An Approach for Assessing Logs by Software Fault Injection

D. Cotroneo*, R. Natella*†, A. Pecchia*†, S. Russo*†
*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed. 1,
Via Cinthia, 80126, Naples, Italy
{cotroneo, roberto.natella, antonio.pecchia, stefano.russo}@unina.it

***Abstract*—Nowadays, an increasing number of systems needs to be kept running for long periods without showing failures, but several factors compromise their correct behavior during the operational phase. Logs play a key role to address dependability issues of current systems and to enable proactive actions against failures (e.g., proactive maintenance, failure prediction). Nevertheless, they may lack any information in case of *software faults*, which escape the testing phase and are activated on the field by complex environmental conditions.**

In this paper, we evaluate built-in logging capabilities of a software system, namely the Apache Web Server, by means of an extensive software fault injection campaign. We experience that, in most of cases, software faults lead to failures without leaving any information in Apache logs. For this reason, we provide a few guidelines for developers that can be used during the development cycle, in order to improve the effectiveness of logs during the operational phase.

***Index Terms*—Field Failure Data Analysis, Software Fault Injection, Log Evaluation, Web Server.**

I. INTRODUCTION

Nowadays, an increasing number of systems needs to be kept running for long periods without showing failures, as in the case of mission critical software. At the same time, the need to provide users with rich and high-performance services leads to a growing complexity of both system architecture and software. As a result, several factors compromise the correct behavior of the system during the operational phase. Latent bugs, misconfiguration, and other operator errors may cause occasional failures [1], [2], [3].

An in-depth understanding of the nature of failures plays a key role to address dependability issues of current systems; therefore it represents an important enabler for proactive actions aiming at anticipating upcoming failures. For instance, data about failures enable the identification of the most failure-prone components and their Time To Repair [4], which can be used to schedule proactive maintenance, as well as to predict the occurrence of failures [5].

A viable solution to gain this understanding is provided by Field Failure Data Analysis (FFDA). An important source of failure data is represented by system and application logs, which are often the only available source of information about the system running state. The effectiveness of FFDA results strictly relies on the quality of logs and their content. Collected events can be broadly classified in two categories: (i) informational events, (ii) exceptional events. The former provides information about current operations and/or state of the system. The latter provides data about errors that occur at run-time, i.e., events related to erroneous behaviors that may prevent a computation to correctly succeed. For example, at the operating system level, these events may notify I/O errors and resources exhaustion. With regard to the application layer, log events usually notify unreachable servers or DBs, unavailable services, expired timeouts and so on. These events are intended to ease the analysis of problems caused by execution environment.

This paper addresses the effectiveness of logging mechanisms in presence of software faults (also known as bugs). It is difficult to locate them in advance during the testing phase. As a matter of fact, they can be activated *on the field* by complex environmental conditions that are difficult to reproduce [1]. Although software faults manifest during the operational phase, logs may lack any information about them [6], [7]. For example, in C/C++ programs, wrong pointer manipulation can originate a crash before any useful information is logged.

This paper reports a practical experience involving the Apache Web Server. We evaluate its built-in logging capabilities by means of an extensive software fault injection campaign. During experiments, we collect data (e.g., memory dumps) in order to (i) devise the most common failure modes and (ii) identify possible logging deficiencies. We experience that, in most of cases, software faults lead to failures without leaving any information in Apache logs. For this reason, we provide a few guidelines for developers that can be used during the development cycle, in order to improve the effectiveness of logs during the operational phase.

The paper is organized as follows. Section II discusses related work about FFDA and fault injection. Section III describes the proposed logging evaluation/improvement approach. Section IV describes the fault injection framework. Section V describes our experience with the Apache Web Server. Section VI concludes the work.

II. Related Work

Several research studies focused on log analysis for complex software systems. There are also past works devoted to evaluation of logging mechanisms. In [8], it is shown that event logs may be incorrect and may lack suitable data for analysis, leading to misleading conclusions. A study on Unix workstations and servers [9] also recognizes that event logs may be incomplete or imperfect, and it describes an approach for combining different sources of data to improve availability estimations. In [10], a study on a networked Windows NT system shows that most reboots (58%) do not show any specific reason, thus highlighting the need for better logging techniques. A study on supercomputers [11] shows that logs may lack useful information for enabling effective failure detection and diagnosis. It also suggests that it would be useful to include *operational context* information (i.e., the time at which the log was produced such as scheduled downtime, production time, and so on) along with log entries. These works evaluate logging mechanisms by means of log files collected during the operational time. In this work we investigate how logging mechanisms can be evaluated and improved *before* the operational phase by means of fault injection, which can reveal deficiencies affecting FFDA results (e.g., unlogged failures, misleading logs).

In [12], a log format is proposed for enabling *autonomic computing*, i.e., log messages have to be well-structured and categorized into standard classes, in order to be suitable for automatic analysis. A log quality metric is also proposed for evaluating how comprehensive and expressive log files are, although there is not a systematic approach for analyzing large log files from complex systems.

Fault injection is a well known approach for dependability evaluation. In the last decade, a framework has been formulated for rigorous evaluation of computer systems with emphasis on software faults, namely *dependability benchmarking*, which has been used for comparisons between systems and for risk assessment [13], [14]. Fault injection has also been used for fault removal in fault tolerance mechanisms [15], [16]. To the best of our knowledge, no previous work was devoted to the systematic improvement of logging mechanisms in complex systems. In [17], logging mechanisms are compared with other failure detection techniques for web applications using fault injection. Although logs are able to detect failures due to resource exhaustion and environment conditions, they provide little coverage with respect to emulated software failures (e.g., a deadlock). This motivates an in-depth analysis of logging mechanisms through injection of realistic software faults.

III. Overall Approach

In this paper we present an approach based on the extensive injection of software faults into the source code of a complex system. This is done to figure out potential failure modes not covered by logging mechanisms. In turn, this analysis is used to improve logging by inserting additional mechanisms into those source locations that are the most affected by error propagation. It may be too difficult to correct all faults in advance, since they can be activated under rare and complex conditions [1]. Logging their effects during the operational phase is useful to cope with them.

Fig. 1 shows the causal relationship between faults and failures. Logging mechanisms can be placed within the execution path leading to a failure state (e.g., an assertion about the correctness of a variable; a log is produced if the assertion is violated). As an alternative, logs can be produced when a failure state is reached (i.e., a wrong service is perceived by the users).

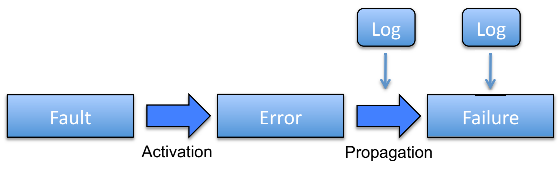


Fig. 1: Fault propagation chain and logging mechanisms placement.

To this aim, we identify the most likely failure modes using fault injection, and we suggest how to place additional logging mechanisms to cope with them in order to increase the probability that a software fault will be logged after its activation.

The proposed approach is composed by the following steps:

- *Execution of the fault injection campaign, and collection of failure data.*
- *Classification of failures*. We evaluate the effectiveness of logging mechanisms by means of the presence of log entries due to the injected fault, in order to identify not logged failure modes. The ratio between the number of logged failures and the total number of failures represents the *relative coverage* of existing logging mechanisms, which can be automatically evaluated. Future work will encompass definition of further metrics based on log relevance (i.e., to evaluate if a log is misleading). Failures are also classified by type (e.g., crash, hang) in order to simplify subsequent analysis.
- *Ranking of failure locations by frequency of occurrence*, in order to better focus development efforts. A failure location is a source code location in which a failure surfaced.
- *Reverse analysis of failure modes, and addition of logging mechanisms*; logs are placed in the source code by following appropriate guidelines.

In this paper, we will give a few examples of guidelines, based on the close examination of failures experienced in the case study. Future work will encompass the definition of a rich guidelines library, in order to make it easier the logging improvement process.

IV. Fault Injection Framework

The fault injection approach adopted in this paper takes advantage of past work in the field of Software Fault Injection, in

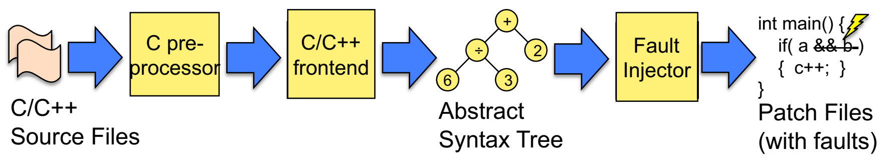


Fig. 2: Steps followed by the fault injection tool.

particular the G-SWFIT technique described in [18]. G-SWFIT copes with the representativeness of software faults, by using *fault operators* based on real faults found in several open-source projects. Representativeness is a major concern, since we aim to help developers to improve logging mechanisms against software faults. The more representative are injected faults, the more logging mechanisms will be able to provide information about real faults when they are triggered.

We inject software faults by means of modifications in the source code of the program, which is different than G-SWFIT. In G-SWFIT, faults are injected by means of changes in the binary code corresponding to programming mistakes in the high-level source code. G-SWFIT is suitable for OTS software items when the source code is not available, but there may be discrepancies between high-level software faults and binary changes (in [18], on the average there are 9% more binary changes not corresponding to high-level software faults, due to the usage of C macros in the target source code). Injection of software faults in the source code avoids those inaccuracies. Moreover, G-SWFIT requires additional efforts to be adapted to the system of interest, because of hardware/OS/compiler heterogeneity. Instead, injection in the source code is portable among all platforms supported by the original program, without any additional efforts. It should be noted that both techniques need to be adapted to the particular programming language of the target program, in order to represent high-level faults. The drawbacks of our approach are (i) the injection time (source files have to be re-compiled after fault injection) and, (ii) the need for the source code (if it is not available, the approach can not applied). We chose the high-level fault injection approach, because of source code availability in our case study, and the greater accuracy of injected faults.

We developed a support tool[1] in order to automate software fault injection. The tool takes as input a source code file, and it produces a set of faulty source code files, each containing a different software fault. Faulty source code files can be used to compile several times the Web Server, each time including an individual software fault.

Fig. 2 summarizes the steps followed by the fault injection tool for each source file. First, a C preprocessor translates the C macros contained in the source code (e.g., inclusion of header files, macros for conditional compilation, constants), in order to produce a complete compilation unit. A C/C++ front-end[2] elaborates the compilation unit, and it produces an Abstract Syntax Tree (AST) representation suitable for the subsequent processing by the Fault Injector program. The Fault Injector searches for fault locations in the AST, and applies fault operators if specific criteria are met. Fault operators are based on previous studies on software faults [13], [18]. In this paper, the most frequent software faults are considered for fault injection. Fault operators used by the Fault Injector are listed in table I.

TABLE I: Fault operators (see also [18]).

Acronym	**Explanation**
OMFC	Missing function call
OMVIV	Missing variable initialization using a value
OMVAV	Missing variable assignment using a value
OMVAE	Missing variable assignment with an expression
OMIA	Missing IF construct around statements
OMIFS	Missing IF construct plus statements
OMIEB	Missing IF construct plus statements plus ELSE before statements
OMLAC	Missing AND clause in expression used as branch condition
OMLOC	Missing OR clause in expression used as branch condition
OMLPA	Missing small and localized part of the algorithm
OWVAV	Wrong value assigned to variable
OWPFV	Wrong variable used in parameter of function call
OWAEP	Wrong arithmetic expression in parameter of a function call

V. Case Study

A. *Experimental setup*

Fig. 3 shows the testbed configuration adopted in this work. It is made up of two Personal Computers, (i) a Server Machine (Intel Pentium 4 3.2 GHz with Hyper-Threading, 4 GB RAM, 1000 Mb/s Network Interface), and (ii) a Client Machine (Intel Pentium 4 2.4 GHz, 768 MB RAM, 100 Mb/s Network Interface). An Ethernet LAN interconnects these machines. The Apache Web Server[3] version 2.2.11 is evaluated in this

[1]Available at http://wpage.unina.it/roberto.natella/

[2]In compiler theory, a front-end is the part of a compiler which builds the internal representation of a program, by means of lexical, syntactic, and semantic analysis.

[3]http://httpd.apache.org/

paper, and the httperf[4] tool version 0.9.0 is used to generate HTTP requests for the Web Server. The Apache Web Server and Test Manager program execute on the Server Machine. For each experiment, the Test Manager executes the following operations:

1) It replaces the Apache executable file with a faulty one. Each experiment involves exactly one software fault.
2) It starts the Web Server, then it starts the workload generator on the Client Machine. It stops the Web Server after the client has received a response for all requests, or the response time exceeds a timeout. The timeout is set to 5 seconds, which is much larger than the expected response time. The start and stop phases have also to be completed within a timeout of 5 seconds.
3) After the experiment, it stores log files produced by the Web Server. If one or more Web Server processes crashed or blocked during the experiment, it also stores a memory dump (i.e., a snapshot of processes raw memory at the time of abortion). It finally cleans up stale resources (e.g., zombie processes, unallocated semaphores) before the next experiment.

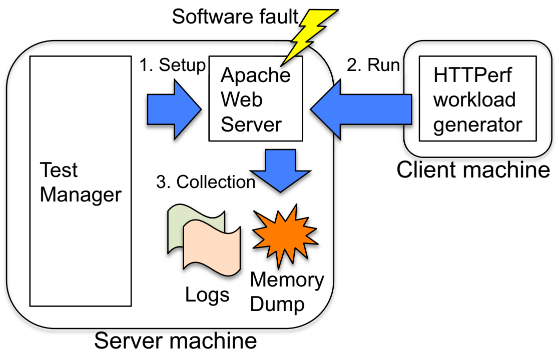


Fig. 3: Testbed setup

B. *Logging mechanisms evaluation*

Preliminary experiments aim to evaluate Apache built-in logging capabilities with respect to the injected software faults. Table II reports the total amount of experiments grouped by fault operators.

TABLE II: Number of fault injection experiments.

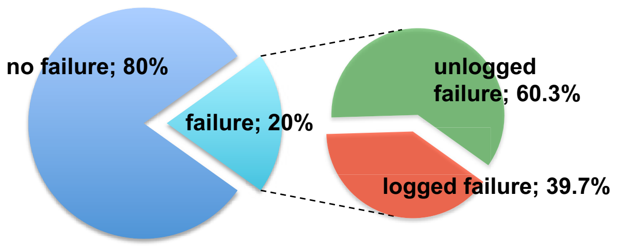
Fault	Locations	Fault	Locations
OMFC	774	OMIA	975
OMIEB	314	OMIFS	884
OMLAC	133	OMLOC	213
OMLPA	2,715	OMVAE	1,442
OMVAV	242	OMVIV	83
OWAEP	383	OWPFV	1,767
OWVAV	324	Total	10,249

[4]http://www.hpl.hp.com/research/linux/httperf/

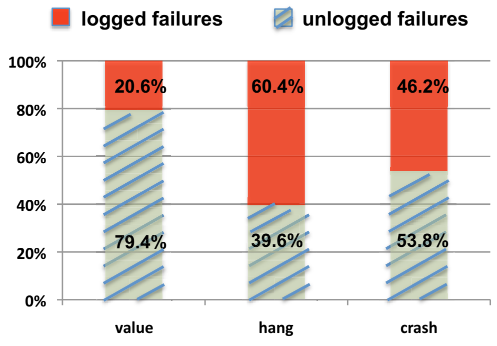
Experiments are classified with respect to the following outcomes:

- *Crash*: the injected fault causes the unexpected termination of one or more Web Server processes. Memory dumps are generated by the operating system;
- *Hang*: one or more of the HTTP requests, or the Web Server start/stop phases, are not executed within the timeout. We force memory dumps to be generated;
- *Value*: all the error conditions that are not the result of a crash or a hang (e.g., protocol error perceived by the client, Web Server termination with an error code);
- *No failure*: all the requests supplied by the workload generator are correctly executed.

Fig. 4 depicts the percentage of faults resulting in a failure outcome (20%). For failure outcomes, we further investigate Apache logfiles in order to figure out the presence of log entries. We experience that 39.7% of failures lead to an effective notification in logs. Therefore, for the most of failures due to software faults, built-in Apache logging mechanisms do not provide any information. Fig. 4b provide an in-depth view on unlogged and logged failures.



(a) Logged and unlogged experiments.



(b) Experiments breakup by failure classes.

Fig. 4: Experimental fault injection results.

For value failures, a large number of cases is unlogged (79.4%). Most of them (55.7%) actually occur during the system start-up phase, when the Web Server aborts without crashing and no logs are provided. Logged value failures (20.6%) mainly concern errors with HTTP protocol handling (e.g., header corruption) or filesystem accesses (e.g., wrong resource path).

Unlogged hang failures (39.6%) are mainly due to algorithmic errors resulting in infinite loops. Logged hangs (60.4%), instead, usually involve OS resources (e.g., sockets, IPCs). As for example, logs are produced when the server tries to write to a badly initialized socket. In this case, the fault prevents the server from replying to the client.

Crash failures are mainly due to wrong pointer manipulations. In most of cases (53.8%), no log entries are produced. An in-depth understanding on the occurred failure thus relies on memory dumps. Logged crashes (46.2%) are due to the termination of one or more Web Server child processes, enabling the parent process to notice their failure. Nevertheless, no information is provided about failure location or about failure causes. Log messages suggest to inspect memory dumps from the operating system, which may be not always available on the field during the operational phase.

C. Logging mechanisms improvement

We aim at improving Apache log mechanisms by providing developers with logging guidelines to be followed during the coding phase. These guidelines are intended to make logs more suitable to detect software faults resulting in field failures.

To deal with the most common causes of hang and crash failures, we analyze memory dumps collected during the preliminary experiments. In order to perform a precise analysis, we separated dumps into two classes, hang and crash failures, respectively. For each dump, we identify the function that the process was executing when the failure occurred. This information is obtained by analyzing the stack.

Table III reports the 10 most frequent functions executed during hang failures, in descending order.

TABLE III: Most hang-prone functions (by # of occurrences).

Function	# occ.	Function	# occ.
apr_socket_accept	364	add_any_filter_handle	9
apr_socket_sendfile	59	ap_allow_standard_methods	9
ap_escape_logitem	56	ap_byterange_filter	9
apr_socket_sendv	25	ap_invoke_filter_init	9
ap_directory_walk	17	ap_set_listner	9

We provide practical examples showing how to improve logs with respect to hang failures, in order to draw more general logging guidelines. First of all, it should be pointed out that the `apr_socket_accept` function is excluded from the analysis. We do so because, in case of hang failure, when we force memory dumps generation, most of the Web Server processes are still correctly waiting for incoming connections, and only one process is actually in hang because of the injected fault. This explains the large number of occurrences of this function.

Example 1: `ap_escape_logitem`. Several injected faults result in a never ending `for` cycle within this function. Fig. 5 shows the involved lines of code. A viable solution to track this type of errors is to check if the current number of iterations exceeds a maximum value. Logging an error message eases the failure analysis process (lines of code 1-2, 19-22).

```
 1 int current_iterations = 0;
 2 char * p = s;
 3 for (; *s; ++s){
 4    if (TEST_CHAR(*s, T_ESCAPE_LOGITEM)) {
 5       *d++ = '\\';
 6       switch(*s) {
 7       case '\b':
 8          *d++ = 'b';
 9          break;
10       // ... omissis ...
11       default:
12          c2x(*s, 'x', d);
13          d += 3;
14       }
15    }
16    else{
17       *d++ = *s;
18    }
19    if(current_iterations++ == MAX_ITERATIONS) {
20       log('ap_escape_logitem: MAX_ITERATIONS exceeded');
21       log('Potential cause: malformed input (%s)', p);
22    }
23 }
```

Fig. 5: ap_escape_logitem (util.c, lines 1795-1826)

Example 2: `ap_invoke_filter_init`. There are faults involving wrong pointer assignment, that result in a never ending `while` cycle within the function. Fig. 6 shows the involved lines of code. Like the previous example, it is possible to track this type of errors, by checking the total number of current iterations. A log message can be generated as shown by the boldface line of codes (1, 11-14).

```
 1 int current_iterations = 0;
 2 while (filters){
 3    if (filters->frec->filter_init_func) {
 4       int result =
 5          filters->frec->filter_init_func(filters);
 6       if (result != OK) {
 7          return result;
 8       }
 9    }
10    filters = filters->next;
11    if(current_iterations++ == MAX_ITERATIONS) {
12       log('ap_invoke_filter_: MAX_ITERATIONS exceeded');
13       log('Potential cause: linked list corruption');
14    }
15 }
```

Fig. 6: ap_invoke_filter_init (config.c, lines 311-319)

The most common cause of unlogged hang failures is due to algorithmic errors that result in infinite loops. The following guideline addresses this issue: *"a cycle including complex variable manipulations (e.g., strings, pointers) should provide a check on the number of iterations"*.

We perform a similar analysis on memory dumps produced by crash failures. Table III reports the 10 most frequent functions executed during a crash, in descending order.

Example 3: `apr_palloc`. Fig. 7 reports a snippet from the most frequent function in case of crash failures. Since this function is called in a large number of source locations (i.e., 64

TABLE IV: Most crash-prone functions (by # of occurrences).

Function	# occ.	Function	# occ.
apr_palloc	435	apr_pollset_add	221
ap_escape_logitem	386	add_any_filter_handle	216
apr_socket_addr_get	353	ap_read_request	213
ap_directory_walk	304	core_create_req	198
ap_core_output_filter	258	ap_core_input_filter	175

times within the server source code), there is a high potential of error propagation towards it. A viable solution to log this type of failure is to figure out the presence of a NULL pointer before its usage. This can be done as shown in Fig. 7 by the boldface lines of code (2-5).

```
1 size = APR_ALIGN_DEFAULT(size);
2 if(pool==NULL) {
3    log('apr_palloc: using NULL pointer');
4    log_stack();
5 }
6 active = pool->active;
```

Fig. 7: apr_palloc (apr_pools.c, lines 637-638)

Inserting a check before every pointer usage requires a high development effort, and it results in a too high overhead at runtime. Our approach identifies source code locations most likely to fail due to NULL pointers. This information enables developers to narrow the number of checks to be inserted. The following guideline has been defined: *"most likely NULL pointers should be checked before the usage"*. Logging the contents of the uppermost stack locations (i.e., by means of the `log_stack()` function) is also useful to identify the function that ultimately caused the failure.

VI. Conclusions and Future Work

In this paper we investigated the problem of evaluating the quality of logs within the Apache Web Server. An extensive fault injection campaign demonstrates that, in the most of cases, logs are not able to provide information about failures due to software faults. Therefore, we proposed an approach for improving logs, based on the analysis of most frequent failure locations identified during the campaign. The approach enabled us to draw a few general guidelines that can be followed by developers during the development phase.

Future work will be devoted to the definition of a wider set of guidelines, not only by an in-depth analysis of the Apache case study but also of other complex software systems. Our ultimate objective is to demonstrate that the proposed approach can actually improve the effectiveness of logs of current systems.

Acknowledgments

This work has been partially supported by the Consorzio Interuniversitario Nazionale per l'Informatica (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the framework of the private-public laboratory COSMIC ("Centro di ricerca sui sistemi Open Source per le applicazioni ed i Servizi MIssion Critical", DM23318, www.cosmiclab.it). It has also been supported by the "Iniziativa Software" Project, an Italian Research project which involves Finmeccanica company and several Italian universities (www.iniziativasoftware.it).

References

[1] J. Gray. Why do computers stop and what can be done about it. In *Proc. of Symp. on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[2] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, pages 772–, Florence, Italy, 2004. IEEE Computer Society.

[3] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, June 2008.

[4] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symp. on Internet Technologies and Systems*, 2003.

[5] F. Salfner and M. Malek. Using hidden semi-markov models for effective online failure prediction. In *Proc. of the 26th IEEE Symp. on Reliable Distributed Systems*, 2007.

[6] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore. Investigation of failure causes in workload-driven reliability testing. In *Foundations of Software Engineering*, pages 78–85. ACM Press New York, 2007.

[7] D. Cotroneo, S. Orlando, and S. Russo. Failure classification and analysis of the java virtual machine. In *Proc. of 26th Intl. Conf. on Distributed Computing Systems*, 2006.

[8] M. F. Buckley and D. P. Siewiorek. VAX/VMS event monitoring and analysis. In *Proc. of 26th Annual Intl. Symp. on Fault-Tolerant Computing*, 1995.

[9] C. Simache and M. Kaâniche. Availability assessment of sunOS/solaris unix systems based on syslogd and wtmpx log files: A case study. In *Pacific Rim Intl. Symp. on Dependable Computing*, pages 49–56. IEEE Computer Society, 2005.

[10] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proc. of the 18th Symp. on Reliable Distributed Systems*, pages 178–187. IEEE Computer Society, 1999.

[11] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, pages 575–584. IEEE Computer Society, 2007.

[12] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. In *Proc. of the IEEE Parallel and Distributed Processing Symp.*, 2004.

[13] J. Durães and H. Madeira. Generic faultloads based on software faults for dependability benchmarking. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2004.

[14] R. L. O. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental risk assessment and comparison using software fault injection. In *Proc. of Intl. Conf. on Dependable Systems and Networks*, 2007.

[15] J. Christmansson and P. Santhanam. Error injection aimed at fault removal in fault tolerance mechanisms-criteria for error selection using field data on software faults. In *Proc. of 7th Intl. Symp. on Software Reliability Engineering*, pages 175–184.

[16] E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, 14(4):73–83, 1997.

[17] L.M. Silva. Comparing error detection techniques for web applications: An experimental study. *7th IEEE Intl. Symp. on Network Computing and Applications*, pages 144–151, 2008.

[18] J.A. Duraes and H.S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.