

Bad Words: Finding Faults In Spirit’s Syslogs

Jon Stearley
Sandia National Laboratories
Albuquerque, NM 87111 USA
Email: jrstea@sandia.gov

Adam J. Oliner
Stanford University
Department of Computer Science
Stanford, CA 94305-9025 USA
Email: oliner@cs.stanford.edu

Abstract—Accurate fault detection is a key element of resilient computing. Syslogs provide key information regarding faults, and are found on nearly all computing systems. Discovering new fault types requires expert human effort, however, as no previous algorithm has been shown to localize faults in time and space with an operationally acceptable false positive rate. We present experiments on three weeks of syslogs from Sandia’s 512-node “Spirit” Linux cluster, showing one algorithm that localizes 50% of faults with 75% precision, corresponding to an excellent false positive rate of 0.05%. The salient characteristics of this algorithm are (1) calculation of nodewise information entropy, and (2) encoding of word position. The key observation is that similar computers correctly executing similar work should produce similar logs.

I. INTRODUCTION

Nearly all systems encounter faults, accidental conditions that cause a component to fail to perform its intended function [20](7C). This is certainly true of high-performance computing clusters designed for performance rather than reliability. As the component count and complexity of these systems increase, faults become more frequent and difficult to localize. Whereas it is not difficult to detect and localize faults with known signatures, the root causes of new fault types can be extremely challenging to find, resulting in repeated application interrupts and unscheduled downtimes.

Syslogs are a rich source of fault information, evidenced by the fact that they are one of the first places systems administrators look for clues when problems occur. Syslog is the most widely used mechanism by which system and application programmers may emit arbitrary text messages to a centralized message repository. Although the possibility of unstructured message content makes syslog flexible for programmers, the resulting variety of messages are a bane to systems administrators and automated detection algorithms, alike. For example, some programmers use the word “error” only in reference to faults justifying human response, while others use it in a frustratingly liberal fashion. In fact, programmers may not know when human responses are justified, because it is often the interaction of multiple interdependent programs that determines this condition. Additionally, the time interleaving of messages emitted from multiuser, multinode,

multitasking supercomputers produces logs with wildly varying time properties. Indeed, syslogs are regarded by many system administrators as a necessary evil.

In this paper, we present an algorithm that can accurately detect faults in syslogs by leveraging two important insights. First, the context of a word in a message is important. We encode words as *terms*, the concatenation of the word text and its token position in the message. Second, similar computers correctly executing similar workloads tend to generate similar logs. By aggregating logs by node and computing certain information-theoretic quantities, explained later, we achieve unprecedented results: 50% accuracy at 75% recall. This corresponds to a false positive rate of 0.05%. Using experiments on syslog data from Sandia’s “Spirit” supercomputer, we explain why using nodewise term entropy is so effective.

II. BACKGROUND

Because syslogs do contain critical fault and misuse information, and are used so widely in computing systems, they have a rich research history. The most common approach is to monitor syslogs for known fault types, usually via regular expressions [1], [8], [15], [21]. The maintenance of precise expressions requires diligent expert attention, especially as logs change with time, due to software and hardware upgrades and configuration changes. Variations in user behavior can also directly affect log content. But the maintenance and monitoring tasks are simple compared to the process used to discover previously unknown fault or misuse messages, which is most commonly based on an expert’s intuitions and experience, coupled with decades-old tools such as UNIX `grep` and `ls`.

In order to facilitate this discovery task, researchers have considered a wide variety of visualization methods. These include plotting of message rates [4] (also, see `splunk.com`) and three-dimensional renderings of host groups [18], [19]. Other approaches include parallel coordinates, spring layouts, and self-organizing maps [7]. Efforts have even been made to auralize log streams [6] using sounds such as frogs croaking and birds singing. Many of these approaches are interesting and may be valuable, but it is difficult to quantify their effectiveness.

Extensive research has also been conducted on the time properties of logs. It is well known that some faults generate large numbers of messages in short periods of time, and much work capitalizes on this property [3], [10]. Hardware failures

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Work was funded in part by the U.S. Department of Energy High Performance Computer Science Fellowship.

(such as disk, memory, and network) can be particularly prolific. The majority of fault types, however, are not bursty [14]. Another supposed property on which researchers have capitalized is the normal periodicity of messages [9], [13], yet many messages are not periodic, but rather are driven by the varying workload. Most of these works assume that message types are well known: that message content is easily tokenized into a message-type-identifying ID. While this is true of some logs (such as from network and phone switches), this is certainly not the case for syslog.

Less work has been done in the area of unstructured message content. Attempts to apply techniques originally developed for genomic sequence mining to logs [17], [24] have run up against scaling problems. Vaarandi applied clustering [22] and a priori data mining [23] to the problem of automatically categorizing messages. Vaarandi was the first to encode word positions in his analyses (e.g. the first word of the message, the second, etc.), thereby effectively capturing a simple form of message context. This paper extends the understanding of how useful this encoding can be.

Reuning [16] and Liao [11] have each applied simple term weighting schemes to the goal of intrusion detection in logs, but Reuning concludes that his false positive rate renders the approach unusable in practice. This paper applies a more complex weighting scheme (“log.entropy,” explained later), which has been shown to be highly effective for information retrieval tasks [2].

III. DATA

Over a 23-day period of production operation, Sandia’s 512-node Linux cluster, called “Spirit,” generated 8.3 million (911MB) syslog messages [12]. Administrators held weekly meetings in which they identified and discussed all known faults, including hardware failures (disks, fans, memory, and network cards), misconfigurations (software, BIOS, and hardware inconsistencies), parallel job failures, and other miscellaneous events. More information on this log, as well as four others, can be found elsewhere [14].

Let a *nodehour* be one hour of log data generated by a single node. Using the faults identified above, we generated a list of single-word regular expressions that identify those messages that were indeed symptoms of faults, and applied them to identify 365 distinct nodehours as containing faults. We use this list of fault-containing nodehours as the ground truth for experiment scoring (described later). The detection algorithms are unsupervised, however, and do not know a priori which nodehours contain faults.

Software present on Spirit nodes detects a number of known fault conditions and sends mail from the node to alert the administrators. This, in turn, generates syslog messages. Presenting nodehours containing such mail, however, is not operationally useful (the alarm was already raised). In order to deal with this, we ignore all email-related syslog messages in this study. This is accomplished by simply discarding all messages that start with “postfix:”, which is Spirit’s mail transfer agent.

The lack of publicly-available, labeled log data is a significant barrier to major advances in the log analysis research community. A primary reason logs are not shared is the technical difficulty of exposing enough information to enable effective research, while not exposing enough to put the sharing site at risk [5]. We scrubbed the logs in this study clean of sensitive information and they are available upon request. Our analysis tools are also publicly available at <http://www.cs.sandia.gov/~jrstear/sisyphus>.

IV. PREPROCESSING

We first index the logs to create a sparse $M \times N$ matrix, where non-zero values in the matrix indicate how many times word i (rows) occurs during nodehour j (columns). A *word* is a whitespace-separated character sequence in the unstructured text messages. We use message timestamp and host name (*node*) solely to determine nodehour; they are not included in the index. Words occurring only once in the entire log set are excluded; most words occur infrequently [22]. This results in a matrix with 36,115 rows (words), 243,409 columns (nodehours), and only 0.045% nonzero entries.

Word position is a simple form of message context information. In order to study the impact of encoding word position, we generated a term-nodehour matrix in addition to the word-nodehour matrix. A *term* is the concatenation of a word with its position in the message; position is encoded as a four digit hexadecimal prefix. For example, the term “0007CONDITION” indicates that the word “CONDITION” occurs as the seventh word of a message. Terms that occur only once in the entire log set are excluded from the index. This matrix has 39,493 rows, 243,409 columns, and 0.075% non-zero entries.

It is conventional for the first word in syslog messages to be of the form “pname[pid]:” where “pname” is the name of the process generating the msg and “pid” is its process ID. Given this convention, we study the detection performance impact of including it in the index versus ignoring it. In all cases, we eliminate the pid during indexing, rendering first words as “pname[*]:” (and thus the first terms are “0001pname[*]:”).

V. OBJECTIVE

Our goal is to automatically divide the nodehours into two classes: those that contain faults, and those that do not. The confusion matrix shown in Table I forms the basis for scoring how well a classifier performs. Let an *alarm* be a nodehour document that is presented by the classifier as containing a fault. Let *actionable* denote a nodehour that does, in fact, contain a fault.

Precision is the percentage of alarms (positive presentation class) that are actionable (positive true class), and is calculated as $P = TP/(TP + FP)$. There exists some threshold for P below which a classifier is practically useless because too many false alarms are raised (this threshold is site and application dependent). Recall is the percentage of actionable nodehours presented as alarms, and is calculated as $R = TP/(TP + FN)$. Plots relating these two quantities

		True Class	
		Positive (Actionable)	Negative (Ignorable)
Presentation Class	Positive (Alarm)	True Positive (TP)	False Positive (FP)
	Negative (Ignore)	False Negative (FN)	True Negative (TN)

TABLE I
CLASSIFIER SCORING MATRIX

are called precision-recall plots. An ideal classifier exhibiting perfect precision and detection would be represented by a single point at (1,1). In practice, however, there are a range of precision and detection values as the classifier presentation threshold is varied from maximum to minimum (left and right ends of the lines on PR plots, respectively).

System administrators commonly begin this classification task by examining log file sizes—an unusually large size implies unusually high message rates (a known symptom of some faults). We therefore use the number of bytes in each nodehour log as the baseline classifier—any classifier that does not significantly exceed the performance of this simple *bytes* algorithm is not worth pursuing.

VI. EXPERIMENTS

Given an $M \times N$ matrix \mathbf{X} , the magnitude of the j^{th} column is $|x_j| = \sqrt{\sum_{i=1}^m x(i,j)^2}$. This calculation forms the basis of our classifiers.

We calculated magnitudes of the word and term index matrices, but these do not significantly outperform *bytes* as a classifier, and are thus omitted for brevity.

We also explored the *tf.idf* weighting scheme [2]. In this scheme, the elements of \mathbf{X} are first scaled by $g(i) = \log_2(1 + \frac{n}{df_i})$, where df_i is the number of nodehours in which term i occurs. So $|x_j| = \sqrt{\sum_{i=1}^m (g(i)x(i,j))^2}$. The intent of this scheme is to give a greater weight to those terms that are useful in distinguishing between nodehours. This also does not yield a classifier that significantly outperforms *bytes*.

Next, we explored *log.entropy* weighting [2], which involves the following two steps. First, we diminish the dominance of terms having high occurrence counts by taking their logarithm. Second, we scale by their information entropy:

$$g(i) = 1 + \frac{1}{\log_2(n)} \sum_{j=1}^n p_{ij} \log_2(p_{ij}),$$

where $p_{ij} = \frac{x(i,j)}{\sum_{j=1}^n x(i,j)}$ is the fraction of term i 's total occurrences which are in document j . Terms that occur the same number of times in all documents receive a weight of zero, and terms which occur in only one document get a weight of one. Terms can occur in all documents and still receive a weight of nearly one if a large majority of their occurrences are focused among a small number of documents. Using this weighting, $|x_j| = \sqrt{\sum_{i=1}^m (g(i)\log_2(x(i,j)))^2}$. This also does not yield outstanding results.

Finally, we calculated each of the above again, but using weights based on aggregates of nodehours rather than raw nodehours. Let \mathbf{Y} be a node-aggregated $M \times N$ matrix, where N is the total number of nodes, and column j is equal to the element-wise sum of columns of \mathbf{X} corresponding to node j . Compute $g(i)$ as above, using $y(i,j)$ instead of $x(i,j)$. Similarly, generate a time-aggregated $M \times H$ matrix \mathbf{Z} , where H is the total number of hours. Using these aggregated weighting factors, *tf.idf* still does not produce noteworthy results, but *log.entropy* does.

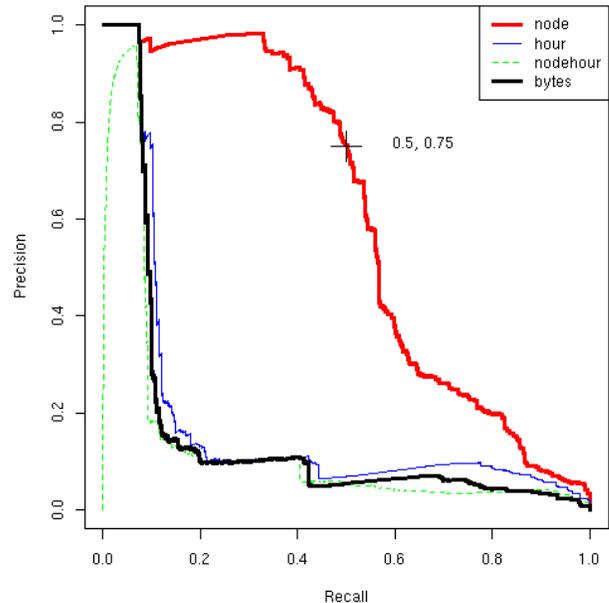


Fig. 1. Using bytes per nodehour as a fault detection criteria yields high precision only for bursty fault types (roughly 10% in this experiment). Information magnitude using distribution of terms over hours or nodehours yields similarly poor performance. In contrast, nodewise information magnitude detects an impressive 50% of faults with 75% precision, corresponding to an excellent false positive rate of 0.05%.

VII. RESULTS

Column magnitudes resulting from *log.entropy* using $y(i)$ are referred to as the *node* classifier (the columns of \mathbf{Y} correspond to individual nodes). Magnitudes resulting from *log.entropy* using $z(i)$ are referred to as the *hour* classifier (the columns of \mathbf{Z} correspond to individual hours). Magnitudes using *log.entropy* on $x(i)$ are referred to as *nodehour* (\mathbf{X} is our original nodehour index matrix). Figure 1 shows the precision-recall performance of these classifiers. The *bytes* classifier maintains a high alarm precision rate for only a very small subset of fault conditions, corresponding to a specific disk failure mode resulting in a large burst of messages. The *hour* and *nodehour* classifiers perform similarly poorly to *bytes*. In comparison, the *node* classifier yields outstanding performance. The point at (0.5, 0.75) indicates that we detect 50% of fault-containing nodehours before alarm precision falls below 75%. For this data, this corresponds to an excellent false positive rate of 0.05%.

The *node* classifier is now in production use on Sandia’s Red Storm and Spirit supercomputers, and has detected a wide range of faults in near real-time (statistics are computed via `cron` every 15 minutes, which has proven to be an acceptable lag time).

Why does aggregating by node have such a positive effect? Some non-fault events also result in message bursts, such as when a computer boots. Every time one of the nodes in this data set boots, it emits 506 messages in one minute. This causes a significant number of false positives for rate-based methods, because rebooting is often a symptom of human intervention, rather than a consequence of a fault. We need a method that detects faults soon after (or before) the fault occurs, rather than after remedial action has been taken. Roughly speaking, all nodes reboot an equal number of times, so the terms involved are evenly distributed across the nodes. The *node* classifier described above results in these terms being weighted low (near zero), greatly diminishing their effect on the magnitude calculations. This is a practical example of how the *node* classifier is beneficial—when similar bursts of messages appear on all nodes over time, it weights those messages low. Bursts of messages that do not occur on all nodes still receive a high weight (near one), as they should.

In Figure 2, we plot two key properties of terms: nodewise information weight $g(i)$ on the vertical axis and total occurrence rate on the horizontal axis. The aforementioned terms associated with booting appear in the lower right region of the plot (low information but high occurrence rate). Terms with both a high information weight *and* a high occurrence rate contribute most significantly to nodehour information magnitudes. Terms that occur only once must also have an information weight of one (it can only occur on a single node). Similarly, for a term to have an information weight of zero it must occur on all nodes (in this case, 512). The negatively sloped boundary at the left of the diagram is a consequence of this bounding relationship between information weight and occurrence rate. Colors are used to group terms into similar information weight ranges, based on arbitrary thresholds. Terms arising from the first word of each message are depicted using “+” symbols (addressed in detail later).

We now turn to the advantage of using terms instead of words, which is quantified in Figure 3. The *Wnode* classifier is calculated in the same way as *node*, but using the word index as \mathbf{X} instead of the term index. Indexing terms results in a slightly larger matrix than words, but the precision benefits are well worth the cost. As a specific example, consider the following two messages:

- CROND[23597]: LAuS error - do_command.c:226 - laus_attach: (19) laus_attach: No such device
- Event Log Daemon:[2907]: Fatal drive error, SCSI port 1 ID 0

The first message is distributed evenly across all nodes, and does not indicate a fault condition (rather, it indicates a benign misconfiguration). The second, however, indicates disk failure, which only occurred on two nodes. The word “error” in the first message is indexed as the term “0003error”, whereas that

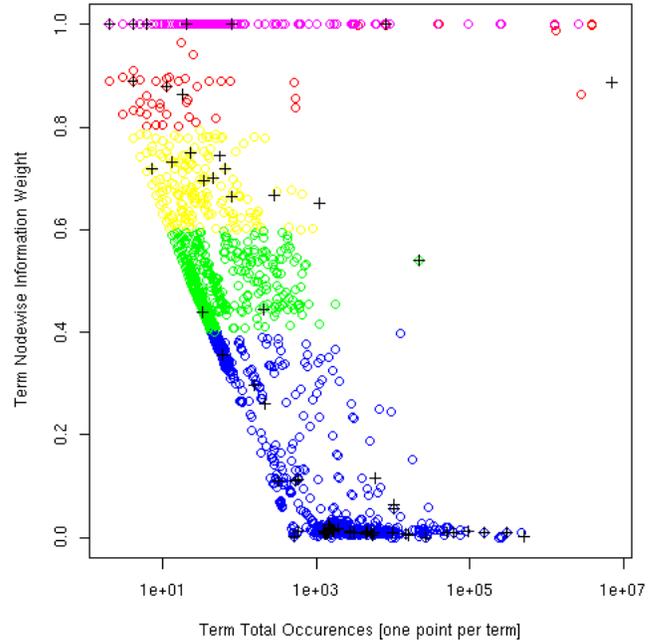


Fig. 2. Terms exhibit a wide range of information weights $g(i)$ and occurrence counts. Terms occurring many times on few hosts (upper right corner) contribute most significantly to nodehour information magnitude. In contrast, terms having only high occurrence rate *or* high information weight contribute little to information magnitude. The negatively sloped boundary condition is described in Section VII.

same word in the second message is indexed as the term “0006error”. The former occurs in a variety of fault and non-fault messages (described below), but the latter occurs only in messages indicating faults, and has an entropy weight nearly one, and thus appears at upper right of Figure 2. Word position information helps to distinguish the terms as being part of fundamentally different messages. By encoding word position in this manner, we retain a simple form of message context and thereby achieve greater precision.

Encoding more context, such as term tuples, may offer additional benefits. It turns out that the term “0003error” also occurs in the message “kernel: EXT3-fs error (device cciss0(104,3)): ext3_get_inode_loc: unable to read inode block - inode=97638, block=196619”, which is another symptom of a type of disk failure. Because “0003error” occurs in both fault and non-fault messages, it receives a nodewise information weight of 0.86. The term “0007unable,” on the other hand, only occurs in the disk failure messages and only on one of the 512 hosts, and so it receives a nodewise information weight of 1.

These examples hint at the complexity of fault detection in syslog data: the text messages are entirely unstructured. In production use, we configure some terms to be ignored from magnitude calculations (e.g. memory addresses in hexadecimal, which almost always receive high nodewise information weighting but provide little detection value). Experience suggests that a small number of such configurations is sufficient to yield accurate detection results, but some platform-specific

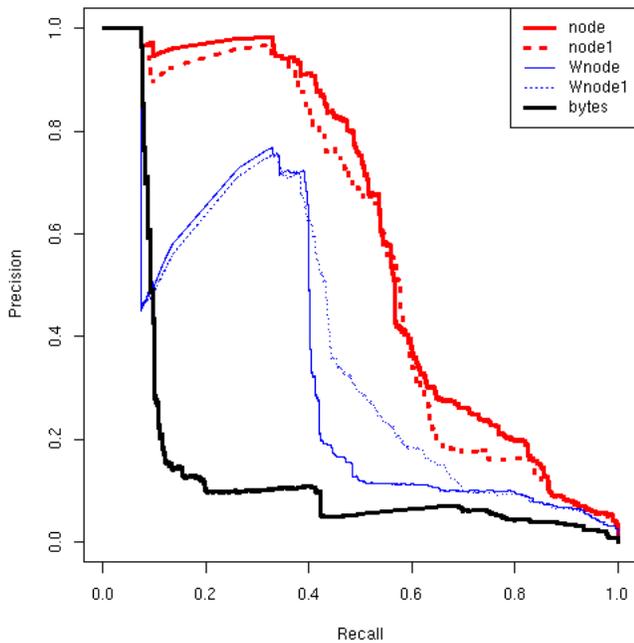


Fig. 3. Using terms in the nodewise information magnitude detector (*node*) yields significantly higher precision than using just words (*Wnode*). Furthermore, omitting the first word of each message (*node*) yields a slight advantage over including it (*node1*).

tuning in this manner is necessary to achieve an operationally acceptable precision rate.

Regarding the first message word, consider Figure 3, which shows that omitting the first message word yields a small precision advantage. The *Wnode1* classifier is the same as *Wnode*, except that the first message word is included in the calculations (similarly for *node1* versus *node*). An example motivating this result is the fact that 80% of boot messages are emitted by the Linux kernel, and thus begin with “kernel:”. The disk failure message described above, however, also begins with “kernel:”. Due to the very large bursts of disk failure messages (two nodes emit 3.8 million of them over 28 total nodehours), the term “0001kernel” receives a high nodewise information weight. Including the term in magnitude calculations erroneously results in boot nodehours receiving a higher magnitude value than if the term were ignored from the calculations. There are other terms with similar properties. Any program that emits a large number of normal messages, but occasionally emits fault messages, will have this effect. Additionally, Figure 2 shows that the majority of first-word terms have low information weight (depicted using “+”).

We next divide nodehours into three groups, (1) those containing hardware faults, (2) those containing software faults, and (3) those containing no faults. We plot the distributions of these groups with respect to their nodewise information magnitude (the *node* classifier) in Figure 4. Ideally, all fault nodehours would be distributed at the right and non-fault nodehours would be at the left. An overlap of fault and non-fault distributions corresponds to decreasing precision in

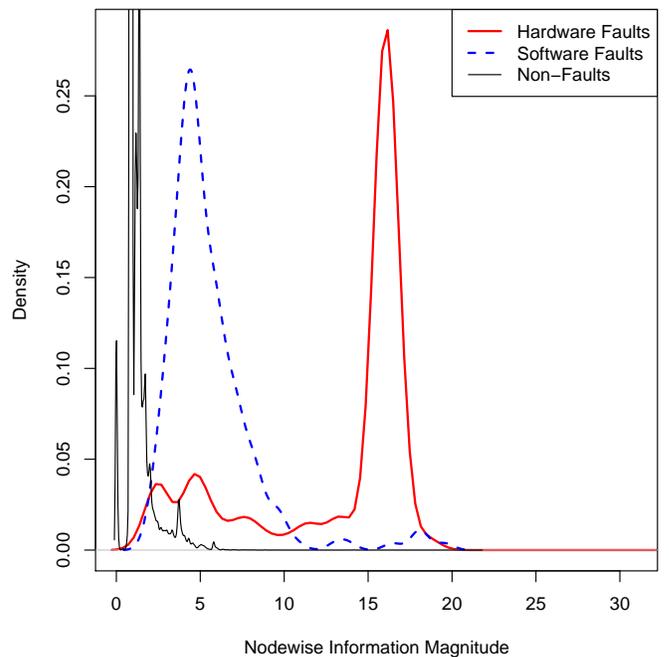


Fig. 4. The nodewise information magnitude classifier effectively separates nodehours not containing faults from nodehours containing either hardware or software faults.

Figures 1 and 3. Both hardware and software fault nodehours are well-separated from non-fault nodehours. This is significant because most hardware faults are extremely bursty, as previously discussed. In fact, the nodehours containing the disk failure messages are outside the range shown in Figure 4, tightly concentrated at a value of 163. Thus, the *node* classifier is effective at detecting more than just bursty faults.

Figure 5 plots nodehour information magnitude versus time. The high density of non-fault nodehours towards the bottom of the plot corresponds to the sharp peak of non-fault nodehours in Figure 4. Blank vertical bands correspond to periods of time during which no logs were emitted due to system downtime. Plotting versus time provides an intuitive and intriguing overview of the information trends present in this data set.

VIII. CONCLUSIONS

Similar computers correctly executing similar workloads tend to produce similar logs. For example, we found that compute nodes in a Linux cluster, running jobs for users, generated logs with similar content to one another during non-faulty operation. We have quantitatively shown that a non-uniform distribution of terms across supercomputer nodes (high nodewise information) is useful for fault detection in syslogs, as is the encoding of word position (terms). These facts, coupled with simple file indexing and matrix computations, yield a classifier that, on our data set, detects 50% of faults while maintaining a precision of at least 75%. This approach could be used to help inform resilience measures that a fault has occurred or is imminent. While some log messages provide only postmortem information, others indicate conditions that

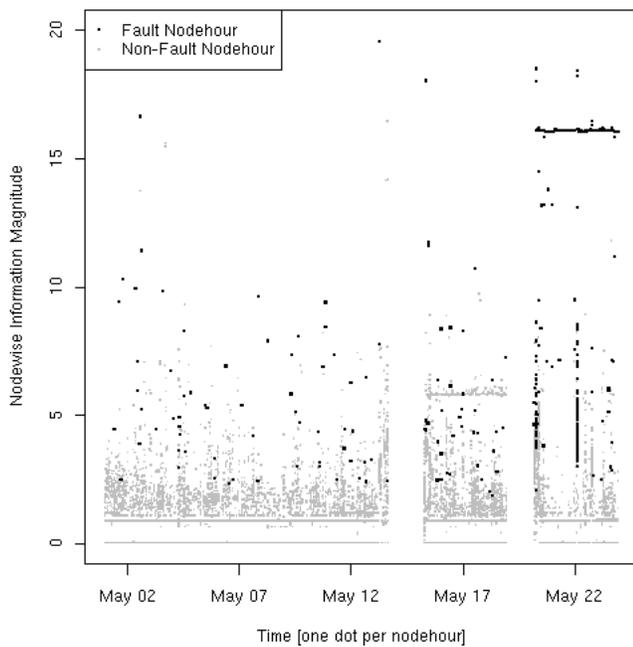


Fig. 5. Most nodes emit low-information logs most of the time. Fault-containing nodehours tend to rise to the top.

are progressing towards fault. In both cases, quick detection and response minimizes the impact on users.

ACKNOWLEDGMENTS

First and foremost, Jon would like to thank his savior Jesus Christ, the revealer of all mysteries. Thanks also to Jerry Smith, Sophia Corwell, and Tim Draelos for contributing their time and insight.

ERRATA

Example messages for terms "0003error" and "0006error" on page 4 did not appear correctly in a previous revision of this paper, but have been corrected herein.

REFERENCES

- [1] Logsurfer - a tool for real-time monitoring of text-based logfiles. <http://www.cert.dfn.de/eng/logsurf/>.
- [2] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Rev.*, 41(2):335–362, 1999.
- [3] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *In The 2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, 2005.
- [4] A. L. Couch. Visualizing huge tracefiles with Xscal. In *USENIX LISA'96 Conference Proceedings*, 1996.
- [5] U. Flegel. Pseudonymizing UNIX log files. In *InfraSec '02: Proceedings of the International Conference on Infrastructure Security*, pages 162–179, London, UK, 2002. Springer-Verlag.

- [6] M. Gilfix and A. L. Couch. Peep (the network auralizer): Monitoring your network with sound. In *USENIX LISA'00 Conference Proceedings*, 2000.
- [7] L. Girardin and D. Brodbeck. A visual approach for monitoring logs. In *USENIX LISA'98 Conference Proceedings*, 1998.
- [8] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *USENIX LISA'93 Conference Proceedings*, 1993.
- [9] J. L. Hellerstein, S. Ma, and C. Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 41(3), 2002.
- [10] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In IEEE, editor, *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005.
- [11] Y. Liao and V. R. Vemuri. Using text categorization techniques for intrusion detection. In *11th USENIX Security Symposium, August 5–9, 2002.*, pages 51–59, 2002.
- [12] C. Lonvick. The BSD syslog protocol. Request for Comments 3164, The Internet Society, Network Working Group, August 2001. RFC3164.
- [13] S. Ma and J. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of the 2001 International Conference on Data Engineering (ICDE'01)*, pages 409–416, 2001.
- [14] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 575–584, Washington, DC, USA, 2007. IEEE Computer Society.
- [15] J. Prewett. Listening to your cluster with LoGS. In *Proceedings of the 2004 Linux HPC Revolution Conference*, 2004.
- [16] J. R. Reuning. Applying term weight techniques to event log analysis for intrusion detection. Master's thesis, University of North Carolina at Chapel Hill, July 2004.
- [17] J. Stearley. Towards informatic analysis of syslogs. In *Proceedings of the 2004 IEEE Conference on Cluster Computing*, 2004.
- [18] T. Takada and H. Koide. Information visualization system for monitoring and auditing computer logs. In *IEEE Conference on Information Visualization*, 2002.
- [19] T. Takada and H. Koide. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *USENIX LISA'02 Conference Proceedings*, 2002.
- [20] S. C. C. . (Terms and D. J. R. (Chair). *The IEEE Standard Dictionary of Electrical and Electronics Terms*, volume IEEE Std 100-1996. IEEE Publishing, 1996.
- [21] R. Vaarandi. SEC - a lightweight event correlation tool. In *IEEE IPOM'02 Proceedings*, 2002.
- [22] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IEEE IPOM'03 Proceedings*, 2003.
- [23] R. Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems*, volume 3283, pages 293–308, 2004.
- [24] A. Wespi, M. Dacier, and H. Debar. An intrusion-detection system based on the teiresias pattern-discovery algorithm. In *EICAR Annual Conference Proceedings*, pages 1–15, 1999.