An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis [*]

Swarup Kumar Sahoo
University of Illinois
201 N. Goodwin Ave.
Urbana, IL 61801
ssahoo2@illinois.edu

John Criswell
University of Illinois
201 N. Goodwin Ave.
Urbana, IL 61801
criswell@illinois.edu

Vikram Adve
University of Illinois
201 N. Goodwin Ave.
Urbana, IL 61801
vadve@illinois.edu

ABSTRACT

Reproducing bug symptoms is a prerequisite for performing automatic bug diagnosis. Do bugs have characteristics that ease or hinder automatic bug diagnosis? In this paper, we conduct a thorough empirical study of several key characteristics of bugs that affect reproducibility at the production site. We examine randomly selected bug reports of six server applications and consider their implications on automatic bug diagnosis tools. Our results are promising. From the study, we find that nearly 82% of bug symptoms can be reproduced deterministically by re-running with the same set of inputs at the production site. We further find that very few input requests are needed to reproduce most failures; in fact, just one input request after session establishment suffices to reproduce the failure in nearly 77% of the cases. We describe the implications of the results on reproducing software failures and designing automated diagnosis tools for production runs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.5 [**Reliability**]: Fault-tolerance

General Terms

Reliability, Testing, Verification, Fault-tolerance

Keywords

bug characteristics, network servers, bug reports, bug diagnosis, checkpointing

1. INTRODUCTION

1.1 Motivation

In-field software failures are becoming increasingly common with the increasing complexity of software. In addition to causing severe inconvenience to customers, such faults result in billions of dollars of lost revenue to service providers [10]. To this end, increasing the reliability of systems is becoming critically important. In spite of tremendous improvements in software engineering, testing and software reliability [34, 29, 30, 22, 31, 24, 33, 16, 27, 14, 13, 12, 7, 26, 36, 38, 40, 21, 15, 11], many software bugs still escape testing and enter production. As others have noted [36], performing off-site analysis of production run failures at development sites has several limitations: 1) it is difficult to reproduce failures at the development site due to differences in the environment, 2) customers have privacy concerns over what information can be released for off-site diagnosis, and 3) the same bug may generate a different failure at multiple production sites; it is cumbersome for the development team to investigate every failure that occurs without any automated feedback regarding the root cause of the failure.

The use of automatic software bug diagnosis techniques is a promising solution for fixing bugs found in production code. Automatic bug diagnosis has the potential to identify root causes of failure (both during development and production runs), create reduced test cases for filing bug reports, and automatically repair the software while it is in production. As an example, Triage [36] uses techniques based on checkpointing/rollback and re-execution to perform in-production bug diagnosis.

One challenge for automated diagnosis tools is that they generally require mechanisms to roll back and replay program inputs repeatedly in order to evaluate different root causes. Unfortunately, using checkpointing for such roll back limits the practical usefulness of such tools. First, checkpointing that is lightweight enough to be used for software diagnosis or debugging requires operating system support [34], which means that such tools cannot work on commodity operating systems that lack such support. Multithreaded processes also complicate the checkpointing procedure. In addition, diagnosis cannot be done if the fault happens before the checkpoint is taken. It would be very useful if diagnosis tools could simply *restart* the target program and replay a small subset of the inputs to reproduce and diagnose the failures. Exploring this question requires an understanding of real-world bug behavior. *One perhaps surprising implication of the present study is that such a*

[*] This work has been supported by NSF contracts CNS 07-20743, CNS 07-09122, CNS 07-16768 and CCF 08-11693, and by the Gigascale Systems Research Center (funded under FCRP, an SRC program).

simple approach is likely to work for most bugs in real-world server programs, as described in Section 7.

There also exist techniques like Delta Debugging [39] to generate a minimal set of test inputs from some failing test case. In general, such tools are too slow to be deployed in a production environment. Again, the knowledge of what types of inputs and how many inputs are commonly needed to produce failures can help us to build automated tools which use heuristics to efficiently select a minimal test case. However, in order to be widely deployable in production run environments, bug diagnosis tools must be able to quickly isolate the inputs that trigger a fault, reduce the inputs to just those that trigger the fault, and be able to reproduce the fault with reasonable reliability.

Server applications make these challenges more difficult as they run for long periods of time, handle large amounts of data over that time, and perform concurrent processing of input. Do the failures in server applications lend themselves to being automatically reproduced? Are there characteristics of server applications that ease or frustrate attempts to perform automatic bug diagnosis? Are there characteristics of inputs that ease the procedure to find a minimal test case? Our study, an empirical examination of real-world bugs in server applications, aims to answer these questions.

1.2 Goals of The Study

The main goal of this study is to answer and analyze the following questions, focusing on server software:

1. How many inputs are needed to trigger the symptoms of a software bug?
2. How long does it take for the symptom to occur after the first faulty input is used by the application?
3. What kinds of symptoms appear as a manifestation of bugs? Are these symptoms sufficient for creating automatic bug diagnosis tools?
4. What fraction of failures are deterministic with respect to the inputs?

The answers to these questions will have implications for designing automated diagnosis tools for server applications. Server applications have several qualities that make them ideal for such a study. They are widely used and mission critical for many businesses. They process a large amount of input data over extended periods of time, making it important to understand how many server inputs are needed to trigger failures, and how reliably they do so. They are also extremely concurrent, making it important to understand whether bug behavior (both normal errors and concurrent programming errors) is deterministic with respect to the inputs. While these qualities make them challenging, a silver lining is that their inputs are well-structured due to the nature of the protocols they use to communicate with clients.

We studied the above questions using public bug databases and committed bug fixes (patches) for six large open-source, server "infrastructure" programs that are widely used to build servers and Web server applications: Squid, Apache, Subversion (SVN), MySQL, the OpenSSH Secure Shell Server (`sshd`), and the Tomcat Application Server. Among the several servers we considered for this study, we attempted to include all the stateful servers. We randomly selected and manually analyzed a total of 266 randomly selected bugs and 30 specifically selected concurrency bugs in these six programs to answer these questions[1].

1.3 Findings and Contributions

The main findings of our study are as follows. Here, we define an "input" as a logical request from the client to the server that needs to be buffered to be able to reproduce a failure (see Section 3):

1. Nearly 77% of the failures due to software bugs can be reproduced with just one input request (excluding session establishment inputs like *login* and *select database* requests). Among the remaining bugs with clear information, only 12 of the failures need more than 3 inputs to trigger a symptom.
2. Among the 30 cases that require more than one input request to trigger a fault, in a majority of them (17 cases), all the necessary inputs are likely to occur within a short duration of time.
3. For most of the failures, the time duration between the first faulty input and the symptom is small.
4. For the majority of overall bugs (nearly 63%), the failure symptom is an incorrect output. In two of the applications (Squid and Tomcat), we find many fewer incorrect output symptoms compared to other applications; our analysis leads us to believe that this is because Squid and Tomcat use many more assertions and exceptions, respectively.
5. Nearly 82% of all the faults we analyzed were reported to have deterministic behavior. This is true, even though Apache, MySQL and Tomcat are extensively multi-threaded and even for stateful applications.
6. Concurrency bugs (e.g., data races or deadlocks) form a small fraction (<2%) of all bugs (even in highly concurrent applications), but they have very different, more complex characteristics: nearly all are non-deterministic, they usually require more inputs to trigger, they have more hangs/crashes and fewer incorrect output symptoms, but about 17% of them show different failure symptoms in different executions for the same inputs.

Our results have several implications for automated bug diagnosis tools:

1. Most bug failures can be automatically reproduced by replaying just the last input. Furthermore, most failures, including both single and multi-input failures, are triggered within a short time after receiving the first faulty input. Therefore, tools can optimize their search for inputs that reproduce the failure by first trying a small number of the most recent inputs received before the failure was detected. Systems can buffer only a small suffix of the inputs for each server instance.
2. In cases where multiple inputs are needed to trigger a fault, the symptoms are not only triggered quickly but the required inputs are also likely to be clustered together in the input stream, allowing an automated tool to optimize the search for the inputs.

[1] A detailed spreadsheet with all these bugs can be found at http://sva.cs.illinois.edu/ICSE2010/bug_statistics.xls

3. New techniques are needed to detect the internal data corruptions that cause incorrect outputs as they are the most common symptom. Inserting more assertions or exceptions manually or automatically appears to be a promising approach.

4. Isolating fault-triggering inputs for concurrency bugs and reproducing failures of such bugs reliably is significantly harder than for other bugs, but tools can exploit the fact that there are fewer incorrect output errors and many more hangs/crashes (i.e., failure detection is somewhat easier).

Based on these results, we also propose a new low-cost technique to reproduce failures during production and development. Automated bug diagnosis tools can buffer a suffix of the input requests to the server and upon failure, *restart the server and replay the suffix of inputs* to attempt to reproduce the failure. A few preliminary experiments suggest this approach may be able to reproduce server failures without checkpointing the server state.

The rest of the paper is organized as follows. Section 2 describes our methodology for choosing server applications and the bugs to study and the key limitations of the work. Section 3 presents our terminology. Section 4 details our primary analysis and classification of the bugs. Section 5 further analyzes the bugs which require multiple inputs to trigger them. Section 6 focuses on the characteristics of concurrency bugs. Section 7 describes the implications our results have on automated bug diagnosis tools. We describe related work in Section 8. We conclude and describe future work in Section 9.

2. METHODOLOGY

In this section, we describe our methodology for selecting the applications and the bugs we study in this paper; we also discuss the limitations of our study. A summary of the applications and software bugs is in Table 1.

2.1 Application Selection

Our study focuses on widely used, large server applications. We aimed our study at open-source applications that provide both a publicly-accessible bug database and access to the server's source code. When possible, we opted to study servers implementing stateful protocols (i.e., protocols requiring the server to maintain application-specific protocol state during an application-defined session) as these servers are the most likely to require more inputs for failure reproduction. Finally, we needed applications with a sufficient number of production bugs to study.

We considered many widely used Linux servers e.g., those implementing the IMAP, SMTP, FTP, DNS, LDAP, and NIS protocols, and selected six server applications. We found that only a few maintain significant state, and we deliberately included the stateful servers in our study. Three out of six servers in our study (Squid, MySQL, Tomcat), or the applications run on top of Tomcat, maintain significant state that affects reported server bugs. All six servers use TCP as the transport-level protocol for client communication.

Apache web server: The Apache web server communicates with clients via the stateless HTTP protocol [18] and via HTTP over SSL for authenticated and encrypted communication. While HTTP is stateless, Apache does maintain some information on the threads and processes it uses to handle requests. Apache can also maintain in-memory and on-disk state for caching recently served web pages.

Squid HTTP proxy server: The Squid web proxy server also communicates with clients via HTTP and HTTP over SSL. Like Apache, Squid can cache recently accessed web pages in memory and on disk. Squid's rate throttling features [4] must also maintain some global state.

MySQL database server: MySQL uses a custom protocol for client communication [2]. For each session, the server must maintain state about the authentication status and credentials of the client, temporary tables, prepared statements, and various parameters like query cache size and SQL modes. MySQL also supports clustering and replication, which can maintain a lot of global state.

Tomcat Servlet Container: Tomcat uses the HTTP protocol (optionally over SSL) for client communication. It maintains internal state about which web applications are loaded and provides facilities with which web applications can maintain session state across individual HTTP requests [8]. Tomcat bug reports are often generated by developers of applications running on Tomcat. Such applications can maintain arbitrary amounts of state, e.g., for e-commerce. This state will affect the behavior and reproducibility of failures due to Tomcat bugs. Tomcat also supports clustering and session replication, which maintains a lot of global state.

OpenSSH secure shell server: The OpenSSH server communicates via the stateful SSH protocol [37]. This protocol provides sessions which maintain one or more virtual channels over which programs on the client can communicate with programs on the server [37]. For each client connection, the server must maintain a small amount of state information about each SSH session as well as the state of each virtual channel maintained by the session.

Subversion version control server: SVN can use HTTP or a custom, stateful SVN protocol [5] tunneled through SSH. For this study, we restricted ourselves to the standalone server using the SVN protocol. The SVN server maintains a small amount of per-session state [5].

2.2 Bug Selection

We selected bugs to analyze for each server as follows.

First, for each application, we selected a recent major version of the software that had been in development and production use for at least a year. We expect these versions to have a more diverse bug sample. In a few cases, a single version of the software did not provide a sufficiently sized sample, so we used multiple versions of the software.

Second, we then selected the set of repaired bugs by using filters provided by the program's Bugzilla database. For all programs except MySQL, we searched for bug reports with a status field of either RESOLVED, VERIFIED, or CLOSED, a resolution field of FIXED, and whose severity was anything other than TRIVIAL or ENHANCEMENT. Since the filters for MySQL are non-standard, we had to adapt our selection criteria to search for bug reports marked FIXED or PATCH APPROVED/QUEUED and with any severity other than FEATURE REQUEST. Fixed bugs will have the most complete and accurate information, but may be biased towards easier-to-fix bugs (see section 2.3).

Third, we randomly selected a subset of bugs from the bug list generated in the previous step for each server using a seeded `rand()` function. We skipped this step for `sshd` and

Application	Description	#LOC	Years in Production	#total bugs after sampling	#bugs Selected
Squid 3.0.x	caching web proxy	93K	5	170	40
Apache 2.0.x	web server	283K	5.7	65	52
OpenSSH sshd 3.6-3.x, 4.x	secure shell server	27K	5.25	61	54
SVN 1.0.0 - 1.6.0	version control server	587K	5	16	12
MySQL 4.x	database server	1,028K	5.7	90	55
Tomcat 5	servlet container and web server	274K	5	70	53
Total		**2,292K**		**472**	**266**

Table 1: Set of Server Applications and Software bugs.

SVN since they had fewer bugs. The detailed information about the number of random bugs selected after this step is shown in the fifth column of Table 1.

Finally, we removed the bugs in development versions of code from the randomly selected bug list. Since our main goal was in-production bug diagnosis, we only focused on bugs in the externally released versions of code. Squid had many development bugs, so we had to select a much larger random sample compared to other servers. Then, we manually removed trivial bugs like build errors, documentation errors, and feature requests. After this final manual filtering, we were left with 266 bugs from 472 bugs in the earlier step. The last column of Table 1 shows the details for each server.

After creating the list of bugs, we analyzed each bug report and its test cases, available patches, and any other external information associated with it. We classified each bug's characteristics by first examining the information in the bug report and then by inspecting the patches and other external information if more information was needed to make a decision. We also used available patches to confirm our analysis of the bug reports. Based on the analysis, we classified each bug based upon the observable failure symptoms the bug caused, the reproducibility of the observable symptoms, and the maximal number of inputs needed to trigger the observable symptoms. For some bugs, we could not classify the bug for some characteristics due to limited information in the bug report. We report, for each characteristic, the number of such ”unclassifiable” bugs by placing them in a separate classification as described in Section 4.

2.3 Limitations

Our methodology's limitations should be considered when using or interpreting our results. Like other empirical studies, our results are limited by the kinds of applications and software bugs we used.

Our study examines a subset of server applications. Our results may not apply to other types of software or to network servers with different architectures e.g., peer-to-peer software. Also, all the servers except one are written in C and/or C++ (Tomcat is written in Java); results may differ for similar servers written in other programming languages.

There is also some potential bug selection bias in our study. First, some samples omit bugs that cause security flaws because their projects e.g., Apache [1] and `sshd` [3], do not record such bugs in their public bug databases. Second, our study omits unfixed bugs from the samples as their bug reports may contain inaccurate information. Since it is possible that unfixed bugs have different properties, it has the potential to introduce bias. Third, the bug samples in our study exclude unreported bugs. We believe this is acceptable because unreported bugs (in a mature server) are likely to be much less frequent than reported bugs. Nevertheless, it is possible that bugs whose failures are difficult to reproduce (which may include non-deterministic bugs and multi-input bugs) are less likely to be reported. Our results are, therefore, only representative of *reported bugs*. On the positive side, our study does not exhibit the bug feature and commit feature biases as described by Bird et. al. [6] as we sampled *all* fixed bugs in the bug database instead of sampling the subset of fixed bugs which are linked with the source code bug fixes or other bug fix information.

Finally, our study relies upon the accuracy and completeness of the bug reports and the materials connected to them. Also, as we have analyzed a large number of bug reports manually, there may be some human error in the results.

3. DEFINITIONS AND TERMINOLOGY

Here we define several terms as they are used in the context of our work. First, an *input request* to a server is a logical input from the client to the server (we identify logical inputs at the application level as reported in the bug reports); examples include a login/authentication request for establishing a session in SSH/MySQL, any command from an SSH or SVN client, an HTTP request, or a MySQL query. Each request may internally involve more than one communication between the client and server and may span several network packets.

We chose logical inputs because bug reports describe test cases as a set of application-level inputs. Translating inputs from bug reports into low-level network protocol messages would have made the study too tedious and error-prone.

Messages coming from sources other than the client e.g., the file system, back-end databases, DNS queries, are not considered to be input requests. These inputs are responses to requests that the server made on behalf of a client i.e., client input is driving these other inputs. Since there is a causal relationship between client inputs and these other inputs, replaying the client inputs will cause the server to perform the same requests.

We also exclude inputs that do not trigger the fault per se but are used, instead, to recreate a persistent environment in which the failure can be reproduced. Examples of such inputs are SVN checkout and SQL table creation commands (unless they are part of the faulty input set). In production systems, the environment is almost always present before the fault is triggered; these inputs are only in the bug report so that a similar environment can be manufactured at the developer's site for off-line diagnosis.

We define *symptoms* of bugs as any incorrect program behavior which is externally visible or detectable. For example, an incorrect output, segmentation fault, program crash, program hang, or assertion violation is a symptom. We use the term *incorrect output* to describe a symptom in which the server completes an operation but the external output of the program differs from the correct behavior. In our analysis, we say that an incorrect output symptom is detected when incorrect output is externally visible to the user or client. Though detecting incorrect outputs can be challenging, the internal data corruption that causes them can be detected through manually or automatically inserted assertions [17,

Application	Memory Error Crash	Other Crash	Assertion violation	Hang	Incorrect Output	Unclear
Squid	6 (15.00%)	5 (12.50%)	11 (27.50%)	0 (0.00%)	18 (45.00%)	0 (0.00%)
Apache	6 (11.54%)	2 (3.85%)	1 (1.92%)	2 (3.85%)	38 (73.08%)	3 (5.77%)
sshd	0 (0.00%)	5 (9.26%)	0 (0.00%)	3 (5.56%)	44 (81.48%)	2 (3.70%)
SVN	1 (8.33%)	2 (16.67%)	1 (8.33%)	1 (8.33%)	7 (58.33%)	0(0.00%)
MySQL	7 (12.73%)	8 (14.55%)	0 (0.00%)	3 (5.45%)	35 (63.64%)	2(3.64%)
Tomcat	10 (18.87%)	0 (0.00%)	12 (22.64%)	3 (5.66%)	25 (47.17%)	3 (5.66%)
Total	30 (11.28%)	22 (8.27%)	25 (9.40%)	12 (4.51%)	167 (62.78%)	10(3.76%)

Table 2: Classification of Bug Symptoms.

32, 13]. Detecting bugs is not a subject of this study; we simply treat incorrect outputs as a symptom.

A failure due to a software bug is observed to be *deterministic* if the fault triggers the same symptom each time the application is run with the same set of input requests in the same order, on a fixed architecture/OS platform and a fixed server configuration. Otherwise, the failure due to a bug is *non-deterministic*. The architecture, OS platform and server configuration are the user-controllable parts of the environment. This definition is reasonable because our goal is reproducibility at the end-user's site, where these parts of the environment and software configuration are fixed. A failure due to a bug is *timing dependent*, if the timing of the input requests in addition to their order determines whether a symptom is triggered and if so, which symptom. This is a special case of a non-deterministic software bug that is input timing dependent. As an example of our terminology, one of the failures in Squid (an assertion violation) occurred when a client sent an input request to the server and then disconnected from the server before the response for the request was written back. This is a timing-dependent bug as the symptom's occurrence depends upon when the disconnect request is sent to the server. The timing of the inputs matters and is under the client's control. In contrast, for a concurrency bug like a data race, the occurrence of the symptom depends upon timing issues that are beyond the client's control (e.g., thread scheduling); we classify such failures as *non-deterministic*, not as *timing dependent* .

Note that our study is conservative with respect to the definition of non-determinism. We classify a bug as non-deterministic if, according to the bug report, the symptom(s) could not be reproduced consistently for any reason (e.g., the same inputs may not be available or parts of the environment might have changed). It is possible that such bugs are deterministic, but we conservatively assume that they are not. Also, our definition of determinism is overly restrictive. We believe that many of the deterministic bugs in our study are actually deterministic across different environments and configurations as, in many cases, failures are reproduced by developers on different systems from the one where the bugs are first detected. But because bug reports often lack sufficient information about the bug's behavior across different environments, we chose to define deterministic to mean reproducibility in a fixed environment.

4. CLASSIFICATION OF SOFTWARE BUGS

We now present the results of our analysis of the selected server application bugs. We classify the software bugs based on three characteristics: observed symptoms (Section 4.1), reproducibility (Section 4.2) and the number of inputs needed to trigger the symptom (Section 4.3). Finally, we analyze the characteristics of a few important subclasses of bugs in (Section 4.4).

4.1 Bug Symptoms

We analyzed the bug reports in detail to determine the symptoms observed for each bug when the failure triggering input requests are sent to the server. For a small number of bugs, two symptoms were possible depending upon which inputs were used to trigger it; for these bugs, we chose to analyze the symptom that was given in the original bug report. The observed symptoms are memory error (segmentation fault, NULL pointer exception, and memory leak), program crash, assertion violation, program hang, and incorrect output. A program crash in this context is an abnormal server termination that is either not caused by a segmentation fault, NULL pointer exception, or memory leak or for which no cause is given in the bug report. We made this distinction because segmentation faults, Java NULL pointer exceptions, and memory leaks all have a memory error as the root cause while other termination errors may have other root causes. Assertion violation symptoms include explicit C/C++ checks and Java exceptions other than NULL pointer exceptions.

Table 2 shows the results. Each column shows the number of bugs that result in the corresponding symptom for each application. The last column shows the number of cases in which the bug report did not clearly identify the symptom; there were only 10 such cases out of 266 bugs. The number in parentheses shows the fraction of the cases resulting in the symptom as a percentage of the total number of bugs.

The results in Table 2 indicate that most of the bugs (nearly 63%) result in incorrect output. Incorrect outputs can be caused by memory errors (e.g., buffer overflows, dangling pointers), integer overflow errors, off-by-one errors in loops, logical errors, etc. The results also show that Squid and Tomcat have many more assertion violations (28% and 23%, respectively), but a lower percentage of incorrect output errors compared to the other servers. We suspect this is because Squid has many more assertions in its code than the other servers and because Tomcat uses Java exceptions.

Implications:

These results lead to two important conclusions:

1. New techniques are needed to efficiently detect the data corruption that causes incorrect output errors at run-time so that future diagnosis tools will be able to detect and diagnose the majority of software errors.

2. The results from Squid and Tomcat suggest that adding assertions or automatically generated program invariants may help detect incorrect output errors.

4.2 Bug Reproducibility

We did a similar analysis to determine the reproducibility of failures due to server bugs; Table 3 summarizes our results. We classified bugs as either deterministic, timing dependent, or non-deterministic. This property is useful, as

Application	Deterministic	Timing-dependent	Non-deterministic	Unclear from bug report
Squid	28 (70.00%)	3 (7.50%)	7 (17.50%)	2 (5.00%)
Apache	41 (78.85%)	0 (0.00%)	4 (7.69%)	7 (13.46%)
sshd	49 (90.74%)	1 (1.85%)	2 (3.70%)	2 (3.70%)
SVN	11 (91.67%)	0 (0.00%)	1 (8.33%)	0 (0.00%)
MySQL	46 (83.64%)	2 (3.64%)	4 (7.27%)	3 (5.45%)
Tomcat	43 (81.13%)	3 (5.66%)	4 (7.55%)	3 (5.66%)
Total	218 (81.95%)	9 (3.38%)	22 (8.27%)	17 (6.39%)

Table 3: Symptom Reproducibility Characteristics.

Application	# 0-input	# 1-input	# 2-input	# 3-input	# >3-input	Unclear	Max #inputs
Squid	8 (20.00%)	20 (50.00%)	1 (2.50%)	2 (5.00%)	2 (5.00%)	7 (17.50%)	3
Apache	3 (5.77%)	38 (73.08%)	1 (1.92%)	0 (0.00%)	2 (3.85%)	8 (15.38%)	2
sshd	4 (7.41%)	28 (51.85%)	14 (25.93%)	2 (3.70%)	2 (3.70%)	4 (7.41%)	12
SVN	0 (0.00%)	0 (0.00%)	9 (75.00%)	3 (25.00%)	0 (0.00%)	0 (0.00%)	3
MySQL	2 (3.64%)	0 (0.00%)	0 (0.00%)	40 (72.73%)	8 (14.55%)	5 (9.09%)	9
Tomcat	6 (11.32%)	34 (64.15%)	2 (3.77%)	1 (1.89%)	4 (7.55%)	6 (11.32%)	3
Total	23 (8.65%)	120 (45.11%)	27 (10.15%)	48 (18.05%)	18 (6.77%)	30 (11.28%)	12 (max)

Table 4: Maximal Number of Inputs Needed to Trigger a Symptom.

it will determine whether diagnosis tools can reliably reproduce the failure symptoms. In some bug reports, the failure could not be reproduced or was very difficult to reproduce (as it occurred infrequently). We conservatively classified such bugs as non-deterministic. Some bug reports contained no information at all about reproducing the failure; these are counted separately in the last column of Table 3. The numbers in parentheses in Table 3 are the corresponding fractions as a percentage of the total number of bugs.

The results are encouraging. More than 82% of the bugs demonstrate deterministic behavior; these results agree with Chandra and Chen's results that nearly 80% of bugs are actually environment-independent [9]. A few bugs (nearly 3%) exhibit timing dependence. Only 8% of the bugs exhibit other non-deterministic behavior.

It should be noted that, in practice, automatic diagnosis tools will normally replay a subset of inputs. While deterministic bugs trigger the same symptom each time when run with the same input sequence, they may also depend upon global state such as memory layout. If using a subset of inputs creates a different global state, then the bug may not trigger the symptom again even though the same root cause is triggered. Section 7 discusses this issue more.

Implications:

The above results indicate that:

1. Because most bugs are deterministic, bug diagnosis tools should be able to reproduce them by replaying inputs.

2. A small percentage of bugs are either timing dependent or non-deterministic. Bug diagnosis tools will need to incorporate new techniques (such as time-stamping inputs or controlling thread scheduling) in order to reproduce failures due to these bugs via input replaying.

4.3 Number of Failure Triggering Inputs

We now analyze and classify the software bugs based on the number of inputs needed to trigger the failure symptom. As mentioned in Section 2, we count each logical input as one input request. We determined the maximal set of inputs needed to trigger the symptom by examining each bug report and the other related external web resources linked to it. When more than one set of inputs could trigger a symptom, we used the largest set to compute the maximal number of inputs. We did not count changes to server configuration files or command line options as inputs. Table 4 shows the results of this analysis. The second, third, fourth, fifth, and sixth columns in Table 4 show the number of cases in which 0, 1, 2, 3, and more than 3 inputs are needed to trigger the corresponding symptom. The zero input column includes cases such as when the server experiences a failure after start-up but before it processes client input requests. There were a few bugs for which it was clear from the bug report that more than one input was needed to trigger the failure, but the exact number of inputs could not be determined as they were difficult to reproduce. We have counted them as requiring more than 3 inputs and included them in the sixth column. The seventh column, as in the previous section, shows the number of bugs for which the bug report contained too little information to determine whether one or more inputs are needed to reproduce the symptom. The last column shows the maximum number of inputs needed to trigger a failure due to any of the bugs we studied for that application. We used only the bugs for which we know the exact number of inputs to trigger a failure to compute the maximum number of inputs. As before, the numbers in parentheses in the table are the corresponding fractions as a percentage of the total number of bugs.

Appl	# ≤1-input	#>1-input	Unclear	Max #ips
Squid	28 (70.00%)	5 (12.50%)	7 (17.50%)	3
Apache	41 (78.85%)	3 (5.77%)	8 (15.38%)	2
sshd	46 (85.19%)	4 (7.41%)	4 (7.41%)	11
SVN	9 (75.00%)	3 (25.00%)	0 (0.00%)	2
MySQL	42 (76.36%)	8 (14.55%)	5 (9.09%)	5
Tomcat	40 (75.47%)	7 (13.21%)	6 (11.32%)	3
Total	206 (77.44%)	30 (11.28%)	30 (11.28%)	11 (max)

Table 5: Maximal Number of Inputs Needed to Trigger a Symptom Excluding Session Setup Inputs.

Our results show that most failures can be triggered using a very small number of inputs. Some symptoms can be triggered with zero inputs. Squid had a few such cases (nearly 20%). For example, when run with a certain combination of options or with a particular configuration, Squid failed after starting execution but before processing any client inputs. The majority of failures in Squid, Apache, and Tomcat can be triggered with just a *single input request*, in the case of sshd and SVN, using just *two input requests*, for MySQL, using *three input requests*. All of the 2 input request failures in sshd and SVN require one input for authentication and session establishment and another final input to trigger the symptom. All of the 3 input request failures in MySQL require two inputs for authentication and database selection, providing a session and execution context in which the final input could trigger the symptom. Table 5 lists the number of failures that can be triggered with no more than one input request excluding session establishment inputs. If we exclude the session setup inputs, then nearly 77% of the bugs in all applications need just a *single input request* to trigger the symptom; in all such cases, we have observed that it is always the last input in the corresponding session/connection which triggers the symptom. Among the remaining cases, nearly 11% of the bugs needed more than one input to trigger the symptom, and the remaining 11% of bugs do not have any clear information about the number of inputs in their bug reports.

Furthermore, considering all the bugs for which we could determine the exact number of inputs, all of the failures for Apache, Squid, SVN and Tomcat can be reproduced using at most 2, 3, 2, and 3 input requests (excluding session establishment inputs), respectively; only two failures in sshd and one in MySQL required more than 3 non-login input requests. In fact, very few bug failures (12 bugs across all server bugs excluding the unclear cases) needed more than three non-login input requests to reproduce the symptom.

Another result (not listed in the tables) is that, for most of the bug reports we studied, the symptom occurs immediately after the last faulty input is processed. In fact, the only exceptions were hangs and time-outs; for these, the time between the last faulty input and when the symptom can be observed is time-dependent but usually small. This suggests that the error propagation chain for these bugs is usually short and that a symptom can usually be detected immediately after the server processes the faulty inputs. These results agree with previous work [20] that found that error propagation chains in the Linux kernel are short in practice.

Implications:

These results have several implications for automatic bug diagnosis tools:

1. Virtually all failures can be triggered by replaying a small number of inputs.

2. Most of the failures can be simply reproduced by first connecting to the server, creating a session if necessary (through authentication and/or a database select request), and replaying a single input.

3. For most of the bugs, the last input request from the session/connection which triggers the fault can be used to reproduce the symptom.

4. Except for bugs which cause a hang or time-out, the failure symptom for a set of faulty inputs will occur immediately after the last faulty input is processed.

4.4 Analysis

Interestingly, when we consider only the subset of non-deterministic or multi-input bugs, many characteristics are very different from the overall bug characteristics. For example, among the 22 non-deterministic bugs, a majority of them (45%) are multi-input bugs and only 40% were single input bugs (remaining cases were unclear). Also, only 23% of the 22 non-deterministic bugs result in incorrect output, most of the failures result in catastrophic failures like crash, segmentation fault, assertion violation and hangs. This implies that majority of non-deterministic bugs need multiple inputs to trigger a failure and many fewer of them result in incorrect outputs as compared to the overall bugs. Of the 30 multi-input bugs, only 40% are deterministic, 27% are timing-dependent, and 33% are non-deterministic. This implies that many fewer multi-input bugs show deterministic behavior compared to the overall bugs.

5. MULTIPLE INPUT BUG ANALYSIS

Bugs that require multiple inputs to trigger a symptom are harder to reproduce and diagnose because the faulty inputs may be interspersed with non-faulty inputs, and the combination of inputs to explore can be large. We did a more detailed study of multi-input bugs to see if there are patterns that can be exploited to reduce the input stream to just the faulty inputs. Specifically, we wanted to see whether the set of inputs for triggering a multi-input failure were likely to be clustered together within an input stream or occur within a short time duration, which can help automatic tools to detect the symptom-triggering inputs more easily. Otherwise, automatic tools will need to use more complex algorithms to track down the faulty inputs.

There were 30 multi-input bugs (5 from Squid, 3 from Apache, 4 from sshd, 3 from SVN, 8 from MySQL, and 7 from Tomcat). For servers like Apache, Squid and Tomcat, any bug in Table 4 with more than one input is considered a multi-input bug. For sshd, SVN, and MySQL, we considered a multi-input bug to be any bug requiring more than two, two and three inputs, respectively, to trigger the symptom. Our rationale is that all of these two and three-input bugs require one and two inputs, respectively, for establishing a session e.g., authentication and/or database selection, and a final, single input for triggering the failure. These bugs are, in essence, single-input bugs with additional session establishment inputs that occur in a known location within the input stream and can be easily buffered for each session.

We classified each bug into one of three categories: *CLUSTERED*, *LIKELY CLUSTERED* and *ARBITRARY*. A bug is classified as CLUSTERED if the input requests must occur within some bound; this bound is often short. For this category, the faulty inputs are always going to be clustered within the input stream within a short period of time (less than a few minutes). We classify a bug as LIKELY CLUSTERED when we know that the faulty input requests are likely to occur within a short duration for most real-world inputs, but there is no bound. For these cases, there are reasonable cases in which the inputs may not be clustered. Inputs are classified as ARBITRARY if there is nothing to indicate that they must be or usually will be clustered within an input stream in real-world usage. This does not mean that the inputs will not be clustered; it simply means that there is no reason to expect that they are likely to be clustered for a single given input stream.

Table 6 shows our detailed analysis results. The second column shows the bug's ID from the bug database, the third column reports the number of inputs needed to trigger the symptom, and the fourth column succinctly describes the steps needed to trigger the symptom. The fifth column ex-

Appl	BugID	#Ip	Steps to trigger the bug	Conclusion	CLASS
Squid	1862	3	Send a POST request with an incomplete body, kill the origin server, and send the remaining body of the request.	All the events will happen within the time a complete request is processed, hence will most likely occur within a short time duration.	CLUSTERED
Squid	2276	>1	Send many NTLM authenticator requests.	The requests can be far apart.	ARBITRARY
Squid	500	3	Start downloading a file. Then start downloading the same file concurrently. Abort downloading the first file.	As the requests happen concurrently, they will occur within the duration of first download.	CLUSTERED
Squid	2096	2	Send a first request to a web page. Then, send a Second request to same webpage at nearly the same time.	Both the inputs will occur within a very short duration, before the completion of first request.	CLUSTERED
Squid	1984	>1	Send a lot of requests for a long time. No mention of any particular inputs causing the crash.	The requests can be far apart spread over a long period of time.	ARBITRARY
Apache	17274	2	First, try to authenticate with LDAP server with wrong login-password. Then try to authenticate again with same login.	Incorrect and correct logins will most likely happen within a short duration.	LIKELY CLUSTERED
Apache	33748	>1	Need a random number of inputs to cause the crash.	The requests can be spread over a long period of time.	ARBITRARY
Apache	34618	>1	Send requests which open a connection to LDAP server. First two requests work, but the third may crash server. Sometimes more such connections are needed for crash.	When many requests are needed, they may not occur within a short duration.	ARBITRARY
sshd	1156	3	First, login to a shell. Run any command, e.g. `sleep 1`; Then, Ctrl+c while the command is running.	Last two commands can happen within a short duration for short running commands.	LIKELY CLUSTERED
sshd	1264	12	Execute 11 commands through a library function cmd() which internally opens a new channel and closes it after executing the command.	All the commands will be sent consecutively and will most likely occur within a short duration.	LIKELY CLUSTERED
sshd	1432	3	Three successive failed login attempt.	All the three requests can occur consecutively (e.g. when someone doesn't remember the password).	LIKELY CLUSTERED
sshd	948	5	The bug is triggered by a denied ssh connection blocked by tcp_wrappers. The bug occurs after 5th blocked connection.	Five blocked connections can happen over a long period of time.	ARBITRARY
SVN	1626	3	First, login to server. Run any svn command, e.g. `svn commit`; Then, Ctrl+c while the command is running.	Last two commands will occur within a short duration, before the svn command completes.	CLUSTERED
SVN	2288	3	First, login to server. Then, run `svn lock test.txt;` followed by `svn commit test.txt;`, when there is no write access on root.	As in many cases, users commits after small changes, the last two commands will be within a short duration in such cases.	LIKELY CLUSTERED
SVN	2700	3	First, login to server. Then, run `svn lock test.txt;` followed by `svn commit test.txt;`.	As in many cases, users commits after small changes, the last two commands will be within a short duration in such cases.	LIKELY CLUSTERED
MySQL	1890	>1	Execute a series of INSERT queries in main thread, Execute some SELECT, UPDATE and DELETE queries on the same table in background thread.	According to the report, error occurs always after some queries execute concurrently, likely to be close together.	LIKELY CLUSTERED
MySQL	5034	5	`prepare stmt1 from "select 1 into @arg15"; execute stmt1; execute stmt1;`.	The execute stmt can occur far apart.	ARBITRARY
MySQL	8510	4	`set sql_mode = ONLY_FULL_GROUP_BY; then select round(sum(a)), count(*) from foo group by a;`.	In cases, when sql_mode is not set in the config file, it can be set long before running group by based queries.	ARBITRARY
MySQL	27164	5	Create InnoDB table; Immediately create MyISAM table containing a POINT column; Insert into the table with the POINT column.	In most cases, insert queries will be immediate after table creation.	LIKELY CLUSTERED
MySQL	4271	5	`prepare stmt1 from <explain complex select>; execute stmt1; execute stmt1;`.	The execute stmt can occur far apart.	ARBITRARY
MySQL	3415	6	In first thread `LOAD DATA INFILE 'file' into mytable;`, in second thread `INSERT INTO mytable2 VALUES(2)`.	As both requests are processed concurrently, they should occur within a short duration.	CLUSTERED
MySQL	15302	4	`load data from master; execute any command.`	As the requests are executed consecutively, they should occur within a short duration.	CLUSTERED
MySQL	186	9	In Master `create temporary table t(a int); reset master;`, in SLAVE `stop slave;reset slave;start slave;`.	reset master can occur after a long time of create.	ARBITRARY
Tomcat	37150	>1	When there is more than 1 simultaneous connection, run a long request like big dir listing.	As the requests are processed concurrently with other connections, they should occur within a short duration.	LIKELY CLUSTERED
Tomcat	27104	>1	Few inputs with clustering and session replication needed to trigger exception.	The inputs can occur independently.	ARBITRARY
Tomcat	37896	>1	When requests are being processed, kill one of the replication servers, all the web servers will fill up the max threads.	All the inputs will occur within socket timeout period, after which the threads will be unblocked.	CLUSTERED
Tomcat	26171	3	Start session through webpage, restart webapp, reload webpage in that session.	The two requests can occur far apart.	ARBITRARY
Tomcat	42497	2	Request a static file, get a 200 response with ETag. Request the same file again, getting a 304 response without ETag.	The two requests can occur far apart.	ARBITRARY
Tomcat	40844	2	Authenticate two users simultaneously with HTTP DIGEST.	The two requests will occur within a short duration.	CLUSTERED
Tomcat	45453	>1	Send requests to make JDBCRealm cache PreparedStatement and preparedRoles. Run two requests allowing two threads to call getRoles simultaneously.	First set of requests can occur far apart.	ARBITRARY

Table 6: Analysis of Multiple-Input Software Bugs.

plains why we classified the bug as we did, and the last column shows the classification we assigned to the bug.

We classified 8 of the 30 bugs as CLUSTERED and 9 bugs as LIKELY CLUSTERED. We deemed 13 of the bugs as ARBITRARY. The faulty inputs for the first two categories can be isolated relatively easily. Even for ARBITRARY cases, it should be noted that server applications normally run on multiple installations. To perform a successful diagnosis, we do not need to reproduce the failure at every possible installation; it is sufficient to reproduce it on a single installation. Thus, it is enough for the automated tools to work if the faulty inputs will cluster in at least one instance.

For designing bug diagnosis tools that replay input, the important factor is the time between the *first* faulty input and the symptom; this delay determines the minimum amount of information a replay tool must record in order to be able to catch all of the faulty inputs. Since we have determined that most faulty inputs are clustered together within an input stream, we know that the time between the first faulty input and the last faulty input is small, and since the symptom will occur shortly after the last faulty input (as described in Section 4.3), we can conclude that the time period between the first faulty input and the symptom is also small.

Implications:

There are two important implications of our results:

1. Most multi-input bugs (except for those in which the symptom is a hang or time-out) will trigger the symptom shortly after the first input. This means that a replay tool only needs to record a small suffix of the input stream to reproduce the failure.

2. The locality of multiple faulty inputs within an input stream makes it easier to create a reduced test case.

6. STUDY OF CONCURRENCY BUGS

Surprisingly, we found only three concurrency bugs out of 160 bugs in the 3 multi-threaded servers (Apache, MySQL and Tomcat) we studied in Section 4. This strongly indicates that there are relatively few concurrency bugs in servers relative to the *total* number of reported bugs (similar observations can be drawn from the data in [28]). One possible reason is that servers generally process each request relatively independently, producing fewer inter-thread interactions than other multi-threaded programs with more intricate sharing behavior.

Nevertheless, concurrency bugs do occur and may have different characteristics from other bugs as they usually involve multiple threads and may be difficult to reproduce. We therefore manually selected and analyzed a set of concurrency bugs to determine if they show behavior similar to that of the bugs in Section 4. We now present the results of our analysis of these bugs. As before, we classified the concurrency bugs based on three characteristics: observed symptoms, reproducibility, and the number of inputs needed to trigger the symptom.

We selected 30 concurrency bugs from Apache, MySQL and Tomcat by searching the bug databases based on a set of keywords like ’race(s),’ ’atomic,’ ’concurrency,’ ’deadlock,’ ’lock(s),’ and ’mutex(s)’ in a manner similar to Lu et al. [28]. The other servers have few concurrency bugs. Out of these 30, 23 were data race bugs and 5 were deadlock bugs (it was not clearly reported for two of the bugs).

Appl.	Deterministic	Timing-dependent	Non-deterministic
Apache	0 (0.00%)	0 (0.00%)	9 (100.00%)
MySQL	2 (18.18%)	2 (18.18%)	7 (63.64%)
Tomcat	0 (0.00%)	0 (0.00%)	10 (100.00%)
Total	2 (6.67%)	2 (6.67%)	26 (86.67%)

Table 8: Symptom Reproducibility Characteristics of Concurrency Bugs.

Table 7 shows the results of classification based on symptoms. Each column shows the number of bugs for the corresponding symptom or combination of symptoms. We include combinations of symptoms because some of these concurrency bugs were reported to produce different, yet specific, symptoms in different executions for the same input (e.g., a segmentation fault in some executions and an incorrect output in others). There are some interesting differences between concurrency and non-concurrency bugs. First, as expected, a much higher fraction of bugs produce hangs; most of them are due to deadlocks. Second, five (17%) of the concurrency bugs produced different, yet specific, symptoms in different executions, as noted above. Third, there are much fewer incorrect outputs with concurrency bugs (20% overall, but 45% in MySQL). It appears that a higher fraction of concurrency bugs produce catastrophic symptoms like crashes or hangs, which can cause service interruption. This observation is similar to that reported by Lu et. al. [28].

Table 8 shows our results from classifying bugs as deterministic, timing-dependent, or non-deterministic. Most of the bugs (87% overall, and 100% in Apache and Tomcat) show non-deterministic behavior. This underscores the difficulty of reproducing the symptoms of concurrency bugs.

Table 9 shows the results of classifying bugs based on the number of inputs required to reproduce the bug (note that the columns differ from Table 4). For some cases, it was not clear from the bug reports exactly how many inputs were needed (for example, bug reports mention that they need to run some test cases in stress mode with multiple threads repeatedly for a few minutes to trigger a symptom). We have conservatively classified these cases as requiring more than eight inputs. The fifth and sixth columns are identical in purpose to those of Table 4. We used only the bugs for which we know the exact number of inputs to trigger them to compute the maximum number of inputs. As before, the numbers in brackets in the table are the corresponding fractions as a percentage of the total number of bugs. There are several differences with our earlier results for non-concurrency bugs. First, all the studied concurrency bugs need multiple non-login inputs (>2) to trigger a symptom (as a comparison, very few non-concurrency bugs needed multiple non-login inputs for reproduction). Some cases need significantly more inputs, but none of the bugs in Apache and MySQL (for which bug report mentions exact number of fault-triggering inputs) needed more than 20 inputs. Second, many bugs needed executions with multiple threads and multiple client connections for some time to reliably trigger the symptoms. (For most of the bugs, though, the bug reports mention that the symptoms can possibly be triggered using 2 or 3 threads and client connections.)

Application	Seg Fault	Crash	Assert. Violation	Hang	Incorrect Output	Seg Fault/ Incorrect Output	Seg Fault/ Assert. Violation	Crash/ Incorrect Output	Crash/ Hang
Apache	1 (11.11%)	0 (0.00%)	0 (0.00%)	4 (44.44%)	1 (11.11%)	1 (11.11%)	1 (11.11%)	1 (11.11%)	0 (0.00%)
MySQL	1 (9.09%)	0 (0.00%)	0 (0.00%)	4 (36.36%)	5 (45.45%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1 (9.09%)
Tomcat	1 (10.00%)	1 (10.00%)	6 (60.00%)	1 (10.00%)	0 (0.00%)	1 (10.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	3 (10.00%)	1 (3.33%)	6 (20.00%)	9 (30.00%)	6 (20.00%)	2 (6.67%)	1 (3.33%)	1 (3.33%)	1 (3.33%)

Table 7: Classification of Bug Symptoms of Concurrency Bugs.

Application	# 0-2 input	# 3-8 input	# >8-input	Unclear	Max #inputs
Apache	0 (0.00%)	0 (0.00%)	6 (66.67%)	3 (33.33%)	20
MySQL	0 (0.00%)	3 (27.27%)	5 (45.45%)	3 (27.27%)	16
Tomcat	0 (0.00%)	0 (0.00%)	6 (60.00%)	4 (40.00%)	15000
Total	0 (0.00%)	3 (10.00%)	17 (56.67%)	10 (33.33%)	15000 (max)

Table 9: Maximal Number of Inputs Needed to Trigger a Symptom for Concurrency Bugs.

6.1 Implications

These results have several implications for automatic bug diagnosis tools for concurrency bugs.

1. As most of the concurrency bugs are non-deterministic and produce different symptoms, bug diagnosis tools will need to develop new techniques to reliably reproduce the symptoms.

2. A somewhat larger suffix of inputs will be needed to trigger the symptoms compared to non-concurrency bugs.

3. For most of the concurrency bugs, we will also need to use inputs from multiple different client connections. This can significantly complicate the process of reproducing the symptom and minimizing test cases.

Finally, note that our methodology successfully identified both non-deterministic behavior and also the need for multiple inputs in these 30 bugs. The *same methodology* found a very low occurrence of both these behaviors among overall reported bugs, as described in the previous section. This is an important validation of those (perhaps surprising) results for overall reported server bugs in Section 4.

7. IMPLICATIONS FOR DESIGNING AUTOMATED BUG DIAGNOSIS TOOLS

Bug diagnosis often involves one or more of the following steps: recording inputs that may lead up to a fault, reducing a small triggering input from the recorded inputs when a fault does occur, and using the inputs to somehow determine the cause of the bug.

New bug diagnosis tools may be able to exploit the bug characteristics we have found to improve their efficiency. We provide potential ways of using our results below.

7.1 Recording Server Input

Tools that automatically create reduced test cases for bugs need an recorded input that triggers the bug. However, recording all of the inputs to a server program is infeasible. Servers process vast amounts of inputs over long periods of time; a way to prioritize which inputs to save and which inputs to discard are needed.

Our results indicate that most bugs can be triggered with a single input, that some protocols require an authentication step before playing that single input, and that the inputs needed for reproducing a multiple-input bug are often clustered close together within the input stream. Additionally, we observed (like previous work [20]) that the symptom is often triggered immediately after the faulty inputs have been processed. Also, as discussed in Section 5, we found that the time duration from first faulty input to symptom is usually short for most bugs.

The implication is that an automatic bug diagnosis tool should be able to catch most bugs without recording the entire input stream. Rather, it should suffice to record a prefix of the per-session input (to replay session establishment and authentication, if necessary) and a suffix of the input (which will most likely contain all of the inputs needed to trigger the symptom).

However, Bug reports often try to reproduce a failure using a subset of the inputs containing all the failure-triggering inputs instead of the complete input stream. One concern is that the ability for a set of inputs to trigger a symptom may depend upon global state in the server that was affected by processing earlier inputs. For example, one set of inputs may cause the server to allocate memory in such a way that a buffer overflow triggers a segmentation fault while a subset of those inputs generate an incorrect output symptom due to a different heap layout. While a simple test case exists, how do we know that it can be reduced from a larger test case? In other words, can a suffix of inputs which contains all the failure-triggering inputs reliably trigger the same symptom that was triggered by the original input stream? We hypothesize that most deterministic bugs will generate identical symptoms even when earlier non-faulty inputs are removed from the input stream. Such bugs have a small number of failure triggering inputs and appear to have small error propagation chains. Moreover, Servers generally process each request relatively independently. Given this, we believe failure behavior of these server bugs is less likely to be affected by differences in global state.

We did a preliminary experiment to see if the global state of the server often affects whether a set of faulty inputs triggers a symptom. We selected four real bugs from four server applications: a stack buffer overflow bug due to an integer overflow (`sshd`), a NULL pointer dereference bug (Apache), and two heap buffer overflow bugs (Squid and NullHTTPD). For each bug, we created an input stream of 99 good inputs and appended to it the faulty input. We then proceeded to feed the input stream into the server, recorded whether the same symptom as reported in the bug report occurred, removed 10 good inputs from the input stream, and re-ran the experiment. We repeated this process until we had reduced the input stream to just 10 faulty inputs.

For all of the bugs we tested, the same symptom occurred after the faulty input was received regardless of the number of good inputs that preceded it. This suggests that the ability to reproduce a bug by replaying a suffix of the inputs to the server does not greatly depend upon global server state.

7.2 Automated Test Case Reduction

Our results may also help develop heuristics for guiding automatic test case reduction. For example, the ddmin minimizing delta debugging algorithm by Zeller and Hildebrandt [39] considers all test inputs as equally likely to trigger a symptom. However, our results indicate that most server bugs are single-input bugs, or require a small number of inputs that are clustered close together. As stated in Section 5, for most of the bugs, we find the time duration from first faulty input to symptom is usually short. Therefore, it may be possible to improve the efficiency of test case reduction algorithms by first systematically testing each small suffix of the recorded input stream to see if any of those inputs trigger the symptom before applying a more general algorithm like ddmin. For most of the bugs, this procedure is likely to be faster than ddmin. Also, some of the tools can possibly take benefit of the fact that very few inputs (usually ≤ 3) are needed to trigger the symptom by first searching small subsets before trying more complex algorithms.

7.3 Automated Bug Diagnosis Tools

The results of our study may also help improve tools such as Triage [36] that use checkpointing and replay to automatically diagnose bugs during production runs. Triage periodically checkpoints a program during normal execution [36]. When it detects an error, Triage instruments the code with additional detectors that help determine the bug's root cause and re-executes the program from the last checkpoint [36].

Our results have several implications for tools like Triage. First, a system like Triage can reduce the input stream to a much smaller set of inputs (as described previously), speeding up the bug diagnosis process by not replaying irrelevant inputs. Second, our results indicate that symptoms can be triggered by restarting the server and replaying a small number of inputs i.e., *without* checkpointing server state. This alleviates the need for checkpointing, However, techniques like checkpointing may not be practical due to various reasons. First, complexity of taking checkpoints (which needs specialized support from application and/or operating system [36, 34]) is a major issue. Second, multi-threaded process also complicates the checkpointing procedure further. Second, they can impose additional overhead on normal program executions. In addition, diagnosis cannot be done if the fault happens before the checkpoint is taken. Using a restart-replay mechanism will enable us to build a much simpler and much powerful automatic bug diagnosis tool.

8. RELATED WORK

Several previous bug characteristic studies have examined the bug reports of both open-source and proprietary software. Gray [19] studied the maintenance reports of customer systems to study trends in system outages; his results show that software is the dominant factor in system outages. Lee and Iyer [23] studied the root causes and error propagation behavior of bugs in the Tandem GUARDIAN90 system and concluded that consistency checks detected slightly more than half of the bugs in software and prevented error propagation for 31% of the bugs [23]. Sullivan and Chillarege [35] studied the root causes and triggers for bugs and determined that most bugs were caused by memory errors. More recent work by Li et. al. [25] aimed to update the results of Sullivan and Chillarege by studying the Mozilla web browser and the Apache web server; they found that memory errors were not the dominate cause of errors in these applications. Furthermore, they found that incorrect program behavior was the most common bug symptom [25]; our study confirms this result. Lu et. al. [28] studied many characteristics of concurrency bugs in open-source software; they found, among other things, that most concurrency bugs can be reliably triggered by controlling the schedule of four or fewer memory accesses. Chandra and Chen [9] studied the bug reports of several versions of Apache, GNOME, and MySQL to determine whether application-agnostic recovery techniques could recover from the majority of bugs detected in production runs. Our study confirms their finding that most bugs are deterministic [9]. As far as we know, no other bug study has examined the number of inputs needed to reproduce bugs or aimed to answer questions about the feasibility of creating automatic diagnosis tools that replay inputs.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we reported the results of an empirical study of bugs found in open-source servers. Our results show that most server bugs are deterministic and that failures due to most bugs (77%) can be reproduced by replaying a single input request (after session establishment if needed). Even for the remaining multi-input bugs, the set of inputs needed for reproducing failures is usually small and are often clustered together within the input stream. For most of the bugs, the time duration between first faulty input and the symptom is small. Our results also showed that many bugs produce incorrect outputs, indicating that better detectors are needed to flag errors in production runs. Most of the concurrency bugs, though, need multiple inputs to reproduce a symptom. Finally, we discuss how the results can be used by automated tools for doing in-production bug diagnosis. One of the key implications of the study is that most of the failures may be reproduced without checkpointing server state.

In future work, we intend to investigate ways of creating light-weight error detectors that can reduce the number of bugs that trigger incorrect outputs. Based on the results of this study, we also plan to build a tool capable of reproducing failures during in-production runs, reducing test case inputs and automatically diagnosing root causes of failures using a repeated restart-replay mechanism.

10. REFERENCES

[1] Apache Bugs Database. Website. https://issues.apache.org/bugzilla/.

[2] MySQL Internals ClientServer Protocol. Website. http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol.

[3] OpenSSH sshd Bugs Database. Website. https://bugzilla.mindrot.org/.

[4] Squid User's Guide. Website. http://www.deckle.co.za/squid-users-guide/Main_Page.

[5] SVN Protocol Description. Website.

http://svn.collab.net/repos/svn/trunk/www/webdav-usage.html.

[6] C. Bird et al. Fair and Balanced?: Bias in Bug-fix Datasets. In *ESEC/FSE '09*, pages 121–130, 2009.

[7] M. Bond. *Diagnosing and Tolerating Bugs in Deployed Systems*. PhD thesis, University of Texas at Austin, 2008.

[8] M. Bond and D. Law. *Tomcat Kick Start*. Sams, 2002.

[9] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN*, 2000.

[10] R. N. Charette. Why Software Fails. *Spectrum, IEEE*, 42(9):42–49, September 2005.

[11] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[12] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN*, 2006.

[13] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *PLDI*, 2006.

[14] M. Dimitrov and H. Zhou. Unified architectural support for soft-error protection or software bug detection. In *PACT*, 2007.

[15] M. Dimitrov and H. Zhou. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *ASPLOS*, 2009.

[16] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. 2003.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 2001.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[19] J. Gray. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, Oct 1990.

[20] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. *DSN*, 0:459, 2003.

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

[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proceedings of the Winter Usenix Conference*, 1992.

[23] I. Lee and R. Iyer. Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system. pages 20–29, Jun 1993.

[24] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 2006.

[25] 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*, 2006.

[26] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

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

[28] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[29] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[30] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.

[31] G. C. Necula, J. Condit, M. Harren, S. Mcpeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 2005.

[32] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, 2002.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. On Comput. Syst.*, 1997.

[34] S. M. Srinivasan, S. Kandula, S. K, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *In USENIX Annual Technical Conference, General Track*, 2004.

[35] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[36] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.

[37] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[38] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02: FSE*, 2002.

[39] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002.

[40] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proc. ACM/IEEE Int'l Symposium on Microarchitecture*, 2004.