Tardis: A Fault-Tolerant Design for Network Control Planes

Zhenyu Zhou★, Theophilus A. Benson‡, Marco Canini°, Balakrishnan Chandrasekaran†*
★Duke University, ‡Brown University, °KAUST, †Vrije Universiteit Amsterdam

ABSTRACT

Guaranteeing high availability of networks virtually hinges on the ability to handle and recover from bugs and failures. Yet, despite the advances in verification, testing, and debugging, production networks remain susceptible to large-scale failures — often due to deterministic bugs.

This paper explores the use of input transformations as a viable method for recovering from such deterministic bugs. In particular, we introduce an online system, *Tardis*, for overcoming deterministic faults by using a blend of program analysis and runtime program data to systematically determine the fault-triggering input events and using domain-specific models to automatically generate transformations of the fault-triggering inputs that are both safe and semantically equivalent. We evaluated *Tardis* on several production network control plane applications (CPAs), including six SDN CPAs and several popular BGP CPAs using 71 realistic bugs. We observe that *Tardis* improves recovery time by 7.44%, introduces a 25% CPU and 0.5% memory overhead, and recovers from 77.26% of the injected realistic and representative bugs, more than twice that of existing solutions.

CCS CONCEPTS

• **Computer systems organization** → **Availability**; • **Networks** → *Network reliability*.

KEYWORDS

Software Defined Networks, control plane, failure recovery, transformation

ACM Reference Format:
Zhenyu Zhou, Theophilus A. Benson, Marco Canini, and Balakrishnan Chandrasekaran. 2021. Tardis: A Fault-Tolerant Design for Network Control Planes. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), September 20–21, 2021, Virtual Event, USA.* https://doi.org/10.1145/3482898.3483355

1 INTRODUCTION

Network reliability is critical, especially, for cloud providers who face an ever-increasing demand for more "nines"[1] of availability [32, 56]. Designing a highly available network is, however, a hard problem: Network devices fail, misconfigurations happen, bugs are endemic in implementations, and errors in specifications are unavoidable [17, 21, 29, 84]. Recent work [1, 25, 49, 89] show that **over 30% of the customer impacting failures** for large scale operational networks are due to **software bugs in the network control planes.**

While there is a growing body of work on detecting and eliminating bugs in the data plane, e.g., configuration verification [18, 22, 23], data plane state verification [41, 42] and fuzzing [24, 49, 77, 92], only few such efforts focus on the control plane. Unlike the data plane, which supports either a simple language that is amenable to verification (i.e., P4) or a set of well-defined rules that can be easily modeled (i.e., OpenFlow rules or forwarding information base (FIB) of router tables), the control plane is generally written in relatively complex languages, e.g., Java or C, which are less amenable to verification and modeling. In fact, efforts to apply modeling to the control plane [10] have demonstrated limited scalability, and attempts at control plane emulation provide only limited coverage across realistic settings [49, 73]. Today, the most promising method for addressing control plane bugs is to perform fuzz testing on production traffic—a concept popularized by Netflix's Chaos-Monkey [6, 75]. such testing techniques, however, still fall short of detecting all bugs—some bugs remain invariably uncaught, and network outages ensue [32, 56].

Motivated by the inability of existing techniques to eliminate control plane application bugs, in this work, we explore the applicability of online techniques to recover from these bugs. There are two common approaches to online recovery: rewriting code (i.e., automated program repair [47, 61, 66]) and input transformations (i.e., failure oblivious computing [11, 67, 69]). The former, code rewriting, is often limited to general and well-understood patterns, e.g., null-pointer exceptions or off-by-one errors, and does not account for the more important class of bugs demonstrated in Table 1, e.g., edge cases or missing logic. The latter, input transformations, addresses a broader set of bugs but requires significant domain knowledge to ensure a principled recovery. Moreover, existing approaches to transformations take either a random and ad-hoc approach [67] or a manual approach [11].

In this paper, we propose *Tardis*,[2] a system that both overcomes the previously discussed limitations of input transformations and generalizes to a wide range of network control plane applications

[*] Zhenyu Zhou is now at Google.

SOSR '21, September 20–21, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/09…$15.00
https://doi.org/10.1145/3482898.3483355

[1] A widely used unit for measuring reliability or availability of computer systems, expressed as a ratio of uptime to the sum of uptime and downtime. Three nines, for instance, refers to 0.999 or 99.9% availability.

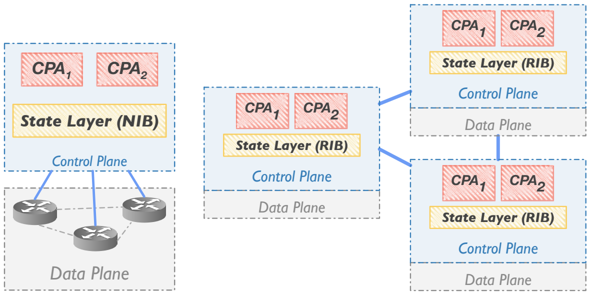
[2] The name *Tardis*, based on the British TV show *Doctor Who*, refers to the system's ability to travel back in time and manipulate history to avert an impending doom—in our case, the crash of a CPA.

(CPAs), deployed in both centralized (e.g., SDNs) and distributed (e.g., BGP) settings. Fundamentally, *Tardis* is a record and replay-based system which (1) automatically identifies the set of input events that trigger the bug using advances in program verification (i.e., symbolic execution) and (2) uses a domain-specific model to automatically search through the space of potential transformations on the input events to identify a candidate list of *semantically equivalent* and *safe* events to replay. We define semantically equivalent and safe events as those events that preserve a set of network-operator-specified network objectives as captured by network invariant checkers.

Tardis's design builds on the following insights. First, network events represent changes in the network state, and different events can be used to exercise the same state transitions. Thus, we can recover from failures by exploring an alternate but equivalent event provided that this alternate event *results* in the same network state. Second, the rich body of work on network verification, i.e., invariant checkers [42, 81], provide a well-understood method for analyzing this network state and, more importantly, determining if any two network states are equivalent. Thus, we can validate transformations by employing network invariant checkers. Specifically, given that we know the initial and final state, we can explore arbitrary transformations of a failure event (e.g., a link failure event) to discover equivalent events which safely bring the network to the intended state.

Tardis operates as a *shim* between the CPA, the state layer (which maintains the network state), and the network (state changes of which generate the events). *Tardis* consists of three key components: first, a novel domain-specific search algorithm for generating arbitrary transformations of the failure triggering event; second, a network invariant checker [42, 81] for detecting semantically safe transformations; and third, a symbolic execution framework for analyzing code to determine the root cause of a fault. *Tardis* intercepts and records events. Such an event recording technique is employed by Google's Orion [19], and it has been proved successful in practice. *Tardis* aims for recovering from two failure types: fail-stop faults and invariant violations. The former is easy to detect, and *Tardis* uses existing fault-detection techniques [42, 81] to detect the latter. When a fault is detected, *Tardis* examines the source code using symbolic execution and determines the events which triggered the failure. Next, *Tardis* uses the fuzzer to generate a set of equivalent events and then employs a domain-specific invariant checker to prune the set of equivalent events to semantically safe ones. Finally, *Tardis* rollbacks the control plane and re-executes the control plane with the transformed event. The rollback and replay continue until the control plane recovers. Thus, *Tardis* provides a best-effort guarantee, which is limited by the space of available transformations. Our evaluations show that both the traditional and SDN network control planes permit a rich enough space of events, which permit sufficient transformations, that allow *Tardis* to provide high availability in the presence of deterministic bugs.

We mainly design *Tardis* for two use cases: (1) help cloud providers such as Google Cloud and AWS, who employ SDNs, to recover their SDN apps; and (2) help network device manufacturers (especially, switch vendors like Cisco) to recover the device's apps such as routing apps. To address the two use cases, we implement *Tardis* in a popular SDN control plane with six SDN-CPAs and in two BGP-CPAs for two traditional distributed control planes (i.e., Quagga and GoBGP). We recreated and injected a total of 71 real faults across both control planes. Our experiments demonstrate that *Tardis* can recover from 87.19% SDN faults and 56.52% BGP faults.



(a) ***SDN*** **(b)** ***TDN***

Figure 1: ***Architecture of SDN (left) and TDN (right) network control planes.***

We summarize our contributions as follows.

⋆ We present *Tardis*, a system that can automatically transform one or more input events to recover from run-time faults. *Tardis* supports both centralized and distributed network control planes.

⋆ We provide an in-depth analysis and taxonomy of bugs in control planes and CPAs to complement anecdotal evidence from large cloud providers and motivate systems like *Tardis*.

⋆ We formulate a search heuristic for automatically generating semantically equivalent network events. Our formulation also provides a principled lens for reasoning about the safety and liveness of these transformations.

⋆ We implement *Tardis* to work with both a popular SDN control plane (Floodlight) and several BGP control planes (Quagga and GoBGP). We recreated a total of 71 real faults and injected them into control-plane applications. In our experiments, *Tardis* recovers from 87% of the SDN faults and 57% of the BGP faults.

2 MOTIVATION AND RATIONALE

Below, we provide a brief background on network control plane applications (§ 2.1), present a survey of bugs in these control planes (§ 2.2), provide a motivating example and rationale for our approach (§ 2.3), and describe the set of bugs addressed by *Tardis* (§ 2.4).

2.1 Network Control Planes

Generally a communication network include two planes: (i) the data plane, which processes each packet and routes them according to predefined rules, and (ii) the control plane, which consists of a set of applications (e.g., load-balancing, firewall, or BGP peering) that generates these predefined rules in response to network events (e.g., switch/link failures or path changes). Network control planes can be broadly classified into two types (cf. Fig. 1): (i) a traditional, *distributed* control plane (e.g., BGP), and (ii) a more modern, logically *centralized* control plane (e.g., Software-Defined Networks).

Table 1: ***A summary of CPA bugs.***

		SDN			BGP	
		ONOS	*CORD*	*Faucet*	*Quagga*	*XORP*
Bug Types	*Det.*	94%	94%	96%	76%	90%
	Non-det.	6%	6%	4%	24%	10%
Triggers	*Network*	20%	50%	40%	38%	38%
	Config	56%	42%	52%	25%	23%
	OS	12%	8%	6%	38%	40%
Symptoms	*Crash Stop*	10%	16%	32%	34%	37%
	Invar. Violation	84%	82%	66%	66%	61%
	Performance	6%	2%	2%	0%	2%
Causes	*Missing Cases*	0%	25%	16%	58%	74%
	Memory	40%	26%	9%	9%	9%
	Concurrency	0%	13%	7%	7%	4%

Software Defined Network (SDN). In this mode, the control and the data plane are on separate devices. The data plane consists of forwarding elements (SDN switches), while the control plane runs on separate x86 servers. The data plane generates and sends events to the control plane, which uses them to build a global view of the network; this state is stored in the network information base (NIB). The control plane runs a set of control plane applications (CPAs) that analyze, process, and react to the data-plane events by inserting new rules into the data plane. In general, the control plane is logically centralized, and it provides the CPAs with a global view of the network, which allows them to make optimal decisions for the events they receive.

Traditional Distributed Network (TDN). In a traditional network, each device contains both a control plane and a data plane. In such a setting, the control plane is distributed across the network, and each control plane hosts several CPAs (e.g., OSPF, BGP, or ISIS processes). In addition to reacting to events from the data plane (e.g., link failures), the CPAs for a traditional control plane also react to messages from other CPAs (e.g., BGP update messages). Given that each control plane only has local information, the CPAs need to exchange messages to allow each CPA to build a global view of the network with which it can determine how to react to events. Each CPA maintains a view of the network in its routing information base (RIB).

Summary. Regardless of the control plane type, networks exhibit two traits. (1) They maintain state relevant for their operation in a separate state layer (e.g., the RIB or NIB in Fig. 1)—we surveyed 47 SDN-CPAs and 6 TDN-CPAs and found that 64% of the SDN-CPAs and 100% of the TDN-CPAs maintain state in an external state layer. (2) They are event-driven—in addition to network events (i.e., from data plane or other CPAs), CPAs also react to events from the operating system (e.g., timers) and events from the configuration interface (e.g., command-line or configuration changes).

2.2 Control-plane Bugs

To understand the types of bugs that occur in practice, in Tab. 1, we survey 150 bugs from three popular SDN control planes and their CPAs (i.e., ONOS [7], Faucet [4], and CORD [63]) and summarize a prior survey [91] on two popular traditional control planes and their CPAs (i.e., Quagga [34] and XORP [27]). Our survey and the prior survey use manual analysis of the control plane's code repository issues to analyze and classify the bugs.

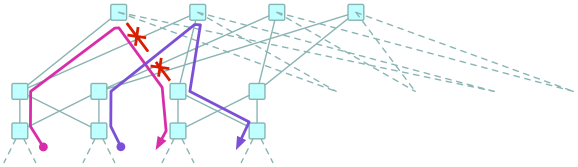


Figure 2: ***Part of a simple Fat-tree with two flows (in pink and purple) routed over link-disjoint paths. Certain events, e.g., a link failure (in red), affect the status quo, i.e., end-to-end connectivity between two hosts or the completion time of the flow that uses the affected link.***

We analyze these bugs across four main dimensions. **(1) Determinism**, whether the bug is deterministic and can be recreated using a predefined set of steps. **(2) Trigger**, the type of event (network, OS, or configuration) which triggered the bug—or the 'bug triggering event.' **(3) Impact/Symptom**—the impact of the bug on the control plane (e.g., crash-stop failure or performance issues in the CPA) or network (e.g., the CPA configures the network in violation of expected behavior). **(4) RootCause**—the programming error which caused the bug; we limit ourselves to the top categories.

We observe the following across both control plane types. (1) Most bugs are deterministic. (2) Moreover, bugs are often due to missing logic, e.g., to handle corner cases. (3) Many bugs (~28%) are due to hardware reboot and network events. These three observations concur with findings from recent publications (e.g., Dalton et al. [14] and Bhardwaj et al. [8]). As illustrated by Google and Microsoft [14, 49], deterministic bugs often severely affect their network's availability: Since most control planes employ some form of state-machine replication, a deterministic bug will manifest in each replica and cripple all of them.

2.3 Motivating Example

To understand the rationale behind transformations and reason about their correctness, we explore the behavior of a well-known CPA for load-balancing network traffic, namely Hedera [3]. Hedera routes flows to maximize aggregate utilization of the network. It improves flow completion times—an *operator-specified objective* — by periodically re-routing large flows over less congested paths. It also satisfies *operator-specified invariants*, e.g., if a path exists between two hosts, Hedera *must* route the traffic between the hosts. Hedera is a simplified version of the CPAs that Microsoft [31] and Google [33] run on their networks.

In Fig. 2, we illustrate a Fat-tree topology; to keep it simple, we do not show the hosts. The figure includes two traffic flows $\mathcal{P}$ (in magenta) and Q (in purple) that flow over link-disjoint paths, but which share a switch. Suppose that at time t both the network objective is satisfied and the invariant holds. Now, if at time $t+1$ an event such as the link failure in Fig. 2 happens, Hedera will receive an event capturing a state transition from a network graph with the link to a graph without the link.

Given the multitude of options to route $\mathcal{P}$, Hedera will pick an option that again maximizes the aggregate bandwidth; the rerouting of the flow over another path also satisfies the invariant. A new network state is established, and this change highlights two observations. First, *there might be more than one way to configure the data plane to satisfy both the objectives and invariants.* Second, *the specific path (or data-plane configuration) does not matter to the operator, as long as the high-level objective is satisfied.*

The first observation that multiple options may exist to meet the CPA's end goals (i.e., satisfying of objectives and invariants) motivates the use of transformations: In case of a CPA bug, we can safely explore a different code path and output to avoid the bug provided that it meets our end goals. *To this end, upon isolating the bug-triggering event, we transform the event and alter the behavior of the application. When no such alternatives exist, recovery can gracefully degrade to using more traditional approaches such as CPA reboots.*

The second observation that the specific option chosen or data-plane configuration effected is irrelevant attests to the safety of transformations. A transformation is deemed safe if the following conditions hold.

(1) The new behavior it elicits still satisfies the invariants.

(2) The state transition it presents is equivalent to the original event; the original event and its transformed counterpart are then said to convey similar semantic intent.

Formal definition of transformations. Given the above, a transformation of a set of input events E conveys the same semantic intent as E, but through a different set of input events $\tilde{E}$. The transformation of the event {`Port` P_1 `(of Sw.` S_1`) Down`}, for instance, to {`Port` P_1 `(of Sw.` S_1`) Up, Port` P_1 `(of Sw.` S_1`) Down`}, preserves the semantic meaning of the original input sequence—that the port P_1 of S_1 is offline. Even though the status of the concerned port went up before going down, the final state conveyed via both the input sequences is the same. In processing the transformed sequence, the CPA exercises, however, a different code path, which might help in averting the buggy behavior observed when processing the original input.

2.4 Target Failure Modes and Scenarios

The CPA bugs described in Tab. 1 manifest via a fail-stop fault, an invariant violation or performance problem. In this paper, we focus on the first two broad classes of faults. First, *fail-stop* faults where a control plane (e.g., a BGP process or an SDN controller) abruptly terminates after processing a bug-triggering input; arbitrarily long delays in responding to an input (i.e., gray failures) also belong to this class. We identify such faults using timeouts or "heartbeat" signals. Second, (network) *invariant violations* where the rules installed by the CPA result in the data plane deviating from "expected behavior," e.g., not dropping malicious packets or not load balancing across parallel links. These deviations are a violation of one or more invariants or objectives established by the operator and, as such, can be detected, in real-time, using invariant checkers [37, 42].

Limitations. We do not handle configuration- or OS-triggered failures, but rely instead on prior work on control plane configuration verification [18, 22, 23] as well as data diversity [40, 51] for detecting them.

3 BACKGROUND

In this section, we discuss prior work on checking for invariants (§ 3.1) and provide an overview on symbolic execution (§ 3.2).

3.1 Checking Network Invariants

There is a rich literature on invariant checkers [37, 41, 42, 81], which analyze network state (via RIB or NIB) to determine if the policy implemented in the network adheres to operator-specified objectives (e.g., loop-free or valid paths between all pairs of destinations). They assume that network operators explicitly specify their objectives. Given such a specification, the checkers analyze either the RIB or the NIB to determine if the network is compliant with the specification. Checkers for TDNs [37] operate at the router level and inspect the RIB as well as router configurations to check for invariant violations, while those for SDNs [41, 42, 81] analyze the NIB of an SDN controller or the NIB created by aggregating RIBs (as in the case of an IGP protocol).

Invariant checkers demonstrate that the operators-specified objectives may be realized through multiple, distinct network states—these distinct network states, satisfying a given objective, can be said to be "equivalent" to one another.

3.2 Symbolic Execution

Symbolic execution [44] is a method of analyzing a software program with the objective of determining how inputs to the program affect its execution (or control flow) along different (code) paths. The term 'symbolic' refers to the use of symbolic values rather than actual inputs in describing the program behavior—expressions, variables, and conditionals. The root of the execution tree begins at the entry point of the symbolically executed code, and each branching in the tree represents the two outcomes of a conditional branch (e.g., `if` block). Each unique path on the execution tree, from the root to a leaf, corresponds to a *code path*.

When a bug manifests in an application, its execution tree and the sequence of inputs (until when the bug manifests) can be used to determine the exact code path where the bug is encountered. Recovery from the bug is feasible by driving execution along a different path. Although more code paths trivially imply more options for recovery, an exhaustive enumeration of all paths is not necessary to recover from a bug—thus effectively sidestepping the scalability issues of symbolic analyses.

4 TARDIS

An effective way to address control-plane bugs is to rewrite, patch, and redeploy the control plane. The process is, however, time-consuming and requires a lot of manual work.

Instead, *Tardis* circumvents runtime bugs in a control plane application (CPA) by transforming one or more bug-triggering inputs. To achieve this goal, *Tardis* runs as a transparent shim (Fig. 3) between the CPAs and the underlying base control plane. This setup allows *Tardis* to monitor the stream of inputs fed to a CPA, the output the CPAs generate (in response to each of those inputs), and the CPA's internal state changes (due to processing the inputs). We note that the architectures of both TDN and SDN control planes both facilitate the use of such a shim (cf. Fig. 3). As illustrated in the figure, the shim runs between the SDN controller

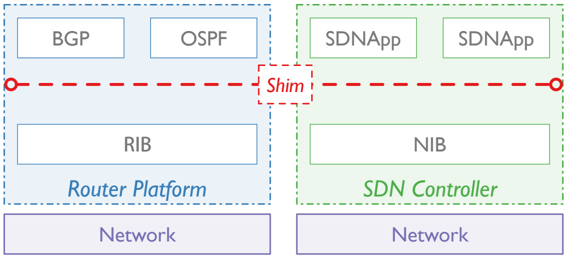


Figure 3: *Architecture of TDN (left) and SDN (right) control planes, indicating where a shim can be added to monitor the input, output, and state-changes of the CPAs.*

core (the router platform) and the SDN-CPAs (TDN-CPAs); thus, it monitors the events fed into the CPAs. These events are not raw OpenFlow or BGP messages, but are rather the transformed messages.[3] To realize this input-transformation-based recovery, *Tardis* is composed of three components (all of which reside in the shim layer—workflow in Fig. 4): RCA-Engine (§5), which uses a fusion of symbolic execution and runtime data to detect the bug-triggering input; the Transformation-Generator (§6), which uses a domain-inspired search heuristic to generate semantically equivalent and safe candidate events to replace the bug-triggering events; and the recovery-orchestrator, which maintains historical data for each CPA and orchestrates rollback and recovery during recovery mode.

Recovery-Orchestrator. The recovery-orchestrator maintains two types of runtime data: the list of input events and historical versions of state-layer changes associated with the input. The list of input events is stored in a queue within the Orchestrator, whereas the historical versions are maintained by augmenting the existing storage layer (i.e., NIB or RIB) to store historical versions. The historical data maintained in the storage layer remains outside of *Tardis*, and our system interacts with it using RPCs.

Tardis normally operates in a *passive mode*, intercepting and maintaining a log of all events to and from the CPA and managing associated state layer changes while incurring minimal overhead. When a CPA bug is triggered, however, *Tardis* enters *recovery mode* and intervenes to recover the CPA and ensure availability.

When the system enters into the recovery mode (refer workflow in Fig. 4), it performs three key operations, namely ⓐ identify the bug-triggering event (or root cause of the failure) and roll back the CPA's state to before the bug-triggering event is processed (§5), ⓑ transform the bug-triggering event into a set of one or more semantically equivalent events (§6), and ⓒ verify the safety and liveness of these transformations before as well as after applying them (§6.4). *Tardis* repeats the operations ⓑ and ⓒ until the CPA is recovered from the fault. Once the CPA has recovered, and liveness is confirmed, *Tardis* transitions back into the passive mode.

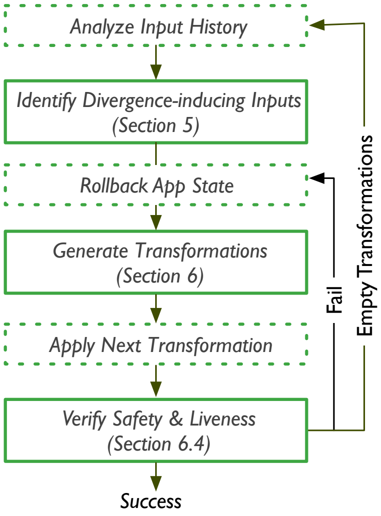

Analyze Input History
Identify Divergence-inducing Inputs (Section 5)
Rollback App State
Generate Transformations (Section 6)
Apply Next Transformation
Verify Safety & Liveness (Section 6.4)
Success
Fail
Empty Transformations

Figure 4: *Workflow for recovery mode.*

5 RCA-ENGINE (ROOT CAUSE ANALYSIS)

The first step in recovering from a failure or a bug is determining the root cause (bug-triggering events) of the fault. Existing approaches [72, 73] either use blackbox techniques [72, 73] (e.g., delta debugging) or explore whitebox techniques (e.g., symbolic execution or model checking). However, the blackbox techniques often take too long because they randomly explore different combinations of events. In fact, our experiments with STS [72] (not included due to space limitations) show that it will take more than 105 seconds to localize the events compared with about 100 ms taken by the approach we propose below. The whitebox techniques, on the other hand, do not scale: We tried employing several symbolic execution techniques [2, 50, 74], but they were unable to analyze the CPA and all its dependent libraries for performing an exhaustive search across all interactions to detect failures.

This work explores a fusion of both techniques. We use symbolic execution to explore just enough of the CPA to understand its structure. Then, we use runtime values from concrete failures to determine the bug-triggering input events. Concretely, we leverage whitebox symbolic execution and limit its analysis to just the CPA. This restricted symbolic execution allows us to understand the different code paths within the CPA. It prevents us, however, from determining where the failures lie or which events trigger failures, both of which require more extensive analysis or modeling of dependent libraries and the ecosystem. To mitigate this limitation, we use runtime data (i.e., real-time variables in the state layer and concrete events in the replay buffer), which provide sufficient concrete information about a failure, to determine the exact code paths that led to failures.

Our approach to identifying the root cause of fault rests on two assumptions: (i) A CPA's state is a function of the sequence of inputs that the application has processed over time, and (ii) a CPA's fault is triggered by an input (event) executing over a specific state, referred to as "buggy state." *Our key contribution, under these assumptions, is the observation that transforming the event that leads the CPA to a*

[3] Per Fig. 3, the shim runs in between the SDN controller and the SDN-CPA in case of a global control plane. For a local control plane, the control plane application (e.g., BGP) can run on a hypervisor (as in [39]) and the shim placed in between the hypervisor and the application.

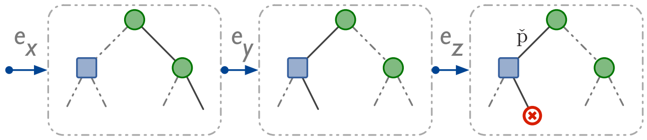


Figure 5: ***Change in code paths (solid black line) of a CPA while processing inputs events leads to a fault (red circle).***

Algorithm 1 *Find a cause of the buggy behavior.*

```
1: procedure FINDROOTCAUSE($\check{p}$, $E$)
2:     $\check{c} \leftarrow$ getPathConditions($\check{p}$)
3:     while $E \neq \emptyset$ do
4:         $(e, s) \leftarrow$ pop($E$)                ▷ Pick next candidate
5:         if side_effect_free($e$) then continue
6:         $p \leftarrow$ getCodePath($e, s$)
7:         $c \leftarrow$ getPathConditions($p$)
8:         if $satisfy(c, \check{c}) = false$ then return $e$
```

buggy state, irrespective of whether that input induces the fault, is a more effective solution.

Suppose that the CPA in Fig. 5 encounters a fault. Per this illustration, e_z encounters a specific state which leads to buggy behavior. Observe that e_y also encounters the same buggy state as e_z but does *not* encounter a fault; thus, a fault requires both a specific application state and a specific class of input event(s). Since e_x is the last event, the SDN-CPA processed before transitioning into the buggy state, we need to identify e_x as the root cause. We claim that altering the event(s) that creates the buggy state, i.e., e_x, is a more attractive solution than the event that triggers the fault, e_z. Stated differently, once the buggy state is reached, several other inputs (besides e_z) may induce a fault.

The first task once a CPA encounters a fault is to determine the buggy path, i.e., the code path ($\check{p}$) along which the execution is currently proceeding (path 3 in Fig. 5). Once we have $\check{p}$, then we identify divergence-inducing inputs. While tracking state changes is relatively straightforward (since *Tardis* acts as a shim between the CPA and the state layer), obtaining insights into the execution is a non-trivial challenge. To this end, we use symbolic analyses of the CPA's implementation to obtain insights such as available code paths, the conditions that need to be satisfied along the paths, and their relation to the state changes effected by the CPA. Using this data, we determine $\check{p}$ by scanning through the CPA's path conditions (obtained from the symbolic analyses) to determine the path condition that both matches the input and satisfies the current snapshot of the state.

Divergence-inducing Inputs. Given a buggy path $\check{p}$ and the associated path conditions, we iterate through the history E of input events of the CPA, in reverse chronological order, and mark a *candidate* input event e (and its associated state-variable changes s) as a root cause (as in Algo. 1) if the following conditions on inputs hold. (1) It is *non-side-effect-free*, i.e., this event modifies one or more of the CPA program's state variables which are associated with the path conditions of $\check{p}$, and (2) before processing this input, the execution of the SDN-CPA proceeded on a fault-free code path. *Tardis* ignores *side-effect-free* events (line 5 in Algo. 1) from consideration, since they do not modify the value of any state variables (and, hence, cannot steer the execution). Essentially, `side_effect_free()` checks to see if the event modifies any state variables. If the event changes state variables, we determine the code path (line 6 in Algo. 1) by using the global state's versioning feature to identify the state associated with the event and then determining the paths matching the state variables. We, then, extract the path conditions associated with the path (line 7 in Algo. 1).

Lastly, we use an SMT solver (line 8 in Algo. 1) to test if an input event steers execution away from the faulty path. The solver checks if the path conditions of the code path associated with an input satisfy that of the buggy path; a failure to satisfy implies that this input event steers execution away from the faulty path. Specifically, the inputs to the solver are the path constraints from when the CPA crashed and the state variables from the global state layer. As we iterate backward through events, we extract the associated state variables from the history within the global state and feed them into the solver.

In Fig. 5, to avert the fault (after processing e_z) we have to change the outcome of the path conditions associated with this buggy path. We mark e_x as the root cause for the buggy behavior because it leads to a different code path (compared to the buggy path of e_z). The algorithm ignores e_y, since this input does not change the path.

6 TRANFORMATION-GENERATOR

Transformations build on the notion of an equivalent and a safe class of inputs, where inputs in the same class both capture the same network state (hence equivalence) and elicit the "same" behavior from the CPA (hence safety), i.e., the output satisfies both operator specified invariants and objectives (§2.3). Hedera's same behavior, for instance, in §2.3, for the two equivalent inputs translate to achieving maximum aggregate bandwidth (objective) and ensuring that flows between any pair of hosts are routed over one of the available paths between the hosts (invariant). In this section, we focus on generating equivalent events in terms of network state and discuss safety in terms of behavior (in §6.4). In Fig. 6, we illustrate the transformation generation and validation functions.

Recall that there are two classes of network inputs. One class of inputs captures and expresses *topology changes/modifications*, i.e., a change in the network topology graph (e.g., a `{Port-up}` or `{BGP Update}` which captures the addition of a new link or path, respectively). The other class provides *local state updates*, i.e., updates information about a node or link (e.g., `{Switch stats}` or `{Packet-In}` which provide information about the number of bytes or the arrival of a flow at a device, respectively).

Our goals in generating transformations are to effectively address both classes of inputs. We achieve that objective using two types of transformations.

6.1 Generating Topology Transformations

The first class transforms an event into another set of events that capture a similar modification to the topology. We can validate these topology transformations by modeling the network as a graph, G, and verifying that both the original input sequence, E, and the transformed sequence, $\tilde{E}$, have the same effect on the graph. Thus,

$G + E = G + \tilde{E}$, where "+" is an operator that applies the messages to the network graph, e.g., applying a 'Link down' event to the network graph removes the link from the graph.

Exploring every single transformation results in a state-space exploration that varies exponentially in the size of the input. Moreover, many transformations are not semantically equivalent (i.e., they violate the protocol hints or do not provide the same effect on the network graph). Thus, we need an efficient search algorithm that can systematically explore the space of potential transformations to quickly identify semantically equivalent events. In essence, we need a search algorithm that can deal with arbitrary control planes and arbitrary event types (and their corresponding cost functions or definitions of equivalence) while finding an optimal solution. Simulated annealing fits these requirements; it provides a statistically optimal solution for arbitrary systems with arbitrary cost functions. Simulated annealing has been used quite successfully in many networking studies published in the last few years [3, 35, 54, 83].

In designing our simulated annealing-based heuristic, we assume a graph G to represent the target network topology. Then, we define a search space in which each state is a graph G_{Trans} with $V_G = V_{G_{Trans}}$. For topology transformations, all states differ in the set of links and nodes that are active. The following intuition guides our exploration of the space: the network is hierarchical (i.e., ports are part of switches, and switches are part of groups, e.g., pods), and semantically equivalent transformations are hierarchically related. For example, the transformation of {Port P_1 (of Sw. S_1) Down}, to {Port P_1 (of Sw. S_1) Up, Port P_1 (of Sw. S_1) Down} must be to the same switch (S_1). A transformation to a different switch (i.e., {Port P_1 (of Sw. S_2) Up, Port P_1 (of Sw. S_2) Down}) will not be semantically equivalent.

Following the intuition from above, our heuristic generates neighbors for the current state using the following procedures.

(1) Toggle an edge (u, v), where $u, v \in V_G$ and $u \neq v$.
(2) Take down all edges connected to a certain vertex u, i.e., let $(u, i) \notin E_G, \forall i \in V_G$.
(3) Bring up all edges connected to a certain vertex u, i.e., let $(u, i) \in E_G, \forall i \in V_G$.

We also have similar procedures, as those listed above, at the vertex level, where we either take down the vertex or its neighbors.

Similar to the classic simulated annealing framework, we have the energy function (also called goal function) defined as the edit distance between the current state (G) and the target network topology (G_0), i.e., $E(G) = dist(G, G_0)$, where $dist()$ indicates the edit distance. A new state could be accepted in two ways, either when $E(G') < E(\hat{G})$, where G' is the new state and $\hat{G}$ is the current state, or with the possibility $e^{-\frac{E(G')-E(\hat{G})}{T}}$.[4] The initial temperature T_0 is set to 1000 and the temperature of t^{th} iteration decays as $T(t) = \frac{T_0}{t+1}$. The algorithm terminates when the temperature drops below 5.

The transformation is the sequence of procedures used to transition from the initial state, G, to the equivalent graph G_{trans}.

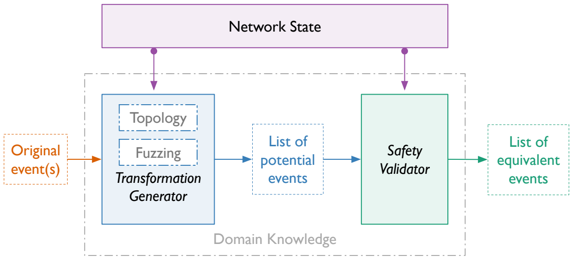


Figure 6: *Transformation generation & validation.*

6.2 Generating Attribute Transformations

The second class transforms the attributes of an event, i.e., a 'Link down' is transformed into another 'Link down' but with different attributes. The protocol specification offers strong hints on attributes that can be fuzzed, i.e., will not impact the intent. For example, the buffer size of a Packet-In message, input to an SDN-CPA, does not affect the intent. We could, hence, transform the message by enlarging its buffer (via padding) and recovering from buffer-size related bugs.

The fuzzer generates transformations for UPDATE messages; we leverage domain knowledge for determining when two UPDATE messages are deemed equivalent. This is generally based on the intent of the update message: for example, in an SDN-CPA the intent may be to express the number of bytes processed by a switch, and as such, the transformation is equivalent if this information is still present. Similarly, for a TDN-CPA, the intent may be to express a relative ordering of paths, and any transformations are valid as long as this relative ordering is maintained. The fuzzer, for instance, utilizes four different types of fields in the TDN messages to search for applicable transformations: (a) numeric values (e.g., MULTI_EXIT_DISC), (b) enumerated list (e.g., ORIGIN), (c) sequences (e.g., AS_PATH), and (d) free-form text (e.g., COMMUNITY).

Our fuzzing search algorithm is motivated by manual fixes employed by operators to "hot patch" their CPAs. We apply a different search algorithm for the different types discussed above. For numeric values, the fuzzer explores random values, and for sequences, it appends or deletes duplicates of a sequence item. In the case of freeform text, the fuzzer uses a set of well-known or predefined values specified by the network operator.[5]

6.3 Generality of Transformations

There is a clear distinction between SDNs and TDNs. While SDNs support a richer set of events, TDNs express most of the key network properties (both topology and local updates) in one event type (i.e., UPDATE). For SDNs, we leverage topology-transformations and attribute-transformations for the topology change and local update events, respectively. Whereas, for TDNs, we leverage topology-transformations and attribute-transformations for the UPDATE event.

[4] This follows the Metropolis Principle, where T is the current temperature.

[5] No global standards dictate how BGP COMMUNITY attributes must be interpreted. Peering ASes rather mutually agree upon a set of acceptable values and interpretations.

6.4 Transformation Safety

Transformations are inputs crafted for correcting buggy behavior. It is only natural, hence, to question if such transformations may themselves be unsafe. How can we check if transformations themselves introduce a fault? To answer this question, we identify two safety properties and defining a notion of *liveness* based on them. A transformation is deemed safe if it satisfies two properties.

$\mathcal{P}_1$: The CPA makes (forward) progress, i.e., it does not encounter a bug when processing the transformed event(s).

$\mathcal{P}_2$: The transformed events themselves do not induce any buggy behavior in the future.

While $\mathcal{P}_1$ applies only to recovery (i.e., recovery mode), $\mathcal{P}_2$ is relevant only after emerging out from the recovery mode and processing new inputs (i.e., passive mode). $\mathcal{P}_2$, similar to the *liveness* property of concurrent systems, assures that the system is not stuck in a loop—making progress, experiencing a fault, and attempting to transform the already transformed input. We can readily identify transformations that violate $\mathcal{P}_1$: If a CPA immediately faults after a transformation, we detect the fault and mark the transformation as the root cause. Satisfying $\mathcal{P}_2$ is, however, difficult: We need to assess the impact of the transformations in the future. We tackle this difficulty with a technique called opportunistic liveness tracking.

Opportunistic Liveness Tracking. The insight behind opportunistic tracking is that CPAs are designed to work with "soft-state," which means that the information in each event has a bounded horizon of relevance. In particular, if the information is not updated or refreshed, the CPAs consider the related state invalid and delete them. For example, information about the status of a link (i.e., link-up for SDNs) or an AS (i.e., five hops aways for TDNs) is considered irrelevant if not updated and refreshed within a predefined interval. Motivated by this, *Tardis* maintains metadata about recovery (e.g., transformations and the bug-triggering event) and tracks a CPA's behavior after recovery until a predefined interval, i.e., *Tardis* discards the metadata after processing N inputs following the transformation. This threshold N defines an opportunistic length of time within which we expect any buggy behavior, induced by the transformation itself, to manifest. A fault within the bounded horizon is attributed to the transformation. Suppose we assume a transformation to be faulty. Now, for any buggy behavior observed beyond the threshold, there is a possibility that we mark some other input, and not the transformation itself, as the root cause. Recovery might still be possible. In our empirical evaluations, we observed that a choice of $N = 16$ suffices, i.e., any impact of the transformation was typically observed within the next 16 inputs to the CPA. In practice, we may, hence, choose a much higher value to be safe, without running into any buffering constraints.

6.5 On Prior Transformation-based Recovery

While inspired by prior work [11] for leveraging transformations, we advance that work along two key dimensions. First, we develop a search heuristic to automatically generate transformations that LegoSDN provides. Second, we introduce techniques to eliminate unsafe and invalid transformations, thus ensuring correctness. A transformation that turns a {Sw. S_1 ↓} into a {[Sw. S_i ↓]*}, for instance, is an unsafe transformation: It shuts down the whole network. *Tardis* ignores this unsafe transformation since the difference between the original and the transformed events are too large.

7 PROTOTYPE

Next, we describe our prototype implementations for both an SDN control plane, with six SDN-CPAs, and two TDN control planes, each with a BGP-based TDN-CPA.

7.1 SDN Prototype

We evaluate *Tardis*'s ability to correct buggy behavior of SDNs by evaluating six different SDN-CPAs: Learning Switch, Firewall, Forwarding, Hedera, RouteFlow, and Load Balancer. We run both *Tardis* and the SDN-CPAs on Floodlight. Learning Switch ('LSwitch') and Forwarding ('Fwding') come bundled with the Floodlight controller. Firewall enforces a preconfigured policy, allowing traffic only between certain end hosts in the network. Hedera implements the flow-scheduling algorithm from [3]. Routeflow ('RtFlow'), from prior work [11], routes flows over the shortest path in the network, and Load Balancer ('LoadBal') balances network traffic between any two endpoints based on some simple heuristics. RtFlow and LoadBal are proactive SDN-CPAs, while the rest are reactive.

Our prototype implementation of *Tardis* for SDN-CPAs runs on top of the Floodlight controller. Since we did not make any changes to the controller's source code, the prototype may be ported to other controllers with modest engineering efforts. We added a simple state layer interface to the SDN-CPAs, providing `GET` and `SET` calls to support querying and modifications of the state-variables associated with the SDN-CPA. To the controller, the interface exposes `COMMIT` and `REVERT` calls allowing the controller to either commit the (control-plane) changes after an SDN-CPA successfully processes an input, or revert the changes, in case of a fault.

We used the Java Path Finder (JPF) model checker [57] and JDart [50] to symbolically execute the SDN-CPAs, and the Z3 SMT solver [15] to implement the constraint-satisfaction checks required for testing whether an input event induces buggy behavior. To symbolically execute the SDN-CPAs, we set the entry point to the event handlers and we make the input events and states symbolic. In our experiment, we observed that, with symbolic execution, we could explore all code paths accessible from the entry points within a reasonable time (i.e., less than 3000 ms).

Checking Invariants. The invariant checker module builds on that of prior work [81]. We modified the checker to flag violations and convert each violation to a fail-stop fault. The modification simplifies the recovery logic: whether it is a fail-stop fault or an invariant violation, recovery follows the same sequence of steps (refer to 'recovery mode' in §4).

7.2 TDN Prototype

We evaluate *Tardis* with two different BGP implementations—Quagga and GoBGP. In extending *Tardis* for the BGP use case, we exploit three key insights. First, the current state-layer for TDNs is a data-structure for storing data extracted from the `UPDATE` messages. Second, we extend this data-structure to make it version-aware and support similar `COMMIT` and `REVERT` semantics as the SDN storage layer. Second, local equivalence and invariants are based on the

Table 2: ***Details of faults, uncovered in prior work, injected into SDN-CPAs for evaluating Tardis.***

Label	*Type*	*Prior work*	*Source of buggy behavior*
S_A, S_B S_C	*Memory* *Management Errors (MME)*	STS [73] LegoSDN [11]	De-referencing a null pointer (or reference), or accessing an invalid memory location, e.g., indexing out of bounds of an array.
S_E	*Network Blackholes (NB)*	STS [73] PathDump [79]	Invalid or inconsistent switch configurations affected by a faulty SDN-CPA.
S_F	*Copy-Paste Errors (CPE)*	CP-Miner [48] Provenance [85]	Code copied by a developer from one location to another, without a careful testing.
S_G	*Forwarding Errors (FE)*	ATPG [92]	Same as that of S_E.
S_D, S_H	*Non-deterministic (ND)*	MED [93] LegoSDN [11]	Transient faults, e.g., race conditions in multi-threaded SDN-CPAs. Faults $\{S_A, S_B, S_C, S_E, S_F, S_G\}$ are deterministic.

relative ordering of paths. Thus we can check local invariants by analyzing the paths stored in the state-layer; we perform this check using C-BGP [68]. Third, CPAs in a TDN setting often run as distinct processes communicating through IPCs and RPCs; we design our shim layer for intercepting such calls based on techniques from prior work [39].

We use a bespoke symbolic execution tool to extract path constraints and reuse the Z3 SMT solver to implement the constraints-satisfaction. We set the entry point as the UPDATE handler and make the input events and states symbolic.

8 EVALUATION

Our evaluation of *Tardis*, is motivated by the following questions: (i) How does *Tardis* perform against existing recovery techniques (§8.2)? (ii) Does *Tardis* generalize across both control planes (§8.2 and §8.3)? (iii) How does *Tardis* operate with partial access to CPA state (§8.4)? (iv) Where do the overheads of *Tardis* come from (§8.4)? (v) What are the performance implications of *Tardis*'s search algorithm for generating transformations (§8.4)?

8.1 Experiment Setup

SDN Setup. We emulated the data plane (a Fat-tree topology, with $k = 4$) using Mininet [46]. We performed our experiments on a Linux (Ubuntu 14.04 LTS) server with 12 processor cores and 16 GB of memory. Unless otherwise mentioned, we injected all bugs in Tab. 2 in all event handlers across all SDN-CPAs and report the statistics (median and standard deviation) gathered from ten different trials.

TDN Setup (BGP): We injected the fail-stop bugs in Tab. 3 into the UPDATE message handler of two widely used BGP implementations: GoBGP (v2.5.0) and Quagga (v1.2.3). We ran the modified BGP implementations on an 8-core machine with 32 GB of RAM, running Linux kernel 4.15.0. We replayed BGP traces from RIPE NCC [70] archived on May 20, 2019 to a testbed consisting of one BGP router—the GoBGP (v2.5.0) or Quagga (v1.2.3) implementation. To use *Tardis* for recovering from BGP faults, we do not require any coordination or support from other networks; our one-node setting, hence, suffices for these experiments.

8.1.1 *Fault Injection.*

Below, we describe *what* faults we inject, and *how* and *where* we inject them.

Table 3: ***Prevalence of a few different bug categories among the bugs actually observed in the Internet.***

Category	Prevalence	Bug Labels (for Fig. 8)
Malformed Message	39.13%	$\{D_A, D_B, D_C, D_G, D_H\}$
Unknown Attribute	8.70%	$\{D_E, D_F, D_H\}$
Disordered Messages	8.70%	$\{D_D\}$

What? We use faults uncovered in prior work. We inject real bugs discovered in open source CPA artifacts by prior work (Tab. 2 for SDNs and Tab. 3 for TDNs): Injecting real bugs from multiple CPA artifacts enables us to understand the performance of *Tardis* across broad and representative failures. For TDNs (i.e., BGP), we selected the dominant categories of bugs (Tab. 3) among those observed on the Internet; the prevalence values in the table reflect the frequency of that bug type across 23 publicly documented BGP bugs over the last 13 years in the Internet. Moreover, recent studies by Google highlight that several of these bugs (e.g., NB type) have significantly impacted network availability [25].

How? We inject bugs via monkey patching and binary rewrites. Our fault injector monkey patches the source code with snippets of buggy code based on bugs in open-source SDN-CPAs and description of bugs in TDN-CPAs. To induce the MME and CPE fault types, the injector monkey patches the source code to throw exceptions or perform an out-of-bounds memory access. The injector carefully removes code or drop outputs, generating invariant violations (e.g., NB and FE fault types). We inject both deterministic and non-deterministic faults; we used a random number generator to help with the non-determinism required for the latter.

Where? We inject faults in all event-handlers. We introduce bugs in the most frequently traversed code paths of every event-handlers of the SDN-CPAs. In the case of the TDN-CPAs we focus on the UPDATE message handler as all reported bugs are in that handler's implementation. Thus, we comprehensively test *Tardis*'s ability to avert faults, even those in well-tested and commonly used paths encountered in invoking the SDN-CPAs.

Table 4: ***Success rate of fault recovery.***

Label	*App. Reboot*	*LegoSDN*	*RSM*	*Tardis*
S_A	0%	33.3%	0%	82.4%
S_B	0%	33.3%	0%	83.6%
S_C	0%	33.3%	0%	83.0%
S_D	100%	100.0%	100%	100.0%
S_E	0%	33.3%	0%	84.3%
S_F	0%	33.3%	0%	81.6%
S_G	0%	33.3%	0%	82.6%
S_H	100%	100.0%	100%	100.0%

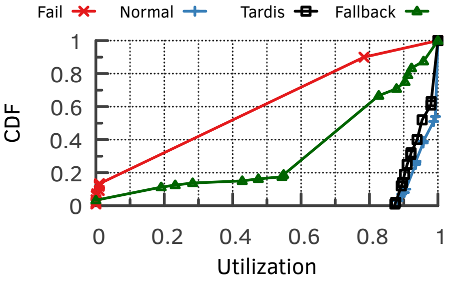


Figure 7: ***Tardis performs better than fast-failover methods ("Fallback"); variation in link utilizations observed after Tardis performs recovery is similar to that observed in a fault-free ("Normal") scenario.***

8.2 Evaluation of SDN Realization

8.2.1 Tardis vs Control Plane Recovery. We begin by highlighting the need for *Tardis*. Our approach recovers from a wider range of faults (Tab. 4) compared to all other online fault-tolerance techniques, namely application reboot (e.g., [9]), replicated state-machine or RSM (e.g., Ravana [36, 52]), and checkpoint-replay (e.g., LegoSDN [11]). RSM and reboot fail to avert any deterministic fault (with labels S_A, S_B, S_C, S_E, S_F, and S_G), since they repeat (without altering) the input sequence on recovery. Besides, they also fail to correct data-plane inconsistencies caused by the fault. LegoSDN alters the input event sequence and, hence, can recover even from deterministic faults. But that LegoSDN only alters the last input, has implications for recovery: Of the 48 faults injected (8 bugs across the 6 SDN-CPAs) LegoSDN fails to recover from 24 faults (Tab. 4).

We analyzed the 6 bugs (in Tab. 4) that *Tardis* could not recover from, in all trials. We identified the key reason to be the lack of alternative code paths—specifically, 2 of the 6 SDN-CPAs have faults in a method containing only one code path. Such lack of path diversity is suggestive of a simplistic, immature implementation. In some cases, *Tardis* fails when the bugs lead to a byzantine fault; such faults cannot, however, be detected using invariant checkers.

8.2.2 Tardis vs Data Plane Recovery. Networks today, typically, handle hardware issues such as link failures by pre-installing back-up or fall-back paths (e.g, [20, 30, 32, 58]). We, hence, return to our motivating example (§2), and simulate fast-failover methods by installing fall-back paths, as required, after the link failure. Fig. 7 plots the CDF of link utilizations after Hedera re-routes flows following the link failure event (cf. Fig. 2). We emulated TCP flows between non-overlapping pairs of hosts, selected uniformly at random, and we configured flow sizes to follow a power-law distribution and sampled inter-arrival times from a Poisson distribution. The fault-free ("Normal") scenario presents the baseline for comparison. *Tardis* performs better than fast-failover methods ("Fallback") and performs well compared to the baseline—variation in link utilizations is similar to that observed in the baseline ("Normal"). While fast-failover techniques are better than the *fail-open* scenario ("Fail"), they cannot always satisfy network objectives: Nearly 40% of the links are underutilized (i.e., with less than 80% utilization) and a few are virtually unused.

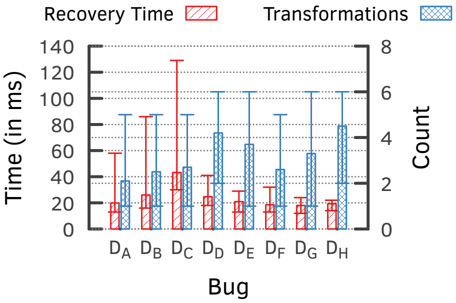


Figure 8: ***Times spent and counts of transformations used by Tardis for recovering from real bugs in a GoBGP implementation. Although not shown, similar results hold for the Quagga implementation.***

8.3 Evaluation of BGP Realization

We now demonstrate *Tardis*'s ability to handle run-time faults in BGP (e.g., [12], [53], and [94]), for evaluating our approach for a traditional, decentralized network (TDN) control plane.

Fig. 8 reports the median recovery times and counts of transformations used by *Tardis* (across 10 trials) to recover from real bugs in a GoBGP implementation. Per this figure, we recover from three dominant BGP bug types, in both BGP implementations, with sub-second recovery times. In most cases, exploring a few different transformations seem to suffice for finding a good solution. *Tardis* recovers quickly from real bugs, and the approach is much preferable to encountering router crashes and resulting widespread routing instabilities.

8.4 Implications of System Design

Below, we analyze some of the key design choices. We focus on the results from the SDN use case which has worse performance than the TDN.

State Layer. Access to the CPA's state variables enables *Tardis* to determine the root cause effectively. While most modern control planes (e.g., [7, 64, 90]) force developers to externalize global state variables in a database, as observed in §2.1, some developers only externalize a subset of the CPA's state. Next, we explore the impact of partially externalizing state variables on *Tardis*'s recovery. Given

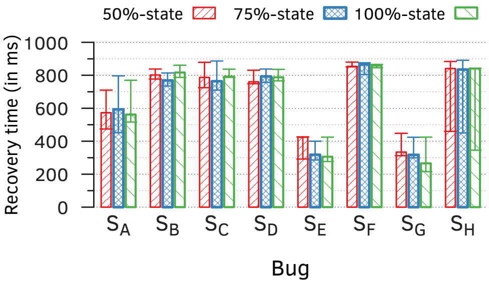


Figure 9: *Impact of the extent to which state variables are externalized on the recovery time.*

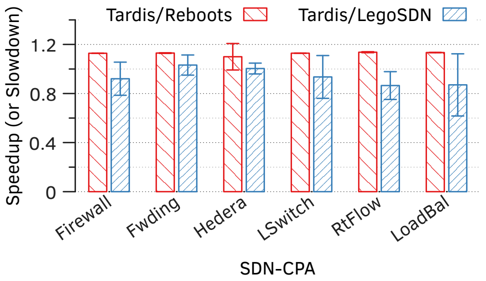


Figure 10: *Comparison of Tardis's recovery time with that of reboots and LegoSDN.*

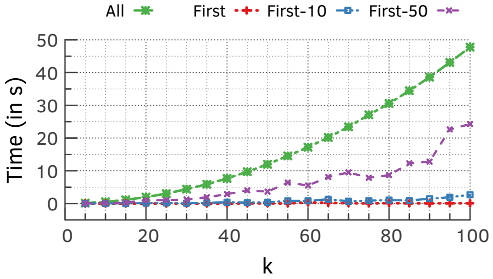


Figure 11: *Cost, in terms of time, of the simulated annealing-based heuristic for finding transformations.*

that the state variables are required for root-cause analysis (§5), we expect *Tardis*'s accuracy to reduce as less state information is made available to it; less accuracy, in turn, will increase its recovery time, because *Tardis* will need to explore more events and transformations. In Fig. 9, we observe that externalizing only 50% of the state ensures that recovery time increases by no more than 20%. Surprisingly, we also observe that recovery can be faster for a subset of the experiments, since less state implies less complexity.

Overheads. To quantify the overheads introduced, we divide the time spent by *Tardis* in recovering from various faults by that of reboots (and LegoSDN) to compute the recovery "slowdowns."

Per Fig. 10 *Tardis* is, unsurprisingly, slower than reboots: The latter only entails restarting the process, compared to the various functionality (e.g., finding a root cause) implemented by the former. Although reboot is faster, since it does not fix most errors (Tab. 4) there will likely be an extended downtime—this is, in essence, similar to what was experienced at Google and Facebook when the systems kept rebooting, and they had several hours of downtime because rebooting did not fix the problem.

Tardis is faster than LegoSDN, since the latter relies on checkpointing, which incurs significant overhead. The recovery time ratios for Forwarding and Hedera, however, show that *Tardis* is slightly slower compared to LegoSDN: These two SDN-CPAs persist comparatively more states resulting in *Tardis* requiring relatively more cycles to analyze these states during recovery. Even in these cases, the overheads introduced by *Tardis* are marginal and justified given its efficiency in fault recovery.

Our experiments also show that *Tardis* uses approximately 25% CPU of a single core and 0.5% memory—these resource utilizations can be controlled, as required, by an administrator by scaling different modules out on multiple machines in a cluster.

We analyze the performance of the Transformation-Generator's search algorithm (§6) in Fig. 11. To this end, we investigate its runtime across topologies of varying sizes—specifically a data center (i.e., Fat-tree topology with vary k values). We also augment this plot with the time spent in running the first n transformations (where n is either 1, 10, or 50), since we could terminate the search earlier, for limiting recovery time to some predefined budget. Our experiment results show, unsurprisingly, that the total search time increases with k. Our algorithm finds all transformations within 500 ms for a topology with $k = 10$, albeit it becomes prohibitively expensive for the much larger topologies. We note, however, that *Tardis* only required a small set of transformations. In practice, we observed that *Tardis* requires fewer than 20 transformations—fewer than 8 for the TDNs use cases. Furthermore, we do not need to wait until all searches finish before beginning replay in practice. Our algorithm can generate the first 10 events within a second even for the largest topology in our evaluation. Our search algorithm is, therefore, performant enough in practice to enable *Tardis* to recover from actual bugs.

9 RELATED WORK

Verification & Troubleshooting. There exists significant work on detecting and identifying bugs in SDNs [5, 10, 16, 37, 38, 42, 45, 51, 55, 59, 60, 71, 78, 82, 87]. Proving the absence of bugs is intractable, and testing can reduce, not eliminate, bugs. *Tardis* stands to benefit from a decrease in bugs: In the event of a (less-likely) fault, *Tardis* will have more fault-free code paths to use in recovering from the fault. Prior efforts also focused on localizing problems in SDNs [28, 72, 73, 80, 86], but these are offline techniques and do not attempt to recover from or circumvent a fault.

Automated Program Repair. An alternative and equally appealing approach is to modify the source code of the program rather than the inputs to the program. Existing work [47, 61, 66] on program repair focuses on developing general techinques for repairing arbitrary programs which limits them to addressing common errors, e.g., off-by-one or buffer overflows. However, as we show in § 2.2, network control planes are often plagued by more complex problems (e.g., missing logic) which require extensive changes. *Tardis* trades off generality for coverage: By narrowly focusing on input transformations for network control plane applications, *Tardis* is able to cover a broader range of important bugs.

Provenance-based recovery. Wu et al. [85] and Han et al. [26] propose provenance-based solutions to *suggest* "fixes" for bugs. In case of a run-time fault, however, the query to elicit fixes might have little or no information to retrieve fixes, and, unlike *Tardis*, recovery entails manual intervention.

Redundancy & Programming Models. Prior work has also investigated the use of redundancy, replication and programming models to tackle faults [36, 39, 62, 65, 76]. Unfortunately these approaches are explicitly designed to tackle non-deterministic bugs, whereas *Tardis* handles both deterministic and non-deterministic bugs.

Execution steering. Crystalball [88] introduced the idea of executing a model checker in parallel with a running system and steering the system's execution to prevent inconsistencies. The authors concede, however, that memory usage is a limiting factor. Further, a Crystalball-compatible system must be implemented in Mace [43], which might require significant engineering efforts. LegoSDN [11] attempts to transform crash-inducing inputs to avoid a fault, but it assumes that the last-processed input is the root cause, severely limiting its applicability. Bouncer [13] filters out malicious inputs, but mainly focuses on illegal memory writes.

10 LIMITATIONS

Recovery Limitations. *Tardis* is only able to recover if the code contains sufficient path diversity and the transformations are able to explore these paths. While we find this true for the TDNs, we observed scenarios where SDN-CPAs did not contain sufficient diversity. This observation highlights the difference between mature CPAs (i.e., TDNs) and non-mature (SDN-CPA). We anticipate that *Tardis* will benefit as the implementations of SDN-CPAs mature and improve.

Bug Type. Although, we focussed on bugs triggered by network events, other major sources of bugs include configurations and operating systems. We note, however, that existing work on verification addresses configuration bugs, and we plan to extend our approach to operating systems.

Semantic Limitations. We are unable to recover from route leaks and hijacks, which account for 18% of the reported BGP issues, due to our inability to validate ownership of IP address prefixes. We are presently only able to recover, therefore, from failures caused by non-malicious inputs.

11 CONCLUSION

The demands for high availability of a network infrastructure emphasize the need for robust, fault-tolerant control-plane applications (CPAs) for managing these networks. Despite prior work on testing, troubleshooting, programming models, and fault tolerance, the scope of prior work misses a key requirement: *support for recovering from bugs, especially of the deterministic type, at run time.* *Tardis* addresses this gap by introducing novel methods for effectively localizing the bug triggering input events and automated techniques for generating an alternative set of semantically equivalent and safe input events. *Tardis* rolls back the CPA and uses these alternative events for recovery. To demonstrate the effectiveness of *Tardis*, we evaluated it using a combination of 71 realistic failures injected into six SDN-CPAs and two TDN-CPAs. In our evaluations, *Tardis* recovered from more bugs than prior approaches. *Tardis* performed better than widely used fast-failover methods for SDN-CPAs, and for TDN-CPAs *Tardis* provided quick recovery (i.e., within 140 ms), avoiding least-preferred, expensive router crashes.

12 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Ryan Beckett, for their insightful comments. We also thank Ayush Bhardwaj for helping us with designing our experiments. This work was supported by NSF award CNS-1749785.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. 2021. Running BGP in Data Centers at Scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI '10)*.

[4] Josh Bailey and Stephen Stuart. 2016. Faucet: Deploying SDN in the Enterprise. *Queue* (Oct. 2016).

[5] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*.

[6] Cory Bennett and Ariel Tseitlin. 2012. Chaos Monkey Released into the Wild. https://web.archive.org/web/20120730195043/http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*.

[8] Ayush Bhardwaj, Zhenyu Zhou, and Theophilus A Benson. 2021. A Comprehensive Study of Bugs in Software Defined Networks. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '21)*.

[9] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2004. Microreboot — A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI '04)*.

[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*.

[11] Balakrishnan Chandrasekaran, Brendan Tschaen, and Theophilus Benson. 2016. Isolating and Tolerating SDN Application Failures with LegoSDN. In *Proceedings of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '16)*.

[12] Cisco. 2019. Cisco Bug: CSCuz62898 - Crash in BGP due to Regular Expressions. https://quickview.cloudapps.cisco.com/quickview/bug/CSCuz62898.

[13] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. 2007. Bouncer: Securing Software by Blocking Bad Input. *ACM SIGOPS Operating Systems Review* (2007).

[14] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*.

[16] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks. *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[17] Gary Eastwood. 2017. How A Typo Took Down Amazon's Cloud. https://web.archive.org/web/20180519161511/https://www.networkworld.com/article/3179831/cloud-computing/how-a-typo-took-down-amazons-cloud.html.

[18] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[19] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. 2021. Orion: Google's Software-Defined Networking Control Plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*.

[20] Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. 2018. Local Fast Failover Routing With Low Stretch. *SIGCOMM Comput. Commun. Rev.* (April 2018).

[21] FOX 46. 2018. American Airlines: PSA Computer Systems Stabilized After Glitch. https://tinyurl.com/yb6r6yz6.

[22] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically Repairing Network Control Planes Using an Abstract Representation. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[23] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*.

[24] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated Network Protocol Fuzzing Framework. *IJCSNS* (2010).

[25] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*.

[26] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer. 2017. FRAPpuccino: Fault-detection through Runtime Analysis of Provenance. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '17)*.

[27] Mark Handley, Orion Hodson, and Eddie Kohler. 2003. XORP: An Open Platform for Network Research. *SIGCOMM Comput. Commun. Rev.* (Jan. 2003).

[28] Brandon Heller, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, Nikhil Handigol, James McCauley, Kyriakos Zarifis, and Peyman Kazemian. 2013. Leveraging SDN Layering to Systematically Troubleshoot Networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*.

[29] Bill Hethcock. 2016. Southwest Airlines Computer Outage Costs Could Reach $82M. https://www.bizjournals.com/dallas/news/2016/08/11/southwest-airlinescomputer-outage-costs-could.html.

[30] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. 2017. SWIFT: Predictive Fast Reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[31] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. *SIGCOMM Comput. Commun. Rev.* (Aug. 2013).

[32] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. 2018. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-defined WAN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*.

[33] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*.

[34] P. Jakma and D. Lamparter. 2014. Introduction to the quagga routing suite. *IEEE Network* (2014).

[35] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. 2016. Optimizing Bulk Transfers with Software-Defined Optical WAN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*.

[36] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. 2015. Ravana: Controller Fault-tolerance in Software-defined Networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*.

[37] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*.

[38] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*.

[39] Eric Keller, Minlan Yu, Matthew Caesar, and Jennifer Rexford. 2009. Virtually Eliminating Router Bugs. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09)*.

[40] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*.

[41] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*.

[42] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*.

[43] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*.

[44] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* (July 1976).

[45] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. 2012. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*.

[46] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*.

[47] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*.

[48] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 04)*.

[49] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. 599–613. https://doi.org/10.1145/3132747.3132759

[50] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[51] Kshiteej Mahajan, Rishabh Poddar, Mohan Dhawan, and Vijay Mann. 2016. Jury: Validating Controller Actions in Software-Defined Networks. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[52] André Mantas and Fernando M. V. Ramos. 2016. Consistent and Fault-Tolerant SDN with Unmodified Switches. *CoRR* abs/1602.04211 (2016). http://arxiv.org/abs/1602.04211

[53] Robert McMillan. 2010. Cisco Patches Bug that Crashed 1% of Internet. https://www.computerworld.com/article/2515200/cisco-patches-bug-that-crashed-1--of-internet.html.

[54] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. 2014. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)*.

[55] Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. 2015. SDNRacer: Detecting Concurrency Violations in Software-defined Networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*.

[56] Jeffrey C. Mogul, Rebecca Isaacs, and Brent Welch. 2017. Thinking About Availability in Large Service Infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*.

[57] NASA. 2007. Java Path Finder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/start.

[58] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. 2007. Fast Local Rerouting for Handling Transient Link Failures. *IEEE/ACM Trans. Netw.* (April 2007).

[59] Tim Nelson, Andrew D Ferguson, and Shriram Krishnamurthi. 2015. Static Differential Program Analysis for Software-Defined Networks. In *International Symposium on Formal Methods*.

[60] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-defined Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*.

[61] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*.

[62] Jan Henry Nyström. 2009. *Analysing Fault Tolerance for Erlang Applications*. Ph.D. Dissertation. Acta Universitatis Upsaliensis.

[63] opencord.org. 2019. About - Open Cord. https://opencord.org/about/, last accessed on November 4, 2019.

[64] OpenDaylight Project. 2013. The OpenDaylight Platform. https://www.opendaylight.org.

[65] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. 2017. SCL: Simplifying Distributed SDN Control Planes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[66] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles (SOSP '09)*.

[67] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*.

[68] B. Quoitin and S. Uhlig. 2005. Modeling the routing of an autonomous system with C-BGP. *IEEE Network* (2005).

[69] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. 2004. Enhancing Server Availability and Security Through Failure-oblivious Computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI '04)*.

[70] RIPE NCC. 2019. RIS Raw Data. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[71] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*.

[72] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. 2016. Minimizing Faulty Executions of Distributed Systems. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*.

[73] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. 2014. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '14)*.

[74] Koushik Sen and Gul Agha. 2006. CUTE and JCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*.

[75] Nick Shelly, Brendan Tschaen, Klaus-Tycho Förster, Michael Chang, Theophilus Benson, and Laurent Vanbever. 2015. Destroying Networks for Fun (and Profit). In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets-XIV)*.

[76] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. 2014. Rosemary: A Robust, Secure, and High-performance Network Operating System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*.

[77] Apoorv Shukla, S. Jawad Saidi, Stefan Schmid, Marco Canini, Thomas Zinner, and Anja Feldmann. 2020. Toward Consistent SDNs: A Case for Network State Fuzzing. *IEEE Transactions on Network and Service Management* (2020).

[78] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '16)*.

[79] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with PathDump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[80] Praveen Tammana, Chandra Nagarajan, Pavan Mamillapalli, Ramana Kompella, and Myungjin Lee. 2018. Fault Localization in Large-Scale Network Policy Deployment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*.

[81] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon-Myung Kang. 2016. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*.

[82] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*.

[83] Da Wei, Lei Xu, Xin Jin, Yiran Li, and Wei Xu. 2016. A 12-rack, 180-server Datacenter Network (DCN) Using Multiwavelength Optical Switching and Full Stack Optimization. In *2016 Optical Fiber Communications Conference and Exhibition (OFC)*.

[84] Elizabeth Weise. 2017. Massive Amazon Cloud Service Outage Disrupts Sites. https://tinyurl.com/y8z6erfj.

[85] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Automated Network Repair with Meta Provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets-XIV)*.

[86] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. 2011. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC '11)*.

[87] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. 2017. Attacking the Brain: Races in the SDN Control Plane. In *USENIX Security Symposium*.

[88] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. 2009. Crystalball: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*.

[89] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[90] Soheil Hassas Yeganeh and Yashar Ganjali. 2016. Beehive: Simple Distributed Programming in Software-Defined Networks. In *Proceedings of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '16)*.

[91] Zuoning Yin, Matthew Caesar, and Yuanyuan Zhou. 2010. Towards Understanding Bugs in Open Source Router Software. *SIGCOMM Comput. Commun. Rev.* (June 2010).

[92] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*.

[93] Q. Zhi and W. Xu. 2016. MED: The Monitor-Emulator-Debugger for Software-Defined Networks. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*.

[94] Earl Zmijewski. 2009. Oracle Dyn: Longer is not Always Better. https://dyn.com/blog/longer-is-not-better/.