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
Kyriakos Zarifis
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-8
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-8.html

February 10, 2013

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[†]
Kyriakos Zarifis[‡] Scott Shenker[†*]
[†]*UC Berkeley* [◇]*Big Switch Networks* [*]*ISCI* [‡]*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 one might improve SDN network troubleshooting by presenting a technique, retrospective causal inference, for automatically identifying a minimal sequence of inputs responsible for triggering a given bug in the control software. Retrospective causal inference works by iteratively pruning inputs from the history of the execution, and coping with divergent histories by reasoning about the functional equivalence of events.

We apply retrospective causal inference to three open source SDN control platforms—Floodlight, 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? Would the United States have abolished slavery if Abraham Lincoln had not been elected? That is, were those prior events intrinsic to the precipitation of the historical outcome, or were they 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 learned of the link failure?" And unlike human history, it is often possible to find answers to such causality questions. In this paper we address the problem of programmatically answering them 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 [19]) 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 [15], providing isolation and resource arbitration between multiple tenants [6], and globally optimizing network utilization [25]. Most of this complexity comes from fundamentally difficult distributed systems challenges such as asynchrony and partial failure. Even Google's Urs Höelzle, a leading networking and distributed systems technologist, attests that [25] "[coordination between replicated controllers] is going to cause some angst, and justifiably, in the industry."

The troubleshooting process 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 [20]. Troubleshooters find little immediate use from traces containing many inputs prior to a fault, since they are often forced to manually filter extraneous inputs before they can start fruitfully exploring what might be the root cause. It is no surprise that when asked to describe their ideal tool, most network admins said "automated troubleshooting" [51].

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. A bug, in this context, 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. We call this an *invalid* configuration. We presume that the control plane functions properly in most circumstances, so that these policy violations are rare. Bugs may be triggered by uncommon sequences of inputs, such as a simultaneous link failure or controller reboot. The act of troubleshooting is identifying which set of inputs triggered the bug. Debugging then involves 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 (*e.g.* link failures) 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 [32]. 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. In particular, we generalize delta debugging [50]—an algorithm for minimizing test cases that are inserted at a single point in time to a single program—to a distributed environment, where inputs are spread across time and involve multiple machines.

The main difficulty in pruning historical inputs is coping with divergent histories. Traditional replay techniques [13, 18] 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.

Our approach is to record and replay 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.

The output of our approach, minimized input traces, represents a noteworthy improvement over the status quo; 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 [46] and liblog [18] enable you to 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 [24] 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 inferenceto three open source SDN control platforms: Floodlight [4], POX [35], and NOX [21]. Of the five bugs we encountered in a five day investigation, retrospective causal inference reduced the size of the input trace to 36% of its original size in the worst case and 2% of its original size in the best case.

2. BACKGROUND

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

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 [31], 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, 11, 38], 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 (up to 100,000 v-switches in a large datacenter) as a single logical switch for each tenant. When an entire datacenter 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 complex and bug-prone.

3. RETROSPECTIVE CAUSAL INFERENCE

To illustrate the mechanics of retrospective causal inference, we start by describing an example bug in the Floodlight open source control platform [16]. 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.

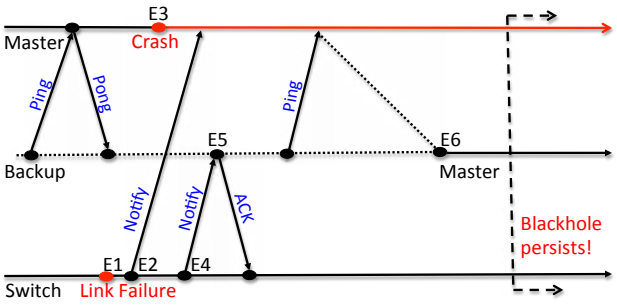

Figure 1: Floodlight failover bug. External inputs are depicted as red dots, internal events are depicted as black dots, and the dotted message line depicts a timeout.

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 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).

There were only two external inputs (E1,E3) shown in our example. However, a developer or operator encountering this bug in practice would not be given this concise version of events. Instead, the trace would a contain 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 [20], and 500 VM migrations per hour [42], there would be at least $8.5 \cdot 60 + 500 \approx 1000$ inputs reflected in an hour-long trace.

Given a trace of the system execution similar to the Floodlight case, our goal is to prune events that are not necessary for triggering errant behavior. We define errant behavior in terms of *correctness violations*: configurations of the network that are inconsistent with the policy. In the example, the correctness violation is between a reachability policy specified in the logical switch ("A can talk to B") and the blackhole in the physical network ("A's packets to B enter the blackhole and do not arrive at B").

Specifically, our technique identifies a minimal sequence of inputs to the controllers that is sufficient for triggering a known correctness violation. We refer to such inputs as a *minimal causal sequence* (MCS). Going back to our example, suppose the log includes many more (extraneous) inputs. Whenever an extraneous event is pruned, the blackhole will still persist: when the controller crash is pruned, the blackhole will be resolved properly, and when the link failure is pruned, no blackhole will occur. The MCS returned is therefore the controller crash and the link failure in conjunction.

[1]Note that this issue was originally documented by the developers of Floodlight [16].

3.1 Delta Debugging

Delta debugging [49], a technique from the software engineering community, gets us part of the way to our goal: given a single input (*e.g.* an HTML page) for a non-distributed program (*e.g.* Firefox), it performs a divide-and-conquer search, repeatedly running the program on subsets of the input until it finds a minimal subset (*e.g.* a single tag) that is sufficient for triggering a known bug. Specifically, it finds a locally minimal causal sequence [49], meaning that if any input from the sequence is pruned, no correctness violation occurs. The delta debugging algorithm is shown in Figure 2 (with '$test$' replaced by '$replay$').

Our problem differs from the original formulation of delta debugging in two dimensions. First, delta debugging assumes that input is inserted at a single point in time. In contrast, input to SDN controllers includes many messages spread throughout time. Second, delta debugging assumes a single program under test. Our input depends on causal relationships across concurrently running nodes.

For the purposes of this section, we model our problem as follows. We are given a single, globally ordered trace of events that ends in in a correctness violation, and we return a minimal causal subsequence of the trace. The trace includes input events (*e.g.* link failures), control message sends and receipts between switches and controllers, and internal state changes (*e.g.* the backup deciding to elevate itself to master in the Floodlight case) labeled with the control process that made the state change. In §4.1, we describe how we obtain the globally ordered trace in practice.

In the rest of this section we describe how we replay inputs to control software and cope with alterations to the causal history of an execution.

3.2 Simulated Execution

Unlike the example applications described by the original delta debugging paper [49], the system we are troubleshooting is not a single program–it is all the nodes and links of a distributed system, including controllers, switches, and end-hosts. The asynchrony of distributed systems makes it difficult to reliably replay orderings of events without great care. We therefore 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. The simulator uses these interposition points

Figure 2: Automated Delta Debugging Algorithm From [49]

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

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

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

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

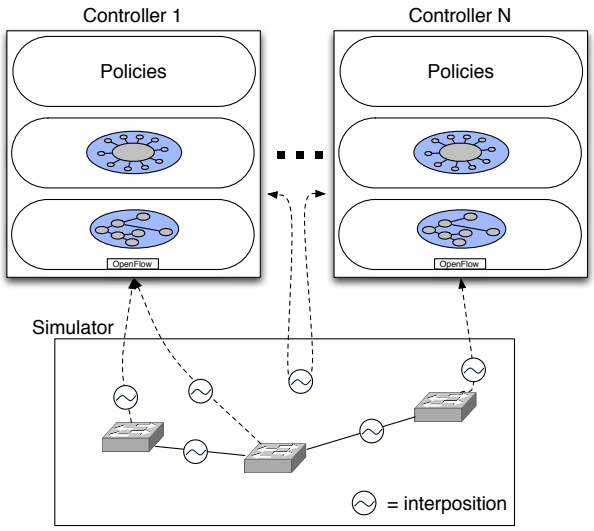

Figure 3: Simulation infrastructure. We simulate network devices in software, and interpose on all communication channels.

to delay, drop, or reorder messages as needed for replay. The overall simulation architecture is depicted in Figure 3.

Given a sequence of inputs (*e.g.* link failures, controller crashes, host migrations, or policy changes) and an invariant checking probe (provided by tools such as HSA [27,28] or Anteater [29,34]), delta debugging finds a minimal causal sequence responsible for triggering the policy violation. The simulator is responsible for replaying intermediate input subsequences chosen by delta debugging. For example, the simulator replays link failures by disconnecting the edge in the simulated network, and sending a port status message from the adjacent switches to their parent controller(s).

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.

3.3 Replay

The timing of the inputs injected by the simulator is crucial for reliably reproducing the correctness violation. Naïvely injecting inputs often fails to trigger the original correctness violation, even without having pruned any events. In particular, we tried and failed to reproduce errors when scheduling inputs with the following simple algorithm:

$$t'_0 = 0$$
$$t'_i = t'_{i-1} + |t_i - t_{i-1}|$$

where t'_i is the simulation's clock value when it injects the i^{th} input, and t_i is the timestamp of the i^{th} input from the original run. In other words, simply maintaining the relative timing between inputs is not sufficient.

The problem with the simple scheduling algorithm is that it does not take into account events that are internal to the control software, such as message receipts, timers going off, or internal state changes like the backup node in the Floodlight example deciding to elevate itself to master; if the ordering of inputs and internal events is perturbed, the final output may differ. Consider for example that if a controller's garbage collector happens to run while we replay inputs, it may delay an internal state transition until after the simulator injects an input that depended on it in the original run.

The challenge is to maintain causal relationships. Formally, to reliably reproduce the original correctness violation we need to inject an external input e at exactly the point when all other events (both external and internal) that precede it in the happens-before relation ($\{i \mid i \rightarrow e\}$) from the original execution have occurred [43].

[2]Handling invalid inputs is crucial for ensuring that the delta debugging algorithm we employ [49] is guaranteed to find a minimal causal sequence, since it assumes that no unresolved test outcomes occur. Zeller wrote a follow-on paper [50] that removes the need for this assumption, but incurs an additional factor of N in complexity in doing so.

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.

While the input and internal events from the original run are given to us, we become aware of internal events throughout replay by (i) monitoring control message receipts between controllers and switches, and (ii) interposing on the controllers' logging library and notifying the simulator whenever the control software executes a log statement (which serve to mark relevant state transitions). Note that to achieve truly deterministic replay, these log statements would need to be highly granular, capturing information such as thread scheduling decisions; we show in §5 however that pre-existing, course granular log statements are often sufficient to successfully reproduce bugs.

3.4 Fingerprinting

Replay is made substantially more complicated by the fact that the delta debugging algorithm is pruning inputs from the history of the execution, thereby changing the resulting internal events generated by the control software. In particular, internal events may differ syntactically (*e.g.* sequence numbers of control packets may all differ), old internal events from the original execution may not occur after pruning, and new internal events that were not observed at all in the original execution may appear.

Our first observation is 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 correctness 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.

These masks define equivalence classes of internal events. Formally, we consider an internal event i' observed in an altered trace equivalent to an internal event i from the original trace iff all unmasked fields have the same value and i occurs between i''s preceding and succeeding inputs in the happens-before relation.

3.5 Handling Absent Internal Events

Given an equivalence relation over internal events, replay is responsible for maintaining equivalent happens-before constraints from the original execution. But syntactic differences are not the only possible change induced by pruning: internal events from the original may also cease to appear.

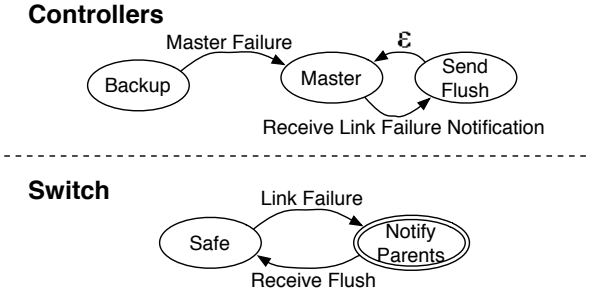

Figure 4: Simplified state machines for the switch and controllers in the example Floodlight bug. Double outlined states represent presence of the blackhole.

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 4. If we prune the link failure input, the master will never receive a link failure notification and transition to and from 'Send Flush'.

In the hope that absent internal events are not actually relevant for triggering the correctness violation, we proceed with replay. Specifically, our approach is to wait for expected equivalent internal events, but time out and proceed if they do not occur within a certain time ε.

In most cases this approach successfully reproduces the original correctness violation, assuming ε is larger than variations in execution speeds between internal events. If the value of ε is too large, however, we may end up waiting too long for the happens-before predecessors of an input e_i such that a successor of e_i occurs before we have injected e_i, thereby violating the remaining happens-before constraints.

If the event scheduling algorithm detects that it has waited too long, it replays the trace from the beginning up until the immediately prior input,[4] this time knowing exactly 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 5.

3.6 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 trace. Ultimately, new events 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

[3] One consequence of applying masks is that bugs involving masked fields are outside the purview of retrospective causal inference.

[4] An alternative would be to take a snapshot of the controllers' state at every injected input and start from the latest snapshot.

```
subsequence = [e1, e2, ..., ej]
// ei is always an input

function replay(subsequence):
  bootstrap the simulation
  for ei in subsequence:
    if ei is an internal event and
          ei is not marked absent:
      Δ = |ei.time - ei-1.time| + ε
      wait up to Δ seconds for ei
      if ei did not occur:
        mark ei absent
    else if ei is an input:
      if a successor of ei occurred:
        // waited too long
        return replay(subsequence)
      else:
        inject ei
```

Figure 5: Replay Algorithm Pseudocode. In practice we account for other vagaries not shown here.

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 that only leads to the correctness violation when we inject the next input before a new internal event, and one that only leads to the correctness 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 correctness 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 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 correctness violation. This is reasonable because not even branching on new internal events is guaranteed to find the shortest fault-inducing input sequence: there may be other unknown paths through the state machine leading to the correctness violation that are completely disjoint from the original execution.

Luckily, crucially ambiguous new internal events are not problematic for the control software we evaluated, as we show in §5. We conjecture that ambiguous new internal events are rare because SDN is a control plane system, and is designed to quiesce quickly (*i.e.* take a small number of internal transitions after any input event, and stop at highly connected vertices). Concretely, SDN programs are often structured as (mostly independent) event handlers, meaning that pruning input events simply triggers a subset of the original event handlers. As an illustration, consider the state machines in Figure 4: the controllers quickly converge to a single state (either "Master" or "Backup"), as do the switches ("Safe").

3.7 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 [49]. 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 $O(n^3)$.

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.

SDN platform developers can reduce the probability that the replay algorithm will need to back up by placing causal annotations on internal events [17]: with explicit causal information, the replay algorithm can know *a priori* whether certain internal are dependent on pruned inputs.

3.8 Limitations

Having detailed the specifics of retrospective causal inference, 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. In practice many relevant internal state changes are already marked by logging statements, but developers may need to add additional logging statements to ensure reliable replay.

Non-determinism Within Individual Controllers. Our tool 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), which we find plenty of in §5. The upshot of this is that our technique is not able to minimize all possible failures, such as data races between threads. Nonetheless, the worst case for us is that the developer ends up with what they started: an unpruned log.

Troubleshooting vs. Debugging. Our technique is a trou-

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

bleshooting tool, not a debugger; by this we mean that retrospective causal inference helps identify and localize inputs that trigger erroneous behavior, but it does not directly identify which line(s) of code cause the error.

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 such inputs. If there is a bug in your switch, you will need to contact your hardware vendor; if you have a bug in your policy specification, you will need to take a closer look at what you specified.

Globally vs. Locally Minimal Input Sequences. Our approach is not guaranteed to find the globally minimal causal sequence from an input trace, since this requires $O(2^N)$ computation in the worst case. The delta debugging algorithm we employ does provably find a locally minimal causal sequence [49], meaning that if any input from the sequence is pruned, no correctness violation occurs.

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

4. SYSTEM DESIGN AND USAGE SCENARIOS

STS (the SDN Troubleshooting Simulator) is our realization of the techniques described in §3. STS is implemented in roughly 10,000 lines of Python in addition to the Hassel network invariant checking library [28]. We have made the code for STS publicly available at `http://ucb-sts.github.com/sts/`. To date, three industrial SDN companies have expressed interest in adopting it.

In the rest of this section, we highlight salient points of STS's design, and illustrate a workflow for users of STS.

We show the input types supported by STS in Table 2. Our software switches notify controllers about link, switch, and host migration events by sending OpenFlow messages [39]. Although the software switches do support packet forwarding, we have not focused on simulating high-throughput dataplane behavior.

We designed STS to be as resilient to non-determinism as is practically feasible, while avoiding modifications to control software whenever possible. When sending data over multiple sockets, the operating system exhibits non-determinism in the order it schedules the socket I/O operations. STS optionally ensures a deterministic order of messages by multiplexing all sockets in the controller process onto a single true socket.[5] STS currently overrides socket functionality within the control software itself.[6] In the future we plan to implement deterministic message ordering without code modifications by loading a shim layer on top of libc (similar to liblog [18]).

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[7]: whenever a control process executes a log statement, we notify STS that a new state transition is about to occur, and optionally block the process. STS then sends an acknowledgment to unblock the controller after logging the state change. If blocking was enabled during recording, we force the control software to block at internal state transition points again during replay until STS gives explicit acknowledgment.

Routing the `gettimeofday()` syscall through STS makes replay more resilient to alterations in execution speeds.[8] As an added benefit, overriding `gettimeofday()` allows us to 'compress' runtime in some cases (similar to time-warped emulation [22]).

If the control software under test utilizes random number generators, we attempt to manually replace any such functionality with deterministic algorithms if possible. Our current implementation does not account other sources of non-determinism, such as asynchronous signals, or interruptable instructions (*e.g.* x86's block memory instructions [13]).

Developers and operators can use STS in a number of ways. Here we illustrate a general workflow.

4.1 Bug Exploration

Use of STS begins with bug exploration. STS itself is well-suited for finding input traces that trigger bugs: it readily simulates common network input events, and ships with a suite of invariant checking algorithms [28]. With complete control over event orderings, STS is especially useful for exploring corner cases. Along these lines, Amin Vahdat has testified to the value of Google's SDN network simulator [10]:

> "One of the key benefits we have is a very nice emulation and simulation environment where the exact same control software that would be running on servers might be controlling a combination of real and emulated switching devices. And then we can inject a number of failure scenarios under controlled circumstances to really accelerate our test work."

[5] Alternatively, we could employ a mutex [33].

[6] Only supported for POX at the moment.

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

[8] 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

Developers can use STS to generate randomly chosen input sequences [36], feed them to controller(s), and monitor invariants at chosen intervals. Driving the execution of the system in this way allows STS to record a totally-ordered log of the events to be replayed later.

Developers can also run STS interactively to generate replayable integration tests, similar to Nebut et al. [37]. In interactive mode developers can examine the state of any part of the simulated network, observe and manipulate messages, and follow their intuition to induce orderings that they believe may trigger bugs.

Generating integration tests in this fashion frees developers to be more agile and spend spend less time writing test cases. As developers and operators encounter additional failure cases they can add them to a suite of integration tests later used to validate the correct behavior of future versions of the software. Since STS makes limited assumptions about the control software under test, the overall SDN community could potentially collect a common repository of test cases.

4.2 Replay

Having discovered a bug, developers can use STS to replay the inputs that triggered the bug. Repeated replay in conjunction with print statements or source-level debuggers is how troubleshooters can ultimately find the buggy line(s) of code (as envisioned by [46]).

Replay with STS has the potential to change how developers elicit help from mailing lists or coworkers. The status quo is to describe the conditions needed to reproduce the bug as carefully as possible and hope that others are able to replicate the issue. With STS, developers can record errant executions in the simulated environment, and exchange traces to be replayed again at other developer's machines.

4.3 Minimizing Input Traces

Moving beyond network replay, STS's main value is in automatically minimizing input traces. Troubleshooting can be highly time-consuming and challenging when the developer has no intuition as to where the problem might arise and only a large input log to work with. For instance, stepping through a 1000 event trace in a source level debugger can involve taking thousands of individual steps. When inspecting log files, developers are often confronted with dozens of lines of debug output per event and hundreds of thousands of log lines overall. With retrospective causal inference, much of the heavy lifting can be performed automatically before the developers begin to diagnose the root cause.

At the least, retrospective causal inference reduces the runtime of test cases and eliminates distracting events during replay. More importantly, minimal causal sequences give developers intuition about what code path is throwing the network into an invalid configuration. In our own experience investigating the bugs described in §5, we had little understanding of what the problem was at first. After identifying the MCS, it became easier to understand what corner case was triggered, and how the bug might be resolved.

Minimal causal sequences also serve to consolidate redundant test cases: if two test failures have the same minimal causal sequence, it is likely that the same underlying bug is responsible [50]. This eliminates time wasted investigating bug reports with the same root cause.

4.4 Analysis of Production Logs

Input generation, interactive execution, replay, and test case minimization are implemented and used in STS today. Forensic analysis of production logs, while not currently implemented, may be another valuable use case of STS. Here we present a sketch of how forensic analysis could be performed with retrospective causal inference.

While retrospective causal inference takes as input a single, totally-ordered log of the events in the distributed system, production systems maintain a log at each node. Instrumentation and preprocessing steps are therefore needed.

Production systems would need to include Lamport clocks on each message [32] or have sufficiently accurate clock synchronization [12] to obtain a partial global ordering consistent with the happens-before relation.[9] Contrast this with STS's testing mode, where a global event ordering is obtained by logging all events at a single location.

The distributed logs would also need to make a clear distinction between internal events and external input events. Further, the input events would need to be logged in sufficient detail for STS to reproduce a synthetic version of the input that is indistinguishable (in terms of control plane messages) from the original input event.

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 is to employ perfect failure detectors [7], which log a failure iff the failure actually occurred. Alternatively, one could employ root cause analysis algorithms [47] or manual inspection to consolidate redundant alarms.

Finally, some care is needed to prevent the logs from growing so large that retrospective causal inference's runtime becomes intractable. Here, causally consistent snapshots [8] can minimize the number of inputs retrospective causal inference needs to examine. Specifically, with causally consistent snapshots of the distributed system taken at regular intervals, STS can bootstrap its simulation from the last snapshot before the failure. If the MCS starting from this snapshot is empty, it can iteratively move backwards, starting from earlier snapshots.

5. EVALUATION

We have applied STS to three open source SDN control

[9] 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 [14].

Bug Name	Topology	Replay Success Rate	Total Inputs	MCS Size
POX list removal	2 switch mesh	20/20	76 (69)	2 (2)
POX in-flight blackhole	2 switch mesh	15/20 [20/20*]	68 (26)	25 (11)
POX migration blackhole	4 switch mesh	20/20*	117 (29)	3 (2)
NOX discovery loop	4 switch mesh	18/20	358 (150)	58 (18)
Floodlight loop	3 switch mesh	15/50	548 (284)	404 (?)

Table 3: Overview of Case Studies. Totals shown in parentheses are with dataplane permit events excluded. *with multiplexed sockets and logging interposition enabled.

platforms: POX [35], NOX [21], and Floodlight [4]. 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.

5.1 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.

There were 76 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 intitialization handshakes), this is not entirely unsurprising. 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.

5.2 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. We encountered this bug on a simple topology, depicted in Figure 7, consisting of two hosts A and B and two switches S1 and S2 connected by a single link.

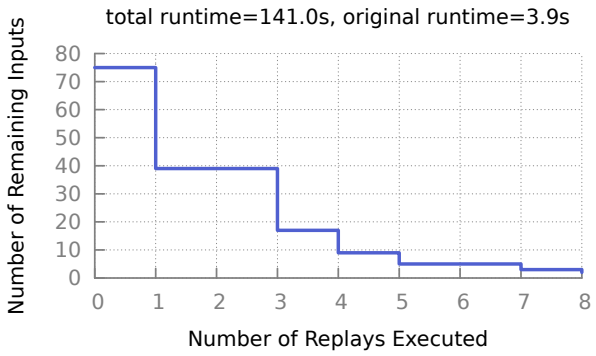

Figure 6: Minimizing the POX list remove trace.

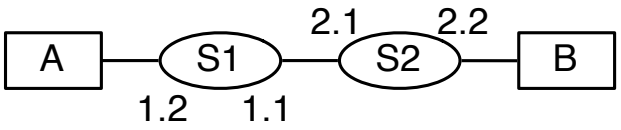

Figure 7: Topology for POX in-flight blackhole. Numbers denote port labels.

There were 68 inputs in the initial trace, and retrospective causal inference returned a 25 input MCS (runtime shown in Figure 8). With the MCS in hand we took out paper and pencil to decipher what had transpired.

Before the discovery module had learned of the link connecting the two switches, there were six traffic injection events between hosts (A→ B and B→A). At the point when the link was discovered, POX had previously learned of B's location at port 2.2, and correctly unlearned a previous location for A at port 2.1 (which it now knew to be a switch-switch link).

Directly after the link discovery we observed an *in-flight* packet arriving from A→B at port 2.1 (without a prior flow notification from S1). This was POX's first error. Upon examining the code, we found that it did not account for in-flight packets concurrent with link discovery. As a result,

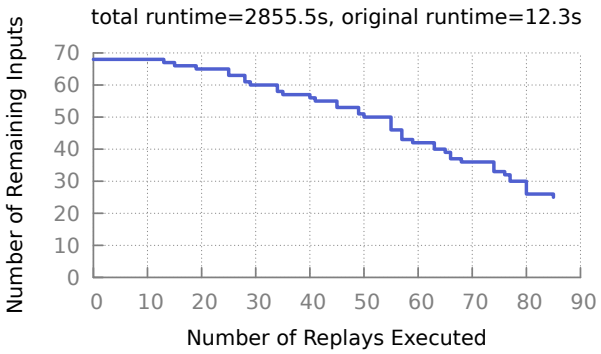

Figure 8: Minimizing the POX in-flight blackhole.

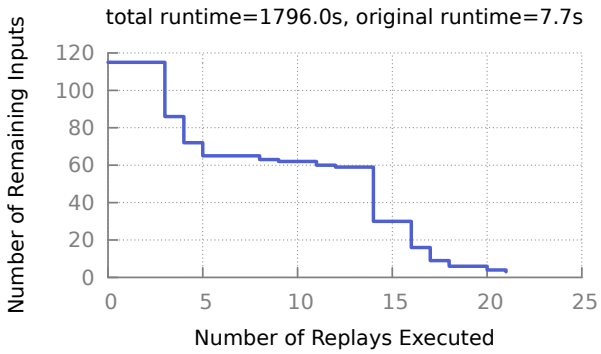

Figure 9: Minimizing the POX migration blackhole.

POX incorrectly learned A's location at 2.1, even though it knew that the link could not have hosts attached. If the first packet had instead originated at 1.1, POX would not have made this mistake.

The next event we observed was another *in-flight* packet from B→A arriving at port 1.1. S1 notified POX of the unmatched flow, and POX appropriately printed a log statement indicating that a packet had arrived at an internal switch port without a previously installed flow entry. What happened next puzzled us though. POX proceeded to install a path for this new B→A flow, but the path itself contained a loop: POX installed a B→A entry going out both 1.1→2.1 and 2.2→2.1, whereas it should have installed only the latter (given A's current known location). The default behavior of OpenFlow switches is to ignore matching route entries (with wildcarded in ports) that forward out the same port the packets arrived on. This is where we started observing the blackhole: now whenever B sent traffic to A, it would be dropped at S1 until the faulty routing entry would eventually expire 30 seconds later.

We investigated the code that handled in-flight packets arriving on switch-switch ports. The log statement that we had observed earlier was inside a nested conditional, and the code for installing the path was below and outside of the nested conditional conditional. What struck us was that there was a commented out return statement directly after the log statement. The comment above it read: "Should flood instead of dropping". We tried reinserting the return statement and replaying, and the blackhole ceased to appear.

In summary, we found that the crucial triggering events were two in-flight packets (set in motion by prior traffic injection events): POX incorrectly learned a host location as a result of the first in-flight packet, and failed to return out of a nested conditional as result the second in-flight packet. We have sent the replayable trace generated by retrospective causal inference to the lead developer of POX, and await his response. We suspect that these fine-grained race conditions had not been triggered before because message timing in Mininet [23] or real hardware is not delayed arbitrarily as it was in STS.

5.3 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 117 inputs. Before investigating the bug we ran retrospective causal inference, and ended up with a 3 input MCS (shown in Figure 9): 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 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.

5.4 NOX Discovery Loop

The next SDN control platform we examined was NOX, the original OpenFlow controller. NOX is also a single machine control platform, but unlike POX it has been used fairly extensively in real networks.

Similar to POX we exercised NOX's routing module ('sprouting'), since it draws in a large number of other components. Routing learns link and host locations, installs all-to-all paths between hosts on a per-flow basis, and is designed to be resilient to looped topologies.

We initially tested NOX on a two node topology, but did not find any immediate problems. We then extended the topology to a four-node mesh, and discovered a routing loop between two switches (involving routes for two hosts) within roughly 20 runs of randomly generated inputs.

Our initial input size was 358 inputs, a minute's worth of execution. Retrospective causal inference returned a 58 input MCS. The most salient inputs in the MCS were 3 dataplane packet drops mid-way through the execution, interspersed with 14 traffic injections. We are in the process

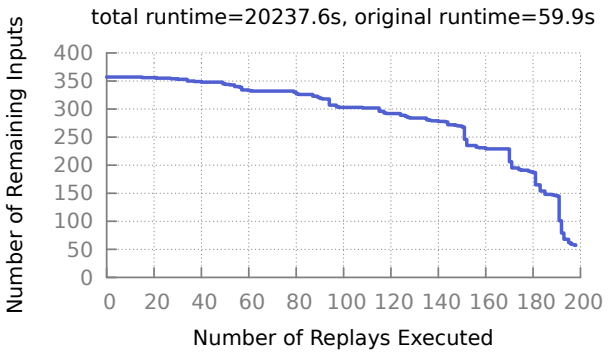

Figure 10: Minimizing the NOX discovery loop.

of pinpointing the exact root cause with NOX developers, based on the 58 input MCS.

5.5 Floodlight discovery loop.

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 777 total events (548 input, 229 internal) that results in a 3-node forwarding loop being set up.

Floodlight makes use multiple kernel level threads, and thus can exhibit non-deterministic behavior. Thus, it is not surprising that we do not achieve full reproducibility of this bug during replay without further instrumentation. On average, 15/50 (30%) of replays reproduce the bug. To proceed with MCS isolation, we replayed the execution up to 13 replays for each subsequence chosen by delta debugging. Statistically, this enables STS to correctly diagnose violations in >99% of cases.[10]

Retrospective causal inference was able to reduce the number of input events from 548 to 404 in 168 iterations (note that this is not the final result; the algorithm crashed due to a trivial error a few hours before the deadline). Comparing output traces of successful and unsuccessful runs, we noticed that the bug seems to correlate with specific thread level race conditions between state updates in the *LinkDiscovery* module and the *Forwarding* module. We are in the process of investigating the actual root cause.

This experiment provides a baseline for a worst case scenario of our system. STS exercised an unmodified, multi-threaded controller that it does not have deterministic control over. The bug appears to depend on fine-grained thread-level races conditions that are difficult to guarantee. Still, STSwas instrumental in pointing out a previously unknown bug, and reducing the input size.

Overall Results. The overall results of our case studies are shown in Table 3. 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 blackhole, we were able to eliminate the relevant non-determinism by employing multiplexed sockets and waiting on POX's logging messages. We expect that we would see similar improvements if we applied these techniques to Floodlight.

We show the initial input size and MCS input size in the last two columns. We also show the input sizes excluding dataplane forwarding permit events, since these inputs are an artefact of how we currently store and replay event traces. We plan on making dataplane permits a default.

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 and a half hours. Nonetheless, even long iteration sizes are often preferable to spending software developer's time on manual diagnosis.

5.6 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 numbers 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 εvalues.

6. 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 [31]. 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

[10] $ln(1-0.99)/ln(1-0.30) \approx 13$

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 sector.

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.[11] 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.

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 [39], 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 fingerprinting. 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.

7. RELATED WORK

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

Software Engineering The software engineering community has developed a long line of tools for automating aspects of the troubleshooting process.

Sherlog [48] 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 multi-threaded (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 [45]. All of these techniques focus on troubleshooting single, non-distributed systems.

Rx [40] 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.

Systems and Networking The systems and networking community has also developed a substantial literature on tools for testing and troubleshooting.

We share the common goal of improving troubleshooting of software-defined networks with OFRewind [46] and recent project ndb [24]. OFRewind provides record and replay

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

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 [41] allow developers to programmatically check whether their expectations about the structure of recorded causal traces hold. MagPie [3] automatically identify 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 [23], ns-3 [1], and ModelNet [44] 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 [47] and dependency inference [26] 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.

Programming Languages Finally, the programming languages community has developed numerous verification and static analysis techniques.

Model checkers such as Mace [30] 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 take as input a particular event trace that is known to trigger a bug, and systematically enumerate subsequences of that event trace in polynomial time.

8. 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 the 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. Thus, people will be troubleshooting and debugging SDN control software for many years to come, until they become as stable as compilers are now.

Current techniques for troubleshooting SDN networks 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 have applied this system to three open source SDN platforms. Of the five bugs we encountered in a five day investigation, our technique reduced the size of the input trace to 36% of its original size in the worst case and 2% of its original size in the best case.

9. 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] ComputerWorld. Interview with Amin Vahdat. http://tinyurl.com/cb4qa6u.

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

[12] 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.

[13] 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 OSR '02.

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

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

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

[17] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive Network Tracing Framework. NSDI '07.

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

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

[20] 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.

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

[22] D. Gupta, K. Yocum, M. Mcnett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To Infinity and Beyond: TimeWarped Network Emulation. NSDI '06.

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

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

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

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

[27] P. Kazemian, M. Change, H. Zheng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. NSDI '13.

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

[29] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. NSDI '13.

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

[31] 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.

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

[33] C. Lin, M. Caesar, and J. Van der Merwe. Towards Interactive Debugging for ISP Networks. HotNets '09.

[34] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. SIGCOMM '11.

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

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

[37] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel. Automatic Test Generation: a Use Case Driven Approach. IEEE TSE '06.

[38] Nicira NVP. http://tinyurl.com/c9jbkuu.

[39] OpenFlow. http://www.openflow.org/.

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

[41] 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.

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

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

[44] 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.

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

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

[47] 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.

[48] 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.

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

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

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