Towards Understanding Bugs in Open Source Router Software

Zuoning Yin and Matthew Caesar
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zyin2, caesar}@cs.uiuc.edu

Yuanyuan Zhou
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
yyzhou@cs.ucsd.edu

ABSTRACT

Software errors and vulnerabilities in core Internet routers have led to several high-profile attacks on the Internet infrastructure and numerous outages. Building an understanding of bugs in open-source router software is a first step towards addressing these problems. In this paper, we study router bugs found in two widely-used open-source router implementations. We evaluate the root cause of bugs, ease of diagnosis and detectability, ease of prevention and avoidance, and their effect on network behavior.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design

General Terms: Reliability

Keywords: Software errors, Internet routing, router software, protocols

1. INTRODUCTION

The Internet is an extremely large distributed system, comprising tens of thousands of competing ISPs, and hundreds of millions of end-hosts. The ability of this system to cope with such massive scales, to weather its continuous churn, and to adapt to new applications and threats, lies in the complex intertwining of systems and protocols that make up its design. Unforeseen and poorly-understood cross-dependencies between protocols and networks, the need to support a vast array of policy configurations and advanced features, the need to operate in untrusted environments, and the need to interoperate with a wide variety of other vendors and legacy code have caused Internet protocols to become exceedingly complicated.

Much of the Internet's complexity lies in software running on its constituent routers. Internet routers typically run an operating system (e.g. Cisco IOS or JunOS), along with a suite of protocol daemons which compute routes and distribute information about network state. Unfortunately, like any complex software, router software is prone to implementation errors, which have led to a number of recent high-profile vulnerabilities and outages [2, 12]. As one recent example [5], on February 16th, a small ISP network (called Supronet) performed a configuration change to shift traffic from one of its links to another ISP. It modified the configuration by increasing the size of the *AS Path* (the AS Path is a field of routing updates that store the list of ISP-level hops used to reach the destination) in routing updates it advertised out that link. The result was a particularly long AS Path, which could be correctly handled by the MikroTik routers deployed in Supronet. Unfortunately, Cisco routers contained a bug that would cause them to reboot when receiving a long AS Path. Worse still, after rebooting, Cisco routers would try to receive those "unexpected" updates again and reboot again, triggering continuous oscillations. While Supronet, which is in the Czech Republic, performed this configuration change around midnight (to minimize impact of accidental outages on their own local networks), it was mid-day/early evening in east Asia and U.S. when the fault occurred. The result was a hundred-fold increase in instability, traffic loss, and outages affecting nearly every country in the world. In general, the Internet was not designed to be resilient to buggy protocols, and hence bugs in router software can be devastating to network operation.

A bug characteristic study is usually the first step to understand the reliability issues in a particular software domain. Understanding software errors can support progress on reliable designs and related tools to address those reliability problems. There has been substantial work on studying properties of bugs [25, 10, 21, 23] in traditional software systems. Bug characterization studies [25] done by IBM in the early 1990's demonstrated the necessity of dealing with memory bugs, which motivated many commercial and open-source memory bug detection tools such as Purify [17] and DIDUCE [16]. A study of operating system bugs by Chou et al. [10] revealed that device drivers had substantially higher error rates than the rest of the kernel, motivating work on reliable device drivers such as Nooks [26]. There have also been characteristic studies of concurrency bugs [23], motivating new work on concurrency bug detection tools [22].

These characterization studies have shed light on the differences in bug characteristics across different software domains. Unfortunately, today we lack a characterization study of bugs in router software. Router software possesses several unique features as compared to traditional software. Router software is highly distributed, running across a large number of routers and disjoint ISP networks. It is also highly customizable, with rich configuration languages, which can lead to a highly diverse set of execution paths and significant implementation complexity. Router software also has extremely high requirements on availability and performance, and is often tightly coupled with an operating system.

A bug characterization study of routers may lead to deeper understanding of the kinds of bugs they suffer from, how they affect network-wide performance and operational considerations, how well current diagnosis and recovery schemes react to the bugs, how future router designs can be made robust to bugs, and may shed light on bugs occurring in other kinds of distributed systems. As a first step towards this goal, we propose a taxonomy of bug properties, and use it to perform a characteristic study of router bugs found in two widely-used open-source router implementations: Quagga [4] and XORP [15]. We leveraged static code analysis techniques [1, 20] coupled with manual classification of bugs in publicly available Bugzilla repositories to study the nature of router bugs. To make our study more general, we also evaluate bug and vulnerability reports on Cisco IOS. Finally, to understand complexity in the data plane of software routers, as well as inter-router communication within the control plane, we also study bugs in the Linux IP stack.

To the best of our knowledge, this paper is the first to characterize bugs in router software. We analyze common programmer errors that lead to router bugs, the effect bugs have on network operation, the efficacy of troubleshooting and working around bugs in live networks, and what components of router code are most prone to bugs. We give a taxonomy of router bugs, which characterizes the trigger condition, the impact to routing, and the efficacy of various detection and recovery schemes to a bug. We then make a number of observations on the origin, effect, detection and prevention of router bugs.

Roadmap: We first analyze two representative router software bugs in Section 2. We then describe our methodology for classifying bugs in Section 3. After that, we present the results of our characteristic study in Section 4, and conclude in Section 5.

2. EXAMPLE ROUTER BUGS

We start by describing two example bugs to show how we classify the properties of the bugs, and to motivate the categories we use to taxonomize router bugs.

Timer rollback causes wrong routing state: In the OSPF routing protocol, each link state update is tagged with the time it is advertised. This time is used to ensure consistency of routing databases: if two updates for the same link are received, the most recent one is used to be added to the local router's link-state database. According to bug report # 134 in Quagga's Bugzilla database, resetting the router's local time to an earlier time caused data packets to be lost. By inspecting the source code, we found that in Quagga versions prior to 0.98.6, this time was stored as the difference from the last-advertised wall-clock time, and hence resetting the system clock to an earlier time caused the advertised time in updates to become negative. Since the time was stored as an unsigned value, it could overflow and become very large. Since such large values could not be overwritten by later updates, these advertisements got "stuck" in router topology databases. From this information, there are several bug properties that immediately follow. Since routing messages can be lost, we know this problem would affect both control and data planes of the network, as it could cause router state to become inconsistent with network topology, and cause data packets to be lost. Since OSPF areas re-advertise messages at area boundaries, the *scope* of the problem would be contained within a single OSPF area, as such times are filtered at area boundaries. This problem was located in Quagga's timer code, and was fixed by having Quagga use "time since startup" instead of wall-clock time, when computing this value.

Control/data-plane inconsistency after interface flap: Some routing protocols require state on either side of peering sessions to be maintained in a consistent fashion. According to a bug report for Cisco IOS version 12.4, a very fast session flap can cause this state to become inconsistent, leading to lost routes. The report indicates the route can be lost when multiple conditions exist (a) an interface goes down for a very short period time (less than 500ms) (b) OSPF runs a shortest path calculation during this period and removes routes via that adjacency (c) the router's neighbor does not notice the flap, and believes the session remains up (d) the link-state advertisement damping is activated, and the damping timer does not expire until after the 500ms period. When these conditions hold, the local router drops routes traversing the link (since it ran the shortest-path computation while the interface was down), but does not advertise a new route (since the interface is up when it checks to see if a new advertisement is necessary), leading to a mismatch between how packets are forwarded (data plane) and the set of routes advertised by routers (control plane).

3. METHODOLOGY

3.1 Taxonomy

We classify bugs along five dimensions: (i) **Root cause:** This category helps us to understand the underlying factors that lead programmers to introduce errors in router software. This dimension is defined similarly to that of traditional software where bugs are usually classified into one of the subcategories of memory, concurrency and semantic bugs. For concurrency bugs, our definition is restricted to bugs that manifest at a single instance and doesn't include bugs that manifest during the interaction across multiple devices. We define a semantic bug as an inconsistency with the original design requirements or the programmers intention [21]. (ii) **Trigger condition:** This category studies what kind of inputs or events lead to buggy behavior. Studying this may give insights into ways to avoid or work around bugs at runtime. Those inputs or events could be routing updates, configuration changes, shell inputs, status changes of interfaces, etc. (iii) **Effect:** This category studies the effect the bug has on router level and network-wide behavior. We start by studying the particular end-result the bug has on router behavior. Next, we study whether the bug could be exploited by an attacker. Then, we study the scope of effect, and determine whether the bug affects the control plane, the data plane, or if it leads to a control/data plane inconsistency. (iv) **Code location:** Next, we study the particular location in the code base which contains the bugs. Doing this may give insights into which parts of router software are most prone to bugs. Given router software is commonly decomposed into several well-defined modules, we performed this task by classifying bugs according to which of these modules they appeared in. (v) **Operational issues:** Finally, we study how difficult it is to detect and work around the bug, in terms of what sorts of traces must be collected from the network to isolate the bug, and whether a simple or previously-proposed *model* of router/network behavior (rcc [13], FIB-RIB consistency checks, watchdog timers) can detect the problem. We also study the *localizability* of the bug, in terms of whether the bug can be isolated to a single router, area, or ISP, and taxonomize various techniques operators can use to *work around* the bug at runtime.

3.2 Router software

Unfortunately, vendors of commercial router software are (understandably) reticent to share their source code with us, limiting our abilities to comprehensively study these code bases. Hence, we instead focus on *open-source* router software. In particular, we study two *diverse* implementations (Quagga and XORP) which differ in their internal design, in an attempt to characterize bug features that are general across router software, as opposed to being biased to a particular implementation. Moreover, studying open source router software is becoming more interesting in its own right, as it is becoming more widely used in commercial routers [6] and third-party routing systems [11]. It also allows us to leverage static code analysis techniques, as well as allowing us to study the exact structure of the implementation errors in the code base. However, we expect our study to shed light on closed-source routers as well, since the open-source routers we studied are similar: they run the same well-specified protocols, use similar configuration languages and interfaces, and similar internal data structures and modules [9]. Furthermore, Quagga and XORP hit on many of the different configuration interfaces and implementation decisions used by various vendors, and hence collectively represent a wide space of router designs. For example, XORP processes updates in an event-driven fashion (similar to Juniper), while Quagga uses a timer to stage processing (similar to Cisco [27]). Also, XORP's configuration language is similar to Juniper's, while Quagga's is similar to Cisco's. We compare our results across the two routers, as well as investigating differences arising from their implementation structure and design decisions. To validate the generality of our results to closed-source router software, we also study a portion of bugs in Cisco IOS. To evaluate generality to security-related bugs, we also study security advisories on Cisco IOS posted on the NANOG mailing list. Finally, to characterize bugs arising in software router data planes, we characterize bugs from the IP stack [3] in the Linux kernel. We were unable to acquire bug samples from Juniper, as their bug database was not open.

3.3 Bug sources

To perform our characterization study of bugs, we need a set of bug "samples" to analyze. Finding bugs in software is in itself a highly challenging problem. To avoid this problem, we manually read each bug report from the Bugzilla repository, studied the affected region in source code, and used this information to classify and summarize the bug. Manual analysis requires substantial effort, as compared to using automated scripts to collect bug statistics based on the prevalence of keywords. However, using manual analysis allows us to derive more precise and richer information about the bugs we studied. We also supplement our study with static code analysis tools such as Coverity Prevent [1] and CPMiner [20] to evaluate effectiveness and accuracy of static analysis in discovering router software bugs.

After compiling our list of bugs from these sources, we then performed several steps to refine the list. First, we filtered out reports marked as *NEW*, *UNCONFIRMED*, or *INVALID*, since these bugs almost always had insufficient or incorrect information. Next, we manually removed trivial bugs from the list which did not affect router operation. These bugs included, for example, typos in documentation and compile errors. By doing the above steps, we eliminated around 63% of bug reports from Quagga, XORP and the Linux IP stack. We then randomly select 210 bugs (80 from Quagga, 80 from XORP and 50 from Linux IP stack) as a representative subset for detailed inspection. To ensure this number of bugs was representative, we also apply the taxonomy on smaller randomly chosen subsets of bugs (30 and 50), and achieved similar results (within a factor of 3.6% and 2.1% across all categories).

Bug set	# sampled	Open src?	Taxonomy
Quagga	80	Yes	Full
XORP	80	Yes	Full
Cisco IOS	104	No	Partial
Linux IP stack	50	Yes	Partial
Cisco Vulnerabilities	50	No	Partial

Table 1: Our bug sources

Since we did not have access to Cisco IOS source code, we used a more limited taxonomy for those bugs, including only effect, trigger condition and workaround. We chose to study the latest stable (mainline) release train, version 12.4. To reduce the number of bugs to a tractable amount, we selected non-trivial bugs (bugs with severity level 1 or 2 assigned to it by Cisco engineers) with keywords "OSPF" or "BGP" in either the title or body of bug reports. We then performed manual classification on each of the resulting 104 bugs. We also studied router vulnerabilities by selecting all Cisco Security Advisories posted on the NANOG mailing list from Jan 03 2005 through Sep 31 2008. We then randomly selected 50 bugs from those vulnerabilities. Therefore, we studied 364 bugs from Quagga, XORP, Cisco IOS and the Linux IP stack with our taxonomy.

3.4 Limitations

Performing a characterization study of router bugs presents a number of challenges. Given the rich variety of bug features, and the difficulty in isolating them, we must rely on manual classification. Therefore, inevitably, our study has some limitations.

First, we do not have access to commercial code. To deal with this, characterizations of traditional software focus their efforts on open-source counterparts, with the motivation that open-source implementations are becoming more widely used [11, 6]. We do study some bugs from Cisco routers with a partial taxonomy, however we remain unable to directly study commercial code. Second, for representativeness, we select bugs *randomly* from the Bugzilla bug databases with the goal to provide an unbiased sample of bugs that have been detected and fixed by the implementers. However, characteristics of unreported bugs are not visible from the bug databases. We deal with this partially by using static analysis to collect information about memory errors directly from source code. However, our study may not reflect characteristics of unfixed/unreported bugs of types not detectable by these tools. Finally, though we examine every possible piece of information we have about the bug, we cannot claim our taxonomy is complete.

Overall, our conclusions cannot be applied to *all* router software. However, we believe the main results of our study could give insights on a large class of existing router software. In addition, most of the characteristics we observe were consistent across both Quagga and XORP, along with substantial similarity to Cisco IOS bug reports, indicating the validity of our methodology to some degree. We do not

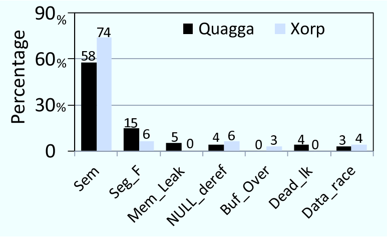

Figure 1: Root cause of bugs.

emphasize any quantitative characteristic results, and readers should view our findings together with our methodology and selected applications.

4. RESULTS

Here, we present the results in the five dimensions mentioned in Section 3.1.

4.1 Root Cause

Overall (Figure 1), we found that most bugs (72%) are due to semantic errors. Of these bugs, the majority (78%) were caused by missing cases, where the programmer overlooked a necessary step of the protocol. Next, we found that 21% of bugs were due to memory errors, 3% of which were caused by memory leaks, and 16% were due to invalid memory accesses (e.g., dereferencing an invalid pointer). With respect to concurrency bugs, we found a total of 6%, with 2% arising from deadlocks and 4% arising from data races. We found that concurrency bugs are fairly rare, since most protocol-specific execution occurs only within a single process. All concurrency bugs found were due to interactions between the router software and the kernel.

4.2 Trigger

We consider a bug to be triggered by an event or input, if the event could cause the code containing the bug to be executed, with the particular inputs to that code causing the bug to generate incorrect behavior. We classify a routers inputs into several categories, and count the number of bugs that are triggered by each (Figure 2): receipt of routing updates (39%, 46% of which were sensitive to the precise ordering or timing of these updates), timer expiry events (22%), commands entered at the vty shell (52%), changes made to the configuration file (63%), failure/repair of interfaces (12%), operating system events (such as resetting the system clock, 3%), and heavy update load (7%). These percentages do not sum to one because many bugs can be triggered by multiple different kinds of inputs. For example 52% of bugs triggered by routing updates could also be triggered by configuration file changes. Cisco IOS has similar results, but has more bugs triggered by configuration changes (92%). We found in many cases multiple specific configuration commands had to be simultaneously present at a single router (32%), or across multiple routers (14%).

4.3 Effect

As shown in Figure 3, we found the majority (58%) of bugs are non-fail-stop bugs: routers kept running but began behaving incorrectly. 39% of bugs caused the router to stop running (crash, hang or deadlock). 81% of bugs affect both control plane and data plane. Cisco IOS bug reports also show a similar trend. The results show that most router bugs do not cause fail-stop failures. We also roughly characterized security vulnerabilities introduced by Quagga and XORP bugs, by counting how many bugs could be *exploited* by a remote attacker. We found that for 26% of bugs, their symptoms could be made worse by the existence of a single compromised router within the network. In the Cisco IOS bug list, only 6% of bugs were vulnerabilities. For completeness, we also study bugs in the software router's data plane (the Linux IP stack). Overall, we found that most bugs (82%) in the Linux IP stack would affect the data plane of a software router. Nearly half of the bugs we studied (48%) will cause the kernel to stop running.

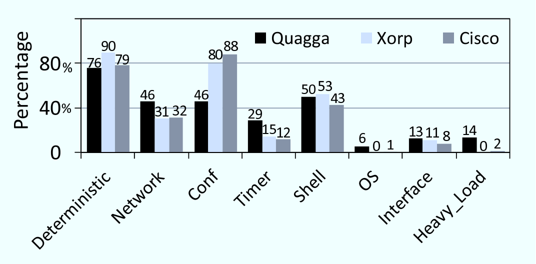

Figure 2: Trigger condition of bugs.

4.4 Code Location

We counted the number of bugs falling into each of the logical components of the source code as a means to analyze code location (as shown in Figure 4). We observed that 75% of bugs are related to the handling of protocol, policy and configuration, demonstrating the complex challenge of correctly implementing these features. For Linux IP stack, a substantial number of bugs are related to security- and filter-related functions such as Netfilter (16%) and IPSec (16%). To further understand the relative frequency of bugs across modules, we studied the complexity of those components in the router software. More specifically, we count the number of Lines of Code (LoC) that made up each component (Figure 4). Over half (50%) of the code implements interfaces to human operators (parsing configuration files, the vty terminal and logging). Interestingly, we find that the number of lines of code is not a good indicator of the likelihood of containing a bug. For example, policy-related logic comprises 4% of code yet comprises 28% of bugs, indicating policy-related code may be more prone to bugs than other parts of router code. To address this, future router and network designs may wish to focus on reducing policy and protocol complexity, for example by simplifying policy configuration [15, 7] or automating protocol implementation [14]. Besides, we observe that a significant amount of bugs reside in the interaction between different protocols (route redistribution). While recent work [19] finds that route redistribution, which translates routing information between different protocols, performs a highly crucial function and is often used to achieve ends not possible with traditional routing protocols, our results indicate that this glue logic is also particularly prone to router bugs, with 10% of bugs in Quagga, 18% in XORP and 20% in Cisco are associated with route redistribution code. In addition, a sizable amount of bugs were discovered due to their causing an interoperability problem between different vendors (e.g., the Supronet bug mentioned in Section 1). One reason for these bugs may be due to insufficient testing of interactions across protocols as well as insufficient testing of router implementations across different vendors.

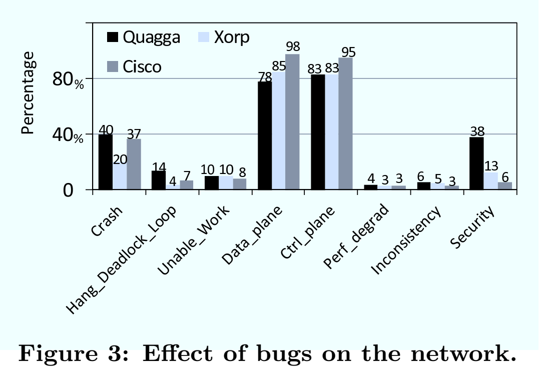

Figure 3: Effect of bugs on the network.

4.5 Operational Issues

Detection: Some bugs cause behavior that is clearly incorrect, for example bugs that cause crashes, making them easy to detect. However, other bugs may require more advanced forms of detection or domain-specific knowledge of network policies or protocol operation in order to be detected. To analyze this, we assumed network software running at a router could be instrumented with a simple *model* of correct behavior. This model consists of a simple set of invariants that router outputs should obey given their inputs, that would catch some (but not all) incorrect behavior. We then manually determined, for each bug sampled, whether the simple model can detect the bug from its manifestation. We found 45% of the examined bugs can be detected by incorporating several simple data-plane and control-plane checks into router behavior. More specifically, a sizable amount (13%) of bugs could be detected by adding some simple sanity checks into router software: 4% could be detected by checking consistency between the FIB and RIB, 9% could be caught by a simple *watchdog timer* which detects if the router is hung or unresponsive. Also, 32% could be caught by detecting that a crash has occurred.

Next, we evaluated the ability of a network operator to detect the bug by appropriate placement of vantage points. We found that if vantage points are placed correctly, 55% of bugs could be detected by solely observing control messages from routing protocols. However, 27% of bugs could not be detected by such means and required operators to look at router *logs*.

Next, we applied two static analysis tools [1, 20] to multiple versions of the two software router. Doing this may shed insights on the sorts of bugs that would be detected with more widespread use of static analysis tools, and to evaluate what sorts of bugs were overlooked by contributors to the Bugzilla databases. Static analysis tools are typically written to target certain classes of errors; for example, Coverity Prevent primarily detects memory access violations, while CPMiner detects copy-paste errors. Hence, these tools detected no semantic bugs when we applied it to router code. Coverity Prevent has some concurrency checkers but they are not effective in detecting real concurrency bugs in our study.

First, we manually cross checked defects found from static analysis of memory errors against the Bugzilla bug repositories. We selected the bug reports which describe a memory bug and have a patch available. With this criterion, we found 12 memory bugs from Quagga Bugzilla. However, Coverity Prevent reported none of these bugs even though it reported many memory defects which are not reported by Bugzilla. Second, focusing on the memory defects, we found that the Coverity Prevent had a false positive rate of 60%-96%, but accurately reported on average 99 non-trivial memory defects. We further analyzed the memory bugs missed by Coverity Prevent. We find that most of them only manifest under a particular interleaving of messages and events. We also tried CPMiner [20], but found nearly all defects reported by it were false positives. This may be because open source router software are smaller than large-scale software like Mozilla and Linux, and contains fewer modules, reducing the frequency with which code would be copied and pasted. While static analysis tools are still very helpful for router code, it is difficult for them to detect bugs manifesting under complex interleavings of messages and events.

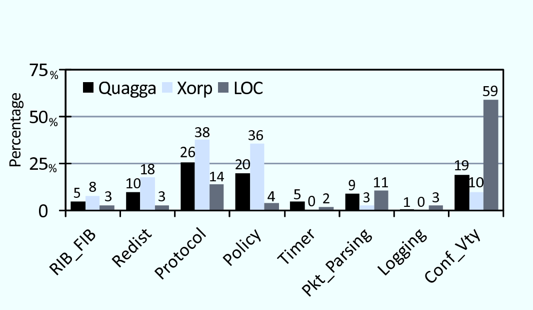

Figure 4: Code location of bugs.

In addition, all concurrency bugs we found are due to the interaction between kernel and routers. For example, when Quagga's bgpd process (BGP daemon) installs a route into the kernel's routing table, it enters a queue. One particular bug (# 268) allowed route withdrawals and advertisements to get reordered within the queue, due to a race between the kernel and bgpd. This in turn caused routes to be leaked (since reordering a withdrawal with a previous route advertisement could prevent the route from ever getting withdrawn). While Coverity Prevent contains techniques to localize concurrency problems, they were not effective in detecting the concurrency bugs in our study. This is also true for other modern concurrency bug detection tools. The fundamental reason is that all these tools only consider concurrency bugs that manifest across threads, they cannot detect concurrency bugs that manifest across processes (e.g., between the OS and applications). Considering the existence of many such bugs, new concurrency bug detection tools should be developed to address this scenario.

Diagnosis: Once a network is determined to contain a bug, network operators may wish to narrow down on the bug's location within the network. To evaluate this, we studied how well a single buggy router could be isolated within a typical ISP's network topology. Here, we assume that vantage points are placed randomly in the network, which can send data-plane probes and observe control updates. We found that 51% of bugs could be deterministically localized to a single router. However, a sizable amount of bugs could not: 22% of bugs could not be localized to a single router, and could only be localized to a set of routers within the ISP.

Once the bug has been isolated, a key challenge lies in reproducing the bug, because the state of router software is determined by a distributed computation, which may only arise due to certain orderings of network events, or only after running for long periods of time. We found that 41% of bugs required some form of interactions with other routers to be reproduced, while the remaining 59% could be reproduced at a single router being fed with routing updates. In addition 17% of bugs required specific routing updates or specific update orderings/timings to trigger the problem. These bugs are more challenging to reproduce. In addition, we found that a fairly large fraction of bugs (17%) are nondeterministic. Such bugs are especially difficult to localize, as the bug may not manifest every time.

Workaround: Most bugs can be "worked around" by network operators using simple manual techniques. After discovering a bug, and while waiting for the router vendor to repair it, the ISP may wish to temporarily work around the problem, to minimize damage the bug causes. Since bug reports often do not contain sufficient information to determine how to avoid the bug, and hence the percentages we give represent *lower bounds*. We found that 9% of bugs could be avoided by rolling back to an earlier version of router software, 6% could be avoided by changing the operating system on which the software router was running, 37% could be avoided by using the techniques of network virtualization [18] to change to a different functionally equivalent protocol at a low overhead (e.g., changing from OSPF to IS-IS), 18% could be avoided by restarting the operating system, 15% could be avoided by restarting the daemon, and 24% could be avoided by modifying the configuration in a functionally-equivalent way (in fact, even more could be worked around with a non-functionally equivalent change: for 54% of the Cisco bugs we studied, developers documented a configuration-change workaround). Again these percentages sum to a number larger than one because some bugs can be worked around in several alternative ways. Therefore, networks and configuration techniques should be designed to simplify workarounds. Network operators may be able to speed reaction by maintaining "fallback" configurations for use during times of errors. Developing automated techniques to automatically determine semantic-equivalent or semantic-similar configurations may be a useful research direction.

Moreover, future routers may be designed to perform *runtime adaptation* [24, 8], where the router automatically detects the problem and modifies its execution environment to avoid the bug. Here, we count how many bugs could be avoided by simple forms of runtime adaptation: 6% could be avoided by dynamically resizing internal buffers (e.g., to mitigate buffer overruns), 14% could be avoided by skipping particular configuration commands in the configuration file, 2% could be avoided by performing an automatic garbage collection procedure, 4% could be avoided by disabling assertions, 1% could be avoided by disabling logging.

5. CONCLUSION

To the best of our knowledge, this paper presents the first characteristic study of router bugs. We provide a methodology for classifying bugs based on a combination of static analysis and manual classification, a taxonomy of bugs with an emphasis on their network-specific properties, and present results for 364 bugs collected from two widely-used open-source software routers, Cisco IOS, and the Linux IP stack. Our results show that router bugs have particularly drastic consequences for network operation, commonly leading to outages and harming correctness of forwarding. Bugs are also challenging to detect and diagnose, though once they are detected they can often be temporarily worked around via simple configuration changes. For future work, we plan to apply the insights of this study towards designing more reliable router software. Moreover, we plan to extend the scope of our study by considering other sorts of network equipment, such as DNS servers.

6. REFERENCES

[1] Coverity prevent. http://www.coverity.com/html/coverity-prevent.html.

[2] The internet outage and attacks of october 2002. http://www.isoc-chicago.org/internetoutage.pdf.

[3] Linux kernel tracker. http://bugzilla.kernel.org/.

[4] Quagga software. http://www.quagga.net.

[5] Reckless driving on the internet. http://www.renesys.com/blog/2009/02/the-flap-heard-around-the-worl.shtml.

[6] Vyatta (open-source router vendor). www.vyatta.com.

[7] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing policy specification language (rpsl). June 1999.

[8] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI*, June 2006.

[9] J. Boney. *Cisco IOS in a Nutshell.* O'Reilly Media, Inc., 2005.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, October 2001.

[11] J. V. der Merwe, A. Cepleanu, K. D'Souza, B. Freeman, and A. Greenberg. Dynamic connectivity management with an intelligent route service control point. In *SIGCOMM Workshop on Internet Network Management(INM)*, September 2006.

[12] J. Duffy. BGP bug bites juniper software. In *Network World*, December 2007.

[13] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[14] T. G. Griffin and J. L. Sobrinho. Metarouting. In *SIGCOMM*, August 2005.

[15] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing extensible IP router software. In *NSDI*, May 2005.

[16] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, May 2002.

[17] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix*, 1992.

[18] E. Keller, M. Yu, M. Caesar, and J. Rexford. Virtually eliminating router bugs. In *CONEXT*, December 2009.

[19] F. Le, G. Xie, D. Pei, J. Wang, and H. Zhang. Shedding light on the glue logic of the internet routing architecture. In *SIGCOMM*, August 2008.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, December 2004.

[21] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID*, October 2006.

[22] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, October 2007.

[23] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, March 2008.

[24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – a safe method to survive software failures. In *SOSP*, October 2005.

[25] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *International Symposium on Fault-Tolerant Computing*, 1992.

[26] M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP'03*.

[27] R. Teixeira, A. Shaikh, T. Griffin, and G. M. Voelker. Network sensitivity to hot-potato disruptions. In *SIGCOMM*, August 2004.