A Comprehensive Study of Bugs in Software Defined Networks

Ayush Bhardwaj  
Brown University  
ayush_bhardwaj@brown.edu

Zhenyu Zhou  
Duke University  
zzy@cs.duke.edu

Theophilus A. Benson  
Brown University  
tab@cs.brown.edu

Abstract—Software-defined networking (SDN) enables innovative and impressive solutions in the networking domain by decoupling the control plane from the data plane. In an SDN environment, the network control logic for load balancing, routing, and access control is written in software running on a decoupled control plane. As with any software development cycle, the SDN control plane is prone to bugs that impact the network’s performance and availability. Yet, as a community, we lack holistic, in-depth studies of bugs within the SDN ecosystem. A bug taxonomy is one of the most promising ways to lay the foundations required for (1) evaluating and directing emerging research directions on fault detection and recovery, and (2) informing operational practices of network administrators. This paper takes the first step towards laying this foundation by providing a comprehensive study and analysis of over 500 ‘critical’ bugs (including ~150 with manual analysis) in three of the most widely-used SDN controllers, i.e., FAUCET, ONOS, and CORD. We create a taxonomy of these SDN bugs, analyze their operational impact, and implications for the developers. We use our taxonomy to analyze the effectiveness and coverage of several prominent SDN fault tolerance and diagnosis techniques. This study is the first of its kind in scale and coverage to the best of our knowledge.

Index Terms—SDN, Bugs, Fault-Tolerance, Taxonomy

I. INTRODUCTION

Software-defined Networking (SDN) has enabled a paradigm shift from legacy networking to programmable networks which has transformed ISP networks [1]–[3]. Clouds [4]–[6] and content provider networks [7]–[9]. Adoption of SDN by all major companies has enabled them to: (1) simplify provisioning and management of their networks, (2) better utilize network resources available for disposal, and (3) lower CAPEX (Capital Expenditures) and OPEX (Operating Expenditures).

SDN’s key principle is to decouple functionality for routing, security, and performance from the networking hardware, i.e., router and switches. This functionality is rewritten in specialized software and deployed at a centralized location, called the controller. Today, modern SDN controllers are complex pieces of software comprising millions of lines of code. With key networking functionality softwarized and deployed on controllers, it is no surprise that any bugs within the SDN controller can lead to network performance and availability issues.

In fact, recent studies by Google [7] and Facebook [10] have shown that 30% of the outages in their SDN deployments are due to software bugs in SDN control planes. Despite the mounting evidence from industry and analysis of opensource bugs [7], [10]–[12], the community is lacking a systematic and detailed analysis of critical bugs within the SDN ecosystem.

This paper provides an in-depth analysis of over 500 critical bugs across three popular and prominent controllers within the SDN ecosystem. We created a taxonomy of bugs through our analysis, evaluated existing SDN fault-tolerant frameworks, and identified classes of bugs that require more research. Our taxonomy provides the building blocks for designing representative and informed fault-injectors for testing SDN controllers.

Our study is motivated by the following key research questions

- RQ1: What are the characteristics of bugs in SDNs?
- RQ2: What is the operational impact of these bugs?
- RQ3: How are these bugs triggered, and what strategies are used to fix them?
- RQ4: How can network operators benefit from this study?
- RQ5: How effective are emerging research prototypes?

In answering these questions, this work lays the foundation for richer and more advanced bug-tolerant SDN systems.

Our key findings are:

- Contrary to the growing work [13], [14] that effectively tackle non-deterministic bugs, our study shows that there is evidence, to the contrary, that most of the critical bugs are deterministic in nature.
- While there is a growing number of SDNs fault tolerance frameworks, e.g., Ravana [13] or STS [12], these are focused on tackling bugs triggered by network-events. Unfortunately, they fall short in tackling bugs triggered by other types of events, e.g., configuration or OS events, e.g., timers. In Section VII-C we show that while most existing approaches can detect bugs, recovering from these bugs remains an unsolved question and new tools are necessary to fill this gap.
- SDN controllers are prone to bugs like any large software system. However, the specific subset of bugs and their distributions within SDNs are different from traditional server applications and distributed software. For example, in server applications, most bugs are due to configuration [15], [16], whereas, in SDNs, we found external calls and network events form a major portion of the bugs,
which requires a redesign of monitoring techniques to monitor all external interactions in addition to network events.

- One of the critical advantages of SDN over legacy networks is the global visibility [17] and the broader optimizations that it enables. However, we observe that the result of many of these bugs (e.g., bugs triggered by network events (19.8%)) is that this visibility is significantly lowered. In essence, these bugs eliminate a crucial benefit of SDNs.

Our analysis of the SDN bug corpus is largely driven by manual analysis and categorization of the different bugs across controller platforms. To ensure that our results generalize, we employ NLP-based analysis across a larger set of bugs.

Given the questions above, we re-used well-established taxonomies [18], [19] (Table I) and extended them to incorporate networking specific issues. The contributions of our characterization study can be summarized as follows:

- We provide a holistic view of SDN bugs to allow developers and researchers to leverage our conclusions to improve the SDN fault tolerance landscape. (§ IV)
- We extract guidelines and operational hints for managing and operating SDN networks (e.g., guidelines for Controller selection). (§ VII-A)
- We evaluate and analyze the coverage and efficacy of several existing SDN fault tolerant and recovery techniques. (§ VII-C)
- We identified the feasibility and effectiveness of designing NLP-based techniques for root cause diagnosis. (§ VII-B)

RoadMap. The rest of this paper is structured as follows: In section II, we discuss our target systems, our methodology, and our approach for automated analysis. In section III we analyze bugs by their type. In section IV we explore the operational impact of these bugs. In Section V we analyze the events that trigger them. In section VI we analyze code repositories to understand their software engineering practices. In section VII we discuss the implications of these bugs. In section VII we discuss the limitations of and threats to our study. We conclude in sections IX and X by describing the related and summarize conclusions.

II. METHODOLOGY

In this section, we discuss the controller frameworks that we analyze (§ II-A) and present our analysis techniques (§ II-B).

A. Target Systems

In Figure 1 we present an overview of the SDN ecosystem. The ecosystem comprises of three components: (i) SDN Applications, which provide specific network functionality, e.g., routing [20], load balancing [21] or access control [22]. (ii) SDN controller framework, which manages interactions between the SDN Applications and the underlying network devices (e.g., switches). (iii) the network data plane, which consists of the switches and routers running within the network. The interactions between the SDN control plane (Applications and controller) and the data plane occur through the exchange of SDN control messages (i.e., OpenFlow messages [23] or XMPP messages [24]). Many SDN controller frameworks build on third-party libraries to provide additional functionality, e.g., state management or packet processing. Thus controllers come bundled with a plethora of additional third-party libraries and services (indicated as a yellow box in Figure 1). SDN controllers are fundamentally event-driven: the arrows in Figure 1 demonstrate the various sources of input events that a controller reacts to (configuration, network events, the kernel through system calls, and application libraries through function calls).

Although there are approximately 32 controllers, we focus our study on three of the four most mature and popular open-source controllers: ODL, CORD, ONOS, and FAUCET. We selected ONOS and CORD over ODL because they are used by major operators, e.g., Comcast [25], Google [26], etc., in a large scale real-world production environments. Moreover, unlike ONOS or ODL, CORD is specially tailored for emerging technologies (e.g., 5G-MEC [27], [28]) – thus providing a different perspective. We selected FAUCET because it is used at Google [30] and provides a unique perspective from the other controllers because it has a more compact structure and is written in Python. Next, we elaborate on the design of each of the three SDN controller frameworks:

- **FAUCET** [30] boasts a monolithic and compact codebase that migrates existing network functionalities like routing protocols, neighbor discovery, etc., into vendor-independent data planes. FAUCET manages flow decisions by utilizing multiple Access Control Lists (ACL) and multi-table processing [31].
- **ONOS** (Open Network Operating System) [32] builds on four major goals: modularity, configuration flexibility, isolation of subsystems, and protocol agnosticism. ONOS utilizes an intent-based API that captures policy directives for controlling network function. These intent-based APIs are realized through a set of state transition machines. Each subsystem employs a different state machine. This is distinct from FAUCET’s monolithic but compact design.
- **Open CORD (Central Office Re-architected as a Datacenter)** [33] is a specialized version of ONOS developed...
for Telecom Central Office (CO) to replace purpose-built hardware with cost-effective, agile networks. CORD is composed of four open-source projects, including Openstack, ONOS, Docker, and XOS. CORD provides a unique subsystem, based on XOS [54], to orchestrate coordination across these four code-bases.

B. Data Set and Methodology

The controller frameworks maintain a structured bug tracking and code management system — ONOS and CORD use JIRA for bugs and Gerrit for rolling out fixes, whereas FAUCET uses Github for bug tracking and managing fixes. While Jira includes tags that allow us to analyze bugs based on developer-identified severity levels, for Github, we used a keyword approach [35] to extract severity levels.

Data. As of April 2020, the FAUCET, ONOS, and CORD communities have identified 251, 186, and 358 critical bugs, respectively, which include both open and close bugs. In examining the bugs in ONOS and CORD, we found that: (1) Over time, the number of critical bugs keeps increasing. This motivates a need for more principled analysis. (2) We observe that a burst of bugs occurs around release dates. For example, in the first quarter of 2017, we observed a burst in CORD bugs which coincided with a release [36]. This highlights the need for longitudinal analysis across different releases.

For our study, we randomly selected 50 closed1 bugs from each controller for manual analysis. Moreover, we further verified the automatic analysis with an extended data set containing over 500 critical bugs.

C. Bug Autoclassification with NLP

To scale and automate classification, we re-use an NLP technique that prior bug studies have used, i.e., Word2Vec [37], to classify bugs and validate our taxonomy. We summarize the steps as follows:

- First, we pre-process the bug data to extract features. There are three classic approaches for keyword extracting including Latent Dirichlet Allocation (LDA) [38], Hierarchical Dirichlet Process (HDP) [39] and Non-negative Matrix Factorization (NMF) [40] based on Term Frequency Inverse Document Frequency (TF-IDF) [41]. We choose the last approach because previous work [42], [43] has demonstrated its potential to analyze similar data.
- Second, we train a Word2Vec model, which provides a mechanism for automatically determining similar words.

Given a bug description, these two steps allow us to map each bug to a numerical vector in a Euclidean space. After mapping bugs to Euclidean space, we can employ classic Machine Learning (ML) techniques, e.g., Support Vector Machine (SVM) or Decision Tree (DT), to automatically classify the bugs.

1) Bug Labeling: We utilize the following dimensions to classify the bugs: bug type, outcome, fix, and trigger. In Table I, we summarize these dimensions. These dimensions align with the recent work to characterize bugs in cloud systems [18], [19] which provide a similar classification as Orthogonal Defect Classification (ODC) [44]. At a high level, we classify bugs based on determinism to understand their reproducibility. For the root-cause and fixes, we classify bugs based on the controller code-base or logic’s impact: some problems require changes to logic while others do not—similarly, some bugs are due to existing logic or absence of any logic (e.g., edge cases). To verify the fixes, we manually analyzed the source code patches and fixes. For the triggers, we identify four key events that initiate bugs. These events align with a canonical SDN controller (Figure 1). For the symptoms, we focus on the type of failure triggered by the bug.

Each bug receives at most one tag from each of the dimensions in Table I.

2) Validation: We validated the automated classification techniques with cross-validation by splitting our data set into 2/3 for training and 1/3 for testing. We explored several classic ML techniques, including Support Vector Machine (SVM) and Decision Tree (DT), Principal Component Analysis (PCA), and AdaBoost. In our experiments, we found that SVM model with normalization provided the best accuracy for predicting bug types and symptoms, with accuracies of 96% and 86%, respectively. Unfortunately, we found it hard to find any algorithm to predict bug fixes accurately, and we believe this is because bug descriptions generally provide little data about the fixes.

III. RQ1: BUG TYPE

We begin by classifying bugs according to determinism. Deterministic bugs are defined as bugs that are clearly reproducible with a fixed set of input actions, whereas non-deterministic bugs are inconsistent and cannot be consistently reproduced by replaying the same set of input events/actions. The key observation is that all frameworks are dominated by deterministic bugs: FAUCET (96%), ONOS (94%), and CORD (94%). One potential reason for this is that many controllers employ standard state-machine-based techniques [13], [14], [45], [46], e.g., Paxos [45] or Raft [46], which tackle and mask most non-deterministic bugs.

Takeaway. Given the dominance of deterministic bugs, we believe that record-and-replay-based recovery techniques [47] will have limited applicability on most SDN controllers. Instead, we recommend failure recovery systems which alter controller input events [12], [48], environments [49], [50], or source code [51]–[55].

IV. RQ2: OPERATIONAL IMPACT OF SDN BUGS

In this section, we explore the bugs’ symptoms and characterize them based on the controller’s behavior. The analysis of symptoms and controller behavior provides us with a first step towards understanding each bug’s operational impact.
Classification | Categories
---|---
Bug Type | Deterministic, Non-deterministic
Root Cause | Controller Logic-bugs: Load, Concurrency, Memory, Missing Logic
Non Controller logic-bugs: Human (misconfiguration), Ecosystem Interaction (Third-Party, Application Libraries or System Calls)
Symptoms | Performance, Fail-stop, Error Message, Byzantine (Wrong Behavior)
Fix | No Logic Changes: Rollback Upgrades, Upgrade Packages
Add New Logic: Add Logic
Change Existing Logic: Add Synchronization, Fix Configuration, Add Compatibility, Workaround
Trigger | Configuration, External Calls, Network Events (OpenFlow Message), Hardware Reboots

Table I: Bug Taxonomy.

Byzantine Failures (61.33%): A majority of the bugs lead to the following unexpected behavior: (i) gray failures – a partial outage of the controller (52.17%), where some controller functionality is working while others are not. For example, in FAUCET-1623 [56], where the controller continues to manage flows but is unable to manage broadcast packets because of an unhandled edge case, a bug in the mirroring interface (shown in Figure 3). (ii) stalling (20.65%), where the controller temporarily freezes, and (iii) incorrect behavior (27.18%). Unlike stalling or partial outages, incorrect behavior is difficult to detect and diagnose because they do not generate error messages or trigger any normal alerts.

Takeaway. These bugs, in general, highlight the need of formal network verification; however, early works on verification [57]–[59] focus on the datapath or provide limited validation of runtime behavior. Our analysis indicates a need for more runtime verification of controller behavior.

Fail-stop (20%): Bugs that cause fail-stop failures or controller crashes are the most dire bugs as they directly impact the network’s availability and lead to production downtime. In Figure 2, we analyze the root cause of these bugs. In FAUCET, these bugs are caused by human mistakes or ecosystem interactions. This implies that crashes are due to the edge cases related to certain external scenarios. In contrast with FAUCET, in ONOS and CORD, a majority of the bugs are due to incorrect controller-logic, e.g., load, memory, and missing code logic. For example, a misconfiguration led to a null pointer exception in CORD’s host and multicast handlers (CORD-2470 [60]), which crashed the CORD controller. Despite CORD being based on ONOS, we observe a key difference between ONOS and CORD: in general, CORD has significantly more bugs due to “missing code logic,” demonstrating a level of immaturity in the codebase.

Takeaway: Fail-stop bugs are the easiest to detect but have disastrous consequences. Our initial analysis shows that exploring designs to improve memory safety (e.g., memory safe languages like RUST [61] and programming styles [62]) will significantly improve availability.

Error Message (14.7%): In general, we ignore these bugs because they result in warnings that have no direct operational impact. The main observation is that CORD has the best exception handling, which leads to fewer error messages.

Performance (4%): From Figure 2, we observe that most of the bugs that result in slow controller performance can be triaged to one or two root causes. From the Figure, we also observe that different controllers have different root causes. A key surprise is that increased system load is not the main cause of slow performance. Instead, increased-system load leads to other failures, i.e., fail-stop and byzantine failures. We observe that poor performance is due to FAUCET’s interactions with the ecosystem, concurrency bugs in ONOS, and memory errors in CORD. Thus broadly speaking, these bugs in FAUCET are due to factors generally beyond the developer’s interactions, whereas in ONOS and CORD, they are due to poor programming logic.

Takeaway. Performance bugs [63] can cascade into a variety of dire bugs, e.g., byzantine, crash, etc., that can introduce SDN control plane instability. These bugs require active monitoring and health check system; however, such systems introduce significant overheads. For some of these

Figure 3: Patch for FAUCET-1623 [56], where interface mirroring didn’t mirror output broadcast packets which was fixed by adding a case for mirrored ports.
bugs, e.g., Concurrency bugs, we can explore alternative and potentially lighter-weight techniques, e.g., semantic exploration techniques [64]. For example, a CORD concurrency bug (CORD-1734 [65]) where multiple interleaved threads caused performance degradation.

Figure 4: Patch for CORD-1734 [65], where multiple threads were negatively impacting the performance of all API calls. This was attributed to reliance of python on global locks, so as a fix the maximum number of workers were reduced to 1.

**New Research Directions:** We summarize new research areas based on our observations:

- There are still gaps between the industrial demands and the modern invariant checkers as illustrated with FAUCET-1623 [56] (discussed above). To tackle such bugs with more complex behavior, we need more complex invariant checkers because most existing checkers focus on reachability-based and QoS-based invariants [57], [59].
- We identified a need for more fine-grained failure-indicators and failure-detectors that detect component level availability and correctness. These techniques need to be more expressive than simple heart-beats; they should verify subcomponent correctness. Specifically, for the failures that are due to load and ecosystem interactions, we may predict these crashes by analyzing metrics or existing sylogs. Given this, it would be interesting to evaluate the potential of extending existing log-based failure prediction systems [66]–[68] or metrics-based systems [69], [70] to SDNs.
- We highlighted a need for research into extending fault prediction based on system load to the SDN-domain to address issues with load and cascading errors, e.g., ONOS-4859 [71] that suffers from ineffective use of memory.

V. RQ3: Bug Triggers and Code Fixes

This section analyzes the events that trigger a bug, the code fixes applied to fix the bug($\S$V-B), and the time to fix them ($\S$V-B).

### A. Analysis of Bug Triggers

Recall, in Section II-A we showed that SDN controllers are event-driven and, in general, these controllers only react to the events listed in §II-A. Below we analyze each of these events and discuss the implications for our study.

**Configuration (38.8%):** We observed that many bugs are triggered when the controller attempts to process system configurations. This fact is astounding because a critical motivation for SDN is to move towards automation and eliminate configuration-based errors [73]–[79].

In Table III, we analyze the type of configurations. We observe that for ONOS and CORD, most of the configuration bugs are due to the configuration of the controller and third-party services.

Interestingly, we observe that only 25% of the configuration-related bugs can be fixed by changing the controller configuration. This implies that research on misconfiguration [77]–[79] focuses on detecting the impact of an application’s configuration on the system and will have limited applicability because third-party code bases’ configuration impacts the system.

**Takeaway.** These observations highlight a need for more research on techniques for diagnosing and debugging the cross-layer impact of configurations. These cross-layer approaches should be coupled with preventive systems such as [80] which detect latent configuration bugs by employing fuzzy-testing.

**External Calls (33%):** For the external calls, we observed that 41.4% of the code fixes attempt to make the controller more compatible with external libraries by changing function calls or arguments to match the external API or by upgrading the external packages. The use of code patches to fix this interdependence highlights the highly dynamic open-source ecosystem. Interestingly, we also observe that the misconfiguration of the communication between multiple modules is a non-trivial source of these problems. For example, in FAUCET-355 [81] (Figure 5), Guage crashed because of a misconfigured data type between Gauge and InfluxDB [82].

Moreover, as highlighted in prior work [83], a majority of open-source projects utilize outdated dependencies, which often makes the system vulnerable to attacks. SDNs are no exception; for example, in CVE-2018-1000615 [84] we observe that an outdated version of OVSDB [85] lead to a Denial of Service (DoS) attack on ONOS. In Table III we provide a broader analysis of vulnerabilities in ONOS using dependency-check tool [86] and cross-checking with NVD [87]. Our analysis shows that ONOS’ vulnerability increased over time as more dependencies were added with version updates. These vulnerabilities were fixed by changing the libraries, which makes them more critical.

**Takeaway.** A strong implication of this analysis is a need to design techniques to discover, track, and detect API mismatches. While techniques existing for tracking dependen-

<table>
<thead>
<tr>
<th>Sub-categories of Configuration Bugs</th>
<th>FAUCET</th>
<th>ONOS</th>
<th>CORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>52.9%</td>
<td>60%</td>
<td>64.2%</td>
</tr>
<tr>
<td>Data Plane</td>
<td>11.7%</td>
<td>15%</td>
<td>14.2%</td>
</tr>
<tr>
<td>Third Party</td>
<td>35.4%</td>
<td>25%</td>
<td>21.6%</td>
</tr>
</tbody>
</table>

Table III: Sub-Categories for Configuration Bugs.
cies [88], [89], these techniques do not update the code when dependencies are intentionally updated.

**Network Events (19.8%):** Despite being designed to handle network events explicitly, the controller contains a non-trivial number of bugs (19.8%) that are triggered by when it processes network events. Specifically, these bugs are tried while the controller is attempting to process OpenFlow messages. These bugs are often addressed by adding additional logic or adding exception handling code, indicating that the existing code is missing crucial logic for handling edge cases.

**Takeaways.** These observations highlight a need for novel fault tolerance techniques that either automatically rewrite code, or alter properties of the network event such that different code paths and cases are explored.

**Hardware Reboots (8.4%):** Hardware often reboots for a variety of reasons. Unsurprisingly a non-negligible set of bugs (8.4%) are due to these reboot events. Surprisingly, we observed that hardware reboot-triggered bugs are related to reboots of the optical components (e.g., ONU, OLT etc.), which points to the importance of tracking bindings between hardware configurations and their corresponding components in the abstraction layer (e.g., VOLTHA [90]). For example, in VOL-549 [91] (Figure 6), the VOLTHA core thread gets stuck waiting for the adapter to connect if OLT reboots after initial activation. This bug was fixed by adding a timeout variable.

**Takeaways:** Anecdotal evidence suggests that such bugs exist because testing environments lack representative failures and equipments [92]. This is a clear sign that emerging approaches to apply Chaos-Monkey style [93] fuzz testing to SDNs are needed, and more work should be done to extend the practicality of such techniques.

**Broader Takeaways for Research:** A significant set of bugs are due to interactions between the controller and external services (e.g., configuration files, network events, or function calls). These observations suggest that these controllers lack sufficient code for checking for valid inputs. Additionally, these bugs demonstrate a tight-coupling between the controller and the broader environment. As the environment evolves, care must be taken to ensure that the controller’s codebase evolves accordingly. We need better tools to track dependencies and highlighting mismatches. Additionally, the developers of the SDN controllers need to introduce better error-guarding logic. Finally, while there is significant work [94]–[97] on addressing system misconfiguration, there is very little work within the SDN space.

**B. Resolution Time for Triggers**

Figure 7 shows the CDF for resolution times for bugs on the basis of the triggers categorised in Table I. In the above analysis, we observed that most bugs are triggered by configuration, but we also found it has the longest tail, which reveals that they are the most severe bug trigger category that could take considerable time to be resolved. It is observed that ONOS has a longer tail as compared to CORD in most of the trigger categories (Configuration, External call, Network event) which could be attributed to its more complex structure (LoC, classes, functionalities). For example, we found a serious ONOS-5992 [98] which impacted multiple versions before it could be fixed, and the fix required addressing multiple bugs: In this bug, killing one ONOS instance resulted in a cluster failure. On the contrary, we observed that bugs triggered by reboot have a longer tail for CORD than ONOS: this was because CORD has specialized code for disaggregated optical equipment, which involves complex configurations, e.g., EPON, GPON [99] and complex logic for tracking the state of these devices.

**VI. ANALYSIS OF SOFTWARE ENGINEERING PRINCIPLES**

In this section, we analyze the software engineering practices of the different controllers. We start with an analysis of technical debt [100] (§ VI-A) and how it impacts code fixes. Then we perform a burn analysis (§ VI-B) of FAUCET to understand how changes to the codebase impact FAUCET’s bugs and how they are triggered.
A. Smell-Analysis for Code-quality

SDN controllers are subject to a large number of code changes over time to meet the evolving demands and fix existing bugs; however, such changes eventually lead to software technical debt \[100\] of software degradation. Code-smells is a popular software engineering technique for analyzing codebases to determine and capture a form of software degradation that is correlated to bugs \[101\]–\[103\]. We perform code-smell analysis on several different release versions of ONOS and analyze ONOS’ software degradation over time. Additionally, we use the refactoring techniques \[104\] within the code-smell analysis to co-relate and understand the type of bug fixes, i.e., No Logic Changes, Add New Logic, Change Existing Logic.

We use Designite \[105\] for our code-smell analysis: Designite utilizes code-quality metrics, and it supports 19 architecture smells along with seven design smells. In Figure 8, we present the code-smell results for various ONOS releases. Next, we describe the smells and focus on those with the most variation across different versions of ONOS.

Broadly, there are two classes of smells: architecture and design. Architecture smells capture system-level impact spanning across multiple components, whereas design smell captures component level impact. Note: while plot Hub-like Modularization \[106\] and Missing Hierarchy \[107\], we do not analyze them because their numbers are low and they have slight variation across controller versions.

1) Architecture smell \[108\]: We observe that while the number of commits per release decreased or became constant (Figure 10), the architecture smells scores (i.e., God Component, and Unstable dependency smell, in Figure 8) remain constant. This constant architecture smell score, despite a decrease in commits, indicates constant technical debt. We believe this constant debt is potentially due to a gap between developer practices for developing patches and refactoring techniques. Next, we elaborate on the specific scores:

   **God component \[109\]**: The God component captures the division of functionality across components and indicates code modularity, i.e., modularity of controller design. We observe in Figure 8 that the God component metric is mainly constant. Although the smell metric indicates the level of controller modularity is not growing, we observe that the average number of classes is growing for controllers; this implies that the controller architecture consists of huge classes that impact overall modularity. For example, while the metric remains stable, the package \texttt{net.intent.impl} had an increase in the number of classes from 49 to 107 from ONOS 1.12 to 2.3.0. We recommend that developers improve their codebase by making logical changes by decomposing huge classes and potentially changing the controller’s Control-Flow graph.

Unstable dependency smells: This smell uses the State Dependency Principle (SDP) \[110\] to capture the stability of dependencies within the controller codebase. Unlike other smells, these can be difficult to refactor because modifying one dependency can lead to cascading changes to other dependencies. Fortunately, we observe in Figure 8 that the unstable dependency smells have decreased steadily from versions 1.12–2.3: this implies that developers can more freely make changes to dependencies without fear of introducing bugs.

2) Design smells \[111\]: As with any software package, ONOS’s initial code releases consist of burst in commits due to prototyping new functionality with limited features and potentially unstable codebase: this is reflected in Figure 8 as an initial spike between versions 1.12–1.14 in the...
Design smells scores (Insufficient modularization, Hub-like modularization, Missing hierarchy, and Broken Hierarchy). However, after version 1.14, we observed a steady decrease in the number of commits and that the Design smells remained unchanged or largely constant. We note that constant design smells are problematic because design smells have a causal relationship with architecture smells [112]: in short, design smells cause architecture smells, and thus to improving design smells will also improve architecture smells.

Insufficient modularization [106]. This metric captures the modularization of an individual class (Note: this differs from the God component, which captures package-level modularization features). In general, developers can improve this score by changing existing logic and decomposing large and complex classes.

Broken Hierarchy [107]. This smell analyzes the relationships between super-types and sub-types and checks to ensure that sub-types do implement features of their types. This smell is generally an indicator of missing logic. For example, in Figure 9, we present Run class which has the ElectionOperations super-type, note that the Run class doesn’t include methods from its supertype ElectionOperation. After a major upgrade (ONOS-6594 [113]) which addressed severe architecture flaws, the Run class (and other related classes) was changed to be a subtype of AsyncLeaderElector – this change fixed the smell.

From Figure 8 we observe an initial spike in broken hierarchy smells (versions 1.12–1.14) demonstrating poor code modularization, and then we observe a reduction (versions 1.14–2.3) which indicates logic changes (add-logic, change existing logic) and restructuring of the existing methods. This conclusion supports the broad set of changes we observe for many of our bug fixes.

```java
public static class Run 
  extends ElectionOperation {

  private String topic;
  private NodeId nodeId;

  public Run(String topic, 
              NodeId nodeId) {
    this.topic = topic;
    this.nodeId = nodeId;
  }
```

Figure 9: Broken Hierarchy in class run as it doesn’t share an IS-A Relation with it’s Super-type.

B. Burn Analysis

This section focuses our burn analysis on FAUCET because of its size (1000’s LOC) and highly modular structure. Both properties make FAUCET an ideal candidate for burn analysis. Unfortunately, due to ONOS and CORD’s complexity and the interleaving of components within individual source files, we are unable to apply burn analysis.

We begin in Figure 11 by characterizing commits and changes to FAUCET’s based on the functionality’s triggering events: (1) Configuration (38%), (2) Network Functionality (35%), (3) External Abstraction (27%).

Unsurprisingly, we observe that most commits focus on increasing network function, which aligns with an SDN controller’s central role, i.e., to provide control over the network. In particular, we observed that most commits are focused on fixing and adding new network functionalities.

The configuration-related commits are the second major category of commits. We believe this can be attributed to complex cross-layer configurations interactions identified in Section V.

Finally, External Libraries’ dynamic nature poses a unique challenge for developers who need to make continual modifications to their code to ensure interoperability. To illustrate, in Table IV we present a list of external dependencies for FAUCET and the number of version changes required. We observe that critical packages, e.g., RYU (network management framework) and chewie (IEEE 802.1x implementation) are subject to most changes and have shorter release cycles than the controller itself. This mismatch implies that the controller will always use outdated versions to introduce correct and security problems (as illustrated in Section V). For example, in FAUCET-2399 [114], an update to chewie prevented the installation of FAUCET. A move towards flexible versioning practices [115] with a balance between agility and predictability in core packages could reduce these bugs.
Figure 11: Distribution of Commits in FAUCET Core Across Three Functional Subsystems of a Controller. A: Configuration, B: Network Functionality, C: External Abstraction.

<table>
<thead>
<tr>
<th>Dependency Name</th>
<th># version changes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chewie</td>
<td>19</td>
<td>802.1X standard implementation</td>
</tr>
<tr>
<td>eventlet</td>
<td>5</td>
<td>networking library</td>
</tr>
<tr>
<td>influxdb</td>
<td>1</td>
<td>time series database</td>
</tr>
<tr>
<td>msgpack</td>
<td>2</td>
<td>binary serialization</td>
</tr>
<tr>
<td>networkx</td>
<td>1</td>
<td>Network Analysis</td>
</tr>
<tr>
<td>pbr</td>
<td>1</td>
<td>management of setuptools packaging</td>
</tr>
<tr>
<td>prometheus_client</td>
<td>8</td>
<td>Monitoring system</td>
</tr>
<tr>
<td>pyyaml</td>
<td>6</td>
<td>YAML Parser</td>
</tr>
<tr>
<td>rys</td>
<td>28</td>
<td>component-based SDN</td>
</tr>
<tr>
<td>beka</td>
<td>5</td>
<td>BGP Speaker</td>
</tr>
<tr>
<td>pytricia</td>
<td>1</td>
<td>IP Address Lookup</td>
</tr>
</tbody>
</table>

Table IV: Burn-down analysis for FAUCET dependency requirements.

VII. BROADER IMPLICATIONS

In this section, we take a step back to understand the broader applicability and implications of our study on network operators. We focus on providing guidelines for (1) selecting controllers (§ VII-A), (2) debugging open issues (§ VII-B), and (3) navigating emerging diagnosis frameworks (§ VII-C).

A. RQ4: Controller Selection Guideline

Inspired by our observations, we provide general guidelines to aid operators in selecting controllers. Our guidelines focus on completeness, functionality, and SDN use cases. We observe that most problems in FAUCET are due to missing logic (specifically 52.5% of bugs), which makes it the least stable of the controllers that we analyzed. Although, CORD and ONOS are based on the same fundamental codebase, we observed that CORD is susceptible to significantly more load-related problems – 30% of bugs in CORD versus 16% in ONOS.

In Table VI, we show two critical use cases that SDN has enabled and the symptoms that affect the core functionality of these use cases. Building on the above observations, we recommend ONOS as the most stable and performant among the analyzed controllers. Unlike CORD, moving towards ONOS will require developers to find appropriate or develop applications due to a lack of rich applications. Moreover, we observed that FAUCET is specialized for a specific use-case, e.g., network slicing [131], [132]. Due to slicing’s inherent isolation, we note that using it outside of this narrow use case will often yield missing functionality and logic errors.

B. RQ4: Automating Operators Diagnosis

In the absence of a tool for holistically diagnosing and resolving bugs, we conclude this section by providing guidelines for expediting root-cause diagnosis and resolution. We do this by analyzing the correlations between the bugs and categories (in Table I) and exploring the uniqueness of the keywords (AKA labels) in the bug descriptions.

Correlation Analysis: Figure 12 shows the CDF of correlations between all possible bug and category pairs. The curve illustrates that while most bug-category pairs (93.72% of bugs) are fairly correlated, there is a long tail indicating strong-correlated bug categories (6.28% of bugs). For example, we observed that memory bugs are highly deterministic in nature. More interestingly, the bugs triggered by third-party service calls are highly correlated to the fix “add compatibility”, which fits with the observations that these bugs could be caused by argument mismatch between library versions. Surprisingly, unlike bugs in the core controller, these third-party bugs are correlated with the outcomes “Error message” and “Byzantine”.

Figure 12: CDF of Bug Category Correlation.

Figure 13: Trigger Distribution among the Whole Dataset. A: Configuration, B: System Calls, C: Third Party Calls, D: Network Events, E: Application Calls. B, C and E belong to External Calls.

<table>
<thead>
<tr>
<th>SDN use case</th>
<th>Symptoms</th>
<th>Deterministic</th>
<th>Bug Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>LegoSDN [18]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Ravana [15]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SCL [14]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RoseMary [122]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SCOUT [123]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JURY [125]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DPQoAP [126]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>


Figure 14: Unique Topic Percentage. A: Deterministic, B: Byzantine, C: Add Synchronization, D: Third Party Calls.

Keyword Analysis: To further understand these correlations, we analyzed the topics extracted by the NLP techniques. We hypothesize that these correlations reflect that specific classes of bugs have unique topics or keywords in the bug description. For example, memory bugs often have a null pointer and other similar exceptions in the bug description. In Figure 14, we listed the top bug categories based on topic uniqueness. We observe that these bug categories are the exact bug categories that have a high correlation discussed earlier. We observe that the uniqueness in topics spreads over all bug classifications. Specifically, bugs with Byzantine symptoms introduce significantly different topics and keywords in the bug description. Similarly, some bug types, e.g., deterministic bugs, have remarkably unique topics.

We also apply our NLP model, which is trained with the manually labeled dataset, onto the whole dataset of critical bugs we get from Jira to demonstrate NLP techniques’ potential further. This large dataset contains ~5X bugs compared to our manually labeled dataset. Figure 13 is the distribution of predicted trigger from the whole dataset. The result indicates that configuration error is the major trigger of SDN controller bugs, and when troubleshooting an SDN controller, the operator should pay more attention to potential configuration glitches. Compared to configuration, the bugs triggered by OpenFlow events only contribute a small part. Given the complexity of capturing, replaying the network events to reproduce a previous scenario, it is more clever to examine the network events after ensuring other more critical potential triggers. We also summarized the results for other aspects, such as the deterministic bug is the dominant bug type. Due to the limit of space, we skip the details in this paper.

Takeaway. These correlations and keyword analysis imply that for a non-trivial amount of bugs, being able to identify outcomes, symptoms, and extract keywords from the bug will allow developers and operators to narrow down the potential root causes and fixes. As part of future work, we anticipate that a decision tree can be developed to help restrict and narrow the developer and operator efforts in diagnosis.

C. RQ5: Selecting Recovery Frameworks

In Table VII we present a survey of existing fault tolerance techniques for SDNs. A key observation here is that no one technique can recover from bugs across all root causes effectively. Unsurprisingly, most techniques [13], [14], [48], [123]–[125] are able to recover from events triggered by OpenFlow messages which is the main focus of most SDN research. Yet, there are very few existing works within the SDN domain for interactions with configuration and external calls. We note that while non-SDN techniques, e.g., Lock-in-Pop [133], can address external events or configuration, these techniques need to be modified to address domain-specific issues.

We observe that most existing systems can easily recover from non-deterministic issues. However, there is very little for deterministic issues that account for most of the problems (as shown in Section III).

Table VII: Analysis of Bug Symptoms Accross Related Work.

<table>
<thead>
<tr>
<th>Symptoms</th>
<th>Category</th>
<th>SDN</th>
<th>Cloud</th>
<th>BGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>20%</td>
<td>59%</td>
<td>39%</td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>4%</td>
<td>14%</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Error Message</td>
<td>14.7%</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Byzantine</td>
<td>61.33%</td>
<td>25%</td>
<td>38%</td>
<td></td>
</tr>
</tbody>
</table>
Takeaway. Although we showed a plethora of systems that can diagnose or recover from different types of bugs, in practice, it is not trivial to combine these systems together to form a holistic system for the following reasons:

- Simply layering the systems on each other may introduce inefficiencies or impact accuracy. For example, while SPHINX [134] requires that all input OpenFlow messages to update a “flow graph”-based model, Bouncer [135] proactively filters out some input which may lead to an inconsistent flow graph and, thus, impacts accuracy.[1]

- Additionally, their expected inputs and system models are often fundamentally different; thus, integration is a non-trivial task. For example, while SOFT [156] analyzes output generated by different vendor implementations and CHIMP [137] analyzes output from different SDN applications, it is unclear how to compose the results from SOFT and CHIMP to provide a holistic, cross-layer approach to fault detection.

VIII. THREATS TO VALIDITY AND DISCUSSIONS

Generalizability. While limited, we believe that our analysis generalizes to future controllers because related work has shown that controllers follow a limited set of design principles that are well represented in the controllers that we studied. Specifically, the three controllers that we analyzed provide coverage over the following design choices: specifically, specialized (CORD) versus generalized (ONOS, FAUCET); monolithic (FAUCET) versus modular (ONOS, CORD); and distributed (ONOS, CORD) versus centralized (FAUCET).

Automated SE Analysis. Our automated code analysis is limited by the constraints of existing software engineering analysis tools, which only support specific languages (JAVA) or specific build systems (maven, gradle). For example, we could not perform smell analysis for FAUCET because it is written in Python, and the smell analysis codebase only supports JAVA-based software. Unfortunately, this limitation limits our ability to perform this analysis on a broader set of controllers.

Different bug management systems. The controllers use different bug management systems, e.g., GitHub (FAUCET), JIRA (ONOS, CORD), which could lead to variation in the type of information available. For example, JIRA provides Gerrit reviews, bug status, timestamps, etc while GitHub provides a different subset of data. These subtle differences impact the set of techniques, tools, and analysis that we could apply. For example, we could not analyze FAUCET’s resolution times because their GitHub repository does not provide this information.

Manual Classification. Our work involves both manual and automated analysis. While the automated analysis is susceptible to noise and bias, we note that we only use the automated analysis to support our manual analysis. In fact, most of our takeaways are based on manual analysis, thus minimizing the impact of learning-based noise on our observations. Our manual analysis’s validity is predicated on the fact that the bugs are accurately described and reported.

IX. RELATED WORKS

System-Research. In general, bug studies spanning across various domains [18], [19], [35], [138]–[141] lay the foundation for systems research. While prior studies have focused on distributed systems, we lack similar in-depth and comprehensive studies for SDN controllers. Unsurprisingly, we observed that, despite using a similar classification as prior work [18], [19], bugs in SDN controllers differ significantly in their distributions, motivating the need for studies such as ours.

SDN Bug Studies. Prior work on SDN bugs [11], [12], [142]–[145] analyze a smaller spectrum of bugs compared with our study, which provides a holistic and in-depth analysis of ‘critical’ SDN bugs. While our work focuses on understanding bugs and their implications, others [143]–[145] have developed stochastic models to help quantify the reliability of existing controllers.

X. CONCLUSION

Bugs are a crucial aspect of any software ecosystem, yet within the software-defined networking (SDN) community, we have a poor understanding of our bugs. Without a thorough understanding of these bugs, it is challenging to: (1) understand the efficacy of existing SDN fault tolerance techniques, (2) design representative fault injectors, or (3) identify key areas that are ripe for research. In this paper, our goal is to provide the knowledge required to fill this crucial gap in the community’s understanding of the SDN ecosystem by performing, to date, the largest bug study over three popular controller platforms.

XI. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Marco Vieira, for their helpful comments. This work was supported by NSF award CNS-1749785.

REFERENCES


