platform troubleshooting.

In evaluating W^3 on three popular SDN platforms—Frenetic, Floodlight and POX—we found or reproduced several bugs: isolation breaches and faulty failover logic between distributed controllers. We also demonstrated the feasibility of deploying W^3 on production networks, finding that our tools can enumerate all policy-violations in simulated networks exceeding 25,000 hosts in under 5 seconds.

The rest of this paper is organized as follows. In §2, we present an overview of the SDN stack and its failure modes. In §3 we present correspondence checking and simulation-based causal inference in detail. We discuss the design of W^3 in §4. In §5 we present three bugs found by W^3, as well as a performance evaluation. Finally, in §6 we discuss related work, and in §7 we conclude.

2 SDN Overview

In this section we first sketch the software architecture of SDN networks, and then describe some examples of errors observed in production software-defined networks.

2.1 SDN Architecture

SDN networks are managed by software running on a set of network-attached servers called "controllers". This software is comprised of three distinct layers, as depicted in Figure 1. The two lower layers are part of the SDN platform, and the highest layer is the control application. We now describe the functions of each layer.

Control Application: This component specifies the desired high-level behavior of the network. We term these behavioral specifications "policies". It is the job of the SDN platform to implement these high-level policy-specifications by configuring the forwarding tables of the physical switches. The platform does so in two steps, each implemented in a separate layer.

Physical View: The lowest level of SDN software maintains a graph data-structure known as the 'physical view', that has a one-to-one correspondence with the physical network. When informed from above about a new policy, synchronization logic in this layer generates a set of configuration changes and sends them to the corresponding network devices. Conversely, when a state change occurs in the network (links going down, new ports installed, etc.), this layer notifies the layer above of the change.

Logical View: The middle layer in the SDN software architecture is sometimes called the virtualization layer because it translates the (possibly complicated) physical view into a simpler logical view. A common pattern (and the one we focus on here) is to represent an entire datacenter network as a single logical switch [7]. This allows operators to specify routing, access control, and QoS policies by configuring a single forwarding device. Thus, network policies (emanating from the control application) can be expressed as a set of flow entries $< header, actions >$ where possible actions includes primitives such as forward out a particular port, drop, or encrypt, and the possible ports include all edge ports (either connecting to external networks, or to hosts). Such a specification would dictate how any packet entering the network should be handled: *i.e.* what outgoing edge port it should be forwarded to, and perhaps what middlebox services it should be subject to along the way. In addition to providing a simplified network view, the virtualization layer can support multi-tenancy by providing each tenant with their own logical network to specify policies over. The platform then multiplexes the policies onto the same physical network.

The logical view greatly simplifies the job of specifying policies. However, SDN does not reduce the overall system complexity; it merely moves complexity out of the control application and into the platform, which must transform these high-level policy specifications into the appropriate configuration of each physical device.

The SDN platform not only must handle a complex task, it must do so in a distributed manner running in a highly dynamic environment. Because modern datacenter networks are large (easily reaching thousands of switches and a hundred thousand hosts), the SDN platform must be replicated across many servers. Onix [17], for example, partitions a graph of the network state across either an eventually-consistent DHT or a transactional database, allowing control applications to make their own tradeoffs in choosing consistency models, degree of fault tolerance, and other properties. The large scale of these networks also means that error events such as link failures or software crashes are common. Microsoft, for example, reports 36M error events over one year across 8 datacenters, which implies 8.5 error events per minute per datacenter [13].

Before describing the many ways in which such a system might fail, we note that there are two different kinds of SDN control applications: proactive and reactive. Proactive applications pre-compute forwarding tables for the entire network, and only push down updates

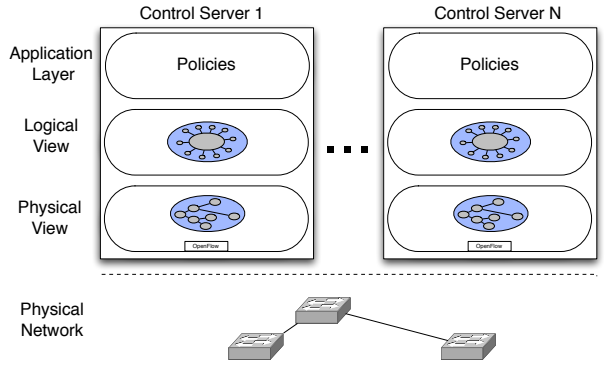

Figure 1: The SDN Software Architecture

periodically to react to link failures, changes in traffic mix, *etc.*. In contrast, reactive control applications forward all new flows to control servers. After a control decision is made, a flow entry is installed in the ingress switch, and the packet is forwarded along. Production SDN deployments are commonly proactive, primarily due to the large scale of datacenter networks and the current capabilities of forwarding hardware. We focus on proactive controllers for the remainder of this paper, although our troubleshooting mechanisms are also applicable to reactive applications. For the purposes of our paper, the key difference between the two cases is which events trigger actions from the SDN platform; for the proactive case, the SDN platform only responds to events that relate to the network view (such as link failures, VM migrations, etc.), while for the reactive case every packet arrival could potentially invoke the SDN platform.

2.2 Platform Failure Modes

W^3 is designed to troubleshoot the SDN platform; here we discuss a few examples of platform failures. As described above, modern SDN platforms differ from 'first-generation' controllers such as NOX [14] in two dimensions: they extend vertically by providing a virtualization layer on top of which control logic resides, and they extend horizontally by distributing state across multiple control servers. Platform errors arise from both of these extensions.

Virtualization. Virtualization errors result from a mismatch between the logical view and the state of the physical network. When an entire datacenter network (up to 10,000 switches) is abstracted into a single logical switch, the mapping between the logical switch and the physical topology is highly complex; for example, a simple configuration change such as "the path from A to B should pass through C" must be implemented as routing entries in a sequence of switches in the physical network. Policy changes that span multiple shards of the physical view (shards being statically-defined partitions of the physical network small enough to be managed by a single control server) require coordination between controllers.

This coordination becomes even more complicated in a multi-tenant environment [7] where each tenant specifies policies on their own logical switch. In such a case, each controller must deal with multiple tenants, and each tenant's policies must be coordinated among multiple controllers. Maintaining isolation between tenants is critical; updates to the physical network must therefore be performed in a consistent fashion to ensure that isolation breaches do not occur for any in-flight packets, despite hardware failures and message delays. The virtualization layer also performs resource arbitration, ensuring that each logical network's QoS policies are each met by the capacity of the physical network.

Controller Coordination. Coordination between controllers entails the same classes of error conditions that arise in general distributed systems: inconsistent reads and writes, race conditions over message arrivals, and unintended consequences of failover logic are common. As an example, suppose a controller fails, and all the switches under its purview are adopted by a new control server. If the new parent neglects to properly query the switches for their current state, or reads stale control information from the data store, it may inadvertently install conflicting flow entries in the network. Errors may also result from non-disjoint partitioning schemes between controllers, or incorrect delegation of control for different portions of the network. High churn in the network topology due to VM migration and hardware failures exacerbate these issues.

In this paper we focus on bugs in the virtualization and controller coordination components of the SDN software stack. We are primarily concerned with corner-case scenarios such as correlated hardware failures, which are the hardest to test *a priori*. Corner-case scenarios, while rare, cannot be ignored because of the distributed nature and large scale of production networks.

3 Approach

W^3 simplifies the troubleshooting process by programmatically localizing the root cause of network problems along three dimensions: *what* network problems exist at

a particular point in time, *where* in the control software a problem first developed, and *when* the triggering event occurred. In this section we present the components of W^3 we have designed for performing software fault localization for SDN.

3.1 Correspondence Checking

The SDN platform transforms high-level policies (specified by the control application) into low-level configuration of they physical infrastructure. The configuration of the physical network should semantically correspond with the policy specified by the application layer, in the sense that the disposition of any packet by the physical forwarding tables should reflect the policies dictated for that packet. We refer to any lack of semantic correspondence between the policy and the configuration as a *policy-violation*.

We leverage the virtual packet algebra pioneered in headerspace analysis [16] to verify whether the application's policies are indeed implemented correctly in the physical network. Our algorithm, correspondence checking, provides a crisp determination of all possible packet inputs that would not behave according to the application's policies if injected into the network at a particular point in time. Furthermore, running correspondence checking between intermediate layers of the SDN stack (logical view versus physical view), allow us to identify the component(s) of the system where policy-violations first manifest themselves.

Formally, the state of the physical network, the physical view, and the logical view can be represented as a graph, $G = (V, E)$. Packets are series of bits, $h \in \{0,1\}^L = H$, where L is the maximum number of bits in the header.

Upon receiving a packet, forwarding elements apply a transformation function, potentially modifying packets before forwarding them on[1]:

$$T : (H \times E) \rightarrow (H \times E_{\emptyset})$$

We use 'Ψ' to denote the collection of all transfer functions present in the network at a particular point in time. In this model, network traversal is simply a composition of transformation functions. For example, if a header h enters the network through edge e, its state after k hops will be:

$$\Psi^k(h,e) = \Psi(\Psi(\ldots\Psi(h,e)\ldots))$$

The externally visible behavior of the network can be expressed as the transitive closure of Ψ:

$$\Omega : (H \times E_{access}) \rightarrow (H \times E_{\emptyset})$$
$$\Omega(h,e) = \Psi^{\infty}(h,e)$$

Here, E_{access} denotes access links adjacent to end-hosts. In words, Ω is a mapping between all possible input packets inserted into the network from an end-hosts, to the final location of the packet after traversing the network.

In SDN, it should always be the case that:

$$\Omega^{view} \sim \Omega^{physical}$$

Informally, this means that any packet injected at an access link in $G^{virtual}$ should arrive at the same final location as the corresponding (encapsulated) packet injected at the corresponding access link in $G^{physical}$. Note that hosts are represented in all layers, although there may not be a one-to-one mapping between the internal vertices of $G^{virtual}$ and $G^{physical}$.

To check correspondence in SDN, we begin by taking a causally consistent snapshot [8] of the physical network. The routing tables of forwarding elements can then be translated into transformation functions. Finally, we feed a symbolic packet x^L to each access link of the network.[2] The end result is a propagation graph representing all possible paths taken by a packet injected at the access link.

The leaves of the propagation graph represent Ω. We verify correspondence in SDN by generating propagation graphs for all SDN layers, and comparing the leaves. Any mismatch in leaves of the propagation graphs represent policy-violations between control applications and network configuration.

It is important to note that correspondence checking assumes that the application's policies are correct; it only checks whether the physical network's configuration is isomorphic with the logical view, but does not check for additional correctness properties such as connectivity and loop-free routing. If the user wishes to explicitly express additional invariants, the HSA framework used by our system can easily check for such properties.

3.2 Simulation-based Causal Inference

Correspondence checking infers all policy-violations in the network at a particular point in time. However, troubleshooters need two additional pieces of the diagnostic puzzle.

[1]Multicast forwarding can expressed by extending the range to sets of output tuples

[2]The rules for process wildcard bits x^n are defined in the HSA paper [16]

First, in large distributed systems with communication delay and hardware failures, transient policy-violations are unavoidable, even common. That is, every time there has been a link failure that the SDN platform has not had time to respond to, or every time there has been a policy change that has not yet propagated down to the physical switches, there is a policy-violation. Most of these policy violations will be temporary, resolving as soon as the SDN platform has had time to respond. Some of these policy violations will persist, and that indicates a problem in the SDN platform. In addition, even some of the ephemeral violations may be harmful (such as those that violate isolation conditions), which again indicate a problem in the SDN platform.

Troubleshooters therefore need a mechanism to differentiate policy-violations that indicate a problem in the SDN platform from those that merely reflect inherent delays in responding to events; we will call all policy-violations that indicate a problem"pernicious" violations.

Second, once pernicious policy-violations are encountered, troubleshooters need to identify the events (link failures, controller failures, VM migrations, *etc.*) that triggered the problem. Moreover, they would like to have these events narrowed down to a minimal set, to make it easier to understand the problem.

Simulation-based causal inference is our mechanism for providing this information. Simulation-based causal inference models the network in a single simulation process, thereby providing arbitrary control over hardware failures, message delays and other failure modes. Simulation-based causal inference distinguishes between "internal" events that happen throughout the normal course of the system execution (*e.g.* message sends and receives), and "external" events injected into the system (*e.g.* link failures). We describe the details of the simulator in the next section.

The key insight behind simulation-based causal inference is that fault localization is significantly easier with the ability to selectively filter out external events from the system execution and observing how the system plays out in isolation. With complete control over the system execution, we are able to programmatically (i) track the lifetime of policy-violations to differentiate persistent from transient errors, and (ii) infer the 'minimal causal set' of events triggering the problem. We describe these components below.

Note that we have implemented policy-violation lifetime tracking, but we are still developing the infrastructure for minimal causal set inference. Nonetheless, we present our algorithm in this section.

3.2.1 Policy-Violation Lifetime Tracking

The first step in simulation-based causal inference involves detecting policy-violations and prioritizing them based on their duration. We do so in a relatively straightforward fashion. First, we take as input a stream of network events (*e.g.* link failures). Event sequences are either synthetically generated or gathered from a production trace of failure and topology change events, as enabled, *e.g.*, by OFRewind [26]. We then replay the execution of the control plane based on the input trace. Throughout the system execution, the simulator periodically invokes correspondence checking to enumerate all policy-violations (defined as any value in $\Omega^{physical}$ not present in $\Omega^{virtual}$, or vice versa). When a policy-violation is detected, the simulator forks off a branch that investigates the future system behavior in a case where no further failure events are played out. Finally, we prioritize the policy-violations based on their duration.

3.2.2 Causal Inference

Having identified persistent policy-violations, simulation-based causal inference seeks to identify the 'minimal causal set' of events leading up to the problem. By 'minimal causal set', we mean the minimally-sized set of events preceding (defined by the 'happens-before' relation [18]) the onset of the policy-violation, such that if any single event were removed the set, the policy-violation would not have resulted.

Simulation-based causal inference captures the notion of causality by attaching vector clocks [21] to all control messages in the simulated execution of the system. Our simulator runs as a single sequential thread; therefore we convert any partially-ordered input trace of network events into a totally-ordered trace by choosing an arbitrary sequential ordering that maintains the 'happens-before' relation.

Without regarding casual constraints, a naive algorithm to infer the minimal causal set would be to iteratively exclude each event from the system execution, and see if the policy-violation still appears. The problem is that we cannot simply "erase history"; antecedent events in the system execution may have depended on the event we are excluding.

Our algorithm therefore proceeds as as follows. First, consider the state of the system at exactly the point where the given policy-violation occurs. The active 'causal branches' in the system at this time are (i) the most recent event (message send, message receive, or internal state change) occurring on every node in the system, and (ii) the message send event for any in-flight control packets in

the network. Our goal is to prune these causal branches until we are left with the minimal causal set at the leaves of the pruned branches.

After having identified the active 'casual branches' in the system at the onset of the policy-violation, our algorithm essentially performs a traversal of the causal graph's (a DAG) topological ordering, pruning leaves that are not responsible for the policy-violation. In pseudo-code:

$causalSet \leftarrow []$
while $\exists$ *leaf in causalGraph* **do**
  remove leaf from causalGraph
  $violation \leftarrow runSimulation(causalGraph)$
  if *not violation* ***then***
    $causalSet += leaf$
  end if
end while

Here, $runSimulation(DAG)$ executes the system from the beginning of the trace, and checks whether the policy-violation occurs at the end of the execution. The beginning of the trace starts at some pre-defined point where the system was known to be in a coalescent state.

CS: Panda and I realized that the minimal casual set algorithm we presented in the OSDI submission is erroneous.

A few things are wrong with this:

- **We can't remove elements of the MCS from the DAG! We should only remove from the DAG if the event is not in the MCS; otherwise, the policy-violation will never be reproduced after the first element is added the MCS, since we took away one of the necessary conditions.**
- **Need to make a distinction between "random events" (e.g. link failures, node failures) and "deterministic events" (e.g., a node is programmed to send a pong when it receives a ping). In the MCS algorithm, it makes no sense to prune out deterministic events... However, the algorithm still needs to take in the entire casual DAG (both random and deterministic) to maintain casual dependencies [can't prune arbitrary random events]**
- **We assume that the random events are independent; we need make this assumption explicit.**

Note that each iteration of the loop can be performed in parallel by cloning the state of the simulator. The serial runtime of the algorithm is therefore linear with the number of events in the input trace.

We also note that Lamport's definition of the 'happens-before' relation is conservatively general, and can include events that are not in fact casually-related. In fact, as described here, our algorithm can be viewed as a generalization of dependency graph algorithms for hardware fault isolation [5]. In future work we hope to leverage domain knowledge of network control plane protocols to provide a more specific definition of causality (*e.g.* OpenFlow control messages with differing transaction ids are not causally related), allowing us to further prune events that are not in fact causally related from the DAG.

3.2.3 Additional Use-Cases

Besides lifetime tracking and causal analysis, our simulation infrastructure has a number of other possible use-cases:

Checking related problems by fuzzing. Input traces can be *fuzzed*, i.e., randomly perturbed, to expose the system to similar error conditions, and confirm that a proposed solution is not just a point-fix.

Investigating pathological environment conditions. The simulator allows for investigation of pathological environment conditions difficult to achieve in a real world test bed (*e.g.*, correlated failure rates, extremely long delays etc.). This enables investigation of situations that have a high potential for triggering errors.

Interactive exploration. Troubleshooters can also interactively bisect the trace or modify specific events to further pinpoint the cause for a failure. This is useful as soon as a suspect event sequence has been identified.

Regression/Integration Test Library. In traditional software engineering practices, integration tests are an important part of the software development cycle: developers feed end-to-end input through the system, and verify that the system execution satisfies certain safety and liveness properties. As additional failure cases are encountered in production, new cases can be added to a suite of integration tests to ensure robust operation of the system in future versions of the system.

Although the practice of accumulating an integration test suite over time is commonplace in other fields of computer science, the field of networking simply did not have the requisite software infrastructure to realize this practice before the emergence of SDN. Simulation-based causal inference can be viewed as our realization of this development practice, applied to network controllers. Our simulator's fine-grained control over failure scenarios allows us to test corner-case network conditions – those that are most difficult to anticipate in traditional unit tests. As known failure cases are accrued over time, we envision simulation-based causal inference being used to validate new and existing SDN platforms.

3.3 Discussion

Correspondence checking and simulation-based causal inference serve to isolate the platform layer and event sequence responsible for a given error. W^3 can be complemented by classical debugging techniques (*e.g.* log messages and source code debugging) to identify the root cause of the failure in the code. These techniques are much more effective when applied a specific event sequence. Once a potential fix has been developed, it can be validated by repeating the problematic execution within W^3. Input fuzzing further helps to validate whether there are related error events that the patch missed.

4 System Design

W^3 is our realization of correspondence checking and simulation-based causal inference as a useful platform to troubleshoot SDN controllers. In this section we discuss our goals in designing simulation-based causal inference, and the challenges we encountered in the process of realizing these goals.

4.1 Design Goals: The 7 rules of W^3

We seek to build a system that facilitates the process of troubleshooting. First and foremost, we hope that W^3 can reproduce difficult bugs observed in production networks, and automate the process of diagnosing their causes. We also envision W^3 being used as a common repository for difficult, corner-case scenarios known to have caused problems for other control platforms in the past. Given these potential use cases, we require the design of the system to be driven by the following requirements:

(1) Realistic Network Sizes. We focus on large, production SDN deployments. As today's datacenters may contain up to 100,000 hosts and 10,000 switches, our simulation infrastructure must be able to support large numbers of switches.

(2) Control plane focus We expect the dynamism in our system to stem from *control plane events*. Typical rates of control plane events must thus be handled, and control plane events must be modeled precisely. Conversely, we don't expect to handle a realistic amount of dataplane traffic, which is intractable for a software solution, and largely irrelevant in current networks (because they are mostly proactive, so control planes are not being driven by packet arrivals).

(3) Controller choice Our system should run with existing production controllers with minimum additional instrumentation. To allow for wider adoption, we don't want to limit ourselves to a particular controller implementation.

(4) Full determinism We want our simulation environment to be fully deterministic, such that repeated simulations with identical initialization values yield provably identical results. This creates a challenge in conjunction with our goal (3).

(5) Comprehensive Failure Modes. W^3 should support a wide range of failure modes at all components in the system, including switch and link failures and message drops, delays and reorderings.

(6) Corner cases investigation The potential state-space in a large-scale network is intractably large. We focus on interesting cases, as recorded, e.g., in production, or found through interactive evaluation. To investigate related error conditions, we *fuzz* the input traces.

(7) Interactivity The system should be fast enough for interactive exploration through an operator.

While none of these requirements were particularly difficult in isolation, taken in aggregate they posed some difficulties, as we now recount.

4.2 Components

As depicted in Figure 2, W^3 combines several components to facilitate the process of troubleshooting SDN platforms: W^3 takes input from production traces, interactive manipulation, and synthetic trace generation, and fuzzes these inputs to ensure that fixes are sufficiently general; W^3's simulator supports large, sophisticated networks; W^3 provides a deterministic, code-agnostic execution environment for running SDN control software; and provides efficient algorithms for checking correspondence throughout the system execution. We now provide an overview of each of these components, and the challenges we encountered in realizing our goals.

Trace Input And Fuzzing. Since a major goal W^3 was to support a wide range of usage scenarios, we provide support for three different methods for generating network trace inputs. The most common method is to insert failure and topology change logs from production deployments into the simulator for replay. Input traces may also be produced synthetically with configurable, random probabilities for network events. Lastly, we support interactive use, where the troubleshooter has complete control over network events, and is thereby free to explore her intuitions in order to reproduce a failure mode she has in mind.

Simulator. We have built a simulator for SDN networks, where network devices and hosts are modeled as lightweight python objects. Within a single thread, we

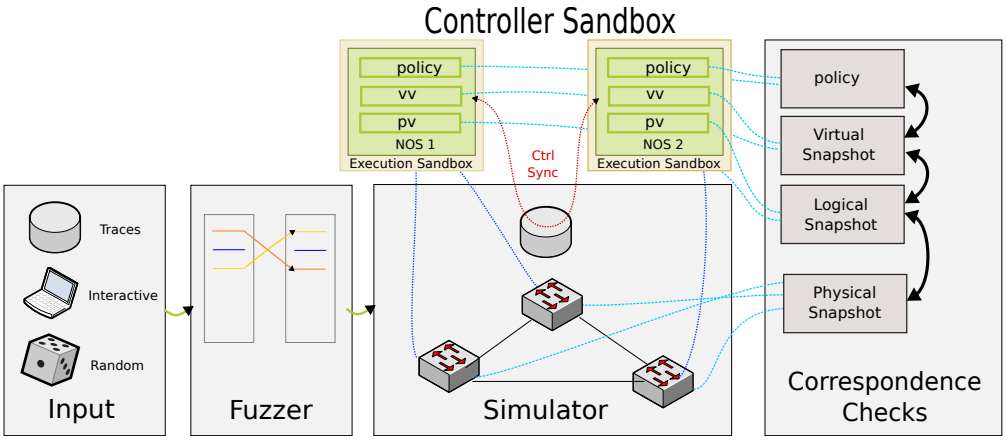

Figure 2: System architecture

are able to deterministically model the execution of very large networks. Our simulated model supports a wide range of failure modes, and provides fine-grained control over event orderings, component failures, and other aspects of the system execution. Our simulator currently supports switch failures, link failures, arbitrary packet reorderings, drops and delays, and a fully general control plane.

The main challenge we encountered in the design of the simulator was the maintaining large numbers of TCP connections to the controller(s). Although the controllers themselves may be spread over multiple physical servers, the main simulator must nonetheless handle all TCP connections between switches and controllers within a single process. We ultimately ended up using epoll to avoid limitations of the UNIX select implementation.

Controller Sandbox. One of our major goals for W^3 was to be able to run any SDN controller on top of the platform, with minimal code changes to the controllers themselves. In addition, control servers running on top of the simulated network must support deterministic execution for reproducible results.

Currently we run applications as UNIX processes outside of the simulator. We note however that there are a number of approaches for achieving deterministic replay for external software. For example: a software determinism layer (e.g. deterministic random number generators) is extremely lightweight, but requires modifications to the external software; binary rewriting does not require any modification to the external software's source code, but incurs moderate performance overhead; and VMs fully support deterministic replay, but only a relatively small number of VMs can be run on a single machine. We hope to leverage this previous work in future versions of W^3. Nonetheless, our architecture does not prevent us from running controllers on different physical servers in case we encounter memory or CPU bottlenecks.

Correspondence Checking. W^3 leverages the hassell library provided by HSA [16] to implement the correspondence checking algorithm. We optimize the code slightly to run efficiently on large networks; in particular, we parallelize symbolic packet propagation to a large number of subtasks. Correspondence checking currently requires a small code change to the controller to fetch the platform's view of the network state.

W^3 is written in roughly 5,000 lines of python, and is publicly available. [anon]

5 Evaluation

We applied W^3 to three open source SDN control platforms: Frenetic [12], POX [22], and Floodlight [4], and quickly found (or reproduced) one bug in each. The bug in Frenetic demonstrates the utility of checking correspondence between high-level policies and low-level configuration (without needing to specify invariants). The bugs in POX and Floodlight demonstrate the importance of simulation-based causal inference's ability to programmatically prioritize persistent policy-violations and infer their minimal causal sets.

For all three cases, only a small code modification to the controller was necessary to retrieve the the platform's state for correspondence checking.

5.1 Case studies

Here we the discuss the three bugs we found with W^3.

Frenetic. Our first example is Frenetic [12], a control platform providing functional-reactive language support for programming OpenFlow networks. Frenetic's language features aim to prevent common OpenFlow programming errors such as race conditions and overlapping flow entries; Frenetic's runtime system handles these low-level details on the application's behalf. Frenetic is a modern SDN controller with a reactive flow installation policy; we present it here to demonstrate that W^3, although focused primarily on proactive controllers, can nonetheless be used to troubleshoot errors in reactive control platforms.

When running learning switch, the simplest Frenetic application, we encountered persistent policy-violations immediately. The propagation graph (Ω) for Frenetic's runtime representation of the network policy had leaves that were not present in the physical network. Upon closer examination, we found that Frenetic's runtime system was neglecting to remove old FLOOD routing entries from its representation of the network policy after the hosts' route had been learned, even though the learning switch application had asked for these entries to be removed. Note that this case was not an overtly malicious bug; the FLOOD entries had indeed been removed from the physical network. The outdated controller state nevertheless went unnoticed; the bug in Frenetic's runtime was not specific to the learning switch, and could have resulted in failure to install flow entries at a later point in time if the application had asked to re-install them. The key takeaway from this example is that correspondence checking is a powerful mechanism for verifying that the controller's representation of the network matches the true network state correctly; without correspondence checking, troubleshooters would need to compare the routing configurations and controller's data structures side-by-side.

POX Our second example is POX [22]. POX is modeled after Onix [17], a production SDN platform; as in Figure 1 POX provides a physical view, a virtualized view, and a naive replication mechanism between distributed control servers.

Because the functionality within POX is relatively young, we chose to fabricate a bug in POX's distribution failover logic, and independently validate that the simulator was able to identify, prioritize, and find the minimal causal set for the fabricated policy-violation.

In particular, we injected the following bug: a controller replica performs updates to switches by (i) updating the persistent datastore storing the state of the network (thereby notifying other replicas of the update), and (ii) pushing the update to the switch. A control server writes a new ACL entry update to the datastore, but crashes before completing step (ii). The switch is adopted by a new replica, but the new control server assumes that the state in the persistent datastore is correct. The ACL entry is therefore never installed in the switch, and a breach of tenant isolation occurs.

We interleaved this event sequence with a normal system execution trace, and determined whether simulation-based causal inference could identify the policy-violation. Throughout the system execution there were a handful of transient policy-violations overlapping with the isolation breach. Nonetheless, our simulator was able to identify the correct policy-violation.

Floodlight: Distributed controller failover race condition More complex bugs and race conditions occur when controllers need to be distributed, and fail-over mechanisms between the individual instances are required for fault tolerance. Consider the following case described in the Floodlight [10] source code[3]: For high availability, Floodlight can run as a distributed controller, with switches connecting to several replica controllers at the same time. In this setup, one controller assumes the role of *master* and thereby gains the authority to issue state changing requests to the switches. The other controllers are in *slave* mode and thus do not perform any state-changes on the switch. Here, a race condition can occur when a switch connects to the controllers shortly after the master controller has died, but before a new master has been selected. In this case, all controllers will be in the slave role and thus will not take responsibility for clearing the switch flow table. At some point, one of the controllers is elevated to to *master* role and will proceed to manage the newly connected switch, based on an inconsistent flow table.

Using W^3 we were able to reproduce the problem. The emulated switches in the simulator support the *role* vendor extension to connect to several controllers. The inter-controller synchronization and heartbeat protocol is proxied through the simulator for control over the timing. After the master controller dies, a new switch is associated with the slave controllers, and integrated into the system with an unmerged flow-table, resulting in a persistent policy-violation between the flow-table representation in the controller and the switch.

5.2 Overhead

In addition to describing bugs, we show that W^3 is able to simulate and check large networks quickly.

Record and Replay Overhead. In contrast to general

[3]Note this issue was independently discovered by the developers of Floodlight.

record-and-replay mechanisms, the amount of recorded state needed for high-fidelity replay is tractable. With proactive flow installation, updates are pushed to routing tables over a relatively long time scale; periodic FIB snapshots along with a log of link state events, control server downtime, and host mobility information suffice for our purposes. As a point of reference, the Cisco 7000 core switch model supports a maximum of 128K MAC entries and 128K ACL entries [1]. Assuming 36 bytes per flow entry, (larger than the OpenFlow 13-tuple), each FIB will contain a maximum of 9216 bytes, uncompressed. A datacenter of 100,000 hosts includes roughly 8,000 switchee [3]. Therefore a snapshot of the FIBs of the entire network takes up roughly 74 MB. The VL2 paper reports 36M network error events over one year over 8 datacenters, which implies 8.5 error events per minute per datacenter [13]. Suppose we took a snapshot of the FIBs in the network every second. Then we would need to store roughly 4GB, uncompressed, per minute, a relatively small growth rate for datacenter logs. This information, in addition to a log of host mobility events (*e.g.* VM migrations) will suffice for our purposes. Note that this is a conservative overestimate.

CS: Notes from Rean Griffith:

- **total vms in a typical datacenter: 1000**
- **migration frequency (migrations/minute): 20 per hour**
- **VM spin ups/downs: 150 power ons per hour (see our OSR 2010 paper for power off estimates)**
- **Do we log VM migrations and how does that log grow (I wasn't able to get any estimates on log-growth data)**

We had an OSR 2010 paper that provided numbers scaled by the number of VMs in an installation: Challenges in building scalable virtualized datacenter management (http://dl.acm.org/citation.cfm?id=1899941)

Correspondence Checking Runtime. Computing the propagation graph for correspondence checking is equivalent to enumerating all possible paths in the network, which scales with the diameter of the network and the number of routing entries per switch. The propagation graph for each host can be computed in parallel however, so the computation is bottlenecked by the serial runtime of computing a single host's propagation graph.

We show the serial runtime of correspondence checking in Figure 3. For this analysis we generated fat tree topologies between 2 and 48 pods wide, with pre-installed PORTLAND [23] routing tables in each switch. Each data point is the minimum of three runs on a single Intel Xeon 2.80GHz core. Note that the number of PORTLAND routing entries per switch scales with the number of pods in the fat-tree. We excluded the time to convert flow tables to HSA transfer functions, since transfer functions can be maintained offline.

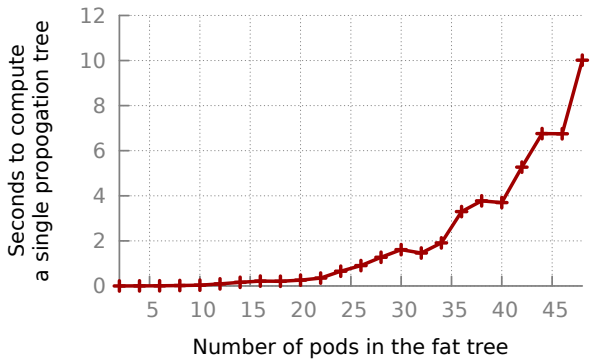

Figure 3: Serial runtime of correspondence checking on PORTLAND fat tree networks. Each datapoint consists of $x^3/4$ hosts and $5x^2/4$ switches (*e.g.* 48 pods means 27,468 hosts attached to 2,880 switches)

As the figure depicts, even for large networks (27,648 hosts) the serial runtime of correspondence checking is reasonable for interactive use. The number of serial tasks to be executed is the number of hosts in the network squared, disregarding ECMP load balancing.

Simulator Scalability. Our design models the entire network within a single process. We show in Figure 4 that this approach nonetheless scales to large networks. For this analysis we generated fat tree topologies between 2 and 48 pods wide, where all switches in the network connected to a single controller. The controller sent each switch an OpenFlow $FLOW\_MOD$ and subsequent $BARRIER\_REQUEST$ message, and waited for the corresponding $BARRIER\_REPLY$. We then measured the time to between the first $FLOW\_MOD$ sent and the last $BARRIER\_REPLY$ received. As expected, the runtime was roughly linear with the number of switches in the network. The figure also shows that the processing time for large networks (5 seconds per simulator round) was well within the bounds for interactive use.

We also tested the extreme limits of the simulator's scalability, pushing up the number of switches until something broke. We encountered what appears to be a limitation of the Linux TCP/IP stack: TCP connection attempts began failing beyond 26,680 sockets. Note that 26,680 switches is an order-of-magnitude larger than the today's biggest networks.

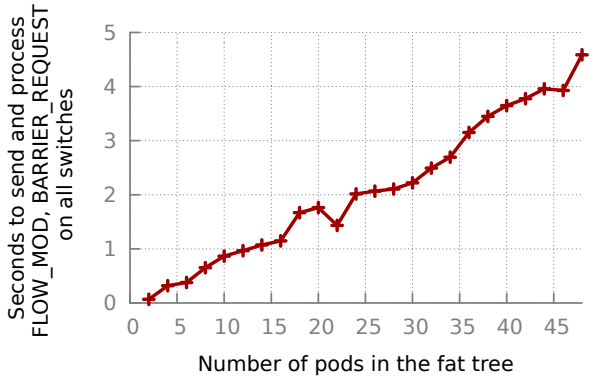

Figure 4: Time to send and process messages between controller and simulated switches. Each datapoint consists of $x^3/4$ hosts and $5x^2/4$ switches (*e.g.* 48 pods means 27,468 hosts attached to 2,880 switches)

5.3 Replay fidelity

On the one hand, since the SDN platform is in software, we can, in theory, reproduce all software-induced policy violations (though not problems resulting from flaky hardware implementing code incorrectly). However, this requires setting up the simulator to emulate the appropriate conditions that led to the policy violation, and that can be quite difficult. We hope to make progress in this area along two dimensions. First, we hope to help the community build up a set of regression tests, so that a wide variety of bug-triggering scenarios are available in a public repository. This would go a long way towards providing adequate test coverage.

Second, we hope to gather error logs from real production deployments which will help us populate this repository; this may require providing novel kinds of anonymization, so that large datacenter operators would be willing to share their problems (since they want their SDN code to work) without revealing the details of their network. This may require a infrastructural counterpart to minimally-causal events; the smallest number of infrastructure components that can reproduce the same bug.

Also, note that our correspondence checking algorithm can not verify time-dependent policies such as "No link should be congested more than 1% of the time", or "No server should receive more than 500MB/s of external traffic". In future work we will extend our correspondence checking algorithm to account for this class of policies.

6 Related Work

This work extends a growing literature on troubleshooting tools for Software-Defined Networks.

The work most closely related to ours is NICE [6]. NICE combines concolic execution and model checking to automate the process of testing NOX applications. This enables one to catch bugs before they are deployed.

Our approach and NICE complement each other in several ways. First, NICE's systematic exploration of failure orderings is potentially of great use for finding corner-case errors, which we could then add to our regression suite. NICE may also be applied directly to the code-base of the SDN platform, but in the case that only a subset of all possible code-paths in the SDN platform can be model-checked due to state-space explosion; our mechanisms allows users to troubleshoot errors *post-hoc* after they are observed in production, so we can find bugs that might be missed due to truncating the state-space exploration. In complement to NICE, correspondence checking helps developers isolate the specific component of the SDN platform responsible for an error, without needing to specify invariants.

Focusing on the physical network, Anteater [20] and HSA [16] are alternative approaches to statically checking invariants in the configuration of switches and routers. Both take take as input a snapshot of the FIB of each network device. To check invariants, Anteater generates a set of constraint functions and feeds them through a SAT solver, while HSA defines an algebra for virtual packets and their transformation through the network. We leverage the HSA work in W^3, and our simulator allows us to detect policy-violations not just in a given set of tables but what tables are produced by a wide range of scenarios.

Also focusing on the physical network, OFRewind [26] develops record and replay techniques for the control plane of OpenFlow networks. Unlike simulation-based causal inference OFRewind focuses specifically on OpenFlow interactions, while we focus on more course-grained replay of failures and topology changes. Running replay within a simulator also allows us to manually modify the execution of the system, rather than playing a static recording.

Another line of work aims to prevent bugs from being introduced in the first place. Frenetic [12] presents a language-based approach to building robust SDN applications. By providing a specialized programming model, Frenetic helps developers avoid writing common classes of bugs, such as 'composition errors' where installed flow entries override each other. Reitblatt et al. [24] developed a technique for ensuring consistent routing updates, guaranteeing that all switches in the network either route a

given packet under the new configuration or under the old configuration, but not both. These abstractions are valuable for preventing common, difficult errors in platform logic.

Several other network simulators exist for testing SDN controllers. Mininet is a platform for emulating OpenFlow switches and hosts within a single VM [19]. The ns-series of network simulators provides a general framework for testing new protocols, topologies, and traffic mixes [2]. We found that these existing simulators did not provide sufficient support for the corner-cases situations which are the focus of our work, such as failures and VM migration.

Many of our ideas originate from the literature on troubleshooting general distributed systems. WiDS checker introduced the notion of recording production executions to be later replayed and verified in a controlled simulation. Pip [25] defines a DSL and collection of annotation tools to reason about causal paths throughout the execution of the distributed system. Finally, end-to-end tracing frameworks such as X-Trace [11] and Pinpoint [9] provide a framework for tracing requests throughout a distributed system in order to infer correctness errors between layers and across components. Our work solves a more constrained problem; we leverage the structure of the SDN stack to enable a simple notion of platform correctness. In addition, these systems assume that invariants should hold at all times; we observe that in an eventually-consistent system such as SDN, transient policy-violations are inevitable. We built simulation-based causal inference to help troubleshooters differentiate ephemeral from persistent errors.

7 Conclusion

SDN is widely heralded as the "future of networking", and its purpose is to make it easy to manage networks. It does so by providing control applications with a simple programmatic interface through which they can specify high-level policies about the network's behavior. While this does indeed make writing control applications simpler, it requires that the underlying SDN platform translates these high-level policies into low-level configuration of the physical switches.

This process of translation is performed by a sophisticated distributed system, comprised of multiple controllers and serving multiple tenants, which must operate at large scales and in the presence of frequent failures. Distributed systems are fundamentally difficult to build, particularly in such a nascent field such as SDN where we have such little experience, and the resulting errors are hard to understand from inspecting logs. Thus, it is important that SDN provide adequate troubleshooting tools so users can determine whether or not the platform is responsible for the problems they are seeing, and SDN vendors can find problems in their own platform code.

In this paper we described a system for troubleshooting called W^3. W^3 employs two techniques to localize the root cause of policy-violations.

- Correspondence-checking identifies which component of SDN platform is responsible for the policy-violation.
- Simulation-based causal inference identifies the minimal causal set network events that trigger the policy-violation.

We have applied this system to several available SDN platforms, and were able to find or reproduce bugs in all the platforms we investigated.

References

[1] Cisco 7000 series datasheet. `http://tinyurl.com/77jorsq`.

[2] The ns-3 network simulator. `http://www.nsnam.org/`.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. SIGCOMM '08. ACM.

[4] BigSwitch Networks. `http://www.bigswitch.com/`.

[5] A. Bouloutas, S. Calo, and A. Finkel. Alarm correlation and fault identification in communication networks. *Communications, IEEE Transactions on*, 42(234):523 –533, feb/mar/apr 1994.

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. NSDI '12.

[7] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. PRESTO '10.

[8] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst., 1985.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. Intl. Conf. on Dependable Systems and Networks, 2002.

[10] Floodlight Controller. `http://floodlight.openflowhub.org/`.

[11] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: a pervasive network tracing framework. NSDI'07.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. ICFP '11.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. SIGCOMM '09.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. CCR, 2008.

[15] V. Jacobson. Unix traceroute man page, 1987.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. NSDI '12.

[17] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. OSDI '10.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[19] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. Hotnets '10.

[20] H. Mai et al. Debugging the data plane with Anteater. SIGCOMM '11.

[21] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[22] J. Mccauley. POX: A python-based openflow controller.

[23] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PORRTLAND: a scalable fault-tolerant layer 2 data center network fabric. SIGCOMM '09.

[24] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! HotNets '11.

[25] P. Reynolds, C. Killian, J. L. Winer, J. C. Mogul, M. A. Shah, and A. Vadhat. Pip: Detecting the unexpected in distributed systems. NSDI '06.

[26] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. ATC '11.