How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software

Colin Scott[†] *Andreas Wundsam*[◇] *Sam Whitlock*[*] *Andrew Or*[†] *Eugene Huang*[†]
Jefferson Lai[†] *Kyriakos Zarifis*[‡] *Scott Shenker*[†*]
[†]*UC Berkeley* [◇]*Big Switch Networks* [*]*ICSI* [‡]*University of Southern California*

Abstract

Software bugs are inevitable in software-defined networking (SDN) control planes, and troubleshooting is a tedious, time-consuming task. In this paper we discuss how to improve control software troubleshooting by presenting a technique, retrospective causal inference, for automatically identifying a minimal sequence of inputs responsible for triggering a given bug.

We apply retrospective causal inference to four open source SDN control platforms—Floodlight, ONOS, POX, and NOX—and illustrate how our technique found minimal causal sequences for the bugs we encountered.

1 Introduction

Would World War I still have occurred if Archduke Ferdinand had not been shot? That is, was the prior event intrinsic to the historical outcome, or was it extraneous? Unfortunately we can never know such historical counterfactuals for sure.

When troubleshooting computer systems, we often need to answer similar questions, *e.g.* "Was this routing loop triggered when the controller discovered the link failure?" And unlike human history, it is often possible to retroactively find explanations for 'how we got into this mess'. In this paper we address the problem of programmatically finding such explanations in the context of software-defined networking.

Based on anecdotal evidence from colleagues and acquaintances in the industry, it seems clear that developers of software-defined networks spend much of their time troubleshooting bugs. This should be no surprise, since software developers in general spend roughly half (49% according to one study [17]) of their time troubleshooting, and spend considerable time on bugs that are difficult to trigger (the same study found that 70% of the reported concurrency bugs take days to months to fix). More fundamentally though, modern SDN control platforms are highly complex, distributing state between replicated servers [14], providing isolation and resource arbitration between multiple tenants [6], and globally optimizing network utilization [22]. Most of this complexity comes from fundamentally difficult distributed systems challenges such as asynchrony and partial failure. Even Google's Urs Höelzle attests that [22] "[coordination between replicated controllers] is going to cause some angst, and justifiably, in the industry."

The troubleshooting process, which starts by analyzing long logs gathered from either operational monitoring or QA testing (we focus on the latter), is hindered by the large number of hardware failures, policy changes, host migrations, and other inputs to SDN control software. As one data point, Microsoft Research reports 8.5 network error events per minute per datacenter [18]. Troubleshooters find little immediate use from logs containing many inputs prior to a fault, since they are forced to manually filter extraneous inputs before they can start fruitfully investigating the root cause. It is no surprise that when asked to describe their ideal tool, most network admins said "automated troubleshooting" [45].

Before continuing, we should clarify what we mean by 'troubleshooting' and 'bugs' in the SDN context. SDN networks are designed to support high-level policies, such as inter-tenant isolation (*i.e.* ACL enforcement). A bug creates situations where the network violates one or more of these high-level policies; that is, even though the control plane has been told to implement a particular policy, the resulting configuration (*i.e.* flow entries in the switches) does not do so properly. These violations could arise from a hardware failure or a policy misconfiguration, and there is active work on developing tools to detect these problems [24, 46]. However, such violations could also arise from bugs in the SDN control plane software; our focus is on trying to troubleshoot this case.

Bugs may be triggered by uncommon sequences of inputs, such as a simultaneous link failure or controller reboot. The act of *troubleshooting* involves identifying which set of inputs is most directly responsible for triggering the bug. *Debugging* is then the act of tracking down the error in the code itself, given a set of triggering inputs. The smaller the set of triggering inputs, the easier debugging will be.

Our focus here is on troubleshooting. When we observe an invalid configuration, which is prima facie evidence for a bug, our goal is to automatically filter out inputs to the SDN software that are not relevant to triggering the bug, leaving a small sequence of inputs that is directly responsible. This would go a long way towards achieving "automated troubleshooting."

If you consider a software-defined network as a distributed state machine, with individual processes sending messages between themselves, one straightforward approach is to account for potential causality: if an external input does not induce any messages before the occurrence of the invalid configuration, it cannot possibly have affected the outcome [28]. Unfortunately, pruning only those inputs without a potential causal relation to the invalid configuration does not significantly reduce the number of inputs.

Our approach is to prune inputs from the original run, replay the remaining inputs to the control software using simulated network devices, and check whether the network re-enters the invalid configuration. Our approach is identical in spirit to delta debugging [43]—an algorithm for minimizing test cases that are inserted at a single point in time to a single program—but involves a substantially different set of technical challenges since networks are distributed asynchronous systems rather than a single program.

The main difficulty in pruning historical inputs is coping with divergent histories. Traditional replay techniques [12, 16] reproduce errors reliably by precisely recording the low-level I/O operations of software under test. Pruning inputs, however, may cause the execution to subtly change (*e.g.* the sequence numbers of packets may all differ), and some state changes that occurred in the original run may not occur. Without the exact same low-level I/O operations, deterministic replay techniques cannot proceed in a sensible manner.

We deal with divergent histories by recording and replaying at the application layer, where we have access to the syntax and semantics of messages passed throughout the distributed system. In this way we can recognize functionally equivalent messages and maintain causal dependencies throughout replay despite altered histories.

Painstaking manual analysis of logs is the *de facto* method of troubleshooting production SDN control software today. As far as we know, our work is the first to programmatically isolate fault-inducing inputs to a distributed system. Record and replay techniques such as OFRewind [40] and liblog [16] allow you to manually step through the original execution and verify whether a set of inputs triggered a bug, but the original run is often so large that the the set of potentially triggering inputs verges on unmanageable. Tracing tools such as ndb [21] provide a historical view into dataplane (mis)behavior. In contrast, our technique provides information about precisely what caused the network to enter an invalid configuration in the first place.

We have applied retrospective causal inference to four open source SDN control platforms: Floodlight [4], ONOS [27], POX [29], and NOX [19]. Of the **five** bugs we encountered in a five day investigation, retrospective causal inference reduced the size of the input trace to 2 events in the best case and 18 events in the worst case.

2 Background

We begin by sketching the architecture of the SDN control plane and illustrating the correctness challenges encountered by operators and implementers.

SDN networks are managed by software running on a set of network-attached servers called 'controllers'. It is the job of the controllers to configure the network in a way that complies with the intentions of network operators. Operators codify their intentions by configuring behavioral specifications we refer to as 'policies'. Policy constraints include connectivity, access control, resource allocations, traffic engineering objectives, and middlebox processing.

For fault-tolerance, production SDN control software is typically distributed across multiple servers. For scalability, the responsibility for managing the network can be partitioned through sharding. Onix [26], for example, partitions a graph of the network state across either an eventually consistent distributed hash table or a transactional database.

In this distributed setting, controllers must coordinate amongst themselves when reacting to state changes in the network or policy changes from above. Coordination in SDN is not exempt from the well-known class of faults inherent to all distributed systems, such as inconsistent reads, race conditions over message arrivals, and unintended consequences of failover logic.

Several production SDN controllers support network virtualization [4, 10, 31], a technology that abstracts the details of the underlying physical network and presents a simplified view of the network to be configured by applications. In this model, multi-tenancy is implemented by providing each tenant with their own abstract view, which are multiplexed onto the same physical network. A common pattern is to treat an entire network as a single logical switch for each tenant. When a large network is abstracted in this way, the mapping between the logical switch and the physical topology is highly complex.

In conjunction, the challenges of maintaining virtualized and distributed objects while ensuring that critical invariants such as isolation between tenants hold at all times make SDN control software highly bug-prone.

In the next section we give a formal definition of our approach to troubleshooting. We then spend the bulk of the paper describing how we realize this formalism.

3 Problem Definition

We represent the forwarding state of the network at a particular time as a configuration c, which contains all the forwarding entries in the network as well as the liveness of the various network elements. The network control plane is a system that takes a sequence of external network events $E = e_1, e_2, \ldots, e_m$ (such as link failures) as input, and produces a sequence of network configurations $C = c_1, c_2, \ldots, c_n$.

An invariant is a predicate P over forwarding state (a safety condition, such as having no loops or blackholes). We say that a configuration c violates the invariant if $P(c)$ does not hold, denoted as $\overline{P}(c)$.

In our formalism a log L of a system execution is a triplet of external events E_L, timing information T_L of the external events, and the resulting sequence of forwarding configurations C_L. A replay of log L involves replaying the external events along with a particular timing T, which need not be identical to the original timings T_L captured in the log. We denote a replay attempt by $replay(E_L, T)$. The output of $replay$ is a sequence of forwarding state configurations $C_R = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$. In the ideal case $replay(E_L, T_L)$ reproduces the same sequence of network configurations as occurred in the original run (*i.e.* $\forall_i . c_i = \hat{c}_i$), but as we discuss later in the paper there are reasons why this does not always hold.

If the original log (E_L, T_L, C_L) violated predicate P (*i.e.* $\exists_{c_i \in C_L} . \overline{P}(c_i)$) then we say $replay(E_L, T) = C_R$ reproduces that invariant violation if $\exists_{\hat{c}_i \in C_R} . \overline{P}(\hat{c}_i)$.

The goal of our work is, when given a log (E_L, T_L, C_L) that violates a predicate P, to find a small sequence of events that reproduces that invariant violation. More formally, we define a minimal causal sequence (MCS) to be a subsequence E_M of E_L and a timing T_M such that $replay(E_M, T_M)$ reproduces the invariant violation, but for all proper subsequences E_N of E_M there is no timing T s.t. $replay(E_N, T)$ reproduces the violation. That is, an MCS is a sequence and timing of external events that reproduces the violation, where one cannot find a subsequence of the MCS that reproduces the violation. Note that an MCS is not necessarily *globally* minimal, in that there could be smaller sequences that reproduce this violation, but are not subsequence of this MCS.

Given a log (E_L, T_L, C_L) exhibiting an invariant violation, our goal is to identify its MCS. We could do this through brute force, trying all subsequence and timings, but this is clearly impractical given the large size of logs and the infinitely large set of possible timings. Thus, we must approximate MCSes by finding good heuristics for which events to eliminate and, more importantly, which timings to test.

In the next section we describe our heuristics for finding MCSes, and then describe the system we built in §5 to implement these heuristics. The key component of this system is a network simulator that can execute the $replay()$ function. Because we do not have access to operational logs , all of our focus is on discovering bugs in control software using this simulator and then finding the relevant MCSes that trigger these bugs.

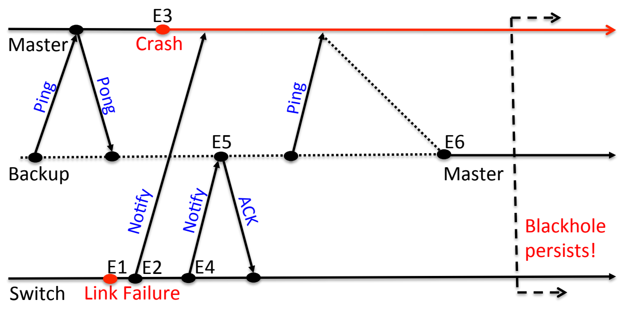

Figure 1: Floodlight failover bug. External inputs are depicted as red dots, internal events are depicted as black dots, and the dotted message line depicts a timeout.

4 Retrospective Causal Inference

Given a log (E_L, T_L, C_L) exhibiting an invariant violation, our goal is to identify its MCS. Achieving this goal involves two tasks: searching for subsequences of E_L, and searching for replay timings for those subsequences.

We illustrate the mechanics of our approach, retrospective causal inference, by describing an example bug in the Floodlight open source control platform [15]. Floodlight is distributed across multiple controllers for high availability, and provides support for virtualization. Switches maintain one hot connection to a master controller and several cold connections to replica controllers. The *master* holds the authority to modify the configuration of switches, while the other controllers are in *backup* mode and do not perform any changes to the switch configurations unless they detect that the master has crashed.

The failover logic in Floodlight is not implemented correctly, leading to the following race condition[1] depicted in Figure 1: a link fails (E1), and the switch attempts to notify the controllers (E2,E4) shortly after the

[1]Note that this issue was originally documented by the developers of Floodlight [15].

master controller has died (E3), but before a new master has been selected (E6). In this case, all live controllers are in the backup role and will not take responsibility for updating the switch flow table (E5). At some point, a backup notices the master failure and elevates itself to the master role (E6). The new master will proceed to manage the switch, but without ever clearing the routing entries for the failed link (resulting in a persistent blackhole). In this example, the MCS is the conjunction of the two external inputs (E1,E3) in conjunction.

A developer or operator encountering this bug in practice would not be given this concise version of events. Instead, the trace would contain a wealth of extraneous inputs, making it difficult to reason about the underlying root cause. In the worst scenario, operators may need to examine logs from a production network, which contain a substantial number of hardware failures, topology changes, and other potential triggering events, all of which may appear characteristic of normal operating conditions at first glance; assuming 8.5 network error events per minute [18], and 500 VM migrations per hour [36], there would be at least $8.5 \cdot 60 + 500 \approx 1000$ inputs reflected in an hour-long trace.

4.1 Searching for Subsequences

Checking random subsequences of E_L would be one viable but inefficient approach to achieving our first task. We can do better by leveraging a divide-and-conquer search technique from the software engineering community: the delta debugging algorithm [43] provides a provably correct means to isolating fault-inducing inputs. In our case, we use delta debugging to iteratively select subsequences of E_L, replay each subsequence with some timing T, and check whether the bug persists. If the bug persists, delta debugging ignores the other inputs, and proceeds with the search for an MCS within this subsequence. In what follows, we use the term *delta-debugging* to refer to our algorithm for finding relevant subsequences. The delta debugging algorithm is shown in Figure 2 (with '*test*' replaced by '*replay*').

It is worth noting that the input subsequences chosen by delta debugging are not always valid. For example, it is not sensible to replay a recovery event without a preceding failure event; nor is it sensible to replay a host migration event without modifying its starting position when a preceding host migration event has been pruned. The simulator checks validity before replaying a given subsequence to account for this possibility. [2] Currently our simulator accounts for validity of all network state change events (shown in Table 2), but does not support policy changes, which have more complex semantics.

4.2 Searching for Timings

Simply exploring subsequences of E_L does not suffice to find minimal causal sequences: the timing of those subsequences throughout invocations of $replay()$ is crucial for reliably reproducing the invariant violation. The most natural approach would be to maintain the original timings, *i.e.* if we define t_i as the timestamp of the i^{th} input from the original run and t'_i as the replay clock value when it injects that same input (which may or may not be the i'th input in the subsequence), then we might just set $t'_i = t_i$.

Unfortunately, this approach does not work in practice because it can violate causal dependencies. The crucial point here is that when we replay only a subsequence of the original inputs, the reaction of the control software can change, such that it behaves differently or takes a different amount of time to respond to the remaining inputs events. Simply maintaining relative timings might result in injecting the remaining inputs too early or late.

Formally, to reliably reproduce the original invariant violation we need to inject an external input e only after all other events (both external and internal) that precede it in the happens-before [28] relation ($\{i \mid i \rightarrow e\}$) from the original execution have occurred [37]. Internal events might include messages sent between the controllers and switches, or state transitions within a single controller. We can obtain the happens-before relation in our logs by having our simulator (to be described later) interpose on all message channels and log a message only after all other logged messages have been delivered.

4.3 Preserving Causality

In practice it is non-trivial to reason about causal dependencies when transforming the original log (which reproduces the violation) to a subsequence of external events (which may or may not reproduce that violation). To make progress we draw inferences about the remaining causal dependencies in the subsequence based on the causal dependencies reflected in the original log. This involves three issues: coping with syntactic differences in internal events across runs, handling internal events from the original execution that may not occur after pruning, and dealing with new internal events that were not observed at all in the original execution.

Functional Equivalence: The first issue is that internal events may differ syntactically (*e.g.* sequence num-

[2] Handling invalid inputs is crucial for ensuring that the delta debugging algorithm finds a minimal causal subsequence. The algorithm we employ [43] makes three assumptions about inputs: monotonicity, unambiguity, and consistency. Violating the first two only leads to slightly inflated MCSes. Zeller wrote a follow-on paper [44] that removes the need for the consistency assumption, but incurs an additional factor of N in complexity in doing so.

Figure 2: Automated Delta Debugging Algorithm From [43]

Input: $T_{\text{✘}}$ s.t. $T_{\text{✘}}$ is a trace and $replay(T_{\text{✘}}) = \text{✘}$. Output: $T'_{\text{✘}} = ddmin(T_{\text{✘}})$ s.t. $T'_{\text{✘}} \subseteq T_{\text{✘}}$, $replay(T'_{\text{✘}}) = \text{✘}$, and $T'_{\text{✘}}$ is minimal.

$$ddmin(T_{\text{✘}}) = ddmin_2(T_{\text{✘}}, \emptyset) \quad \text{where}$$

$$ddmin_2(T'_{\text{✘}}, R) = \begin{cases} T'_{\text{✘}} & \text{if } |T'_{\text{✘}}| = 1 \text{ (“base case”)} \\ ddmin_2\big(T_1, R\big) & \text{else if } replay(T_1 \cup R) = \text{✘} \text{ (“in } T_1\text{”)} \\ ddmin_2\big(T_2, R\big) & \text{else if } replay(T_2 \cup R) = \text{✘} \text{ (“in } T_2\text{”)} \\ ddmin_2\big(T_1, T_2 \cup R\big) \cup ddmin_2\big(T_2, T_1 \cup R\big) & \text{otherwise (“interference”)} \end{cases}$$

where $replay(T)$ denotes the state of the system after executing the trace T, ✘ denotes a correctness violation, $T_1 \subset T'_{\text{✘}}$, $T_2 \subset T'_{\text{✘}}$, $T_1 \cup T_2 = T'_{\text{✘}}$, $T_1 \cap T_2 = \emptyset$, and $|T_1| \approx |T_2| \approx |T'_{\text{✘}}|/2$ hold.

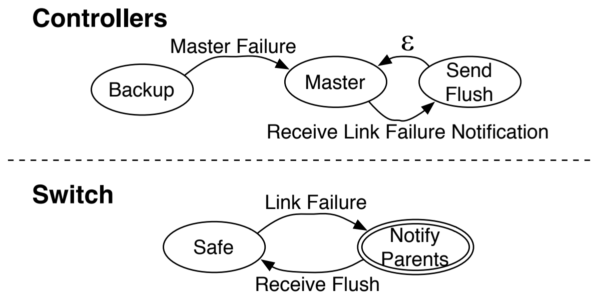

Figure 3: Simplified state machines for the switch and controllers in the example Floodlight bug. Double outlined states represent presence of the blackhole.

Internal message	Masked values
OpenFlow headers	transaction id
OpenFlow FLOW_MODs	cookie, buffer id
Log statements	varargs parameters to printf

Table 1: Example internal messages and their masked values. The masks serve to define equivalence classes.

bers of control packets may all differ) when replaying a subsequence of the original log. We observe that many internal events are *functionally equivalent*, in the sense that they have the same effect on the state of the system with respect to triggering the invariant violation (despite syntactic differences). For example, flow modification messages may cause switches to make the same change to their forwarding behavior even if the transaction identifier of the messages differ.

We leverage this observation by defining domain-specific masks over semantically extraneous fields of internal events.[3] We show four examples of masked values in Table 1. We consider an internal event i' observed in the subsequence log equivalent (in the sense of inheriting all of its happens-before relations) to an internal event i from the original log iff all unmasked fields have the same value and i occurs between i''s preceding and succeeding inputs in the happens-before relation.

Handling Absent Internal Events: Some internal events from the original log which causally "happen before" some external event may be absent when replaying a subsequence of that log. The structure of the control software's state machine (which we do not assume to know) determines whether internal events disappear. Consider the simplified state machines for the switch and controllers from the Floodlight case shown in Figure 3. If we prune the link failure input, the master will never receive a link failure notification and transition to and from 'Send Flush'.

We handle this possibility by waiting for expected equivalent internal events, but timing out and proceeding if they do not occur within a certain time ε. If the event scheduling algorithm detects that it has waited too long (such that a happens-before successor has occurred before a predecessor), it replays the log from the beginning up until the immediately prior input, this time knowing which internal events in the current input interval are and are not going to occur before injecting the next input. We show the overall event scheduling algorithm in Figure 4.

Handling New Internal Events: The last possible change induced by input pruning is the occurrence of new internal events that were not observed in the original log. New events ultimately leave open multiple possibilities for where we should inject the next input. Consider the following case: if i_2 and i_3 are internal events observed during replay that are both in the same equivalence class as a single event i_1 from the original run, we could inject the next input after i_2 or after i_3.

In the general case it is always possible to construct two state machines that lead to differing outcomes: one

[3]One consequence of applying masks is that bugs involving masked fields are outside the purview of our approach.

```
procedure REPLAY(subsequence)
  for $e_i$ in subsequence
    if $e_i$ is an internal event
    and $e_i$ is not marked absent :
      then
        $\Delta \leftarrow |e_i.time - e_{i-1}.time| + \varepsilon$
        wait up to $\Delta$ seconds for $e_i$
        if $e_i$ did not occur :
          then mark $e_i$ as absent
    else if $e_i$ is an input :
      then
        if a successor of $e_i$ occurred :
          comment: waited too long
          then return REPLAY(subsequence)
        else inject $e_i$
```

Figure 4: `Replay` is responsible for replaying subsequences of events chosen by delta debugging and determining if the bug reappears.

that only leads to the invariant violation when we inject the next input before a new internal event, and one that only leads to the invariant violation when we inject the next input after a new internal event. In other words, to be guaranteed to traverse any existing suffix that leads to the invariant violation, it is necessary to recursively branch, trying both possibilities for every new internal event. This implies an exponential number of possibilities to be explored in the worst case.

Exponential search over these possibilities is not a practical option. Our heuristic when waiting for expected internal events is to proceed normally if there are intermediate new internal events, always injecting the next input when its last expected predecessor either occurs or times out. This ensures that we always find suffixes that contain only a subset of the (equivalent) original internal events, but leaves open the possibility of finding divergent suffixes that still lead to the invariant violation.

4.4 Complexity

The delta debugging algorithm terminates after $O(log\,n)$ invocations of *replay* in the best case, where n is the number of inputs in the original trace [43]. In the worst case, delta debugging has $O(n)$ complexity.

If the replay algorithm never needs to back up, it replays n inputs, for an overall runtime of $O(nlog\,n)$ replayed inputs in the best case, and $O(n^2)$ in the worst case. Conversely, if the event scheduling needs to back up in every iteration, another factor of n is added to the runtime: for each input event e_i, it replays inputs $e_1, \ldots, e_i$, for a total of $n \times \frac{n+1}{2} \in O(n^2)$ replayed inputs. In terms of replayed inputs, the overall worst case is therefore $O(n^3)$.

Link failure	Link recovery
Switch failure	Switch recovery
Control server failure	Control server recovery
Dataplane packet injection	Dataplane packet drop
Dataplane packet delay	Dataplane packet permit
Control message delay	Host migration

Table 2: Input types supported by STS

The runtime can be decreased by observing that delta debugging is readily parallizable. Specifically, the worst case runtime could be decreased to $O(n^2)$ by enumerating all subsequences that delta debugging can possibly examine (of which there are $O(n)$), replaying them in parallel, and joining the results.

The runtime can be further decreased by taking snapshots of the controller state at regular intervals. When the replay algorithm detects that it has waited too long, it could then restart from a recent snapshot rather than replaying the entire prefix.

In the next section we describe some of the practical challenges we have overcome to realize our approach. In our initial experiments we have found that applying delta debugging to explore subsequences of E_L and striving to maintain a single timing T that maintains causal dependencies reliably finds small MCSes.

5 Systems Challenges

The core piece of infrastructure we need to realize our approach is a mechanism to replay execution logs. Building such a replay mechanism comes with many challenges; unlike the example applications described by the delta debugging paper [43], the system we are troubleshooting is not a single program–it encompasses all the nodes and links of a distributed system, including controllers, switches, and end-hosts, where asynchrony makes it highly difficult to reliably replay inputs.

Our approach is to simulate the control-plane behavior of network devices (with support for minimal data-plane behavior) on a single machine. We then run the control software on top of this simulator and connect the software switches to the controllers as if they were true network devices, such that the controllers believe they are configuring a true network. This setup allows the simulator to interpose on all communication channels and delay, drop, or reorder messages as needed to induce failure modes during input generation, or to maintain causal dependencies during replay. The overall simulation architecture is depicted in Figure 5.

We begin by using our simulator to perform testing on controllers to find bugs. Our most common use case in-

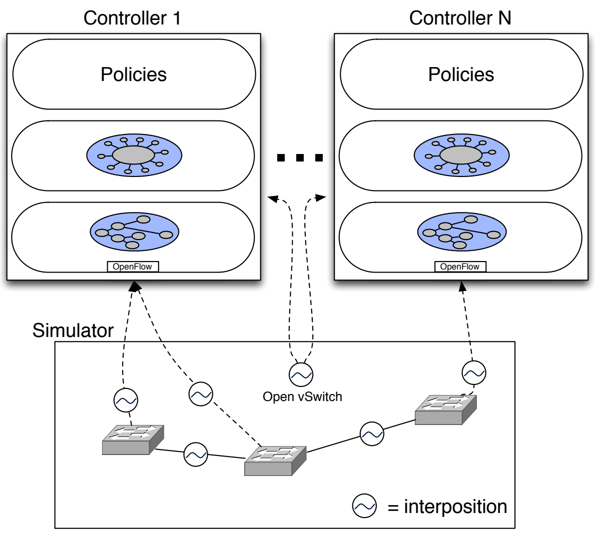

Figure 5: Simulation infrastructure. We simulate network devices in software, and interpose on all communication channels.

volves generating randomly chosen input sequences [30], feeding them to controller(s), and monitoring invariants [24] at chosen intervals. We also run the simulator interactively so that we can examine the state of any part of the simulated network, observe and manipulate messages, and follow our intuition to induce orderings that we believe may trigger bugs. Either way, controlling the inputs from a single location allows the simulator to easily record a global, causally-consistent event ordering.

After discovering an invariant violation of interest, the simulator replays the logged sequence of inputs (such as link failures, controller crashes, or host migrations). For example, the simulator replays link failures by disconnecting the edge in the simulated network, and sending a OpenFlow [32] port status message from the adjacent switches to their parent controller(s).

STS (the SDN Troubleshooting Simulator) is our realization of this simulator. STS is implemented in roughly 18,000 lines of Python in addition to the Hassel network invariant checking library [24]. STS also makes use of Open vSwitch [33] as an interposition point for messages sent between distributed controllers. We have made the code for STS publicly available at ucb-sts.github.com/sts, and have discussed the logistics of deploying it with several SDN companies.

5.1 Mitigating Non-Determinism

We designed STS to be as resilient to non-determinism as is practically feasible, while avoiding modifications to control software whenever possible. Routing the `gettimeofday()` syscall through STS makes replay more resilient to alterations in execution speeds.[4] When sending data over multiple sockets, the operating system exhibits non-determinism in the order it schedules the I/O operations. STS optionally ensures a deterministic order of messages by multiplexing all sockets onto a single true socket. STS currently overrides socket functionality within the control software itself.[5]

STS needs visibility into the control software's internal state changes to reliably reproduce the system execution. We achieve this by making a small change to the control software's logging library[6]: whenever a control process executes a log statement (which indicate that an important state change is about to take place), we notify STS, and optionally block the process. STS then sends an acknowledgment to unblock the controller after taking note of the state change.

5.2 Coping with Non-Determinism

Our current implementation does not account for other sources of non-determinism, such as random number generators, asynchronous signals, or interruptable instructions (*e.g.* x86's block memory instructions [12]). And even if these sources were eliminated, it would not be possible to achieve perfectly deterministic replay in all cases without full visibility into internal events–a daunting instrumentation task.

Fortunately we can cope with cases of excessive non-determinism by replaying each subsequence chosen by delta debugging multiple times. If observing the bug is an independent and identically distributed random variable (an optimistic but not completely unreasonable assumption) with some underlying probability p, then n replays will observe the bug with probability $1-(1-p)^n$. The exponential works strongly in our favor; for example, even if the original bug is triggered in only 20% of replays, the probability that we will not trigger it during an intermediate replay is approximately 1% if we replay 20 times per subsequence.

5.3 Enabling Analysis of Production Logs

STS does not currently support minimization of production logs. Here we present a sketch of how STS might support production logs as input.

While retrospective causal inference takes as input a single, totally-ordered log of the events in the dis-

[4]When the pruned trace differs from the original, we make a best-effort guess at what the return values of these calls should be. For example, if the altered execution invokes `gettimeofday()` more times than we recorded in the initial run, we interpolate the time values of neighboring events

[5]Only supported for POX at the moment.

[6]Only supported for POX and Floodlight at the moment.

tributed system, production systems maintain a log at each node. Production systems would need to include Lamport clocks on each message [28] or have sufficiently accurate clock synchronization [11] to obtain a partial global ordering consistent with the happens-before relation.[7] Inputs would also need to need to be logged in sufficient detail for STS to replay a synthetic version of the input that is indistinguishable (in terms of control plane messages) from the original.

Without care, a single input event may appear multiple times in the distributed logs. A failure of the master node, for example, could be independently detected and logged by all other replicas. The most robust way to avoid redundant input events would be to employ perfect failure detectors [7], which log a failure iff the failure actually occurred.[8] Alternatively, one could employ root cause analysis algorithms [41] or manual inspection to consolidate redundant alarms.

Finally, some care would be needed to prevent the logs from growing so large that retrospective causal inference's runtime becomes intractable. Here, causally consistent snapshots [8] would allow STS to bootstrap its simulation from the last snapshot before the failure rather than replaying from the beginning of the log.

5.4 Limitations

Having detailed the specifics of our approach we now clarify the scope of our technique's use.

Partial Visibility. Our event scheduling algorithm assumes that it has visibility into the occurrence of all relevant internal events. This may involve substantial instrumentation effort beyond pre-existing log statements.

Non-determinism Within Individual Controllers. Our technique is not designed to reproduce bugs involving non-determinism within a single controller (*e.g.* race-conditions between threads); we focus on coarser granularity errors (*e.g.* incorrect failover logic). Nonetheless, the worst case for us is that the developer ends up with what they started: an unpruned log.

Bugs Outside the Control Software. Our goal is not to find the root cause of individual component failures in the system (*e.g.* misbehaving routers, link failures). Instead, we focus on how the distributed system as a whole reacts to the occurrence of such inputs.

Globally vs. Locally Minimal Input Sequences. Our approach is not guaranteed to find the globally minimal causal sequence from an input trace, since this enumerating the powerset of E_L in the worst case (a $O(2^n)$ operation). The delta debugging algorithm we employ does provably find a locally minimal causal sequence [43], meaning that if any input from the sequence is pruned, no invariant violation occurs.

Correctness vs. Performance. We are primarily focused on correctness bugs, not performance bugs.

Bugs Found Through Fuzzing. We generate bug traces primarily through fuzz testing, not from real bugs found in operation. There is a substantial practical hurdle in instrumenting operational systems to produce logs that can be injected into our system.

Scaling. Our discussions with companies with large SDN deployments suggest that scaling to the size of the large logs they collect will be a substantial challenge. On the other hand, the fact that these logs are so large makes the need for finding MCSes even more acute.

6 Evaluation

We have applied STS to three open source and one proprietary SDN control platform: Floodlight [4], ONOS [27], POX [29], and NOX [19]. Over a span of roughly five days of investigation we found a total of five bugs. We show a high-level overview of our results in Table 3, and illustrate in detail how retrospective causal inference found their minimal causal sequences in the rest of this section. Interactive visualizations and replayable event traces for all of these case studies are available at ucb-sts.github.com/experiments.

6.1 POX List Removal

The first SDN control platform we examined was POX, the successor of NOX. POX is a single-machine control platform intended primarily for research prototyping and educational use (*i.e.* not large scale production use). Nevertheless, POX has been deployed on real networks, and has a growing set of users.

The POX application we ran was a layer two routing module ('l2_multi') that learns host locations and installs exact match per-flow paths between known hosts using a variant of the Floyd-Warshall algorithm. It depends on a discovery module, which sends LLDP packets to discovery links in the network, and a spanning tree module, which configures switches to only flood packets for unknown hosts along a spanning tree.

We start with a relatively trivial bug to illustrate that STS is useful for early stage development and testing. We employed STS to generate random sequences of inputs, and found after some time that POX threw an exception due to attempting to remove an element from a list where the element was not present.

[7] Note that a total ordering is not needed, since it is permissible for retrospective causal inference to reorder concurrent events from the production run so long as the happens-before relation is maintained [13].

[8] Perfect failure detectors can be implemented in partially synchronous distributed systems by explicitly killing nodes that are suspected to be down.

Bug Name	Topology	Replay Success Rate	Total Inputs	MCS Size
POX list removal	2 switch mesh	20/20	69	2
POX in-flight blackhole	2 switch mesh	15/20 [20/20*]	26	11
POX migration blackhole	4 switch mesh	20/20*	29	3
NOX discovery loop	4 switch mesh	18/20	150	18
Floodlight loop	3 switch mesh	15/50	284	36

Table 3: Overview of Case Studies.
*with multiplexed sockets, overridden `gettimeofday()`, and logging interposition enabled.

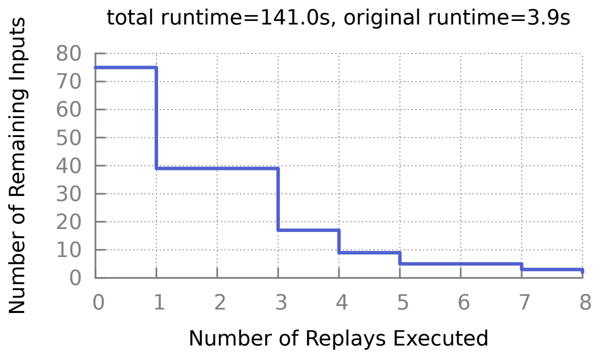

Figure 6: Minimizing the POX list remove trace.

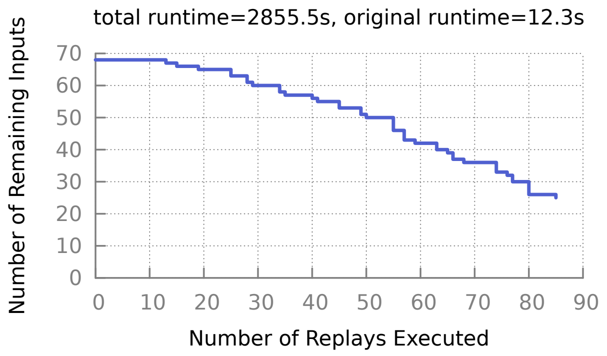

Figure 7: Minimizing the POX in-flight blackhole.

There were 69 randomly generated inputs in the trace leading up to the exception. We invoked retrospective causal inference to identify a two element MCS: a failure of a connected switch followed by a reboot/initialization of the same switch. The nearly logarithmic runtime behavior of retrospective causal inference for this case is shown in Figure 6.

Apparently the developers of POX had not anticipated this particular event sequence. Given the rarity of switch recovery events, and the tediousness of writing unit tests for scenarios such as this (which involved multiple OpenFlow initialization handshakes), this is not surprising. STS made it straightforward to inject inputs at a high semantic level, and the minimized event trace it produced made for a simple integration test.

6.2 POX In-flight Blackhole

We discovered the next bug after roughly 20 runs of randomly generated inputs. We noticed that STS reported a persistent blackhole while POX was bootstrapping its discovery of link and host locations. There were 29 inputs in the initial trace, and retrospective causal inference returned a 11 input MCS (runtime shown in Figure 7).

We provided the MCS to the lead developer of POX. Primarily using the console output, we were able to trace through the code and identify the problem within 7 minutes, and were able to find a fix for the race condition within 40 minutes. By matching the console output with the code, he found that the crucial triggering events were two in-flight packets (set in motion by prior traffic injection events): POX first incorrectly learned a host location as a result of the first in-flight packet showing up immediately after POX discovered that that port belonged to a switch-switch link – apparently the code had not accounted for the possibility of in-flight packets directly following link discovery – and then as a result the second in-flight packet POX failed to return out of a nested conditional that would have otherwise prevented the blackholed routing entries from being installed.

6.3 POX Migration Blackhole

Having examined the POX code in some depth, we noticed that there might be some interesting corner cases related to host migrations. We set up randomly generated inputs, included host migrations this time, and checked for blackholes. Our initial input size was 29 inputs. Before investigating the bug we ran retrospective causal inference, and ended up with a 3 input MCS (shown in Figure 8): a packet injection from a host A, followed by a packet injection by a host B towards A, followed by a host migration of host A. This made it immediately clear what the problem was. After learning the location of A and installing a flow from B to A, the routing entries in

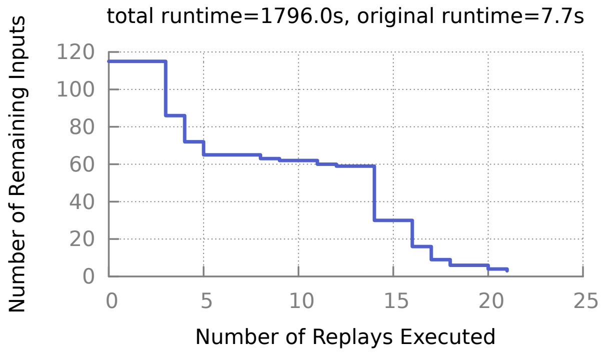

Figure 8: Minimizing the POX migration blackhole.

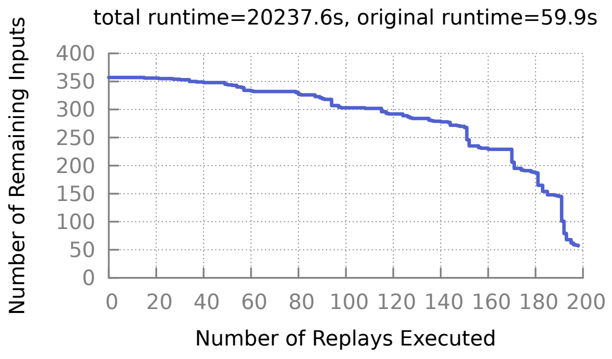

Figure 9: Minimizing the NOX discovery loop.

the path were never removed after A migrated, causing all traffic from B to A to blackhole until the routing entries expired. We did not know it at the time, but this was a known problem, and this particular routing module did not support host migrations. Nonetheless, this case demonstrates how the MCS alone can point to the root cause.

6.4 NOX Discovery Loop

We tested NOX on a four-node mesh, and discovered a routing loop between three switches within roughly 20 runs of randomly generated inputs.

Our initial input size was 68 inputs, and STS returned an 18 input MCS.[9] Our approach to debugging was to reconstruct from the MCS how NOX should have installed routes, then compare how NOX actually installed routes.

The order in which NOX discovered links was crucial: at the point NOX installed the 3-loop, it had only discovered one link towards the destination. Therefore all other switches routed through the one known neighbor switch. This comprised 2 of the 3 links involved in the link.

The destination host only sent one packet, which caused NOX to initially learn its correct location. After NOX flooded the packet though, it became confused about its location. One flooded packet arrived at another switch that was currently not known to be attached to anything, so NOX incorrectly concluded that the host had migrated. Other flooded packets were dropped as a result of link failures in the network and randomly generated network loss. The loop was then installed when the source injected another packet.

This case took us roughly 10 hours to debug. We are confident that without the minimized trace, it would have taken much longer to trace through the subtle sequence of events that were necessary for setting up the network in precisely the right conditions.

6.5 Floodlight discovery loop.

We applied STS to Floodlight's routing application. In about 30 minutes, our fuzzing uncovered a 117 input sequence that caused a persistent 3-node forwarding loop. STS reduced the sequence to 13 input events in 324 replays and 8.5 hours.

We repeatedly replayed the 13 event MCS, while successively adding instrumentation and increasing the log level each run. After about 15 replay attempts, we found that the problem was caused by interference of end-host traffic with ongoing link discovery packets. In our experiment, Floodlight had not discovered an inter-switch link due to dropped LLDP packets, causing an end-host to flap between perceived attachment points.

While this behavior cannot strictly be considered a bug in Floodlight, the case-study nevertheless highlights the benefit of STS over traditional fuzzing techniques: by leveraging repeated replays of a significantly reduced MCS, we were able to diagnose the root cause–a complex interaction between the LinkDiscovery, Forwarding, and DeviceManager modules. We subjected the current (unmodified) open source version of Floodlight (git commit f37046a) to fuzz testing with a three node fully meshed topology and high failure event rates. In an hour-long experiment, the fuzzer found an event sequence with 284 total events that results in a 3-node forwarding loop.

Overall Results. The overall results of our case studies are shown in Table 3. We show the initial input size and MCS input size in the last two columns. For the Replay Success Rate column we repeatedly replayed the original unpruned event trace, and measured how often we were able to reproduce the policy violation. There was indeed non-determinism in some cases, especially Floodlight. For the specific case of POX in-flight

[9]We had difficulty replaying the final MCS. The reason seems clear: this bug depends on the order NOX discovers links, which in turns depends on hashes computed from random memory addresses its discovery modules LLDP selection algorithm!

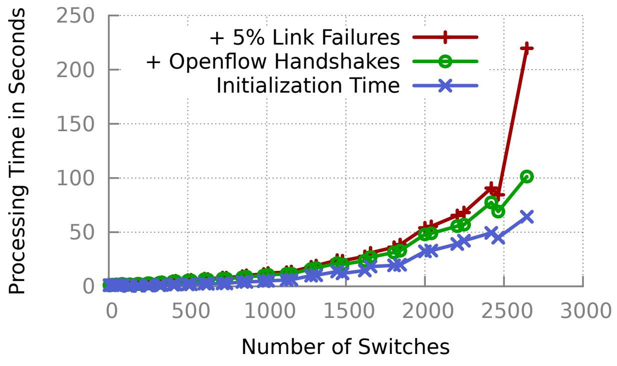

Figure 10: Simulation time for bootstrapping FatTree networks, cutting 5% of links, and processing the controller's response.

blackhole, we were able to eliminate the relevant nondeterminism by employing multiplexed sockets, overriding `gettimeofday()`, and waiting on POX's logging messages. We expect that we would see similar improvements if we applied these techniques to Floodlight.

We measured the runtime of retrospective causal inference for these case studies in Figures 6&8 − −11. While some instances ran in logarithmic time, the worst case was minimizing NOX discovery loop, which took more than 5.5 hours. Nonetheless, we conjecture that even long iteration sizes are often preferable to spending software developer's time on manual diagnosis.

6.6 Simulator Scalability

Simulating networks in a single process potentially prevents STS from triggering bugs that only appear at large scale. We ran STS on large FatTree networks to see where these scaling limits lie. On a machine with 6GB of memory, we booted POX as the controller, and measured the time to create successively larger FatTree topologies, complete the OpenFlow handshakes for each switch, cut 5% of links, and process POX's response to the link failures. As shown in Figure 10, STS's processing time scales roughly linearly up to 2464 switches (a 45-pod FatTree). At that point, the machine started thrashing, but this limitation could easily be removed by running an a machine with more memory.

Note that STS is not designed for simulating high-throughput dataplane traffic; we only forward what is necessary to exercise the controller software. In proactive SDN setups, dataplane events are not relevant for the control software.

6.7 Parameters

Our algorithm leaves an open question as to what value ε should be set to. We experimentally varied ε on the POX in-flight blackhole and the POX list removal bugs. We found for both cases that the number of events we timed out on while isolating the MCS became stable for values above 25 milliseconds. For smaller values, the number of timed out events increased rapidly. We currently set ε to 100 milliseconds.

In general, larger values of ε are preferable to smaller values (disregarding runtime considerations), since we can always detect when we have waited too long (*viz.* when a successor of the next input has occurred), but we cannot detect when we have timed out early on an internal event that is in fact going to occur. Analyzing event frequencies for particular bugs could provide more ideal ε values.

7 Discussion

Or evaluation of retrospective causal inference leaves open several questions, which we discuss here.

Aren't SDN controllers relatively simple and bug free? It is true that the freely available SDN applications we investigated are relatively simple. However, they are most definitely not bug-free since, in a short period of time, we were able to demonstrate bugs in all of them. Production SDN platforms are far more complex than the freely available ones, for a variety of reasons. Larger deployments cannot be managed with reactive microflows, and thus require more complex proactive or hybrid strategies. For fault tolerance, controllers are replicated on multiple physical servers, and sharded for scalability [26]. Multi-tenant virtualization critically requires tenant isolation to be preserved at all times. SDN controller platforms interact with cloud orchestration platforms and must correctly react to concurrent changes on their north- and southbound interfaces. Thus, we expect that these production SDN platforms will continue to be under active development, and have ongoing issues with bugs, for years to come. From our conversations with several SDN controller vendors, we are aware that they all invest significant resources to troubleshooting. Several commercial players have voiced interest in our tool as a way to improve their troubleshooting.

Aren't the bugs described here trivial in nature? Yes, the bugs we found were trivial, but that is evidence that without better troubleshooting tools tracking down even trivial bugs is difficult. We were particularly surprised how quickly our tool was able to identify policy violations in the standard routing modules of *all* investigated platforms, because we assumed that the routing modules would have been well-tested through years of use.

However, these bugs remained undiagnosed because they arise from unexpected interactions between different elements in the control plane (e.g., shortest path routing and topology discovery). We expect more complex bugs to surface once we aim our tool at more complex platforms, such as those used in production settings.

Are simulated failures really indistinguishable from actual failures? There will always be some failure modes observed in practice that are not reproducible without adding significant complexity to the simulator. Our approach is not particularly well-suited to model fine-grained low-level behavior, especially on the data plane, e.g., when switches are dropping packets due to memory or slow path constraints. Conversely, our approach excels at investigating corner cases in distributed control plane interactions, which are the source of many complex bugs. That said, as the system is entirely built in software, it is in principle possible to add more fine grained low-level behavior simulation at the cost of performance (with logical clock speeds adjusted accordingly). Additional experience with production systems will help us determine how to best trade off improved simulation fidelity against degraded performance.

Will this approach work on all control platforms? We make limited assumptions about the controller platform in use. Two of the three investigated controller platforms were exercised with retrospective causal inference without any modifications. Limited changes to the controller platforms (e.g., the possibility to override `gettimeofday()`) can increase replay accuracy further. In general, we expect retrospective causal inference to support controllers conforming to OpenFlow 1.0 out of the box.

Why do you focus on SDN networks? SDN represents both an opportunity and a challenge. In terms of a challenge, SDN control platforms are in their infancy, which means that they have bugs that need to be found and corrected. Based on our conversations with commercial SDN developers, we are confident there is a real need for improved troubleshooting in this field.

In terms of an opportunity, SDN control platforms have two properties that make them particularly amenable to an automated troubleshooting approach such as ours. First, and most importantly, SDN control software is designed to quickly converge to quiescence—that is, SDN controllers become idle when no policy or topology changes occur for a period of time.[10] This means that most inputs are not relevant to triggering a given bug, since the system repeatedly returns to a valid quiescent state; often there is only one critical transition from the last quiescent valid configuration to the first invalid configuration. If this were not the case, it is not clear that the minimal causal subsequences found by our technique would be small, in which case our approach would not yield significant advantages.

[10]Complex bugs may occur when several such processes overlay.

Second, SDN's architecture facilitates the implementation of STS. The syntax and semantics of interfaces between components of the system (*e.g.* OpenFlow between controllers and switches [32], or OpenStack Quantum's API between the control application and the network hypervisor [2]), are open and well-defined–a property that is crucial for codifying functional equivalencies. Moreover, controllers are small in number compared to the size of overall network, which makes it much easier to superimpose on messages.

In future work we hope to measure the effectiveness of our technique on other control plane systems such as NAS controllers that share the same properties.

8 Related Work

Our work spans three fields: software engineering, programming languages, and systems and networking.

Software Engineering & Programming Languages Sherlog [42] takes on-site logs from a single program that ended in a failure as input, and applies static analysis to infer the program execution (both code paths and data values) that lead up to the failure. The authors of delta debugging applied their technique to multithreaded (single-core) programs to identify the minimum set of thread switches from a thread schedule (a single input file) that reproduces a race condition [9]. Chronus presents a simpler search algorithm than delta debugging that is specific to configuration debugging [39]. All of these techniques focus on troubleshooting single, non-distributed systems.

Rx [34] is a technique for improving availability: upon encountering a crash, it starts from a previous checkpoint, fuzzes the environment (*e.g.* random number generator seeds) to avoid triggering the same bug, and restarts the program. Our approach perturbs the inputs rather than the environment prior to a failure.

Model checkers such as Mace [25] and NICE [5] enumerate all possible code paths taken by control software (NOX) and identify concrete inputs that cause the system to enter invalid configurations. Model checking works well for small control programs and a small number of machines, but suffers from exponential state explosion when run on large systems. For example, NICE took 30 hours to model check a network with two switches, two hosts, the MAC-learning control program (98 LoC), and five concurrent messages between the hosts [5]. Rather than exploring all possibilities, we discover bugs through testing and systematically enumerate subsequences of their event traces in polynomial time.

Systems and Networking We share the common goal of improving troubleshooting of software-defined networks with OFRewind [40] and recent project ndb [21]. OFRewind provides record and replay of OpenFlow control channels, and allows humans to manually step through and filter input traces. We focus on testing corner cases and automatically isolating minimal input traces.

ndb provides a trace view into the OpenFlow forwarding tables encountered by historical and current packets in the network. This approach is well suited for troubleshooting hardware problems, where the network configuration is correct but the forwarding behavior is not. In contrast, we focus on bugs in control software; our technique automatically identifies the control plane decisions that installed erroneous routing entries.

Neither ndb nor OFRewind address the problem of diagnostic information overload: with millions of packets on the wire, it can be challenging to pick just the right subset to interactively debug. To the best of our knowledge, retrospective causal inference is the first system that programmatically provides information about precisely what caused the network to enter an invalid configuration in the first place.

Trace analysis frameworks such as Pip [35] allow developers to programmatically check whether their expectations about the structure of recorded causal traces hold. MagPie [3] automatically identifies anomalous traces, as well as unlikely transitions within anomalous traces by constructing a probabilistic state machine from a large collection of traces and identifying low probability paths. Our approach identifies the exact minimal causal set of inputs without depending on probabilistic models.

Network simulators such as Mininet [20], ns-3 [1], and ModelNet [38] are used to prototype and test network software. Our focus on comparing diverged histories requires us to provide precise replay of event sequences, which is in tension with the performance fidelity goals of pre-existing simulators.

Root cause analysis [41] and dependency inference [23] techniques seek to identify the minimum set of failed components (*e.g.* link failures) needed to explain a collection of alarms. Rather than focusing on individual component failures, we seek to minimize inputs that affect the behavior of the overall distributed system.

9 Conclusion

SDN is widely heralded as the "future of networking", because it makes it much easier for operators to manage their networks. SDN does this, however, by pushing complexity into SDN control software itself. Just as sophisticated compilers are hard to write, but make programming easy, SDN platforms make network management easier for operators, but only by forcing the developers of SDN platforms to confront the challenges of asynchrony, partial failure, and other notoriously hard problems that are inherent to all distributed systems.

Current techniques for troubleshooting SDN control software are quite primitive; they essentially involve manual inspection of logs in the hope of identifying the relevant inputs. In this paper we developed a technique for automatically identifying a minimal sequence of inputs responsible for triggering a given bug. We believe our technique will be especially valuable for troubleshooting distributed controllers running complex applications, which are just now becoming available to the public and the broader research community.

References

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

[2] OpenStack Quantum. http://wiki.openstack.org/Quantum.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. OSDI '04.

[4] BigSwitch Networks. http://tinyurl.com/cgepwdj.

[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. NSDI '12.

[6] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. PRESTO '10.

[7] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. JACM '96.

[8] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM TOCS '85.

[9] J. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. SIGSOFT '02.

[10] ConteXtream. http://www.contextream.com/.

[11] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-Distributed Database. OSDI '12.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. SIGOPS '02.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. JACM '85.

[14] Floodlight Controller. http://floodlight.openflowhub.org/.

[15] Floodlight FIXME. Controller.java, line 605. http://tinyurl.com/af6nhjj.

[16] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging For Distributed Applications. ATC '06.

[17] P. Godefroid and N. Nagappan. Concurrency at Microsoft - An Exploratory Survey. CAV '08.

[18] 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, Sec. 3.4. SIGCOMM '09.

[19] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System For Networks. CCR '08.

[20] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments Using Container Based Emulation. CoNEXT '12.

[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the Debugger for my Software-Defined Network? HotSDN '12.

[22] U. Höelzle. OpenFlow at Google. Keynote, Open Networking Summit '12.

[23] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed Diagnosis in Enterprise Networks. SIGCOMM '09.

[24] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. NSDI '12.

[25] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. PLDI '07.

[26] 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.

[27] O. Lab. Open Networking Operating System. `http://onlab.us/tools.html`.

[28] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM '78.

[29] J. Mccauley. POX: A Python-based OpenFlow Controller. `http://www.noxrepo.org/pox/about-pox/`.

[30] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. CACM '90.

[31] Nicira NVP. `http://tinyurl.com/c9jbkuu`.

[32] OpenFlow. `http://www.openflow.org/`.

[33] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending Networking into the Virtualization Layer. HotNets '09.

[34] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—A Safe Method To Survive Software Failures. SIGOPS OSR '05.

[35] 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.

[36] V. Soundararajan and K. Govil. Challenges in Building Scalable Virtualized Datacenter Management. SIGOPS OSR '10.

[37] G. Tel. *Introduction to Distributed Algorithms*. Thm. 2.21. Cambridge University Press, 2000.

[38] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. OSDI '02.

[39] A. Whitaker, R. Cox, and S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. SOSP '04.

[40] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. ATC '11.

[41] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. A Survey of Fault Localization Techniques in Computer Networks. Science of Computer Programming '04.

[42] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. ASPLOS '10.

[43] A. Zeller. Yesterday, my program worked. Today, it does not. Why? ESEC/FSE '99.

[44] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE TSE '02.

[45] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Technical Report TR12-HPNG-061012, Stanford University '12.

[46] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. CoNEXT '12.