Troubleshooting SDN Control Software with Minimal Causal Sequences

Colin Scott[†] Andreas Wundsam[◇] Barath Raghavan[*] Andrew Or[†] Jefferson Lai[†]
Eugene Huang[†] Zhi Liu[+] Ahmed El-Hassany[*] Sam Whitlock[*] H.B. Acharya[*]
Kyriakos Zarifis[‡] Scott Shenker[†*]
[†]*UC Berkeley* [◇]*Big Switch Networks* [*]*ICSI* [‡]*USC* [+]*Tshinghua University*

ABSTRACT

Software bugs are inevitable in software-defined networking control software, and troubleshooting is a tedious, time-consuming task. In this paper we discuss how to improve control software troubleshooting by presenting a technique for automatically identifying a minimal sequence of inputs responsible for triggering a given bug. We apply our technique to five open source SDN control platforms—Floodlight, NOX, POX, Pyretic, ONOS—and illustrate how the minimal causal sequences our system found aided the troubleshooting process.

1. INTRODUCTION

Software-defined networking (SDN) proposes to simplify network management by providing a simple logically-centralized API upon which network management programs can be written. However, the software used to support this API is anything but simple: the SDN control plane (consisting of the network operating system and higher layers) is a complicated distributed system that must react quickly and correctly to failures, host migrations, policy-configuration changes and other events. All complicated distributed systems are prone to bugs, and from our first-hand familiarity with five open source controllers and three major commercial controllers we can attest that SDN is no exception.

When faced with symptoms of a network problem (*e.g.* a persistent loop) that suggest the presence of a bug in the control plane software, software developers need to identify which events are triggering this apparent bug before they can begin to isolate and fix it. This act of "troubleshooting" (which precedes the act of debugging the underlying code) is highly time-consuming, as developers spend hours poring over multigigabyte execution traces.[1] Our aim is to reduce effort spent on troubleshooting SDN control software.

To that end, we have built a troubleshooting system that, given an execution trace, attempts to automatically eliminate trace events that are not causally related to the bug, producing a "minimal causal sequence" (MCS) of triggering events. Once the original trace has been reduced to an MCS (or an approximation thereof), the developer embarks on the debugging process. We claim that the greatly reduced size of the trace makes it easier for the developer to figure out which code path contains the underlying bug, allowing them to focus their effort on the task of fixing the problematic code itself. After the bug has been fixed, the MCS can serve as a test case to prevent regression, and can help identify redundant bug reports where the MCSes are the same.

Our goal of minimizing test cases is in the spirit of delta debugging [47], but our problem is complicated by the distributed nature of SDN control software: our test input is not a single file fed to a single point of execution, but an ongoing sequence of events involving multiple actors. Therefore we must carefully control timings of events in the face of asynchrony and non-determinism while we specutively perturb the original execution, without making assumptions about the language or instrumentation of the control software. As far as we know we are the first to minimize inputs to a distributed system (discussed in §8).

Our troubleshooting system, which we call STS (SDN Troubleshooting System), consists of 21,000 lines of Python, and is designed so that organizations can implement the technology within their existing QA test infrastructure (discussed in §5); over the last year we have worked with a commercial SDN company to integrate STS with their proprietary control software. We evaluate the effectiveness of STS in two ways. First and most significantly, we use STS to troubleshoot seven previously unknown bugs—involving concurrent events during bootstrapping, failure-induced blackholes, internal-state errors, and lock acquisition in a distributed database—that we found by fuzz testing five controllers (Floodlight [13], NOX [20], POX [34], Pyretic [16], ONOS [37]) written in three different languages (Java, C++, Python). Second, we demonstrate the boundaries of where STS works well by finding MCSes for previously known and synthetic bugs that span a range of bug types. In our evaluation, we quantitatively show that STS is able to minimize traces by up to 98% (of the new bugs), and we anecdotally found that reducing traces to MCSes made it easy to understand the root causes of the bugs.

[1]Software developers in general spend roughly half (49% according to one study [18]) of their time troubleshooting and debugging, and spend considerable time troubleshooting bugs that are difficult to trigger (the same study found that 70% of the reported concurrency bugs take days to months to fix).

2. BACKGROUND

Network operating systems, the key component of SDN software infrastructure, consist of control software running on a replicated set of servers, each running a controller instance. Controllers coordinate between themselves, and receive input events and statistics from switches (either physical or virtual), configuration and policy changes via a management interface, and possibly packets from the dataplane. In response to these events, the controllers issue forwarding instructions to switches. All input events are asynchronous, and individual controllers may fail at any time. The controller instances may communicate with each other over the network they manage, or use a separate dedicated network. In either case, controllers may become partitioned.

The goal of a network control plane is to configure the switch forwarding entries so as to enforce one or more invariants, such as connectivity (*i.e.*, ensuring that a route exists between every endpoint pair), isolation and access control (*i.e.*, various limitations on connectivity), and virtualization (*i.e.*, ensuring that packets are handled in a manner consistent with what would happen in the specified virtual network). A bug causes an invariant to be violated. Invariants can be violated because the system was improperly configured (*e.g.*, the management system [2] or a human improperly specified their goals to the SDN control plane), or because there is a bug within the SDN control plane itself. In this paper we focus on troubleshooting bugs in the SDN control plane after it has been given a configuration.[2]

In commercial SDN development, software developers work with a team of QA engineers whose job is to find bugs. The QA engineers exercise automated test scenarios that involve sequences of external events such as failures, migrations, or policy changes on large (software emulated or hardware) network testbeds. If they detect an invariant violation, they hand the resulting trace to a developer for analysis.

The space of possible bugs is enormous, and it is difficult and time consuming to link the symptom of a bug (*e.g.*, a routing loop) to the sequence of events in the QA trace (which includes both the external events and internal monitoring data), since QA traces contain a wealth of extraneous events. Consider that an hour long QA fuzz test emulating event rates observed in production could contain 8.5 network error events per minute [19] and 500 VM migrations per hour [41], for a total of $8.5 \cdot 60 + 500 \approx 1000$ inputs.

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 need to account for timings T because as we prune events we are unable to use the original timings T_L. 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 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 a subsequence of this MCS.

Given a log (E_L, T_L, C_L) exhibiting an invariant violation,[3] 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 heuristics for which events to eliminate and, more importantly, which timings to test.

In the next section we describe our heuristics for finding (approximate) MCSes for use in troubleshooting, and then describe the system we built in §5 to implement these heuristics. The key component of this system is a mock network that can execute $replay()$.

4. MINIMIZING TRACES

Given a log (E_L, T_L, C_L) generated from testing infrastructure exhibiting an invariant violation, our goal is to identify its MCS. This involves two tasks: searching through subsequences of E_L, and searching for replay timings for

[2]This does not preclude STS from troubleshooting misspecified policies so long as invariants [25] are specified separately.

[3]We discuss how these logs are generated in §5.

those subsequences that, if possible, trigger the original invariant violation.

4.1 Searching for Subsequences

Checking random subsequences of E_L would be one viable but inefficient approach to achieving our first task. We do better by leveraging the delta debugging algorithm [47], a divide-and-conquer algorithm for isolating fault-inducing inputs. In our case, we use delta debugging to iteratively select subsequences of E_L and replay each subsequence with some timing T. If the bug persists for a given subsequence, delta debugging ignores the other inputs, and proceeds with the search for an MCS within this subsequence. The delta debugging algorithm is shown in Figure 1.

The input subsequences chosen by delta debugging are not always valid. Of the possible inputs sequences we generate (shown in Table 2), it is not sensible to replay a recovery event without a preceding failure event, nor to replay a host migration event without modifying its starting position when a preceding host migration event has been pruned. Our implementation of delta debugging therefore prunes failure/recovery event pairs as a single unit, and updates initial host locations whenever host migration events are pruned so that hosts do not magically appear at new locations.[4] These two heuristics account for validity of all network events shown in Table 2. We do not yet support network policy changes as events, which have more complex semantic dependencies.[5]

4.2 Searching for Timings

Simply exploring subsequences of E_L is insufficient for finding MCSes: the timing of the inputs within those subsequences during $replay()$ is crucial for reliably reproducing the invariant violation. The most natural approach would be to maintain the original timings between the remaining inputs of each subsequence. Unfortunately, this approach does not work in practice because it can violate causality: the control software may react to remaining inputs differently as a result of us pruning other inputs. To reproduce the invariant violation we need to inject an input event e only after all other events, including internal events triggered by the control software itself, that precede it in the happens-before [28] relation ($\{i \mid i \rightarrow e\}$) from the original execution have occurred [43].

Internal events include (a) message delivery events, either between controllers (*e.g.* database synchronization messages) or between controllers and switches (*e.g.* OpenFlow commands), and (b) state transitions within controllers (*e.g.* a backup node deciding to become master). We obtain visibility into (a) by interposing on all messages within the test environment (to be described in §5). We optionally obtain visibility into (b) by instrumenting controller software with a simple interposition layer (to be described in §5.2). With visibility into internal events and control over input events, we record a totally-ordered trace during testing by logging an event only after all prior events are completed.

Maintaining the happens-before relation from the original trace (which reproduces the violation) throughout replay of subsequences of the trace (which may or may not reproduce that violation) 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. Internal events may differ syntactically (*e.g.* sequence numbers 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_mod` messages may cause switches to make the same change to their forwarding behavior even if their transaction ids differ.

We leverage this observation by defining masks over semantically extraneous fields of internal events.[6] These masks only need to be specified once, and can later be applied programmatically to event traces.

We show the fields we mask in Table 1. We consider an internal event i' observed in the replay equivalent (in the sense of inheriting all of its happens-before relations) to an internal event i from the original log if and only if 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 input may be absent when replaying a subsequence of that log. For instance, if we prune a link failure event, then

Internal message	Masked values
OpenFlow messages	xac id, cookie, buffer id, stats
packet_out/in payload	all values except src, dst, data
Log statements	varargs parameters to printf

Table 1: Internal messages and their masked values.

[4] Handling invalid inputs is crucial for ensuring that the delta debugging algorithm finds a minimal causal subsequence. The algorithm we employ [47] makes three assumptions about inputs: monotonicity, unambiguity, and consistency. An event trace that violates monotonicity may contain events that "undo" the invariant violation triggered by the MCS, and may therefore exhibit slightly inflated MCSes. An event trace that violates unambiguity may exhibit multiple MCSes; delta debugging will return one of them. The most important assumption is consistency, which requires that the test outcome can always be determined. We guarantee neither monotonicity nor unambiguity, but we guarantee consistency by ensuring that subsequences are always semantically valid by applying the two heuristics described above. Zeller wrote a follow-on paper [48] that removes the need for these assumptions, but incurs an additional factor of n in complexity in doing so.

[5] If codifying the semantic dependencies of policy changes turns out to be difficult, one could just employ the more expensive version of delta debugging to account for inconsistency [48].

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

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 1: Automated Delta Debugging Algorithm From [47].

Input Type	Implementation
Switch failure/recovery	TCP teardown
Controller failure/recovery	`SIGKILL`
Link failure/recovery	`ofp_port_status`
Controller partition	`iptables`
Dataplane packet injection	Network namespaces
Dataplane packet drop	Dataplane interposition
Dataplane packet delay	Dataplane interposition
Host migration	`ofp_port_status`
Control message delay	Controlplane interposition
Non-deterministic TCAMs	Modified switches

Table 2: Input types currently supported by STS.

```
procedure PEEK(input subsequence)
  inferred ← [ ]
  for e_i in subsequence
    checkpoint system
    inject e_i
    Δ ← |e_{i+1}.time − e_i.time| + ε
    record events for Δ seconds
    matched ← original events & recorded events
    inferred ← inferred + [e_i] + matched
    restore checkpoint
  return inferred
```

Figure 2: PEEK determines which internal events from the original sequence occur for a given subsequence.

the corresponding link failure notification will never arise.

We handle this by attempting to infer the presence of internal events before we replay each subsequence. Our algorithm (called PEEK()) for inferring the presence of internal events is depicted in Figure 2. The algorithm injects each input, records a checkpoint[7] of the network and the control software's state, allows the system to proceed up until the following input (plus a small time ϵ), records the observed events, and matches the recorded events with the functionally equivalent internal events observed in the original trace. With these inferred causal dependencies, we then replay the subsequence, this time waiting to inject each input until each of its (functionally equivalent) predecessors have occurred.[8]

Handling New Internal Events. The last possible change induced by pruning is the occurrence of new internal events that were not observed in the original log. New events present 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 that only leads to the invariant violation when we inject the next input *before* a new internal event, and another only when we inject *after* a new internal event. In other words, to be guaranteed to traverse any existing suffix that leads to the invariant violation, we must 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 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 a subset of the (equivalent) original internal events, but leaves open the possibility of finding divergent suffixes that lead to the invariant violation.

4.3 Complexity

The delta debugging algorithm terminates after $\Omega(\log n)$ invocations of $replay$ in the best case, and $O(n)$ in the worst case, where n is the number of inputs in the original trace [47]. Each invocation of $replay$ takes $O(n)$ time (one iteration for PEEK() and one iteration for the replay itself), for an overall runtime of $\Omega(n \log n)$ best case and $O(n^2)$ worst case replayed inputs. The runtime can be decreased by parallelizing delta debugging: speculatively replaying subsequences in parallel, and joining the results. Storing periodic checkpoints of the system state throughout testing can also reduce runtime, as it allows us to replay starting from a recent checkpoint rather than the beginning of the trace.

5. SYSTEMS CHALLENGES

Our technique depends on infrastructure for testing SDN control software. All three of the major commercial SDN controller teams that we have first-hand knowledge of em-

[7] We discuss the implementation details of checkpointing in 5.3.

[8] In the case that, due to non-determinism, an internal event occurs during PEEK() but does not occur during replay, we time out on internal events after ϵ seconds of their expected occurrence.

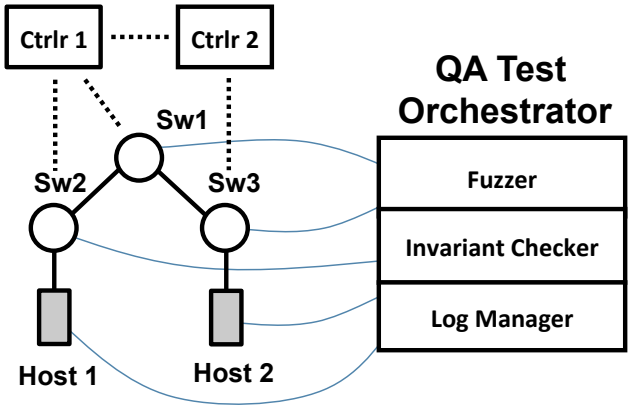


Figure 3: Typical QA testbed. A centralized test orchestrator injects inputs and checks invariants

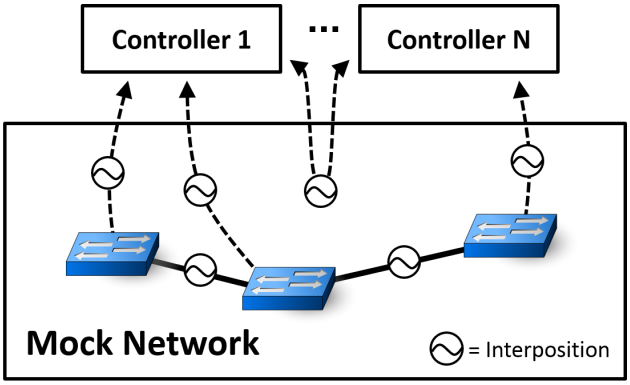


Figure 4: STS runs mock network devices, and interposes on all communication channels.

ploy a team of QA engineers to fuzz test their controller software on testbeds of switches and hosts. As depicted in Figure 3, this fuzz testing infrastructure consists of the controller software under test, the network testbed (which may be software or hardware), and a centralized test orchestrator that chooses random input sequences, drives the behavior of the testbed, periodically checks invariants, and manages log files. When a bug is discovered, an engineer triages it and then sends logs to a developer for further troubleshooting.

Engineering organizations with existing QA testbeds can add delta debugging to their test orchestrator, and optionally add interposition points throughout the testbed to control event ordering during replay. In this way they can continue simulating large scale networks with the switches, middleboxes, hosts, and routing protocols they had already chosen to include in their QA testbed.

We do not have access to such a QA testbed, and instead built our own testing framework to discover bugs and perform replay. Our framework mocks out the control plane behavior of network devices in lightweight software switches and hosts (with support for minimal data plane forwarding). We then run the control software on top of this mock network and connect the software switches to the controllers. This design is similar to production software QA testbeds. Our mock network also implements delta debugging and interposes and buffers messages on all communication channels, allowing it to delay, drop, or reorder messages as needed to induce failure modes during testing. This interposition allows us to replicate many of the failure modes caused by asynchrony in real networks in a more deterministic manner. During replay, we use these buffers to enforce event orderings by matching messages in the buffers by their fingerprints and managing the order in which messages are let through. The overall architecture is depicted in Figure 4.

We begin by using our mock network to perform testing on controllers to find bugs. Most commonly we generate randomly chosen input sequences [35], feed them to controller(s), and monitor invariants at chosen intervals.[9] We also run the mock network interactively so that we can examine the state of any part of the 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 our test framework to record a global, serial event ordering.

After discovering an invariant violation of interest, our system replays the logged sequence of inputs (shown in Table 2). For example, the system replays link failures by disconnecting the edge in the mock network, and sending an `ofp_port_status` [38] message from the adjacent switches to their parent controller(s).

STS is our realization of this system and is implemented in more than 21,000 lines of Python in addition to the Hassel network invariant checking library [25]. STS also optionally makes use of Open vSwitch [39] 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.

5.1 Coping with Non-Determinism

Non-determinism in the execution of concurrent processes stems from differences in system call return values, process scheduling decisions (which can even affect the result of individual instructions, such as x86's interruptible block memory instructions [12]), and asynchronous signal delivery. These sources of non-determinism can affect whether STS is able to reproduce the original bug during replay.

Most testing systems, such as the QA testing frameworks we are trying to improve, do not attempt to mitigate non-determinism. STS's main approach to coping with non-determinism is to replay each subsequence chosen by delta debugging multiple times. If the non-deterministic bug occurs with probability p, we can model the probability that we will observe it within r replays as $1 - (1 - p)^r$. This 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

[9] We currently support the following invariants: (a) all-to-all reachability, (b) loop freeness, (c) blackhole freeness, (d) controller liveness, and (e) POX ACL compliance.

replay is approximately 1% if we replay 20 times per subsequence.[10]

5.2 Mitigating Non-Determinism

When non-determinism acutely impacts replay, one might also seek to mitigate or prevent non-determinism altogether. Deterministic replay techniques [12, 17], which precisely record and replay system call return values and process scheduling decisions, may be deployed to reliably produce bugs in exchange for a loss in performance. Unfortunately these techniques do not easily allow any modifications to the inputs fed to the processes during replay, since changes may cause the remaining execution to subtly differ (*e.g.* the sequence numbers of packets may all differ), and our goal is precisely to modify (eliminate) the inputs fed to the distributed system.

We therefore strived to balance resilience to non-determinism and the need to modify control software. We placed STS in a position to record and replay all network events in serial order, and ensured that all data structures within STS were unaffected by randomness. For example, we avoid using hashmaps that hash keys according to their memory address, and sort all list return values.

We also optionally interpose on the controller software itself. Routing the `gettimeofday()` syscall through STS helps ensure timer accuracy.[11][12] When sending data over multiple sockets, the operating system exhibits non-determinism in the order it schedules I/O operations. STS optionally ensures a deterministic order of messages by multiplexing all sockets onto a single true socket. On the controller side STS currently adds a shim layer atop the control software's socket library,[13] although this could be achieved transparently with a libc shim layer [17].

STS may need visibility into the control software's internal state transitions to properly maintain happens-before relations during replay. We gain visibility by making a small change to the control software's logging library[14]: whenever a control process executes a log statement, which often indicates that an important state transition is about to take place, we notify STS. Such coarse-grained visibility into internal state transitions does not handle all cases, but we find it suffices in practice.[15] We can also optionally use logging interposition as a synchronization barrier, by blocking the process when it executes crucial logging statements until STS explicitly tells the process that it may proceed.

[10] See §6.5 for an experimental evaluation of this model.

[11] 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 timestamps of neighboring events.

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

[13] Only supported for POX at the moment.

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

[15] We discuss this limitation further in §5.6.

5.3 Checkpointing

To efficiently implement the PEEK() algorithm depicted in Figure 2 we assume the ability to record checkpoints of the state of the system under test. We currently implement checkpointing for the POX controller[16] by telling it to `fork()` itself and suspend its child, cloning the sockets of the parent (which constitute shared state between the parent and child processes, since the socket state is managed by the kernel), and later resuming the child. This simple mechanism does not work for controllers that use other shared state such as disk. To handle other shared state one could checkpoint processes within lightweight Unix containers [1]. For distributed controllers, one would also need to implement a consistent cut algorithm [8], which is available in several open source implementations [3].

If developers do not choose to employ checkpointing, they can use our implementation of PEEK() that replays the prefix for each input interval from the beginning of the trace, thereby increasing replay runtime by a factor of n. Alternatively, they can avoid PEEK() and solely use the event scheduling heuristics described in §5.4.

Beyond its use in PEEK(), snapshotting has three advantages. As mentioned in §4.3, only considering events starting from a recent checkpoint rather than the beginning of the execution decreases the number of events to be minimized. By shortening the replay time, checkpointing coincidentally helps cope with the effects of non-determinism, as there is less opportunity for divergence in timing. Lastly, checkpointing can improve the runtime of delta debugging, since many of the subsequences chosen throughout delta debugging's execution share common input prefixes.

5.4 Timing Heuristics

We have found a number of heuristics useful for ensuring that invariant violations are consistently reproduced during replay. These heuristics may be used alongside or instead of PEEK(). We evaluated the effectiveness of these heuristics using visualization tools (described in §5.5) to compare replay executions with and without the heuristics enabled.

Event Scheduling. If we had perfect visibility into the internal state transitions of control software, we would be able to replay inputs at precisely the correct point in the happens-before relation. Unfortunately this is impractical.

We find that keeping the wall-clock spacing between replay events close to the recorded timing helps (but does not alone suffice) to ensure that invariant violations are consistently reproduced. When replaying events, we `sleep()` between each event for the same duration that was recorded in the original trace, less the time it takes to replay each event. Accounting for the extra time it takes to replay events is especially important when we time out on internal events, or when input events take a long time to inject.

Whitelisting keepalive messages. We observed during

[16] We only use the event scheduling heuristics described in §5.4 for the other controllers.

some of our experiments that the control software incorrectly inferred that links or switches had failed during replay, when it had not done so in the original execution. Upon further examination we found in these cases that LLDP and OpenFlow echo packets periodically sent by the control software were staying in STS's buffers too long during replay, such that the control software would time out on them. We therefore added an option to always pass through keepalive messages that mitigates the issue. The limitation of this heuristic is that it cannot be used on bugs involving keepalive messages.

Whitelisting dataplane events. Dataplane forward/drop events constitute a substantial portion of overall events. However, for many of the controller applications we are interested in, dataplane forwarding is only relevant insofar as it triggers control plane events (*e.g.* host discovery). We find that allowing dataplane forward events through by default, *i.e.* never timing out on them during replay, can greatly decrease skew between the wall-clock timing of the original vs. replayed trace.

5.5 Root Causing Tools

Throughout our experimentation with STS, we often found that minimized event traces alone were insufficient to pinpoint the root causes of bugs. We therefore implemented a number of complementary root causing tools within STS, which we use along with Unix utilities to help us complete the final stage of debugging. We illustrate in §6 how exactly we use these tools.

OFRewind. STS supports an interactive replay mode similar to OFRewind [45] that allows troubleshooters to query the state of the network throughout replay, filter subsets of the events, check additional invariants, and even induce new events that were not part of the original event trace. Similar to OFRewind, we do not run concurrent controller processes while the user is interactively performing replay, since proper replay across concurrent processes requires precise timing. Instead, STS replays the exact OpenFlow commands from the original trace to the switches, and creates mock TCP connections that drop whatever messages the switches attempt to send back to the controllers.

Packet Tracing. Especially for SDN controllers that react to flow events, we found it useful to trace the path of individual packets throughout the network. STS includes tracing instrumentation similar to NetSight [21] for this purpose.

OpenFlow Reduction. The OpenFlow commands sent by controller software are often somewhat redundant. For example, controllers may override routing entries, allow them to expire, or periodically flush the contents of flow tables and later repopulate them. STS includes a tool for filtering out such redundant messages, leaving only those commands that are directly relevant for invalid network configurations.

Event Visualization. Understanding the timing of messages and internal state transitions is a crucial part of troubleshooting distributed systems. STS includes two visualization tools designed to aid with this task. First, we include a tool to visualize space-time diagrams [28] of event traces. Second, we include a tool to visually highlight event ordering differences between two or more event traces, which is especially useful for comparing the behavior of intermediate delta debugging replays when the original trace exhibits a high degree of non-determinism.

5.6 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 relevant internal events. For some controllers this may involve substantial instrumentation effort beyond pre-existing log statements, though as we show in our evaluation, most bugs we encountered can be minimized without perfect visibility.

Non-determinism. Non-determinism is fundamental in networks. When non-determinism is present STS (i) replays multiple times per subsequence, and (ii) employs software techniques for mitigating non-determinism, but it may nonetheless output a non-minimal causal sequence. In the common case this is still better than what developers had before, since developers generally do not have tools for reproducing non-deterministic bugs. In the worst case STS leaves the developer where 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.

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

6. EVALUATION

We first demonstrate STS's viability in troubleshooting real bugs. We found seven new bugs by fuzz testing five open source SDN control platforms: ONOS [37] (Java), POX [34] (Python), NOX [20] (C++), Pyretic [16] (Python), and Floodlight [5] (Java), and debugged these with the help of STS. Second, we demonstrate the boundaries of where STS works well and where it does not by finding MCSes for previously known and synthetic bugs that span a range of bug types encountered in practice.

Our ultimate goal is to reduce developer effort spent on troubleshooting bugs. As this is difficult to measure, since developer skills and familiarity with code bases differs widely, we instead quantitatively show how well STS minimizes logs, and qualitatively relay our experience using MCSes to debug the newly found bugs.

We show a high-level overview of our results in Table 3, and illustrate in detail how STS found MCSes in the rest of this section. Interactive visualizations and replayable event traces for all of these case studies are publicly available at ucb-sts.github.com/experiments.

6.1 New Bugs

	Bug Name	Topology	Runtime (s)	Input Size	MCS Size	MCS helpful?
Newly Found	Pyretic loop	3 switch mesh	266.2	36	2	Yes
	POX premature PacketIn	4 switch mesh	249.1	102	2	Yes
	POX in-flight blackhole	2 switch mesh	641.1	46	7	Yes
	POX migration blackhole	4 switch mesh	1796.0	29	3	Yes
	NOX discovery loop	4 switch mesh	4990.9	150	18	Indirectly
	Floodlight loop	3 switch mesh	27930.6	117	13	Yes
	ONOS distributed database locking	2 switch mesh	N/A	1	1	N/A
Known	Floodlight failover bug	2 switch mesh	-	202	2	Yes
	ONOS master election	2 switch mesh	6325.2	30	3	Yes
	POX load balancer error checking	3 switch mesh	2396.7	106	24 (N+1)	Yes
Synthetic	Null pointer on rarely used codepath	20 switch FatTree	2396.7	365	2	Yes
	Overlapping flow entries	2 switch mesh	115.4	27	2	Yes
	Delicate timer interleaving	3 switch mesh	N/A	39	39	No
	Algorithm misimplementation	3 switch mesh	525.2	40	7	Indirectly
	Multithreaded race condition	10 switch mesh	36967.5	1596	2	Indirectly
	Memory leak	2 switch mesh	15022.6	719	30 (M)	Indirectly
	Memory corruption	4 switch mesh	145.7	341	2	Yes

Table 3: Overview of Case Studies.

Pyretic Loop. We discovered a loop when fuzzing Pyretic's hub module, whose purpose is to flood packets along a minimum spanning tree. After minimizing the execution (runtime in Figure 5a), we found that the triggering event was a link failure at the beginning of the trace followed some time later by the recovery of that link. After roughly 9 hours over two days of examining Pyretic's code (which was unfamiliar to us), we found what we believed to be the problem in its logic for computing minimum spanning trees: it appeared that down links weren't properly being accounted for, such that flow entries were installed along a link even though it was down. When the link recovered, a loop was created, as the flow entries were still in place. The loop seemed to persist until Pyretic periodically flushed all flow entries.

We filed a bug report along with a replayable MCS to the developers of Pyretic. They found after roughly five hours of replaying the trace with STS that Pyretic told switches to flood out all links before the entire network topology had been learned (including the down link). By adding a timer before installing entries to allow for links to be discovered, the developers were able to verify that the loop no longer appeared. A long term fix for this issue is currently being discussed by the developers of Pyretic.

POX Premature PacketIn. We discovered this bug accidentally while working on a synthetic test case. During a particular fuzzing run, the `l2_multi` module failed unexpectedly with a `KeyError`. The initial trace had 102 input events. Within 6 iterations and 249 seconds, STS reduced it to an MCS of 2 input events, as shown in Figure 5b.

To analyze the root cause, we repeatedly replayed the MCS trace while adding instrumentation to the POX code. The root cause was a race condition in POX's handshake state machine. The OpenFlow standard requires a 2-message handshake.[17] Afterwards, the switch is free to send arbitrary messages. POX, however, requires an additional series of message exchanges before considering the switch fully connected and notifying the application modules of its presence via a *SwitchUp* event.

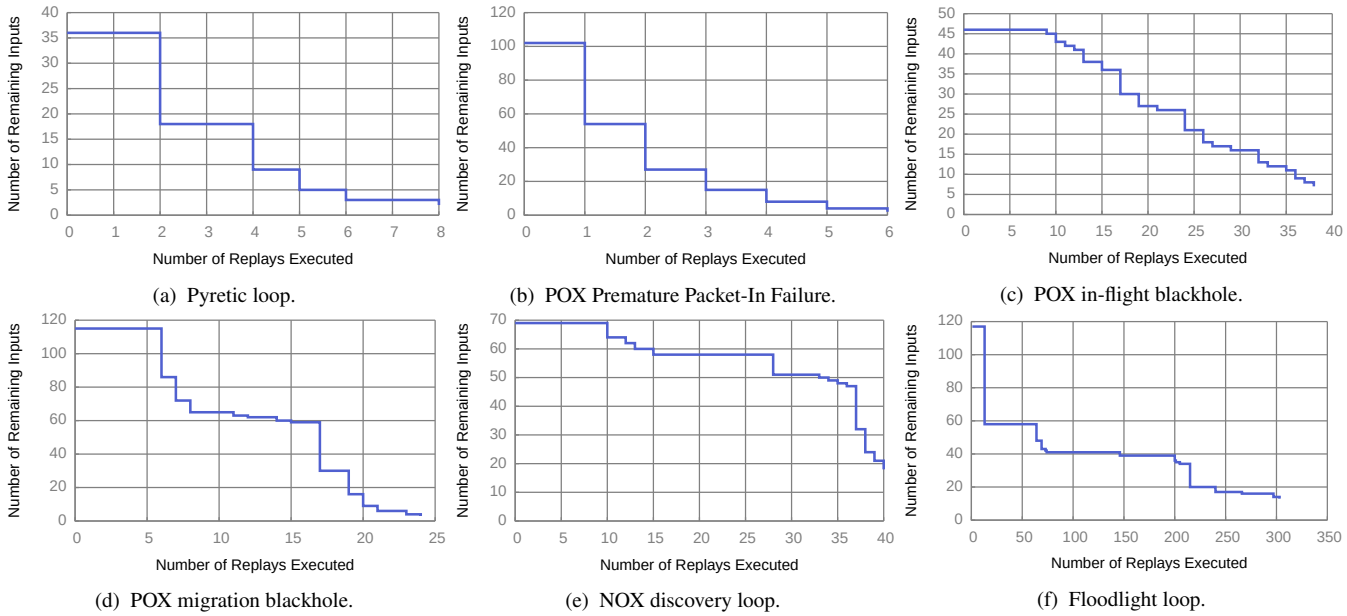
In this case, the switch was slow in completing the second part of the handshake, causing the *SwitchUp* to be delayed. During this window, a *PacketIn* event related to an LLDP packet was forwarded to POX's *discovery* module, which in turned raised a *LinkEvent* to `l2_multi`, which would then fail because it expected *SwitchUp* to occur first. We verified with the lead developer of POX that this was indeed a bug.

This case study demonstrates how even a simple handshake state machine can behave unexpectedly and in a non-trivial manner that is hard to understand without being able to repeat the experiment with a minimal trace. Making heavy use of the MCS replay, a developer unfamiliar with the two subsystems was able to root-cause the bug in ~30 minutes.

POX In-flight Blackhole. We discovered another bug after roughly 20 runs of randomly generated inputs. We noticed a persistent blackhole while POX was bootstrapping its discovery of link and host locations. There were 46 inputs in the initial trace. The initial trace was affected by non-determinism and only replayed successfully 15/20 times. We were able to reliably replay it by employing multiplexed sockets, overriding `gettimeofday()`, and waiting on POX's logging messages. STS returned a 7 input MCS (runtime shown in Figure 5c).

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 port belonged to a switch-switch link—apparently

[17] `ofp_hello` and `ofp_feature_(request|reply)`



(a) Pyretic loop.

(b) POX Premature Packet-In Failure.

(c) POX in-flight blackhole.

(d) POX migration blackhole.

(e) NOX discovery loop.

(f) Floodlight loop.

Figure 5: Minimization results.

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.

POX Migration Blackhole. We noticed after examining POX's code that there might be some 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 115 inputs. STS produced a 3 input MCS (shown in Figure 5d): a packet injection from a host ('A'), followed by a packet injection by another host ('B') towards A, followed by a host migration of 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.

NOX Discovery Loop. Next 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. Our approach to debugging was to reconstruct from the minimized trace how NOX should have installed routes, then compare how NOX actually installed routes. This case took us roughly 10 hours to debug. Unfortunately the final MCS did not reproduce the bug on the first few tries, and we suspect this is due to the fact NOX chooses the order to send LLDP messages randomly, and the loop depends crucially on this order. We instead used the console output from the shortest subsequence that did produce the bug (21 inputs, 3 more than the MCS) to debug this trace.

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. The links adjacent to the neighbor switch formed 2 of the 3 links in the loop.

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.

ONOS distributed database locking. When testing ONOS, a distributed open-source controller, we noticed that ONOS controllers would occasionally reject switch attempts to connect upon initialization. The initial trace was already minimized, as the initial input was the single event of the switches connecting to the controllers with a particular timing. When examining the logs, we found that the particular timing between the switch connects caused both ONOS controllers to encounter a "failed to obtain lock" error from their distributed graph database. We suspect that the ONOS controllers were attempting to concurrently insert the same key, which causes a known error. We modified ONOS's initialization logic to retry when inserting switches, and found that this eliminated the bug.

Floodlight loop. Next we tested Floodlight's routing application. In about 30 minutes, our fuzzing uncovered a 117 input sequence that caused a persistent 3-node forwarding

loop. In this case, the controller exhibited significant non-determinism, which initially precluded STS from efficiently reducing the input size. We worked around this by increasing the number of replays per subsequence to 10. With this, STS reduced the sequence to 13 input events in 324 replays and 8.5 hours (runtime shown in Figure 5f).

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 techniques: by repeatedly replaying a significantly minimized MCS, we were able to diagnose the root cause—a complex interaction between the LinkDiscovery, Forwarding, and DeviceManager modules.

6.2 Known bugs

In addition to our troubleshooting case studies, we evaluate STS's ability to minimize traces on a range of bug types, both known and synthetically injected by us.

Floodlight failover bug. We were able to reproduce a known problem in Floodlight's distributed controller failover logic [14] with STS. In Floodlight 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 change the switch configurations. If a link fails shortly after the master controller has died, all live controllers are in the backup role and will not take responsibility for updating the switch flow table. At some point when a backup notices the master failure and elevates itself to the master role it will proceed to manage the switch, but without ever clearing the routing entries for the failed link, resulting in a persistent blackhole.

We ran two Floodlight controller instances connected to two switches, and injected 200 extraneous link and switch failures, with the controller crash and switch connect event[18] that triggered the blackhole interleaved among them. We were able to successfully isolate the two-event MCS: the controller crash and the link failure.

ONOS master election bug. We reproduced another bug, previously reported in earlier versions and later fixed, in ONOS's master election protocol. If two adjacent switches are connected to two separate controllers, the controllers must decide between themselves who will be responsible for tracking the liveness of the link. They make this decision by electing the controller with the higher ID as the master for that link. When the master dies, and later reboots, it is assigned a new ID. If its new ID is lower than the other controllers', both will incorrectly believe that they are not responsible for tracking the liveness of the link, and the controller with the prior higher ID will incorrectly mark the link as unusable such that no routes will traverse it. This bug depends on initial IDs chosen at random, and ONOS is not instrumented to support deterministic replay of random values. We mitigated this inherent non-determinism by replaying each subsequence 5 times. With this setting, STS was able to minimize the trace to 3 elements.

POX load balancer error checking. We are aware that POX applications do not always check error messages sent by switches rejecting invalid packet forwarding commands. We used this to trigger a bug in POX's load balancer application: we created a network where switches had only 25 entries in their flow table, and proceeded to continue injecting TCP flows into the network. The load balancer application proceeded to install entries for each of these flows. Eventually the switches ran out of flow entry space and responded with error messages. As a result, POX began randomly load balancing each subsequent packet for a given flow over the servers, causing session state to be lost. We were able to minimize the MCS for this bug to 24 elements (two auxiliary flow entries, plus 22 flows needed to overflow the routing tables). A notable aspect of this MCS is that its size is directly proportional to the flow table space, and developers would find across multiple fuzz runs that the MCS was always 24 elements.

6.3 Synthetic bugs

Lastly, we injected synthetic bugs across a range of bug types into POX. For space reasons we only briefly describe these bugs.

Delicate timer interleaving. We injected a crash on a code path that was highly dependent on the interleaving of internal timers triggered within POX. This is a particularly hard case for STS, since we have little control of internal timers. We were able to trigger the code path during fuzzing, but were unable to reproduce the bug during replay after five attempts, and were left with the original 39 input trace. This is the only case where we were unable to replay trace.

Algorithm misimplementation. We modified POX's implementation of Floyd-Warshall to create loops. We noticed that the MCS was inflated by at least two events: a link failure and a link recovery that we did not believe were relevant to triggering the bug we induced. The final MCS also was not replayable on the first try. We suspect that these problems may have been introduced by the fact that the routing implementation depended on the discovery module to find links in the network, and the order in which these links are discovered is non-deterministic.

Overlapping flow entries. We ran two modules in POX: a capability manager in charge of providing upstream DoS protection for servers, and a forwarding application. The capabilities manager installed drop rules upstream for servers

[18] We used a switch connect event rather than a link failure event for logistical reasons, but both can trigger the race condition

that requested it, but these rules had lower priority than the default forwarding rules in the switch. We were able to minimize 27 inputs to the two traffic injection inputs necessary to trigger the routing entry overlap.

Null pointer on rarely used codepath. On a rarely used-code path, we injected a null pointer exception, and were able to successfully minimize a fuzz trace of 365 events to the expected conditions that triggered that code path: a switch failure followed by a switch recovery.

Multithreaded race condition. We created a race condition between multiple threads that was triggered by any packet I/O, regardless of input. With 5 replays per subsequence, we were able to minimize a 1596 input in 10 hours 15 minutes to a replayable 2 element failure/recovery pair as an MCS. The MCS itself though may have been somewhat misleading to a developer (as expected), as the race condition was triggered randomly by any I/O, not just these two inputs events.

Memory leak. We created a case that would take STS very long to minimize: a memory leak that eventually caused a crash in POX. We artificially set the memory leak to happen quickly after allocating 30 (M) objects created upon switch handshakes, and interpersed 691 other input events throughout switch reconnect events. The final MCS found after 4 hours 15 minutes was exactly 30 events, but it was not replayable. We suspect this was because STS was timing out on some expected internal events, which caused POX to reject later switch connection attempts.

Memory corruption. We simulated a case where the receipt of link failure notification on a particular port causes corruption to one of POX's internal data structures. This corruption then causes a crash much later when the data structure is accessed during the during corresponding port up. These bugs are often hard to debug, because considerable time can pass between the event corrupting the data structure and the event triggering the crash, making manual log inspection or source level debugging ineffective. STS proved effective in this case, reducing a larger trace to exactly the 2 events responsible for the crash.

6.4 Overall Results and Discussion

We show our overall results in Table 3. We note that with the exception of 'Delicate timer interleaving', STS was able to significantly reduce the size of the input traces. As described in the case studies, we were able to counter some sources of non-determinism by replaying multiple times per subsequence and adding instrumentation to controllers.

The cases where STS was most useful were those where a developer would have started from the end of the trace and worked backwards, but the actual root cause lies many events in the past (*e.g.* the Memory corruption example). This requires many re-iterations through the code and logs using standard debugging tools (*e.g.* source level debuggers), and is highly tedious on human timescales. In contrast, it was easy to step through a small event trace and manually identify the code paths responsible for a failure.

Max replays per subsequence	Size of final MCS	Total hours
1	65	6.10
2	20	6.37
3	15	7.78
4	12	9.59
5	9	6.38
6	9	11.20
7	9	11.83
8	6	12.35
9	6	11.13
10	6	12.86

Table 4: Effectiveness of replaying subsequences multiple times in mitigating non-determinism.

Bugs that depend on fine-grained thread-interleaving or timers inside of the controller are the worst-case for STS. This is not surprising, as they do not directly depend on the input events from the network, and we do not directly control the internal scheduling and timing of the controllers. The fact that STS has a difficult time reducing these traces is itself indication to the developer that fine-grained non-determinism is at play.

6.5 Coping with Non-determinism

Recall that STS optionally replays each subsequence multiple times throughout delta debugging to mitigate the effects of non-determinism. We evaluate the effectiveness of this approach on the minimization of a synthetic non-deterministic loop created by Floodlight. Table 4 demonstrates that the size of the resulting MCS decreases with the number of replays per subsequence. This suggests that replaying each subsequence multiple times is effective in coping with non-determinism, at the cost of increased runtime.

6.6 Instrumentation Complexity

For POX and Floodlight, we added shim layers to the controller software to redirect `gettimeofday()`, interpose on logging statements, and demultiplex sockets. For Floodlight we needed 722 lines of Java to obtain this indirection, and for POX we needed 415 lines of Python.

6.7 Scalability

Mocking the network 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 exist. On a machine with 6GB of memory, we ran 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 6, 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

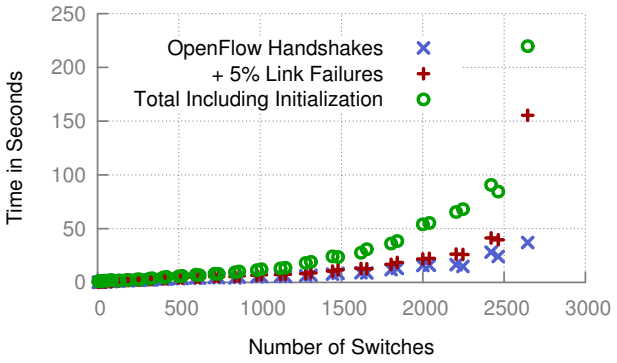


Figure 6: Simulation time for bootstrapping FatTree networks, cutting 5% of links, and processing the controller's response.

removed by running on a machine with >6GB of 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, except perhaps for host discovery.

6.8 Parameters

We found throughout our experimentation that STS leaves open several parameters that need to be set properly in order to effectively find and troubleshoot bugs.

Setting fuzzing parameters. STS's fuzzer allows the user to set the rates different event types are triggered at. In our experiments with STS we found several times that we needed to set these parameters such that we avoided bugs that were not of interest to developers. For example, in one case we discovered that a high dataplane packet drop rate dropped too many LLDP packets, preventing the controller from successfully discovering the topology. Setting fuzzing parameters remains an important part of experiment setup.

Differentiating persistent and transient violations. In networks there is a fundamental delay between the initial occurrence of an event and the time when other nodes are notified of the event. This delay implies that invariant violations such as loops or blackholes can appear before the controller(s) have time to correct the network configuration. In many cases such transient invariant violations are not of interest to developers. We therefore provide a threshold parameter in STS for how long a invariant violation should persist before STS reports it as a problem. In general, setting this threshold depends on the network and the invariants of interest.

Setting ε. 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 bug. 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.

7. DISCUSSION & FUTURE WORK

How much effort do MCSes really save? Based on our experience and conversations with engineers, two facts seem to hold. First, companies dedicate a substantial portion of their best engineers' time on troubleshooting bugs. Second, the larger the trace, the more effort is spent on debugging, as humans can only keep a small number of things in working memory [36]. If one were to consider the task of debugging a large input trace, a common approach taken by engineers is in fact similar to the divide and conquer approach taken by delta debugging which is $O(n^2)$ in the worst case for n inputs, yet human time is highly valuable.

Will this approach work on all controllers? We make limited assumptions about the controller software in use. Three of the five platforms we investigated were exercised with STS without any modifications. Limited changes to the controller platforms (*e.g.* overriding `gettimeofday()`) can increase replay accuracy further. In general, we expect STS to support controllers conforming to OpenFlow 1.0.

Why do you focus on SDN? SDN represents both an opportunity and a challenge. In terms of a challenge, SDN control software—both proprietary and open source—is in its infancy, which means that bugs are pervasive.

In terms of an opportunity, SDN's architecture facilitates the implementation of systems like STS. The interfaces between components of the system (*e.g.* OpenFlow for switches [38] and OpenStack Neutron for management [2]) are well-defined, which is crucial for codifying functional equivalencies. Moreover, the control flow of SDN control software repeatedly returns to a quiescent state after processing inputs, which means that many inputs can be pruned.

Although we focus on troubleshooting SDN control software, we believe that our general approach can be applied to other distributed systems. In future work we hope to validate our technique on other such systems.

Dataflow analysis. Another avenue for future work would be to apply dataflow analysis [29,42] to further prune events. The simplest form of dataflow analysis involves reasoning about happens-before relations: if you consider a software-defined network as a distributed state machine, then one can safely prune an input event if it does not induce any messages before the occurrence of the invalid configuration [28]. Unfortunately we found that pruning inputs in this way does not significantly reduce the number of inputs, but one might go further by analyzing the control software itself.

Optimizing delta debugging. Delta debugging is agnostic to how input sequences are split. Currently we split by time, but it may be more effective to split events by type or by location in the network topology.

Enabling analysis of production logs. STS does not currently support minimization of production (as opposed to QA) logs. Production systems would need to include Lamport clocks on each message [28] or have sufficiently accurate clock synchronization to obtain a happens-before relation. Inputs would also need to need to be logged in suffi-

cient detail for STS to replay a synthetic version. Finally, without care, a single input event may appear multiple times in the distributed logs. 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. RELATED WORK

Our goal is to transform event traces into minimal sequences of inputs. To achieve this we are the first to our knowledge to extend delta debugging [47,48] to a distributed system where there is an ongoing sequence of inputs involving multiple entities rather than a single file fed to a single entity. Delta debugging has been applied to minimize thread context switches leading up to race conditions [10] and to prune irrelevant threads from execution traces [22], but these applications still involve injecting a single input (the thread schedule) to a single entity (the scheduler).

We characterize the other approaches taken by the troubleshooting literature as (i) program flow analysis (ii) instrumentation (tracing), (iii) bug detection (invariant checking), (iv) replay (of buggy executions), and (v) root cause analysis (of network device failures).

Program Flow Analysis. Other than delta debugging, the closest work to ours involves reasoning about control- and dataflow dependencies (which may be recorded at runtime [29], or dynamically inferred [22, 42]) in order to reduce the length of deterministic replay executions. These techniques are tied to a single language because of the extensive effort required for instrumentation and annotation. We avoid making assumptions about the language or instrumentation of the controller software. Moreover, our application of functional equivalence to the space of possible inputs allows us to minimize inputs more aggressively, whereas they are forced to consider all controlflow and dataflow dependencies in the control software.

Instrumentation. Unstructured log files collected at each node are the most common form of diagnostic information. The goal of tracing frameworks [4,9,15,21,40] is to produce structured logs that can be easily analyzed, such as DAGs tracking the events triggered by requests passing through the distributed system. An example within the SDN problem space is NetSight [21], which allows users to retroactively examine the paths dataplane packets take through OpenFlow networks. Tools like NetSight allow developers to understand how, when, and where the dataplane broke. In contrast, we focus on making it easier for developers to understand why the control software misconfigured the network.

Bug Detection. With instrumentation available, it becomes possible to check expectations about the system's state (either offline [31] or online [11, 32]), or about the paths requests take through the system [40]. Within the networking community, this research is primarily focused on verifying routing tables [24–26, 33] or forwarding behavior [49, 50]. We use bug detection techniques (invariant checking) to guide delta debugging's minimization process.

It is also possible to infer performance anomalies by building probabilistic models from collections of traces [4, 9]. Our goal is to produce exact minimal causal sequences without depending on probabilistic models, and we are primarily focused on correctness instead of performance.

Model checkers [6, 27] seek to proactively find bugs by enumerating all possible code paths. After identifying a bug with model checking, finding a minimal code path leading to it is straightforward. However, model checking suffers from exponential state explosion when run on large systems. For example, NICE [6] took 30 hours to model check a network with two switches, two hosts, the NOX MAC-learning control program (98 LoC), and five concurrent messages between the hosts. Rather than exploring all possibilities, we discover bugs through testing of selected scenarios, which makes our problem tractable.

Replay. Crucial diagnostic information is often missing from traces. Record and replay techniques [17, 30] instead allow users to step through (deterministic) executions and interactively examine the state of the system in exchange for runtime recording overhead. Within SDN, OFRewind [45] provides record and replay of OpenFlow channels between controllers and switches. Manually examining long system executions can be tedious, and our goal is to minimize such executions so that developers find it easier to identify the problematic code through replay or other means.

Root Cause Analysis. Without perfect instrumentation, it is often not possible to know exactly what events are occurring (*e.g.* which network components have failed) in a distributed system. Root cause analysis [23, 46] seeks to reconstruct those unknown events from limited monitoring data. Here we know exactly which events occurred, but seek to identify the minimal sequence of events that trigger a particular bug.

It is worth mentioning another goal outside the purview of distributed systems, but closely in line with ours: program slicing [44] is a technique for finding the minimal subset of a program that could possibly affect the result of a particular line of code. Our goal is to slice the temporal dimension of an execution rather than the code dimension.

9. CONCLUSION

SDN's purpose is to make networks easier to manage. 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 control software makes network management easier, but only by forcing the developers of SDN control software to confront the challenges of asynchrony, partial failure, and other notoriously hard problems 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. Here we developed a technique for automatically identify-

ing 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.

We focused on SDN control software, but we believe our techniques are applicable to general distributed systems. As distributed systems proliferate, we hope that our technique helps ameliorate the dearth of tools in this important area.

10. REFERENCES

[1] Linux kernel containers. linuxcontainers.org.

[2] OpenStack Neutron. https://wiki.openstack.org/wiki/Neutron.

[3] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. IPDPS '09.

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

[5] BigSwitch Networks. http://www.bigswitch.com/.

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

[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] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. DSN '02.

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

[11] D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live Debugging of Distributed Systems. CC 09'.

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

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

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

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. ICFP '11.

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

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

[19] 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.

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

[21] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. NSDI '14.

[22] J. Huang and C. Zhang. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. OOPSLA '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, M. Change, H. Zheng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. NSDI '13.

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

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

[27] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. NSDI '07.

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

[29] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward Generating Reducible Replay Logs. PLDI '11.

[30] C.-C. Lin, V. Jalaparti, M. Caesar, and J. Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. ATC '13.

[31] X. Liu. WiDs Checker: combating Bugs in Distributed Systems. NSDI '07.

[32] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: Debugging Deployed Distributed Systems. NSDI '08.

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

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

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

[36] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[37] ON.Lab. Open Networking Operating System. http://onlab.us/tools.html.

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

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

[40] 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.

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

[42] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction. ISSTA '07.

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

[44] M. Weiser. Program Slicing. ICSE '81.

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

[46] 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.

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

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

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

[50] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. NSDI '14.