Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences

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

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, without making assumptions about the language or instrumentation of the software under test. 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.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

Keywords

Test case minimization; Troubleshooting; SDN control software

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 code) is highly time-consuming, as developers spend hours poring over multigigabyte execution traces.[1] Our aim is to reduce effort spent on troubleshooting distributed systems like SDN control software, by automatically eliminating events from buggy traces that are not causally related to the bug, producing a "minimal causal sequence" (MCS) of triggering events.

Our goal of minimizing traces is in the spirit of delta debugging [58], but our problem is complicated by the distributed nature of control software: our input is not a single file fed to a single point of execution, but an ongoing sequence of events involving multiple actors. We therefore need to carefully control the interleaving of events in the face of asynchrony, concurrency and non-determinism in order to reproduce bugs throughout the minimization process. Crucially, we aim to minimize traces without making assumptions about the language or instrumentation of the control software.

We have built a troubleshooting system that, as far as we know, is the first to meet these challenges (as we discuss further in §8). Once it reduces a given execution trace 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 troubleshooting system, which we call STS (SDN Troubleshooting System), consists of 23,000 lines of Python, and is designed so that organizations can implement the technology within their existing QA infrastructure (discussed in §5); over the last year we have worked with a commercial SDN company to integrate STS. We evaluate STS in two ways. First and most significantly, we use STS to troubleshoot seven previously unknown bugs—involving concurrent events, faulty failover logic, broken state machines, and deadlock in a distributed database—that we found by fuzz testing five controllers (Floodlight [16], NOX [23], POX [39], Pyretic [19], ONOS [43]) 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 (non-synthetic) bug traces by up to 98%, and we anecdotally found that reducing traces to MCSes made it easy to understand their root causes.

[1]Software developers in general spend roughly half (49% according to one study [21]) 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).

SIGCOMM'14, August 17–22, 2014, Chicago, Illinois, USA.

ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626304.

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 (*e.g.* link failure notifications) and statistics from switches (either physical or virtual), policy changes via a management interface, and possibly dataplane packets. In response, the controllers issue forwarding instructions to switches. All input events are asynchronous, and individual controllers may fail at any time. The controllers either communicate with each other over the dataplane network, or use a separate dedicated network, and may become partitioned.

The goal of the 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 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), 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 policy 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 (input) events such as failures 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 external events and internal monitoring data), since QA traces contain a wealth of extraneous events. Consider that an hour long QA test emulating event rates observed in production could contain 8.5 network error events per minute [22] and 500 VM migrations per hour [49], 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 control software is a system consisting of one or more controller processes that takes a sequence of external network events $E = e_1 \rightarrow e_2 \rightarrow \cdots e_m$ (*e.g.* link failures) as inputs, 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, *e.g.* loop-freedom). We say that configuration c violates the invariant if $P(c)$ is false, denoted $\overline{P}(c)$.

We are given a log L generated by a centralized QA test orchestrator.[3] The log L contains a sequence of events

$$\tau_L = e_1 \rightarrow i_1 \rightarrow i_2 \rightarrow e_2 \rightarrow \cdots e_m \rightarrow \cdots i_p$$

which includes external events $E_L = e_1, e_2 \cdots e_m$ injected by the orchestrator, and internal events $I_L = i_1, i_2 \cdots i_p$ triggered by the control software (*e.g.* OpenFlow messages). The events E_L include timestamps $\{(e_k, t_k)\}$ from the orchestrator's clock.

[2]This does not preclude us from troubleshooting misspecified policies so long as test invariants [31] are specified separately.

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

A replay of log L involves replaying the external events E_L, possibly taking into account the occurrence of internal events I_L as observed by the orchestrator. We denote a replay attempt by $replay(\tau)$. The output of $replay$ is a sequence of configurations $C_R = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$. Ideally $replay(\tau_L)$ reproduces the original configuration sequence, but this does not always hold.

If the configuration sequence $C_L = c_1, c_2, \ldots, c_n$ associated with the log L violated predicate P (*i.e.* $\exists_{c_i \in C_L}.\overline{P}(c_i)$) then we say $replay(\cdot) = C_R$ reproduces that violation if C_R contains an equivalent faulty configuration (*i.e.* $\exists_{\hat{c}_i \in C_R}.\overline{P}(\hat{c}_i)$).

The goal of our work is, when given a log L that exhibited an invariant violation,[3] to find a small, replayable sequence of events that reproduces that invariant violation. Formally, we define a minimal causal sequence (MCS) to be a sequence τ_M where the external events $E_M \in \tau_M$ are a subsequence of E_L such that $replay(\tau_M)$ reproduces the invariant violation, but for all proper subsequences E_N of E_M there is no sequence τ_N such that $replay(\tau_N)$ reproduces the violation. Note that an MCS is not necessarily *globally* minimal, in that there could be smaller subsequences of E_L that reproduce this violation, but are not a subsequence of this MCS.

We find approximate MCSes by deciding which external events to eliminate and, more importantly, when to inject external events. We describe this process in the next section.

4. MINIMIZING TRACES

Given a log L generated from testing infrastructure,[3] our goal is to find an approximate MCS, so that a human can examine the MCS rather than the full log. This involves two tasks: searching through subsequences of E_L, and deciding when to inject external events for each subsequence so that, whenever possible, the invariant violation is retriggered.

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 employing the delta debugging algorithm [58], a divide-and-conquer algorithm for isolating fault-inducing inputs. 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 we implement 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

[4]Handling invalid inputs is crucial for ensuring that the delta debugging algorithm finds a minimal causal subsequence. The algorithm we employ [58] 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

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 E_S of E_L is insufficient for finding MCSes: the timing of when we inject the external events during $replay$ is crucial for reproducing violations.

Existing Approaches. The most natural approach to scheduling external events is to maintain the original wall-clock timing intervals between them. If this is able to find all minimization opportunities, *i.e.* reproduce the violation for all subsequences that are a supersequence of some MCS, we say that the inputs are isolated. The original applications of delta debugging [6,47,58,59] make this assumption (where a single input is fed to a single program), as well as QuickCheck's input "shrinking" [12] when applied to blackbox systems like synchronous telecommunications protocols [4].

We tried this approach, but were rarely able to reproduce invariant violations. As our case studies demonstrate (§6), this is largely due to the concurrent, asynchronous nature of distributed systems; consider that the network can reorder or delay messages, or that controllers may process multiple inputs simultaneously. Inputs injected according to wall-clock time are not guaranteed to coincide correctly with the current state of the control software.

We must therefore consider the control software's internal events. To deterministically reproduce bugs, we would need visibility into every I/O request and response (*e.g.* clock values or socket reads), as well as all thread scheduling decisions for each controller. This information is the starting point for techniques that seek to minimize thread interleavings leading up to race conditions. These approaches involve iteratively feeding a single input (the thread schedule) to a single entity (a deterministic scheduler) [11, 13, 28], or statically analyzing feasible thread schedules [26].

A crucial constraint of these approaches is that they must keep the inputs fixed; that is, behavior must depend uniquely on the thread schedule. Otherwise, the controllers may take a divergent code path. If this occurs some processes might issue a previously unobserved I/O request, and the replayer will not have a recorded response; worse yet, a divergent process might deschedule itself at a different point than it did originally, so that the remainder of the recorded thread schedule is unusable to the replayer.

Because they keep the inputs fixed, these approaches strive for a subtly different goal than ours: minimizing thread context switches rather than input events. At best, these approaches can indirectly minimize input events by truncating individual thread executions.

With additional information obtained by program flow analysis [27, 34, 50] however, the inputs no longer need to be fixed. The internal events considered by these program flow reduction techniques are individual instructions executed by the programs (obtained by instrumenting the language runtime), in addition to I/O responses and the thread schedule. With this information they can compute program flow dependencies, and thereby remove input events from anywhere in the trace as long as they can prove that doing so cannot possibly cause the faulty execution path to diverge.

While program flow reduction is able to minimize inputs, these techniques are not able to explore alternate code paths that still trigger the invariant violation. They are also overly conservative in removing inputs (*e.g.* EFF takes the transitive closure of all possible dependencies [34]) causing them to miss opportunities to remove dependencies that actually semantically commute.

paper [59] 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 [59].

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.

Allowing Divergence. Our approach is to allow processes to proceed along divergent paths rather than recording all low-level I/O and thread scheduling decisions. This has several advantages. Unlike the other approaches, we can find shorter alternate code paths that still trigger the invariant violation. Previous *best-effort* execution minimization techniques [14, 53] also allow alternate code paths, but do not systematically consider concurrency and asynchrony.[6] We also avoid the performance overhead of recording all I/O requests and later replaying them (*e.g.* EFF incurs ~10x slowdown during replay [34]). Lastly, we avoid the extensive effort required to instrument the control software's language runtime, needed by the other approaches to implement a deterministic thread scheduler, interpose on syscalls, or perform program flow analysis. By avoiding assumptions about the language of the control software, we were able to easily apply our system to five different control platforms written in three different languages.

Accounting for Interleavings. To reproduce the invariant violation (whenever E_S is a supersequence of an MCS) we need to inject each input event e only after all other events, including internal events, that precede it in the happens-before relation [33] from the original execution ($\{i \mid i \rightarrow e\}$) have occurred [51].

The internal events we consider are (a) message delivery events, either between controllers (*e.g.* database synchronization messages) or between controllers and switches (*e.g.* OpenFlow messages), and (b) state transitions within controllers (*e.g.* a backup node deciding to become master). Our replay orchestrator obtains visibility into (a) by interposing on all messages within the test environment (to be described in §5). It optionally obtains partial visibility into (b) by instrumenting controller software with a simple interposition layer (to be described in §5.2).

Given a subsequence E_S, our goal is to find an execution that obeys the original happens-before relation. We do not control the occurrence of internal events, but we can manipulate when they are delivered through our interposition layer,[7] and we also decide when to inject the external events E_S. The key challenges in choosing a schedule stem from the fact that the original execution has been modified: internal events may differ syntactically, some expected internal events may no longer occur, and new internal events may occur 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. For example, `flow_mod` messages may cause switches to make the same change to their forwarding behavior even if their transaction ids differ.

We apply this observation by defining masks over semantically extraneous fields of internal events.[8] We show the fields we mask

[6]PRES explores alternate code paths in best-effort replay of multithreaded executions, but does not minimize executions [45].

[7]In this way we totally order messages. Without interposition on process scheduling however, the system may still be concurrent.

[8]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 $test(T_{✘}) = ✘$. Output: $T'_{✘} = ddmin(T_{✘})$ s.t. $T'_{✘} \subseteq T_{✘}$, $test(T'_{✘}) = ✘$, and $T'_{✘}$ is minimal.

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

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

where $test(T)$ denotes the state of the system after executing the trace T, ✘ denotes an invariant 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 [58]. $\subseteq$ and $\subset$ denote subsequence relations.

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($input\ subsequence$)
  $inferred \leftarrow [\,]$
  for e_i *in subsequence*
    checkpoint system
    inject e_i
    $\Delta \leftarrow |e_{i+1}.time - e_i.time| + \epsilon$
    record events for Δ *seconds*
    $matched \leftarrow original\ events\ \&\ recorded\ events$
    $inferred \leftarrow inferred + [e_i] + matched$
    restore checkpoint
  return $inferred$

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

in Table 1. Note that these masks only need to be specified once, and can later be applied programmatically.

We then consider an internal event i' observed in $replay$ 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 that “happen before” some external input may be absent when replaying a subsequence. For instance, if we prune a link failure, the corresponding notification message will not arise.

To avoid waiting forever we 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[9] 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.[10]

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

[10]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.

Handling New Internal Events. The last possible induced change 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 state transition suffix that leads to the violation, we must recursively branch, trying both possibilities for every new internal event. This implies an exponential worst case number of possibilities to be explored.

Exponential search over these possibilities is not a practical option. Our heuristic 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 state transition suffixes that contain a subsequence of the (equivalent) original internal events, but leaves open the possibility of finding divergent suffixes that lead to the invariant violation.

Recap. We combine these heuristics to replay each subsequence chosen by delta debugging: we compute functional equivalency for all internal events intercepted by our test orchestrator's interposition layer (§5), we invoke PEEK() to infer absent internal events, and with these inferred causal dependencies we $replay$ the input subsequence, waiting to inject each input until each of its (functionally equivalent) predecessors have occurred while allowing new internal events through the interposition layer immediately.

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 [58]. 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

Thus far we have assumed that we are given a faulty execution trace. We now provide an overview of how we obtain traces, and then describe our system for minimizing them.

Obtaining Traces. All three of the commercial SDN companies

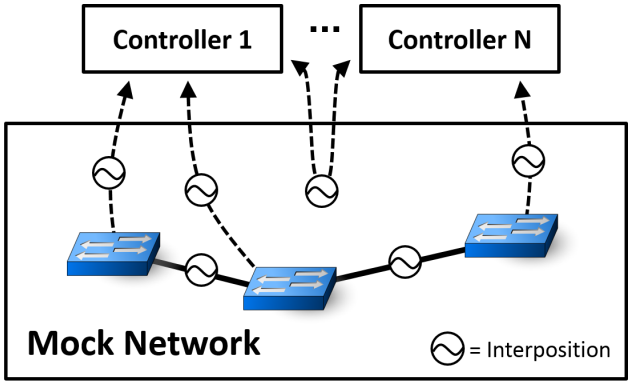


Figure 3: STS runs mock network devices, and interposes on all communication channels.

that we know of employ a team of QA engineers to fuzz test their control software on network testbeds. This fuzz testing infrastructure consists of the control software under test, the network testbed (which may be software or hardware), and a centralized test orchestrator that chooses input sequences, drives the behavior of the testbed, and periodically checks invariants.

We do not have access to such a QA testbed, and instead built our own. Our testbed mocks out the control plane behavior of network devices in lightweight software switches and hosts (with support for minimal dataplane forwarding). We then run the control software on top of this mock network and connect the switches to the controller(s). The mock network manages the execution of events from a single location, which allows it to record a serial event ordering. This design is similar to production software QA testbeds, and is depicted in Figure 3. One distinguishing feature of our design is that the mock network interposes on all communication channels, allowing it to delay or drop messages to induce failure modes that might be seen in real, asynchronous networks.

We use our mock network to find bugs in control software. Most commonly we generate random input sequences based on event probabilities that we assign (*cf.* §6.8), and periodically check invariants on the network state.[11] We also run the mock network interactively so that we can examine the state of the network and manually induce event orderings that we believe may trigger bugs.

Performing Minimization. After discovering an invariant violation, we invoke delta debugging to minimize the recorded trace. We use the testing infrastructure itself to $replay$ each intermediate subsequence. During $replay$ the mock network enforces event orderings as needed to maintain the original happens-before relation, by using its interposition on message channels to manage the order (functionally equivalent) messages are let through, and waiting until the appropriate time to inject inputs. For example, if the original trace included a link failure preceded by the arrival of a heartbeat message, during $replay$ the mock network waits until it observes a functionally equivalent ping probe to arrive, allows the probe through, then tells the switch to fail its link.

STS is our realization of this system, implemented in more than 23,000 lines of Python in addition to the Hassel network invariant checking library [31]. STS also optionally makes use of Open vSwitch [46] as an interposition point between controllers. We have made the code for STS publicly available at ucb-sts.github.com/sts.

Integration With Existing Testbeds. In designing STS we aimed to make it possible for engineering organizations to implement the technology within their existing QA test infrastructure. Organizations 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 running large scale networks with the switches, middleboxes, hosts, and routing protocols they had already chosen to include in their QA testbed.

We avoid making assumptions about the language or instrumentation of the software under test in order to facilitate integration with preexisting software. Many of the heuristics we describe below are approximations that might be made more precise if we had more visibility and control over the system, *e.g.* if we could deterministically specify the thread schedule of each controller.

5.1 Coping with Non-Determinism

Non-determinism in concurrent executions 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 [15]), and asynchronous signal delivery. These sources of non-determinism can affect whether STS is able to reproduce violations during $replay$.

The QA testing frameworks we are trying to improve do not mitigate non-determinism. STS's main approach to coping with non-determinism is to $replay$ each subsequence multiple times. If the non-deterministic bug occurs with probability p, we can model[12] the probability[13] 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 replay is approximately 1% if we replay 20 times per subsequence.

5.2 Mitigating Non-Determinism

When non-determinism is acute, one might seek to prevent it altogether. However, as discussed in §4.2, deterministic replay techniques [15, 20] force the minimization process to stay on the original code path, and incur substantial performance overhead.

Short of ensuring full determinism, we place STS in a position to record and replay all network events in serial order, and ensure that all data structures within STS are 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.[14][15] 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,[16] although this could be achieved transparently with a libc shim layer [20].

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

[11] 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.

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

[13] This probability could be improved by guiding the thread schedule towards known error-prone interleavings [44, 45].

[14] 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.

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

[16] Only supported for POX at the moment.

software's logging library[15]: whenever a control process executes a log statement, which 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.[17] We can also optionally use logging interposition as a synchronization barrier, by blocking the process when it executes logging statements until STS unblocks it.

5.3 Checkpointing

To efficiently implement the PEEK() algorithm depicted in Figure 2 we assume the ability to record checkpoints (snapshots) of the state of the system under test. We currently implement checkpointing for the POX controller[18] by telling it to `fork()` itself and suspend its child, transparently cloning the sockets of the parent (which constitute shared state between the parent and child processes), 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 [9], 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 inputs from the beginning rather than a checkpoint, 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 three heuristics useful for ensuring that invariant violations are consistently reproduced. These heuristics may be used alongside or instead of PEEK().

Event Scheduling. If we had perfect visibility into the internal state transitions of control software, we could replay inputs at precisely the correct point. 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 or time out on each event.

Whitelisting keepalive messages. We observed during 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. To avoid these differences, we added an option to always pass through keepalive messages. The limitation of this heuristic is that it cannot be used on bugs involving keepalive messages.

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

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

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 in wall-clock timing.

5.5 Root Causing Tools

Throughout our experimentation with STS, we often found that MCSes alone were insufficient to pinpoint the root causes of bugs. We therefore implemented a number of complementary root causing tools, which we use along with Unix utilities to finish the debugging process. We illustrate their use in §6.

OFRewind. STS supports an interactive replay mode similar to OFRewind [56] that allows troubleshooters to query the network state, filter events, check additional invariants, and even induce new events that were not part of the original event trace.

Packet Tracing. Especially for controllers that react to flow events, we found it useful to trace the path of individual packets through the network. STS includes tracing instrumentation similar to NetSight [25] for this purpose.

OpenFlow Reduction. The OpenFlow commands sent by controller software are often redundant, *e.g.* they may override routing entries, allow them to expire, or periodically flush and later repopulate them. STS includes a tool for filtering out such redundant messages and displaying only those commands that are directly relevant to triggering invalid network configurations.

Trace Visualization. We often found it informative to visualize the ordering of message deliveries and internal state transitions. We implemented a tool to generate space-time diagrams [33], as well as a tool to highlight ordering differences between multiple traces, which is especially useful for comparing intermediate delta debugging replays in the face of acute 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 software this may require substantial instrumentation beyond preexisting log statements, though as we show in §6, 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 MCS. In the common case this is still better than the tools developers employ in practice. In the worst case STS leaves the developer where they started: an unpruned log.

Lack of Guarantees. Due to partial visibility and non-determinism, we do not provide guarantees on MCS minimality.

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.

Interposition Overhead. Performance overhead from interposing on messages may prevent STS from minimizing bugs triggered by high message rates.[19] Similarly, STS's design may prevent it from minimizing extremely large traces, as we evaluate in §6.

[19] Although this might be mitigated with time warping [24].

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 [43] (Java), POX [39] (Python), NOX [23] (C++), Pyretic [19] (Python), and Floodlight [16] (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 effort spent on troubleshooting bugs. As this is difficult to measure,[20] 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. 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

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 4a), 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, since 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. During a POX fuzz run, the `l2_multi` routing module failed unexpectedly with a `KeyError`. The initial trace had 102 input events, and STS reduced it to an MCS of 2 input events as shown in Figure 4b.

We repeatedly replayed the MCS while adding instrumentation. The root cause was a race condition in POX's handshake state machine. The OpenFlow standard requires a 2-message handshake. POX, however, requires an additional series of message exchanges before notifying applications of its presence via a *SwitchUp* event.

The switch was slow in completing the second part of the handshake, causing the *SwitchUp* to be delayed. During this window, a *PacketIn* (LLDP packet) was forwarded to POX's `discovery` module, which in turned raised a *LinkEvent* to `l2_multi`, which then failed because it expected *SwitchUp* to occur first. We verified with the lead developer of POX that is a true bug.

This case study demonstrates how even a simple handshake state machine can behave in a 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 found a persistent blackhole while POX was bootstrapping its discovery of link and host locations. There were initially 27 inputs. 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 logging messages. STS returned a 11 input MCS (runtime shown in Figure 4c).

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 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 prevented the blackhole 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 added host migrations to the randomly generated inputs and checked for blackholes. Our initial input size was 115 inputs. STS produced a 3 input MCS (shown in Figure 4d): 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 (Figure 4e). 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.

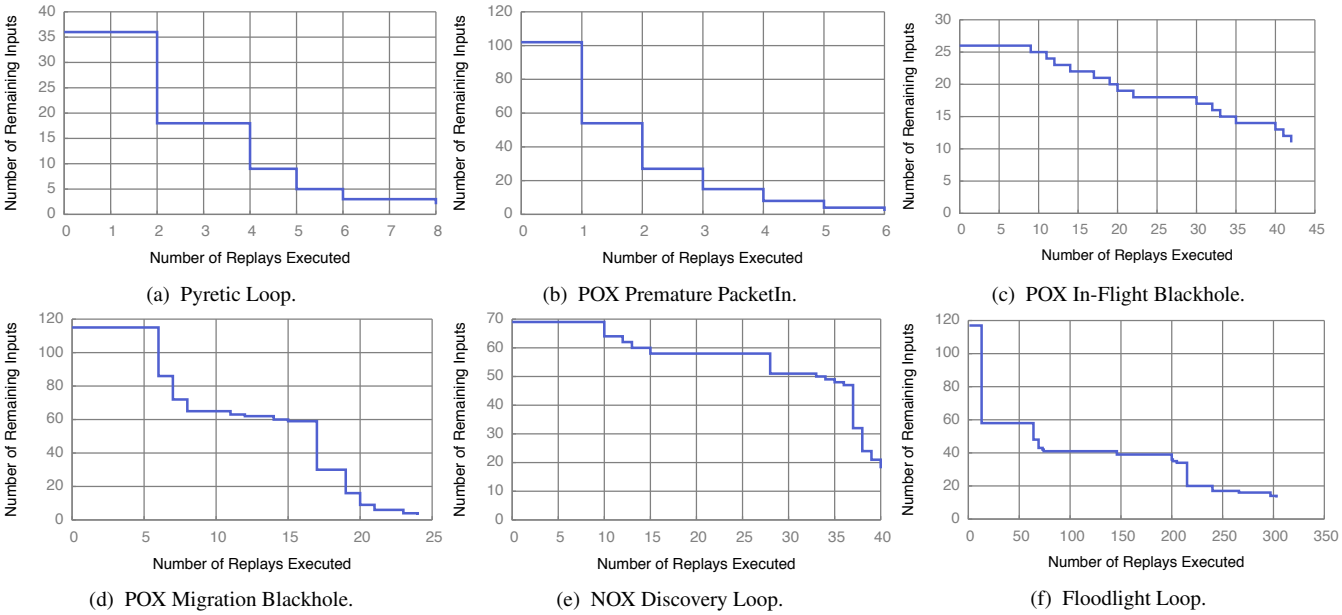
The destination host only sent one packet, which caused NOX to initially learn its correct location. After NOX flooded the packet though, it became confused about its location. One flooded packet arrived at another switch that was currently not known to be attached to anything, so NOX incorrectly concluded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

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

[20] We discuss this point further in §7.

	Bug Name	Topology	Runtime (s)	Input Size	MCS Size	MCS WI	MCS Helpful?
Newly Found	Pyretic Loop	3 switch mesh	266.2	36	1	2	Yes
	POX Premature PacketIn	4 switch mesh	249.1	102	2	NR	Yes
	POX In-Flight Blackhole	2 switch mesh	1478.9	27	11	NR	Yes
	POX Migration Blackhole	4 switch mesh	1796.0	29	3	NR	Yes
	NOX Discovery Loop	4 switch mesh	4990.9	150	18	NR	Indirectly
	Floodlight Loop	3 switch mesh	27930.6	117	13	NR	Yes
	ONOS Database Locking	2 switch mesh	N/A	1	1	1	N/A
Known	Floodlight Failover	2 switch mesh	-	202	2	-	Yes
	ONOS Master Election	2 switch mesh	2746.0	20	2	2	Yes
	POX Load Balancer	3 switch mesh	2396.7	106	24 (N+1)	26	Yes
Synthetic	Delicate Timer Interleaving	3 switch mesh	N/A	39	NR	NR	No
	Reactive Routing Trigger	3 switch mesh	525.2	40	7	2	Indirectly
	Overlapping Flow Entries	2 switch mesh	115.4	27	2	3	Yes
	Null Pointer	20 switch FatTree	157.4	62	2	2	Yes
	Multithreaded Race Condition	10 switch mesh	36967.5	1596	2	2	Indirectly
	Memory Leak	2 switch mesh	15022.6	719	32 (M+2)	33	Indirectly
	Memory Corruption	4 switch mesh	145.7	341	2	2	Yes

Table 3: Overview of Case Studies. 'WI' denotes 'Without Interposition', and 'NR' denotes 'Not Replayable'.



(a) Pyretic Loop. (b) POX Premature PacketIn. (c) POX In-Flight Blackhole.

(d) POX Migration Blackhole. (e) NOX Discovery Loop. (f) Floodlight Loop.

Figure 4: Minimization runtime behavior.

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 4f).

We repeatedly replayed the 13 event MCS while successively adding instrumentation and increasing the log level each run. After 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 the MCS, we were able to diagnose the root cause—a complex interaction between the LinkDiscovery, Forwarding, and DeviceManager modules.

ONOS Database Locking. When testing ONOS, a distributed open-source controller, we noticed that ONOS controllers would occasionally reject switches' attempts to connect. 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.

6.2 Known bugs

Floodlight Failover. We were able to reproduce a known problem [17] in Floodlight's distributed controller failover logic 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[21] 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. 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, so we modified ONOS to hardcode ID values. We were able to successfully minimize the trace to the master crash and recovery event, although we were also able to do so without any interposition on internal events.

POX Load Balancer. 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 (there were two preexisting flow entries in each routing table, so 24 additional flows made the 26 (N+1) entries needed to overflow the table). 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

Delicate Timer Interleaving. We injected a crash on a code path that was highly dependent on internal timers firing within POX. This is a 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. This is the only case where we were unable to replay the trace.

Reactive Routing Trigger. We modified POX's reactive routing module to create a loop upon receiving a particular sequence of dataplane packets. This case is difficult for two reasons: the routing module's behavior depends on the (non-deterministic) order these links are discovered in the network, and the triggering events are multiple dataplane packet arrivals interleaved at a fine granularity. We found that the 7 event 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 noticed that after PEEK() inferred expected internal events, our event scheduler still timed out on some link discovery messages–those that happened to occur during the PEEK() run but did not show up during replay due to non-determinism. We suspected that these timeouts were the cause of the inflated MCS, and confirmed our intuition by turning off interposition on internal events altogether, which yielded a 2 event MCS (although this MCS was still affected by non-determinism).

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 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 a rarely used code path we injected a null pointer exception, and were able to successfully minimize a fuzz trace of 62 events to the expected triggering conditions: control channel congestion followed by decongestion.

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 to a replayable 2 element failure/recovery pair. 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 interspersed 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 created a case where the receipt of a link failure notification on a particular port causes corruption to one of POX's internal data structures. This causes a crash much later when the data structure is accessed during the corresponding port up. These bugs are 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 & Discussion

We show our overall results in Table 3. We note that with the exception of Delicate Timer Interleaving and ONOS Database Locking, STS was able to significantly reduce input traces.

The MCS WI column, showing the MCS sizes we produced when ignoring internal events entirely, indicates that our techniques for interleaving events are often crucial. In one case however—Reactive Routing Trigger—non-determinism was particularly acute, and STS's interposition on internal events actually made minimization worse due to timeouts on inferred internal events that did not occur after PEEK(). In this case we found better results by simply turning off interposition on internal events. For all of the other case studies, either non-determinism was not problematic, or we were able to counteract it by replaying multiple times per subsequence and adding instrumentation.

The cases where STS was most useful were those where a developer would have started from the end of the trace and worked back-

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

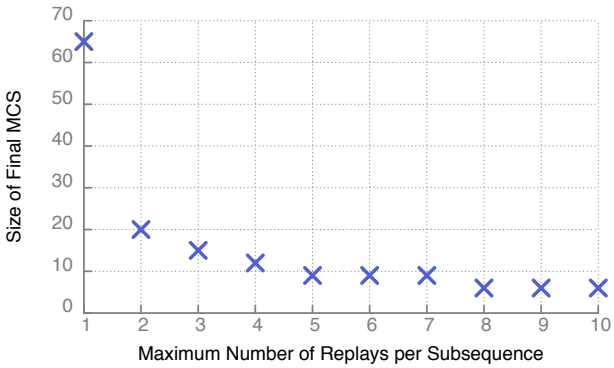


Figure 5: Effectiveness of replaying subsequences multiple times in mitigating non-determinism.

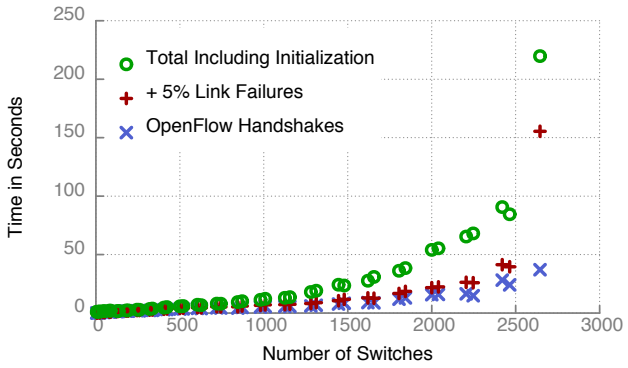


Figure 6: Runtime for bootstrapping FatTree networks, cutting 5% of links, and processing the controller's response.

wards, but the actual root cause lies many events in the past (as in Memory Corruption). 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.

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 to mitigate the effects of non-determinism. We evaluate the effectiveness of this approach by varying the maximum number of replays per subsequence while minimizing a synthetic non-deterministic loop created by Floodlight. Figure 5 demonstrates that the size of the resulting MCS decreases with the maximum number of replays, at the cost of additional runtime; 10 replays per subsequence took 12.8 total hours, versus 6.1 hours without retries.

6.6 Instrumentation Complexity

For POX and Floodlight, we added shim layers to the control software to redirect `gettimeofday()`, interpose on logging statements, and demultiplex sockets. For Floodlight we needed 722 lines of Java, 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 lie. 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 removed by running on a machine with >6GB of memory.

Note that STS is not designed for 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.

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 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 an 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 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

How much effort do MCSes really save? Based on conversations with engineers and our own industrial experience, 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, since humans can only keep a small number of facts in working memory [41]. As one developer puts it, "Automatically shrinking test cases to the minimal case is immensely helpful" [52].

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 (*e.g.* OpenFlow for switches [40] 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 SDN control software, we are currently evaluating our technique on other distributed systems, and believe it to be generally applicable.

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 [33] or have sufficiently accurate clock synchronization to obtain a happens-before relation. Inputs would also need to need to be logged in sufficient 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 [8], which log a failure iff the failure actually occurred.

8. RELATED WORK

Our primary contribution, techniques for interleaving events, made it possible to apply input minimization algorithms (*cf.* Delta Debugging [58, 59] and domain-specific algorithms [12, 47, 55]) to blackbox distributed systems. We described the closest work to us, thread schedule minimization and program flow reduction, in §4.2.

We characterize the other troubleshooting approaches as (i) instrumentation (tracing), (ii) bug detection (invariant checking), (iii) replay, and (iv) root cause analysis (of network device failures).

Instrumentation. Unstructured log files collected at each node are the most common form of diagnostic information. The goal of tracing frameworks [5, 10, 18, 25, 48] is to produce structured logs that can be easily analyzed, such as DAGs tracking requests passing through the distributed system. An example within the SDN space is NetSight [25], 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 in the first place.

Bug Detection. With instrumentation available, it becomes possible to check expectations about the system's state (either offline [36] or online [37]), or about the paths requests take through the system [48]. Within the networking community, this research is primarily focused on verifying routing tables [30–32, 38] or forwarding behavior [60, 61]. 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 [5, 10]. Our goal is to produce exact minimal causal sequences, and we are primarily focused on correctness instead of performance.

Model checkers [7, 42] seek to proactively find safety and liveness violations by analyzing all possible code paths. After identifying a bug with model checking, finding a minimal code path leading to it is straightforward. However, the testing systems we aim to improve do not employ formal methods such as model checking, in part because model checking usually suffers from exponential state explosion when run on large systems,[22] and because large systems often comprise multiple (interacting) languages, which may not be amenable to formal methods. Nonetheless, we are currently exploring the use of model checking to provide provably minimal MCSes.

[22]For example, NICE [7] 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.

Replay. Crucial diagnostic information is often missing from traces. Record and replay techniques [20, 35] instead allow users to step through (deterministic) executions and interactively examine the state of the system in exchange for performance overhead. Within SDN, OFRewind [56] 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 components have failed) in a distributed system. Root cause analysis [29, 57] seeks to reconstruct those unknown events from limited monitoring data. Here we know exactly which events occurred, but seek to identify a minimal sequence of events.

It is worth mentioning another goal outside the purview of distributed systems, but closely in line with ours: program slicing [54] is a technique for finding the minimal subset of a program that could possibly affect the result of a particular line of code. This can be combined with delta debugging to automatically generate minimal unit tests [6]. Our goal is to slice the temporal dimension of an execution rather than the code dimension.

9. CONCLUSION

SDN aims 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 primitive; they essentially involve manual inspection of logs in the hope of identifying the triggering inputs. Here we developed a technique for automatically identifying a minimal sequence of inputs responsible for triggering a given bug, without making assumptions about the language or instrumentation of the software under test. While we focused on SDN control software, we believe our techniques are applicable to general distributed systems.

Acknowledgments. We thank our sheperd Nate foster and the anonymous reviewers for their comments. We also thank Shivaram Venkataraman, Sangjin Han, Justine Sherry, Peter Bailis, Radhika Mittal, Teemu Koponen, Michael Piatek, Ali Ghodsi, and Andrew Ferguson for providing feedback on earlier versions of this text. This research is supported by NSF CNS 1040838, NSF CNS 1015459, and an NSF Graduate Research Fellowship.

10. REFERENCES

[1] Linux kernel containers. linuxcontainers.org.
[2] OpenStack Neutron. http://tinyurl.com/qj8ebuc.
[3] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. IPDPS '09.
[4] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. Erlang '06.
[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. OSDI '04.
[6] M. Burger and A. Zeller. Minimizing Reproduction of Software Failures. ISSTA '11.
[7] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. NSDI '12.

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

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

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

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

[12] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ICFP '00.

[13] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding Race Conditions in Erlang with QuickCheck and PULSE. ICFP '09.

[14] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. ICSE '07.

[15] 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. OSDI '02.

[16] Floodlight Controller. http://tinyurl.com/ntjxa6l.

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

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

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

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

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

[22] 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.

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

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

[25] 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.

[26] J. Huang and C. Zhang. An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs. SAS '11.

[27] J. Huang and C. Zhang. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. OOPSLA '12.

[28] N. Jalbert and K. Sen. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. FSE '10.

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

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

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

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

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

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

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

[36] X. Liu. WiDs Checker: Combating Bugs in Distributed Systems. NSDI '07.

[37] 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.

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

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

[40] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. SIGCOMM CCR '08.

[41] G. A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review '56.

[42] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. SOSP '08.

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

[44] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. ASPLOS '09.

[45] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. SOSP '09.

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

[47] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. PLDI '12.

[48] 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.

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

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

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

[52] A. Thompson. http://tinyurl.com/qgc387k.

[53] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. SOSP '07.

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

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

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

[57] 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.

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

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

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

[61] 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.