Interactive Visualization for System Log Analysis

Armin Samii and Woojong Koh*
University of California, Berkeley

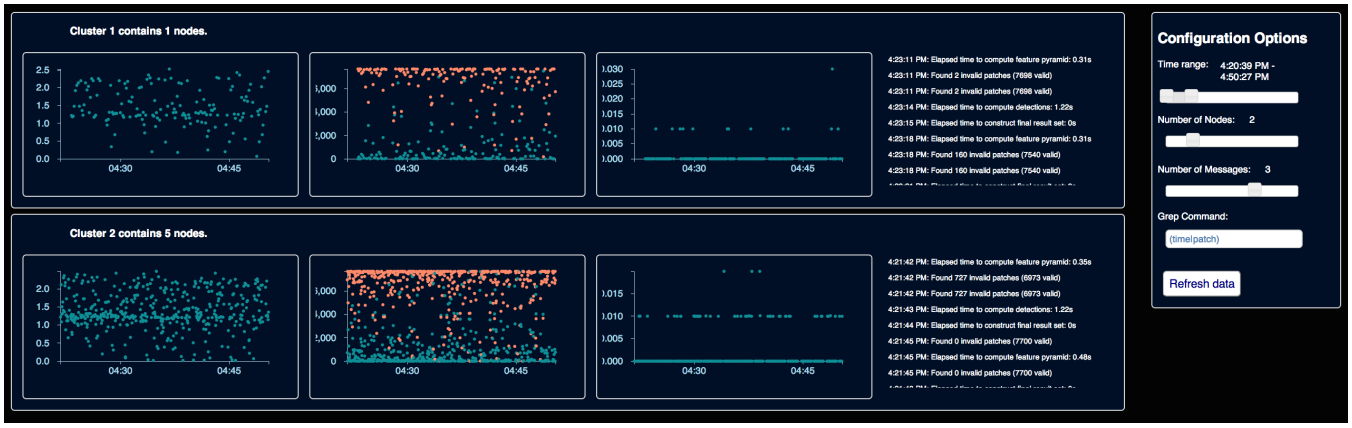

Figure 1: *Our visualization displays arbitrary system log data by parsing the variable parts of each log message. Each row contains messages from a single node. Each column is a certain type of message. Each point is a single log message. The x axis is time; the y axis is the value of the extracted variable. The colors indicate multiple variables in the same message. For example, column two has two colors: blue is the number of valid patches, and red is the number of invalid patches. Notice here that five nodes performed similarly (row 2), whereas one node (row 1) is performing more slowly, and thus the number of log messages is sparse.*

Abstract

We present a visualization interface to assist system administrators in searching system logs. Even with current regular expression matchers such as `grep`, the amount of log data is difficult for humans to understand. We augment `grep` with a visualization of the matched patterns. To further reduce the amount of information displayed, we cluster similarly-performing nodes and only show a single node representative of the entire cluster. The user can then interactively search the log, zoom in on a certain time window, and choose the number of clusters. Our simulations show that our clustering is effective and our system is fast: the clustering successfully isolates anomalously-performing nodes in a variety of situations; the visualization can interactively visualize hundreds of millions of log messages across a hundred nodes at interactive rates.

1 Introduction

The gears of a distributed system are complex and constantly moving. A system administrator uses log files to understand the underlying structure and detect anomalous behavior. These logs are large, distributed across many filesystems, and hard to understand even with today's tools.

Our goal is to aggregate the log data into a visualization that can be easily understood by the system administrator. The two primary use cases are:

1. searching through system logs to find software bugs or hardware failures, and
2. viewing the current system state on-the-go without access to a command line.

To this end, our system contributes the following features:

1. **trustable**: the user is able to view the raw data at the most "zoomed out" state;
2. **fast and distributed**: the proprocessing primarily occurs at each node, with a single central node aggregating preprocessed data; and
3. **interactive**: the user can adjust the paramaters of the visualization at interactive speeds.

With a large enough cluster, this will require focusing the user's attention on interesting parts of the system state. This will require detecting anomalies, which in turn requires a constantly-evolving model of what the normal state of the system is.

Our system is open source and available on github [1].

*e-mail:{samii,wjkoh}@berkeley.edu

[1] `http://www.github.com/wjkoh/cs262a`

2 Related Work

Given the wide range of distributed applications, system log analysis has been a prominent research area for decades. Unfortunately, given the available visualization tools in the past, none are adequate for off-the-shelf use. We divide the related work into two sections: system state analysis and system log analysis. The system state refers to aggregating data about the statistics of resource usage in a system, agnostic to the application. System logs are arbitrary print statements in a piece of code, distributed across one or more nodes.

2.1 System State Analysis

Ganglia[2] is a system monitoring system which can handle up to 2,000 nodes and aggregate statistics in real-time [Massie et al. 2004]. It requires installing software on each running node and can only handle predefined metrics; it does not allow user-defined metrics and does not utilize system logs. The data is not clustered either, so either all nodes are visualized, or all nodes are aggregated. Finally, it is far from interactive: simple options such as setting a time window require many clicks. See Figure 2 for a screenshot of the amount of data shown for a single node.

Munin, based on RRDtool, is similar to Ganglia: after installation on each node, it automatically presents a visualization of the system status [Oetiker 1999]. As is evident from Figure 3, the visualization is extremely dense and illegible as the number of nodes increases.

We use ideas from these system state visualizations, and apply them to system logs. We also recognize that viewing all logs makes an illegible visualization, viewing one node at a time is inefficient, and aggregating all nodes into a single server status hides any anomalous behavior.

2.2 System Log Analysis

Wei Xu and collaborators [Xu et al. 2009b] uses machine learning to detect anomalies by mining log data. When an anomaly is detected, a decision tree visualization is presented to the user which attempts to explain why it is an anomaly. This setup, while useful, is fundamentally untrustable: no algorithm can successfully detect all anomalies, and the user will eventually have to resort to command-line `grep`. We adopt their machine learning features for our clustering algorithm.

HP Operations Analytics[4] provides many similar features to our work, including interactive time-window zooming. It does not seem to cluster any data, nor does it focus on software bugs, but instead analyze single-node server issues. It seems to require modifying source code to work with their log analysis system, which may limit its applicabilty. Unfortunately, it is also closed-source proprietary software so we are not able to test out its effectiveness.

Seaview has the closest goals to ours [Hangal 2011], and provide similar visualizations. However, they do not work with arbitrary log types (notably, log messages with multiple variables), nor does it extend to multiple nodes or large amounts of data. They include no machine learning or reduction of the amount of data shown, other than a planned `grep` text search.

Makanju and collaborators present spatio-temporal clustering of log data as well [Makanju et al. 2012]. Their method does not seem to be, nor claim to be, interactive. Finally, Saganowski and collaborators present a statistical method for preprocessing features that will be used in anomaly detection [Saganowski et al. 2013]. We find that these two methods for clustering, feature extraction, and feature preprocessing to be complementary to ours, and it would fit cleanly in our interactive system.

3 Methods

We automatically extract the source code line which printed each log message, the time it was printed, the node which generated it, and the variable part of each log message.

Our algorithm can be summarized as follows:

1. Compute per-node features
2. Perform k-means clustering
3. Display interactive visualization

We desribe each of these in detail below.

3.1 Feature Extraction

Each node computes its own features, and passes on the feature vector as well as raw log data to the master server.

Following the example of Xu and collaborators [Xu et al. 2009b], we extract a message count vector in the time window requested by the user. The message count vector is the number of unique times each message was printed. This is a strong feature for failing or slowing nodes. The state ratio vector compares how many times one type of message was printed compared to similar ones; for example, the ratio of file opens to file closes. We found that the state ratio vector was not useful for our log data, and so we do not compute it to save CPU time.

Using the extracted variables, we compute basic statistics about each type of message. In a given time window, we compute the:

1. mean,

[2] `http://www.ganglia.info`

[3] `http://oss.oetiker.ch/rrdtool/gallery/index.en.html`

[4] `http://www8.hp.com/us/en/software-solutions/operations-analytics-operations-analysis/`

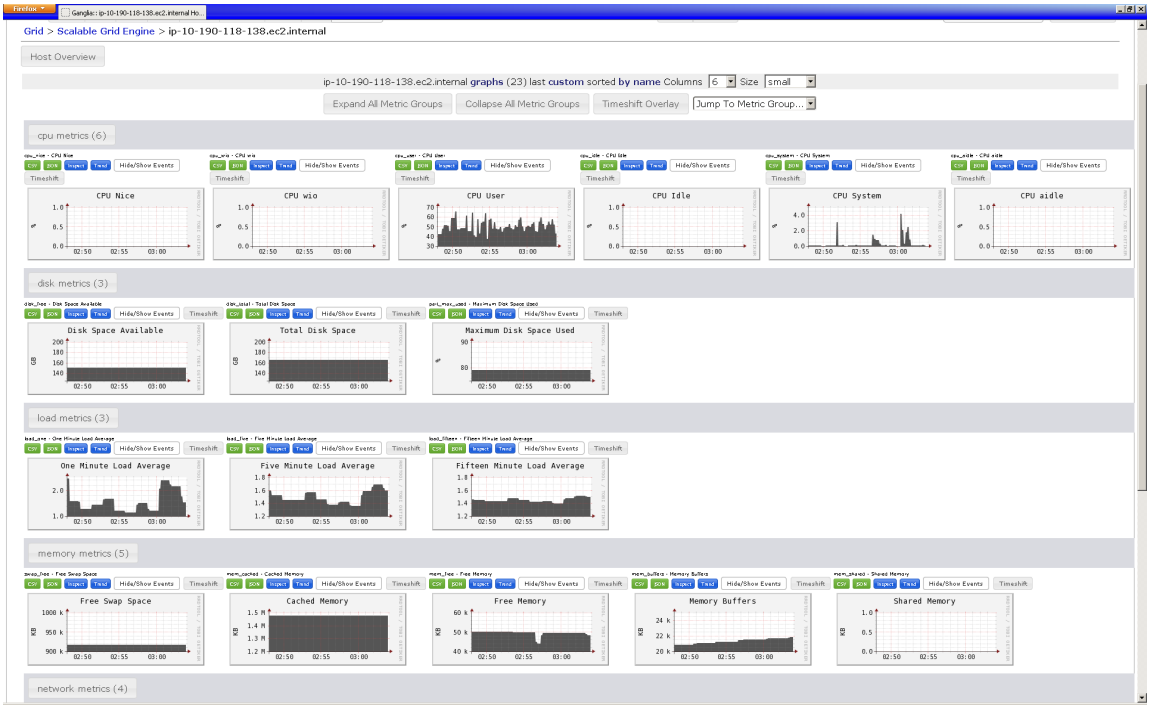

Figure 2: *Screenshot of Ganglia (image courtesy of Wikipedia). This information shows various metrics for a single node.*

Figure 3: *A screenshot of RRDtool attempting to coherently display a lot of information to the user, but ending up extremely cluttered.*[3]

2. standard deviation,
3. slope of a linear regression,
4. z-score, and
5. r-value.

These would indicate anomalies involving, for example, a diverging residual in a machine learning application, a section of code which takes too long to run, and a message which keeps working on the same file rather than working on new files.

3.2 Clustering

We perform k-means clustering in the feature vector space [MacQueen 1967; Lloyd 1982]. With a small number of nodes (on the order of tens), the clustering can be performed on a single machine quickly. On more nodes, a distributed k-means algorithm is required, which can either be exact and slower [Jin et al. 2006] or approximate [Kanungo et al. 2002]. Since the clustering will occur each time the user updates the parameters of the visualization, it is essential that it is fast.

Once the desired number of centroids are found, we compute the node which is closest to the centroid in a least-squares sense and display only that node. Therefore, the visualization master server need only receive $O(k)$ data to pass on to the user, linear with the number of centroids, instead of with the total number of nodes.

3.3 Visualization

The master server collects the data from each node and uses an HTTP Apache Server to pass the data on to the client. We use d3.js to visualize the data [Bostock et al. 2011]. We use a scatterplot where each point is a raw log message, the x axis is time, and the y axis is the parsed variables (see Figure 1). Mousing over a scatter point displays the text of the raw log message. Each row in the visualization is the nearest node to the centroid of the k-means clustering. Each column is a specific type of log message, identified by its line in the source code.

The user can interactively change the time window, the number of desired clusters, and the regular expression pattern. With each change, the clusters are recomputed and new data is shown. Clustering is only performed on the log messages which match the regular expression pattern in the given time window.

Because the visualization gets cluttered if there are too many scatter points in a single graph, each graph at most shows 5,000 messages, each one evenly spaced in time. Zooming in (narrowing the time range) allows finer sampling of the log messages.

4 Implementation

We identify each log message by the node and source code line that generated it. We assume that all logs are generated using the Google Logging Library (GLOG)[5] in order to know which source line generated which log message. This assumption can be dropped using the mining algorithm of Xu and collaborators [Xu et al. 2009a].

4.1 Log Parser

GLOG provides a consistent log format so we were able to build a parser easily using Python's regular expression library. After parsing we need to group every log message by its message type and extract nominal and quantitative data from the messages.

First, we group log messages by its source file name and line number. We then separate dynamic parts of messages from static parts using Ratcliff/Obershelp pattern recognition [Ratcliff and Metzener 1988]. If the dynamic part is a string instead of number (such as a filename), we simply discretize the value. This will succeed in detecting if an anomalous node is opening the same file repeatedly, and other similar issues, but will fail on text data that is semantically meaningful.

4.2 Feature Extraction and Clustering

The parameters requested by a user affect the input to the clustering, so we need to be able to compute feature extraction at interactive rates. Our current implementation only uses the message count vector [Xu et al. 2009b] for our feature vector to achieve such rates. Our message count vector computation uses prefix sums, where each message at time $t = T$ knows the message count in the range $t \in [0, T]$. To find the message count vector in a certain timerange $t \in [t0, t1]$, we find the closest message after $t = t0$ and the closest message before $t = t1$, and subtract the prefix sum at each of those times. This results in $t \in [0, t1] - [0, t0] = t \in [t0, t1]$, the exact range we want. Similar optimizations can be performed for the simple statistical features.

4.3 Visualization

We used d3.js [Bostock et al. 2011] and jQuery UI libraries for our user interfaces. The front end of our framework is a web-based dashboard, enabling easy viewing from a mobile device while on-the-go. However, in our experience, the interaction is much simpler on a large display.

In Figure 4 you can see the user controls of our visualization. The controls allow control of the time window, the number of clusters, and the regular expression command. There is also control over the number of message types displayed, which

[5] `https://code.google.com/p/google-glog/`

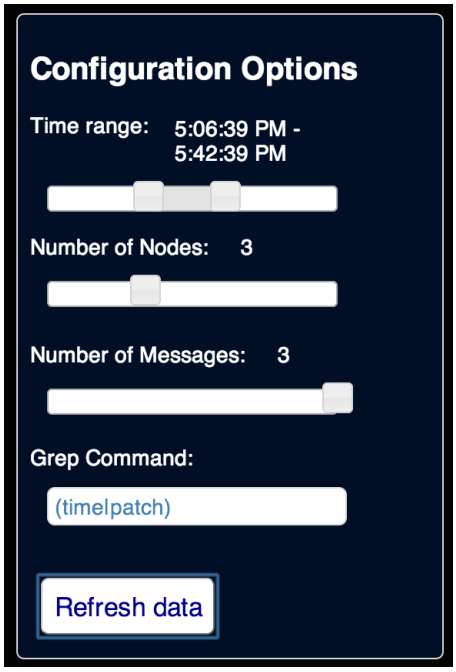

Figure 4: *The configuration options available to the user via our web framework. The user can change these options at interactive rates.*

should only display the most representative message types; we have not yet implemented this algorithm.

5 Results

The success of our system depends on three axes: clustering performance, speed, and user friendliness. The third axis is the most difficult to evaluate, but we currently have one user of the system who is excited to use and improve it. We formally evaluate the other two axes here.

5.1 Anomaly Detection

The metric of success for anomaly detection is as follows: assuming only one node is anomalous, how many clusters are required until that node is in its own cluster? The best-case scenario is two clusters: one for each of the $n-1$ well-performing nodes, and another for the anomalous node.

We simulate five experiments on a seven node cluster using six nodes of real data and one node of simulated anomalous data. The five anomalies tested are:

1. **Failed node**

   One node fails when the algorithm is 80% to completion, and all other nodes run to completion.

2. **Node 5% slower**

   One node runs 5% slower than the speed of the other nodes

3. **Node 1% slower**

   One node runs 1% slower than the speed of the other nodes

4. **50% of log is junk**

   Half of the messages in each log file are random, both in time and in value. One node is still 5% slower. Can the anomalous node be detected?

5. **99% of log is junk**

   Only one out of every hundred messages is real. One node is still 5% slower. Can the anomalous node be detected? Note that the file size here is bloated to 100 times the actual log data.

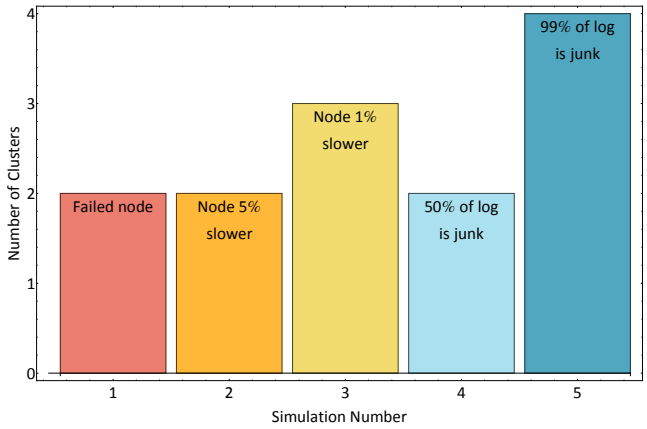

Figure 5: *Results for anomaly detection via clustering. The x axis shows different simulation evaluations; the yaxis is the number of clusters required until the anomalous node is in its own cluster. Two clusters is the best-case scenario. See Section 5.1 for a description of each of the simulation numbers.*

Figure 5 shows the performance of our clustering on each of the above tests. Most of the clustering successfully detects an anomaly with just two clusters. The two failure cases are a 1% slowdown of a node, and when 99% of the log is not useful information and information is outputted at random times. We assume these cases are rare, and are satisfied with successfully clustering a 5% slowdown and 50% junk log data.

We would like to eventually compare our algorithm with the results of Xu and collaborators [Xu et al. 2009b]. Unfortunately, all of their results used private and proprietary logs, so we could not do a direct comparison.

5.2 Timing

Our log files had over 500,000 messages per node. After compression, 30MB of data were sent to the master server

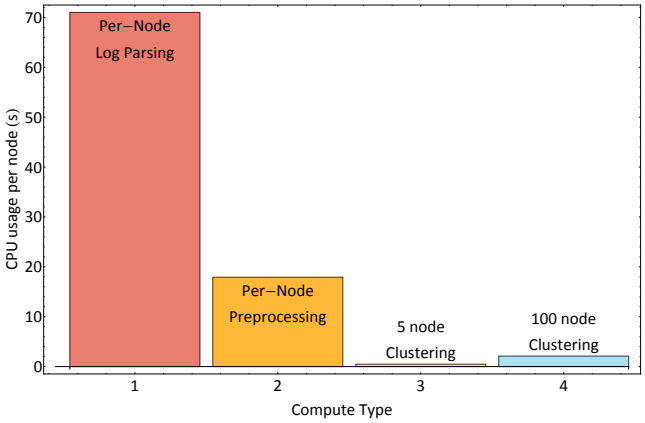

Figure 6: *Timing information. The first column is the per-node log parsing and feature extraction. The second column is the work the master server must do. The last column is the time a user must wait until an updated visualization is displayedafter changing the configuration. See Section 5.2 for more details.*

per node. The visualization was served to the client via an HTTP server, and all javascript visualizations were computed client-side, so the only cost of visualization is generating and transferring the data.

Figure 6 shows basic timing information of our system. From this, it is clear that the per-node cost is small: less than two seconds to compute features per-node. This preprocessing is the bottleneck in our unoptimized implementation. After the distributed preprocessing this data is then passed on to the master node, which prepares to send it to the client via HTTP. Finally, when the user changes the parameters with the web interface, the k-means clustering is recomputed at interactive rates.

When the number of nodes increases to 100, and the number of log messages to 50 million, we are still able to run the clustering in 2.1 seconds on a single thread. We have not found it necessary to use distributed or approximate k-means clustering, but if the number of nodes increases, it will be worthwhile to use a sublinear k-means algorithm.

6 A Note on the Data

Our goal for this project was to assist in analyzing a specific source of log files, which was a machine learning algorithm on terabytes of data. The machine learning algorithm used about 80 processes, each with its own log file, and spread over around a dozen machines. There are regular failures which are currently cumbersome to find and debug. At the same time, we kept our system general enough to work on other types of log files, particularly one with many more than a dozen nodes.

For this particular application, we believe our system can immensely simplify the analysis of the log data. Diverging residuals, crashed nodes, and network failures are the common failure cases, and our system can capture all of these. While other log analysis research often focuses on analysis of a *system* [Xu et al. 2009a; Xu et al. 2009b], we also focus on analysis of an *algorithm* which periodically logs its progress. The failures are then in the variables, not the existence of the message.

7 Conclusions and Future Work

We have presented a system for visualizing arbitrary log files. The log files are automatically and distributedly parsed, features are computed per node, and the results are visualized in an easy-to-use web interface. We cluster similarly-performing nodes into a single visualization to abstract away redundant data.

The visualization has a spectrum of parameters. On one end of this spectrum, the raw log data is shown; on the other end, a single graph shows the most representative state of the system. This is important to maintain *trustworthiness* of the visualization: no data is hidden from the user.

Our visualization is still extremely primitive. Users may have no intuition as to why two clusters are separated. Displaying a cluster confidence may assist in telling the user when two clusters are not very different. On the other hand, if two clusters should be different, we would want to provide an explanation to the user similar to the decision tree visualization of Xu and collaborators [2009b]. As is, it would be very hard for a user to notice a 1% slowdown in a node.

We would like to compute more representative features for the clustering, including higher-order statistics. However, as the feature complexity increases, the interactivity decreases. Finding a middle-ground between interactivity and complexity is an difficult problem: we must be able to compute all of the features without touching all of the data.

Finally, we would need a more thorough evaluation of our visualization. Some questions we have are:

1. is a scatterplot the best visualization choice?
2. are we able to successfully segregate other types of anomalies in different clusters?
3. can multiple messages be aggregated into the same plot?
4. what additional information would assist the human in understanding the visualization?

There is a lot to be desired before our system is usable in the wild. We believe we have taken a major step in the right direction by providing the user with tools to explore the raw log data, rather than trying to predict what they would like to know.

Acknowledgements

The authors would like to thank Ling Huang for helpful discussions and insight into the structure of log files, Sean Arietta for providing us with the log data gathered from his research project, and John Kubiatowicz and Anthony Joseph for their guidance throughout this project. Adrien Truielle at Carnegie Mellon University provided the desktop machine on which we ran our experiments.

References

BOSTOCK, M., OGIEVETSKY, V., AND HEER, J. 2011. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*.

HANGAL, S. 2011. Seaview: Using fine-grained type inference to aid log file analysis.

JIN, R., GOSWAMI, A., AND AGRAWAL, G. 2006. Fast and exact out-of-core and distributed k-means clustering. *Knowledge and Information Systems 10*, 1, 17–40.

KANUNGO, T., MOUNT, D. M., NETANYAHU, N. S., PIATKO, C. D., SILVERMAN, R., AND WU, A. Y. 2002. A local search approximation algorithm for k-means clustering. In *Proceedings of the eighteenth annual symposium on Computational geometry*, ACM, 10–18.

LLOYD, S. 1982. Least squares quantization in pcm. *Information Theory, IEEE Transactions on 28*, 2, 129–137.

MACQUEEN, J. B. 1967. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, L. M. L. Cam and J. Neyman, Eds., vol. 1, 281–297.

MAKANJU, A., ZINCIR-HEYWOOD, A. N., MILIOS, E. E., AND LATZEL, M. 2012. Spatio-temporal decomposition, clustering and identification for alert detection in system logs. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ACM, 621–628.

MASSIE, M. L., CHUN, B. N., AND CULLER, D. E. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 7, 817 – 840.

OETIKER, T., 1999. Rrdtool. `http://oss.oetiker.ch/rrdtool/`.

RATCLIFF, J. W., AND METZENER, D. 1988. Pattern matching: The gestalt approach. *Dr. Dobb's Journal* (July), 46.

SAGANOWSKI, L., GONCERZEWICZ, M., AND ANDRYSIAK, T. 2013. Anomaly detection preprocessor for snort ids system. In *Image Processing and Communications Challenges 4*, R. S. Chora, Ed., vol. 184 of *Advances in Intelligent Systems and Computing*. Springer Berlin Heidelberg, 225–232.

XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. 2009. Online system problem detection by mining patterns of console logs. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, IEEE Computer Society, Washington, DC, USA, ICDM '09, 588–597.

XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ACM, New York, NY, USA, SOSP '09, 117–132.