ARL-TR-8624 ● JAN 2019

US Army Research Laboratory

Naïve Bayes Log File Reduction and Analysis

by Ralph P Ritchey, Gregory G Shearer, and Kenneth D Renard

ARL-TR-8624 ● JAN 2019

US Army Research Laboratory

Naïve Bayes Log File Reduction and Analysis

by Ralph P Ritchey
Computational Information Sciences Directorate, ARL

Gregory G Shearer and Kenneth D Renard
ICF International, Fairfax, VA

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**			
1. REPORT DATE *(DD-MM-YYYY)* January 2019	**2. REPORT TYPE** Technical Report		**3. DATES COVERED (From - To)** August 7, 2017–July 10, 2018
4. TITLE AND SUBTITLE Naïve Bayes Log File Reduction and Analysis			**5a. CONTRACT NUMBER**
			5b. GRANT NUMBER
			5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Ralph P Ritchey, Gregory G Shearer, and Kenneth D Renard			**5d. PROJECT NUMBER**
			5e. TASK NUMBER
			5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory Computational Information Sciences Directorate (ATTN: RDRL-CIN-S) Aberdeen Proving Ground, MD 21005			**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-8624
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			**10. SPONSOR/MONITOR'S ACRONYM(S)**
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			
13. SUPPLEMENTARY NOTES			
14. ABSTRACT The application of Bayes' theorem in computer science dates back to the 1960s and continues to be heavily used in Naïve Bayes classifiers in machine learning. In this report, we propose the use of a Naïve Bayes-based classifier for automated analysis and data reduction of text-based log files generated by various computer systems and the services they provide. The intended application of this technique is to automate the reduction of voluminous log files to a more manageable size and, with reasonable accuracy, retain log lines containing potential indicators of malicious cybersecurity activity or other infrequent interesting activity that should be examined further through other means.			
15. SUBJECT TERMS Naïve Bayes, machine learning, log files, reduction, cybersecurity			

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Ralph P Ritchey
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	47	**19b. TELEPHONE NUMBER (Include area code)** (410) 278-3508

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

With the deepening interest in protecting information, the usage of encryption for network connections has greatly increased[1], at the expense of cybersecurity. While encryption enables information security, the effect is a reduced capability for intrusion detection systems (IDS) to perform deep network packet inspection for detection of potentially malicious behavior contained in the packet payload.[2] This impact of encryption on cybersecurity operations is now triggering a search for additional data sources and detection methodologies to use for cybersecurity purposes.

While the ability to monitor network-based communications between devices is reduced, a potential source of additional data are the logs automatically generated on the devices themselves while they perform their activities. This includes portable devices that have evolved to the point of using operating systems offering the same, or very similar, capabilities as traditional, nonportable devices including logging. Various techniques, such as kernel modules and execution of additional processes, have been and are still used for on-device malicious behavior detection. However, great care must be taken to not negatively impact normal operations and the stability of portable devices by depleting their limited resources. Those issues must be carefully factored into any methodology developed to process and reduce log files.

The information logged by the operating systems (OSs) and processes on devices, whether it is a Linux OS system/messages log file, an Apache HTTP server daemon (HTTPD), or a Windows OS log file, contain information such as timestamps, the specific activity performed, users or IP addresses involved, and, in some situations, detailed error information or other important state information. While the format of a particular log type may be reasonably and consistently formatted for automated processing, the volume within and between log types can become untenable for analysis—especially when hundreds or even thousands of systems need to be monitored simultaneously. The combination of computational and man power required to thoroughly process and analyze the log files can exhaust available resources, potentially allowing malicious activity to be missed.

Based upon the success of other researchers applying principal component analysis (PCA)[3] and other machine learning algorithms to log files, our objective was to develop an automated, lightweight technique using machine learning to perform log file reduction with minimal human supervision resulting in smaller log files that still contain potential indicators of cybersecurity events.

2. Literature Review

A literature review was performed, examining published research regarding the application of machine learning to system log files. The following is an overview of relevant publications:

- Li's paper[4] specifically discusses the use of machine learning for processing log files specific to an Ericsson piece of hardware. Recognizing the unbalanced data, the author mentions clustering and statistical-based algorithms as appropriate options to aid in identifying anomalies. The application of the research builds upon a previous version of Awesome Automatic Log Analysis (AALA) version 1.0 by incorporating additional algorithms and thereby becoming AALA version 2.0. Specifics regarding log file contents were not included, making it difficult to do a thorough comparison with nonEricsson hardware log files to determine direct applicability.

- A paper by Aharon et al.[5] is closely related to our work as the paper describes using "system event logs" as the data source. The authors implemented a more sophisticated algorithm called the Principal Atoms Recognition In Sets (PARIS) capable of grouping log lines into "events". System requirements, resource utilization information, and execution times were not provided.

- Xu et al.[3] discuss the detection of issues using log files in large-scale environments. The authors leveraged a PCA for processing the log files, as well as a static analysis technique to extract log line templates from source code to facilitate improved breaking of individual log lines into more-meaningful features. Accuracy performance information is provided; however, system requirements, resource utilization, and execution times were not.

- Siploa et al.[6] approach anomaly detection in network log data using diffusion maps. The log-file data used came from an Apache web server. The results from using a diffusion map were compared to results obtained using PCA and support vector machines. System requirements, resource utilization information, and execution times were not provided.

- He et al.[7] provide results on log anomaly detection based on using six different machine learning algorithms and different techniques for separating log file contents into groups. Two data sets were used: one from a Hadoop Distributed File System logs, and system logs collected from the Blue Gene/L supercomputer. The paper provides hardware configuration information and details on execution times; however, data regarding resource utilization were not provided. Effectiveness of the various algorithms in identifying anomalies was provided, as well as the impact of grouping techniques on the effectiveness of the algorithms used.

- A known challenge entering into this research project is the skewed data set—a tiny number of true-positives (outliers we want to keep) in comparison to the number of false-positives (normal log entries). The skewed data set relates to one-class classification (OCC) and provided another avenue to find relevant research. Khan[8] provides an overview of published OCC research papers and the approaches taken by various researchers.

While prior research was found, only one paper included information regarding system configuration and some (but not all) data points regarding resource utilization. In addition to determining which machine learning algorithm is suitable for our purpose, we must also know how the algorithms utilize available computing resources. For our intended use case, the algorithm must be accurate while using minimal computing resources to be deemed a suitable solution.

3. Approach

The research for this observational study is contingent upon two assumptions:

- Malicious or suspicious activity is much less common than normal requests and thereby results in far fewer log entries.
- Nonmalicious but still interesting activity is also much less common than normal requests and therefore also results in far fewer log entries.

The approach taken, based upon the assumptions previously listed, was to leverage an unsupervised machine learning algorithm to automatically identify *outliers* in a log file, where outlier is defined as "any observation in a set of data that is inconsistent with the remainder of the observations in that data set".[9] Identifying and reducing a log file to contain only the outliers would result in a greatly reduced dataset retaining the two types of activities identified in the assumptions. The reduced log file may then be fed to other processes or algorithms for further processing if desired.

This approach was broken into two overarching steps: selecting log files to be used for the experiments and refining the unsupervised machine learning configuration to improve performance after an initial experiment to use as a performance baseline. For primary experimentation, Apache HTTPD log files from a production Linux server were selected. This decision was based upon having sufficiently sized logs available to provide meaningful results and our direct experience and knowledge of using Apache on Linux. Other experts were also readily available if we required further insight or interpretation of log data. As a secondary log file type, Linux system/message log files were selected. The system/message log data, centrally collected from a wide set of production servers, would be used to roughly gauge the transportability of our methodology developed from experimenting with Apache logs to a different log file type. Both log file types provided a computationally appropriate amount of data with sufficient variability within features.

The initial unsupervised machine learning algorithm selected was a PCA algorithm based upon the successful use as outlined in a research paper authored by Xu et al.[3] An initial test using an Apache HTTPD server access log file with 75,000 lines was performed using PCA to reduce the dimensionality of the complete feature set down to three dimensions for generating a human interpretable graph (Fig. 1). Viewing the PCA-based graph raised numerous questions: While humans can easily see outliers, due to the scale of the graph are they one, a few, or hundreds of points looking like a single outlier? If it is one or even a few overlapping plotted points, which log line(s) do they correspond to?

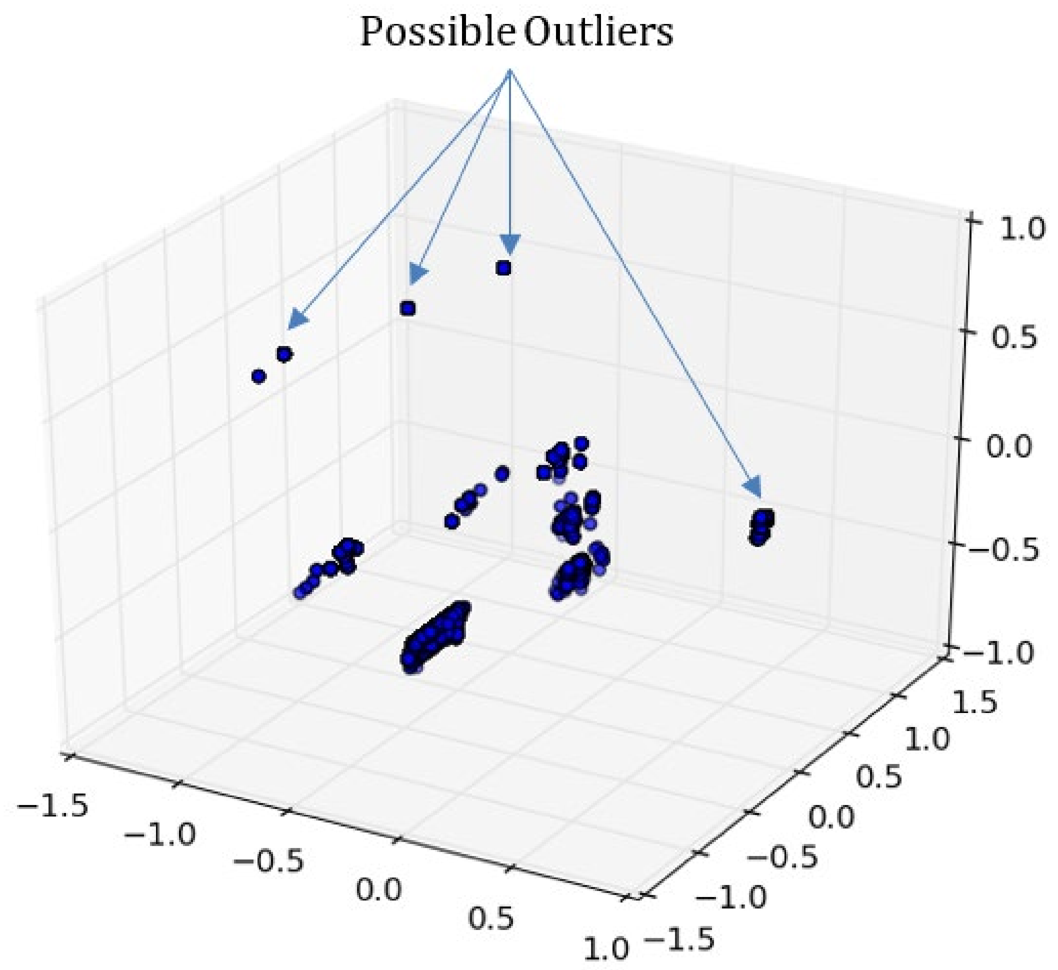

Fig. 1 Static 3-D graph of PCA results from analyzing HTTPD access log

The ability to tie results to specific log lines is critical to determining the performance of log file reduction while still retaining the desired outliers (potential indicators of malicious intent or interesting activity). Compounding these challenges further, executing the PCA against the data exhibited a significant utilization of both CPU and memory, to the point some log files could not be used due to complete exhaustion of available physical memory. This level of resource utilization immediately eliminated PCA as a possible algorithm as completely exhausting an available resource (physical memory) did not meet our goal of limited or reasonable resource utilization. Execution time was also found to be significantly longer (Table A-2 in the Appendix) for PCA than the subsequently used algorithm. Longer execution times would impact battery life on portable devices, further eliminating PCA as a possible algorithm meeting our basic requirements.

Based upon prior internal research applying Naïve Bayes, a decision was made to use Naïve Bayes instead of PCA for experimentation. In addition to lowered resource utilization and significantly shorter execution times, this adjustment provided the capability to directly tie results back to the originating log lines. Extracting outliers from the results was accomplished using a simple threshold technique as their location was restricted to the lower region of the graph, whereas for PCA, outliers could appear anywhere making their identification and extraction more challenging.

3.1 Logalyzer

At the outset of this research, a simple Python script leveraging the NumPy[10] scientific computing library, scikit-learn[11] machine learning library, and the matplotlib[12] graphing library was created. As research continued and refinements to our coding approach were made based on results, it became clear a more flexible coding approach than the initial simple script provided was needed to facilitate easier re-execution of experiments and tracking of adjustments to the code. The core source code from the original script was retooled into a framework, leveraging a simple plugin[13] based approach. The framework was subsequently named *logalyzer* due to the research focusing on analyzing log files.

The *logalyzer* framework consists of one main script separating the execution workflow for log analysis into three discrete steps: transformation, machine learning, and graphing the results. Additional steps can be easily incorporated; however, for the purposes of our research these three steps were sufficient. Having broken the workflow into those three steps using plug-ins, we were able to quickly add a new machine learning plugin, for instance, while leveraging other existing plugins for other steps during our research, facilitating quick support of "what if" scenarios:

1) Transformation: Transformation plugins read a particular type of log file and convert it into a format suitable for use in a machine learning algorithm. These plugins contain the logic for separating each individual log line into features. Variations of the same plugin can be used to test different features of engineering techniques without impacting the rest of the workflow.

2) Machine learning: These plugins receive the output from a transformation plugin as input and execute a machine learning algorithm against that data. While there is no requirement to do so, information or result messages can be printed while the plugin is executing so that the user can redirect to a file for capture and later review.

3) Graphing: The graphing plugin receives the output from the machine learning algorithm and plots the results in a graph or chart, which is then saved as a file. The current (and only) graphing plugin automatically detects whether the results should be plotted in two or three dimensions.

The current implementation of *logalyzer* maintains all processing and data passed between plugins cached in memory. While several gigabytes of memory were used in some of our experiments, steps were taken to optimize memory utilization. For instance, if one-hot encoding is not required by a machine learning plugin, the internal API will notify the transformation plugin, which will then not perform that action during transformation thus reducing memory utilization by gigabytes.

Logalyzer's framework includes command line parameter flexibility. While an extensive listing of available options for both the main script and each available plugin can be displayed with the "`-h`" command line option, not all options must be used all the time. Each individual plugin can provide required or optional parameters specific to that plugin. The "`-p`" option provided by the main script allows the passing of options to the plugins being used in "`<name>=<value>`" pairs. Required parameters are indicated in the available help output and checks are performed at the start of runtime execution to ensure everything that is required for execution of an experiment was properly provided before processing actually begins.

3.2 Data Sampling

Two types of log files were used for experimental purposes while performing this research: Apache HTTPD access log files and Linux `messages` log files. The Apache HTTPD access log file was collected from a production, public-facing Linux server and spans three contiguous days. The Linux `messages` log files span approximately 17 days and is an aggregate `messages` log collection from over 1000 different Linux systems.

Apache HTTPD access log files were selected as the primary log files for experimentation as attacks are common on publicly accessible web servers and the applications they provide access to. The format of the log file is easily broken into features and the logged feature values will vary due to public accessibility. The consistency in formatting and varied values provided an ideal data source for experiments.

The Linux `messages` log was selected as a secondary source of data for use after the primary experiments were completed with the Apache HTTPD access log file. While formatting of the `messages` file is still relatively consistent and easily broken into features, the feature values were not necessarily as varied. The `messages` log file allowed us to perform an experiment to see how easily transportable the methodology used for Apache HTTPD log files may be to a significantly different log file type.

The following sections provide details regarding each of these log files, as well as the approach taken to break each log line into features.

3.2.1 Apache HTTPD Access Log File

The Apache HTTP server[14] is a commonly used web server for Linux-based systems. It is available in the repositories of most Linux distributions, making it readily available to a wide audience. For these reasons, Apache HTTPD access log files were obtained from several servers: one used internally and one accessible by a much wider range of external users. While the logs from the internal server were initially used during *logalyzer* creation, the results included in this report are for the externally facing web server, which reflects a more realistic, real-world use case.

The following are two (sanitized) sample lines from an Apache HTTPD access log file:

```
192.168.1.12 - - [01/Jul/2017:04:16:41 -0400] "POST
/myapp/core/perform_action.php?menuaction=check_status HTTP/1.1" 200 170042
192.168.1.12 - - [01/Jul/2017:04:17:00 -0400] "GET
/myapp/index.php?doaction=get_data&id=2 HTTP/1.1" 200 177274
```

These are typical log lines containing the originating IP address making the request to the web server, timestamp when the connection from the requestor to the server was established, the HTTP request method coupled with the uniform resource identifier (URI)[15] being requested, followed by the HTTP server response code and the number of bytes sent from the HTTP server back to the requestor as a result of the request. The two hyphens between the originating IP address and timestamp fields represent missing information—specifically the RFC 1413[16] `identd` of the originating IP and the `userid` of the person on the originating IP address making the request. Neither of these fields are typically populated or reliable, so for purposes of this research they were ignored and not used as features. Additional information regarding available fields that can be included in an Apache HTTPD log, and how the log files can be configured, can be found at the Apache HTTP server project website.[17]

3.2.2 Linux Messages Log File

Linux-based servers predominantly log important information from running processes and services to a variety of log files contained in `/var/log`. While many of these may be suitable candidates for the research being performed, the `messages` log file was chosen due to being a core log file capturing a wider range of important log entries for a system as a whole. Other log files are typically specific to one process or service, and due to the more centralized role it plays in the overall health and security of a server, the `messages` log file provided a more varied and meaningful source of data for experimentation.

The following are two (sanitized) sample Linux `messages` log file entries:

```
Sep 10 20:59:10 server1 puppet-agent[13006]: Finished catalog run in 3.02
seconds
Sep 10 19:52:55 server1 sshd[4025]: Connection closed by 192.168.1.14
port 41814 [preauth]
```

Similar to the Apache HTTPD log file, the `messages` log file consists of several well-defined sections: a timestamp indicating when the log entry was added to the file, the name of the server the log entry is from (useful when using centralized logging for numerous servers), the name of the process logging the message coupled with the process identification (PID) number of that process, and ending with a highly variable message section. The content of the message field varies significantly between different process types, but typically follows a well-defined format unique to the process type generating the log line.

4. Naïve Bayes-based Algorithm

Bayes' theorem[18] (Eq. 1) forms the mathematical basis the Naïve Bayes classifier algorithm is built upon. The theorem calculates the conditional probability $P(A|B)$ that an event A occurs given that event B is true. For Naïve Bayes classifiers, A represents a classification such as a dog, cat, horse, and so on. B represents the set of features (weight, fur color, height, etc.) used to calculate their respective probabilities of occurrence for a specific class during training. The resulting model generated during training is then used to calculate the probability a set of features provided as input belongs to each class specified during the training phase. The higher the calculated probability, the higher the likelihood the provided set of features belong to a class. The algorithm is referred to as *naïve* because each feature is given equal weighting under the assumption each feature contributes equally to the probability calculation.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}. \tag{1}$$

Bayes' theorem

When using Naïve Bayes classifiers, the following generalized workflow is used:

1) Perform training on a set of previously classified data to calculate the conditional probabilities for the features within each class.

2) Run a different set of data through the now-trained classifier, calculating the probabilities for each class for each set of features based upon the model created during training in the prior step.

3) Examine the probabilities generated for each class for each set of features. The classification with the highest probability value identifies which class that set of features most likely belongs to.

As operating system updates are applied, changes are made to installed applications, and users' use of systems and applications change, the data logged in a log file will evolve. To compensate for logged data evolution, we approached the Naïve Bayes-based classifier general workflow from a slightly different angle. The majority of the data logged will be normal, routine behavior and use. Log entries indicating potentially malicious behavior should occur with a much lower probability in a log file; therefore, we calculated the probability that a log line belonged in the log file being processed. This approach, which we refer to as a modified Naïve Bayes in this report, has the following generalized workflow containing a key variation in Step 2:

1) Train the algorithm on the features from a log file, treating it as a single class to calculate the conditional probabilities using the standard Naïve Bayes methodology.

2) Run the same features from the same log file through the now-trained single class classifier, calculating *the probability that log line belongs to that log file* by summing the probabilities for each feature value and then calculating the average. The calculation for determining the probability is depicted in Eq. 2, where n represents the total number of features for a record, and z_i is the calculated probability value contained in the model generated in the training step for the feature's value.

3) Log lines with a calculated probability below a threshold (i.e., the lowest calculated probabilities) are deemed to be outliers (potential indicators of malicious behavior).

$$\frac{\sum_{i=0}^{n-1} z_i}{n}. \tag{2}$$

Modified class probability calculation

There are several benefits to this approach, which may not be applicable in other situations.

- The content of log files will evolve over time as usage and users change and software is updated. This is especially true for web servers whose code and content are typically updated on a regular basis. By training on the log file being processed, the probabilities used are automatically tuned for that log file, reducing false-positives that may result by using an older training data set against log lines generated by software or other factors that were recently changed on the system.
- There is no need to manually tag each log line in a training set as to the class it belongs to. Depending on the organization, system activity, and duration of time being analyzed, log files can potentially contain millions of log lines making it challenging for one person to perform manual classification for an initial training set, much less subsequent updates to the training sets as the system naturally evolves.
- If remotely deployed, there is no fear of losing sensitive models created from trained data due to a system compromise or consuming valuable storage space on portable devices.

Note: Implementations of Naïve Bayes typically incorporate *Laplace smoothing* (also known as “additive smoothing” or “Lidstone smoothing”) to account for situations where feature values appear in nontraining data that were not encountered in the training data. Laplace smoothing was not needed in our implementation as the same data are used for training and classification, which is unique to application of the algorithm.

5. Results

In this section the results of running the various log files through the modified Naïve Bayes algorithm are presented. Results are broken into two major sections based on the log file type used, with subsections showing results as feature engineering was improved for experimentation.

For all graphs presented in this section, the x-axis represents each log line, in the order read from the log file. The y-axis represents the calculated probability that the given log line belongs in that log file—or in another way to interpret the probability, given the conditional probabilities for the features in that log line, how often did they occur? The lower the value on the y-axis, the higher the likelihood something potentially malicious may have occurred.

5.1 Apache HTTPD Access Log Experiments

Due to better consistency in formatting between log lines, the single-purpose nature of the log file, and the variety of values for the features, the majority of experimentation was performed on Apache HTTPD access log files. Initially, the individual features from each log line were extracted "as is" with no modification. Experimental results looked promising, which triggered additional internal discussion as to whether or not the results could be further improved by performing additional feature engineering—first by breaking the request URI into logical sections, then by removing specific features viewed as not providing significant value. The following sections cover the details specific to the experimental variations run across the logs used for the experiments.

Over 238,000 lines were contained in the original log file used. In addition to using the log file in its entirety, smaller chunks in increasing size were used: 50,000 lines, 75,000 lines, and 135,000 lines. This facilitated answering the question: How is performance impacted as the number of log lines used increases?

HTTPD log files were not available that had been previously analyzed at the level of detail needed for this research to accurately assess performance in regard to how well the technique properly identified interesting or malicious log entries. To roughly assess research results, an inexact methodology was used to automatically determine in bulk an approximate number of potentially malicious or interesting log lines. By filtering out log lines containing HTTPD response codes indicating the request was processed normally, the remaining log lines would be considered "potentially true-positives". The HTTPD response codes filtered out as indicators of normal, uninteresting behavior are

- 200 = OK
- 301 = Moved permanently
- 302 = Found
- 304 = Not modified
- 501 = Not implemented

Due to the automated, inexact methodology of determining true-positives, it was decided to use the terminology "potential true-positives" and "potential false-positives" to help reflect the performance assessments as being inexact. Without manual, in-depth analysis of each log line, which is not feasible at this scale, there is a likelihood of log lines being improperly categorized. For the targeted use of this research intending to be a reduction in the volume of data while retaining true positives, the tolerance for false-positives is higher than false-negatives. It is important to ensure analysts find sufficient breadcrumbs in the reduced logs that would trigger a manual extraction and review of log lines from the original log file.

The log data does include Nessus scan probes. While not necessarily malicious, these log entries do serve as indicators of potentially malicious behavior. These log lines should appear in the "potential true-positives", which will help further validate whether or not the approach is effective.

5.1.1 Intact URI

For the initial experiment, the URI features contained in the HTTPD log file were kept completely intact without any changes made. Four separate executions were performed with an increasing number of log lines (Figs. 2–5). Results between the executions were then reviewed to see how an increase in the number of log lines impacted the graph. The graphs provided visual input to initially determine what the threshold value should be set to, and the log lines falling below the threshold reviewed for accuracy. (Were they interesting or potential indicators of malicious activity?)

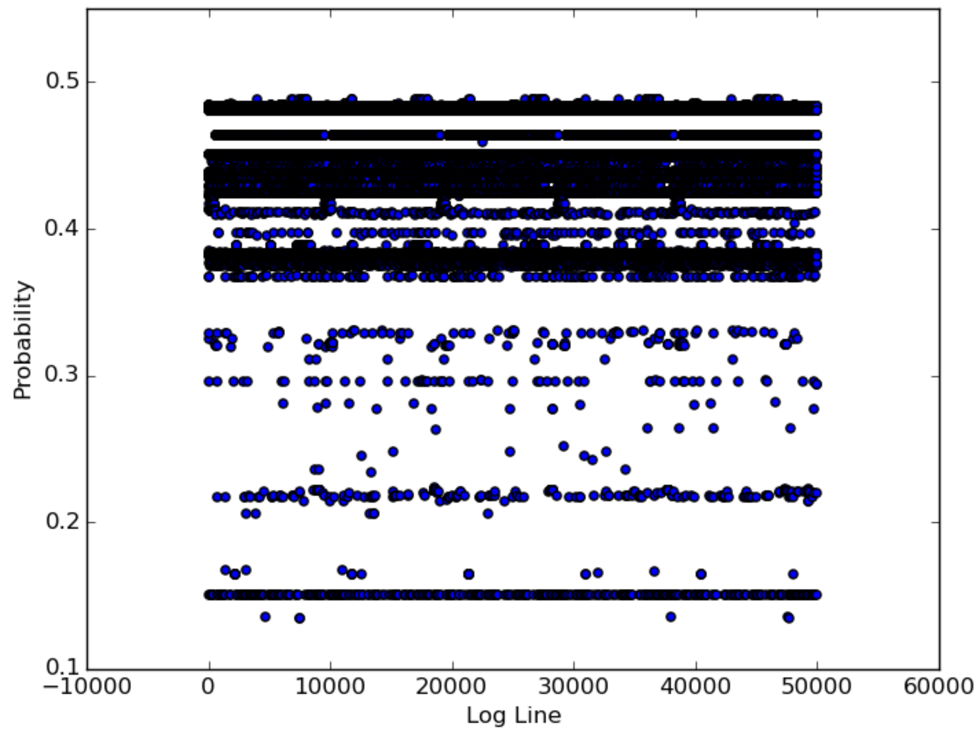

Fig. 2 Apache access log, 50,000 lines, URI intact

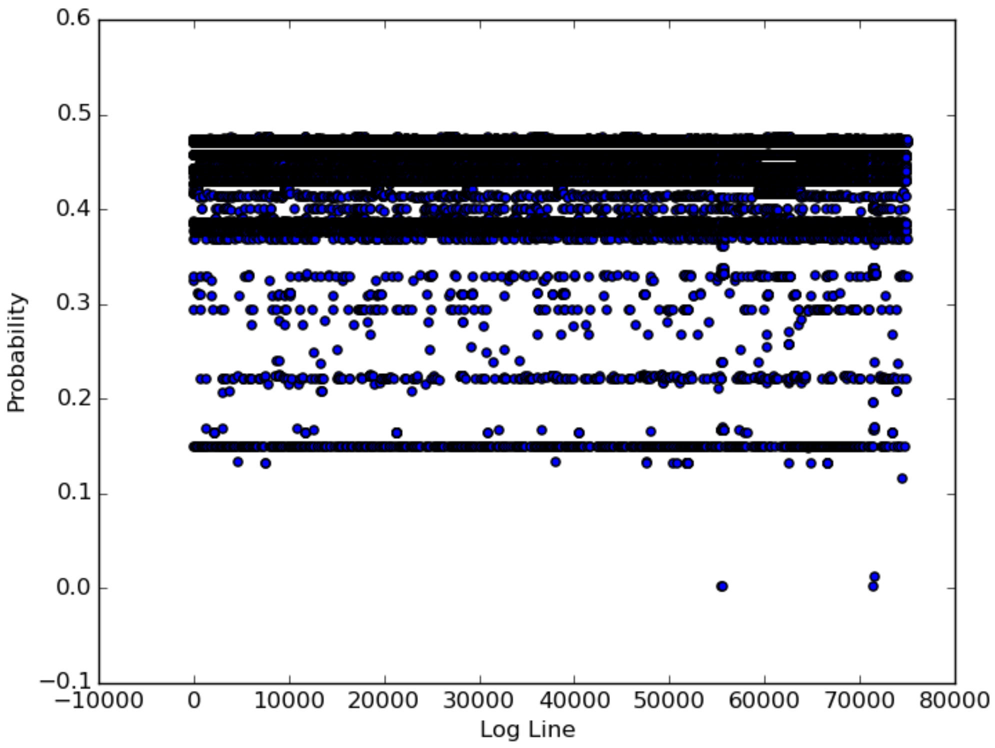

Fig. 3 Apache access log, 75,000 lines, URI intact

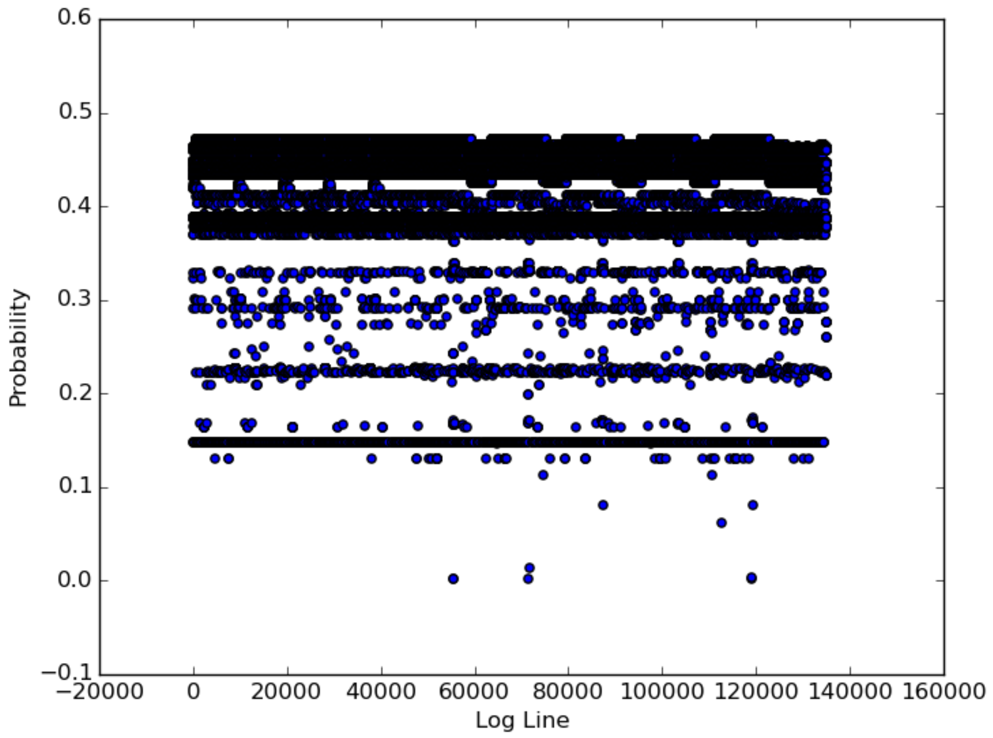

Fig. 4 Apache access log, 135,000 lines, URI intact

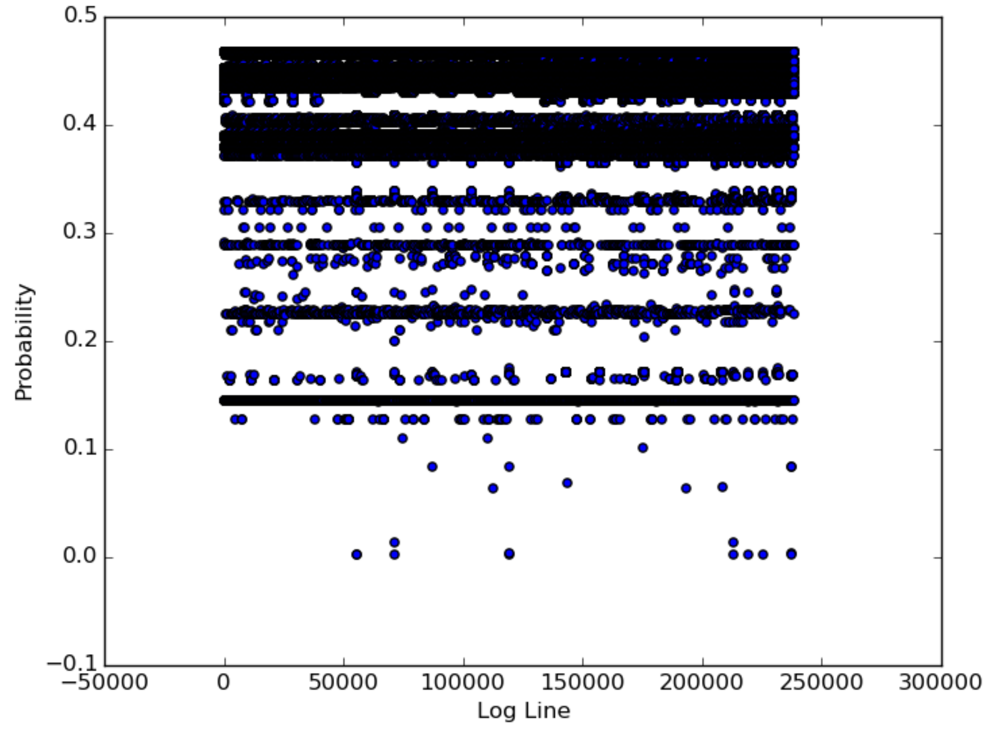

Fig. 5 Apache access log, 238,000 lines, URI intact

With the URI intact, 99.13% of the results (921 lines) from the 50,000-line log file were potential false-positives. Only 8 (13.33%) of the 60 potential true-positives were correctly flagged with probabilities falling below the threshold. These results were obtained using a 0.28 (28% probability) threshold, which was arrived at and used as the standard for this size log file after determining (tuning) the point where the results would suddenly increase with potential false-positives. For each differently sized log file, the same initial tuning was performed during this experiment and that derived threshold value was then used throughout subsequent experiments.

While more potential true-positives (32) were correctly flagged in the 75,000-line log file, the performance dropped to 5.73% as the actual number flagged as falling below the threshold in this experiment is 558. There was a slight performance improvement as the percentage of potentially false-positive log lines falling below the threshold fell to 97.7% versus the 99.13% seen in the 50,000-line log file. Comparing the calculated probability for the same log lines between the 50,000- and 75,000-line log files showed a rough decrease by 0.01 in the 75,000-line log file, which resulted in the significant increase in log lines falling below the threshold.

Another decrease in identification of potential true-positives (59 out of 1583) to 3.73% was seen with the 135,000-line log file; however, the potential false-positive rate essentially remained the same at 97.71%. Interestingly, the identification of potential true-positives (453 out of 5349) increased to 8.47% and the potential false-positive rate dropped to 90.82% for the log file containing slightly over 238,000 log lines.

While the expected increase in the number of flagged potential true-positives does occur as the log file size increased, the performance initially decreased as the number of log lines increased until improving slightly with the full log file. The only conclusions that can be extracted from this experiment is the log files can be reduced to roughly 2% of their original size using this configuration; and as the log file size increases, the number of flagged potential false-positives decreases, while a less-than-desirable rate of performance is seen in flagging potential true-positives.

Reviewing the output to see how many lines with the keyword "`nessus`" were identified as potential true-positives (refer to Table A-4 in the Appendix), only 5 out of a total of 16 (31.25%) were identified correctly for the 70,000-line log file. The 11 log lines not marked as potential true-positives all had a probability of 0.33, which is slightly above the threshold used. For the larger log files, performance was lower at 31% correctly identified as potential true-positives. The smallest log file did not contain any Nessus probe traces.

5.1.2 Deconstructed URI

After reviewing the graphs for the previous experiments, the distinct horizontal banding drew our attention. We realized the banding was most likely caused by the same web applications with the same subpaths being accessed down to the same file or subcomponent. Although the original hypothesis was validated in that the proposed technique appears potentially viable for the intended purpose (log file reduction and a basic level of identification of true-positives), we hypothesized the initial results could be improved by deconstructing the single URI feature into three separate features: 1) the application being accessed, 2) the path/file for a specific component within that application, and 3) the parameters being passed. The basic approach taken when deconstructing the URI is the root (first component) of the URI is typically, but not always, the name or, at some level, the indicator of the web application being accessed by a remote user. The end of the URI (third section) contains any parameters being passed to the application, which may or may not always be present. The remainder of the URI, the center (second) section, is typically an indicator of a specific feature or functionality of the web application being accessed.

For example, deconstructing the example URI (“`GET /myapp/index.php?doaction=get_data&id=2`”) provided in Section 3.2.1 would result in the following three new features replacing the single URI feature:

1) Application: `myapp`
2) Path/file: `index.php`
3) Parameters: `doaction=get_data&id=2`

By separating the URI into the three components, the supposition is the banding witnessed in the previous experiment will become more well defined as applications fall into specific probability ranges based upon how often they are accessed, with the width of the banding being determined by the number of unique paths/files being accessed for each application and the variance in the parameters. An additional supposition is the banding will become more refined as the number of lines increased in the log file being used. Comparing the graphs between the deconstructed URI experiment and the intact URI does show an increased definition in the banding (Figs. 6–9). In reviewing the results from splitting the URI and calculating the occurrences of each unique application, the number of bands roughly align with the number of applications with the highest occurrence rate before a drop off in occurrences occurs.

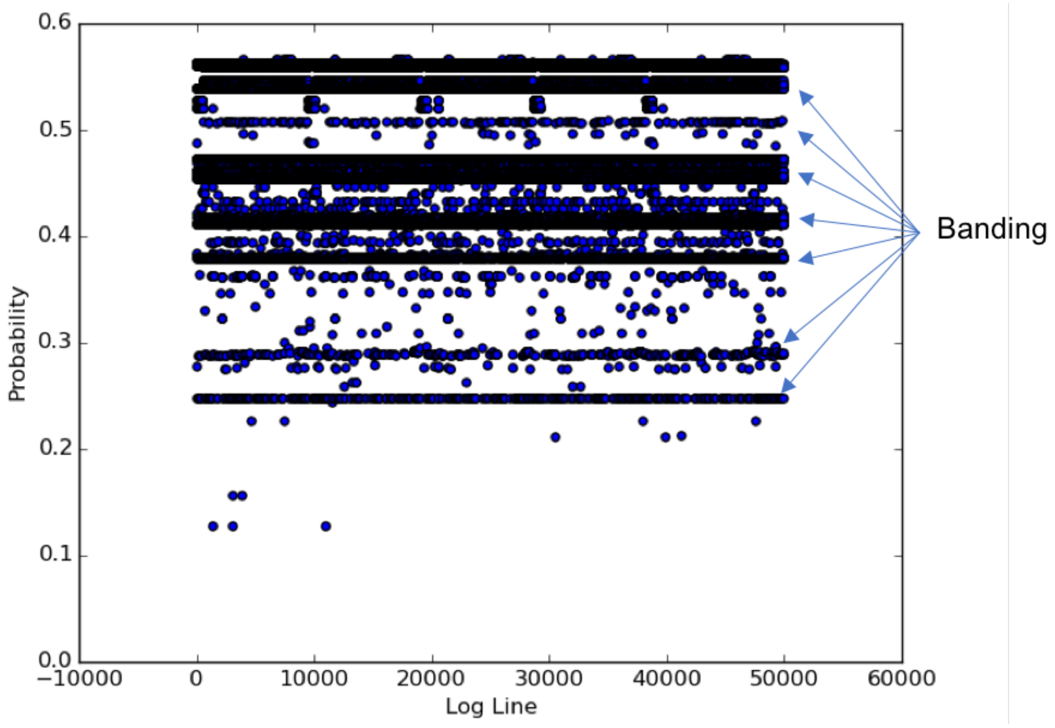

Fig. 6 Apache access log, 50,000 lines, deconstructed URI

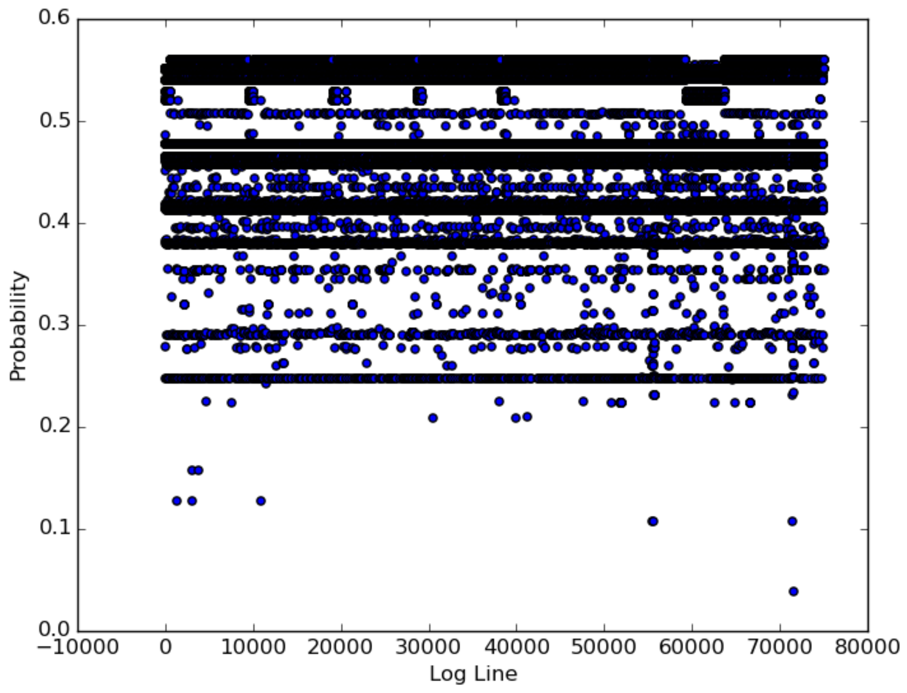

Fig. 7 Apache access log, 75,000 lines, deconstructed URI

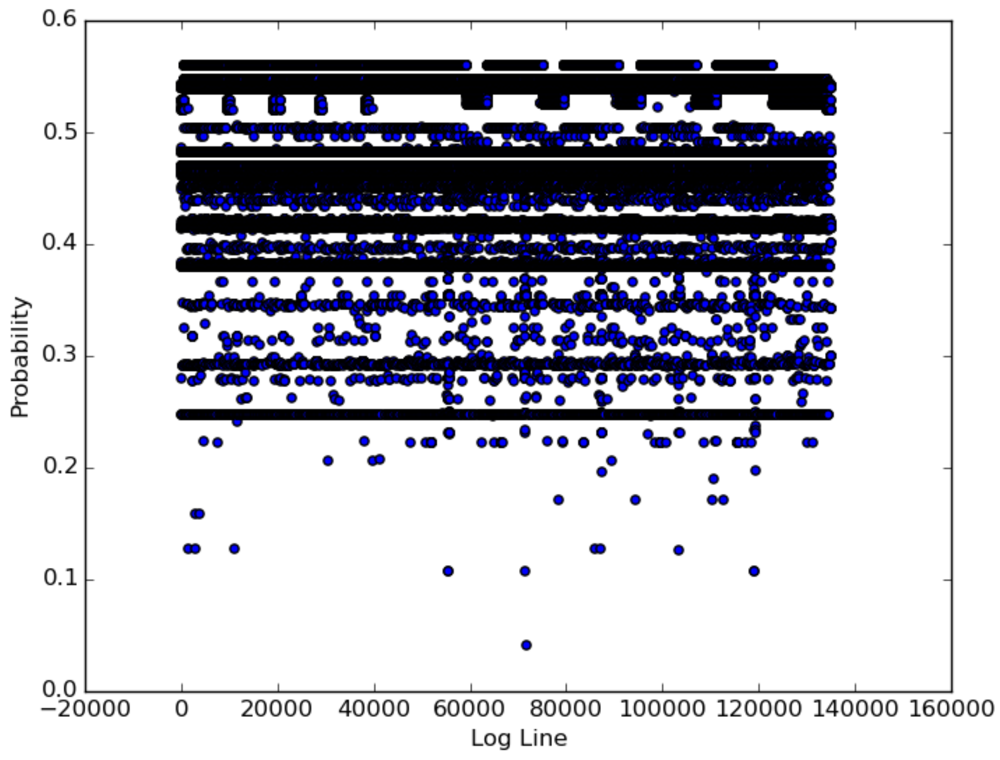

Fig. 8 Apache access log, 135,000 lines, deconstructed URI

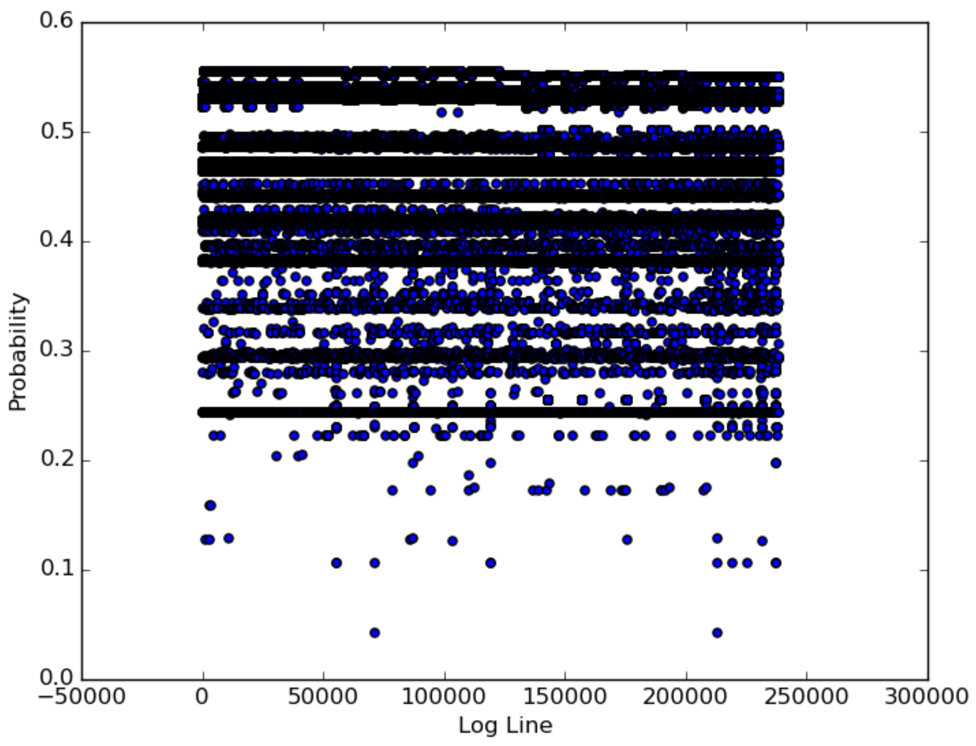

Fig. 9 Apache access log, 238,000 lines, deconstructed URI

With the URI deconstructed, the number of log lines falling below the threshold for each log file size decreased dramatically reducing the number of flagged false-positives by 43% or more in each experiment when compared to the previous experiment where the URI was not split. Excluding the results for the 50,000-line log file, the flagged potential true-positives also showed increased performance in comparison to the previous experiment. Further investigation would need to be performed to determine why the potential true-positive results for the 50,000-line log file dipped slightly instead of improving as it did for the larger log files.

For performance related to the proper identification of Nessus scan probes, the 75,000-line log file doubled performance to 100%. Similar improvement was seen in the larger log file sizes where both increased from 31.25% to 62.5%.

5.1.3 Deconstructed URI, Single Feature Removed

With results from two experiments looking positive, additional time was spent on further feature engineering. The next step taken was to examine the impact on the results if features that were viewed as providing little contributing value were removed (Figs. 10–13). The previous experiment using the deconstructed URI was re-executed with a modification to the Naïve Bayes-based machine learning plugin to display the table of conditional probabilities for each feature value. After reviewing the probabilities, the decision was made to remove the HTTP protocol feature ("`HTTP/1.1`") as there was insignificant variation in the calculated probabilities (98% for "`HTTP/1.1`" and 2% for "`HTTP/1.0`") due to only two values existing for this feature.

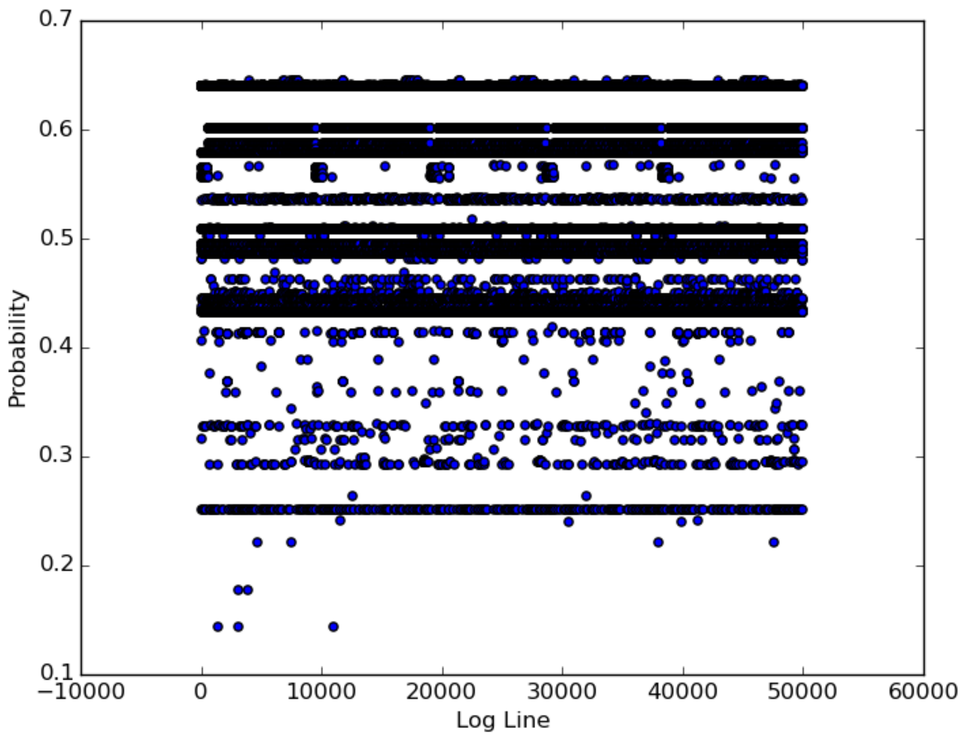

Fig. 10 Apache access log, 50,000 lines, deconstructed URI, one feature removed

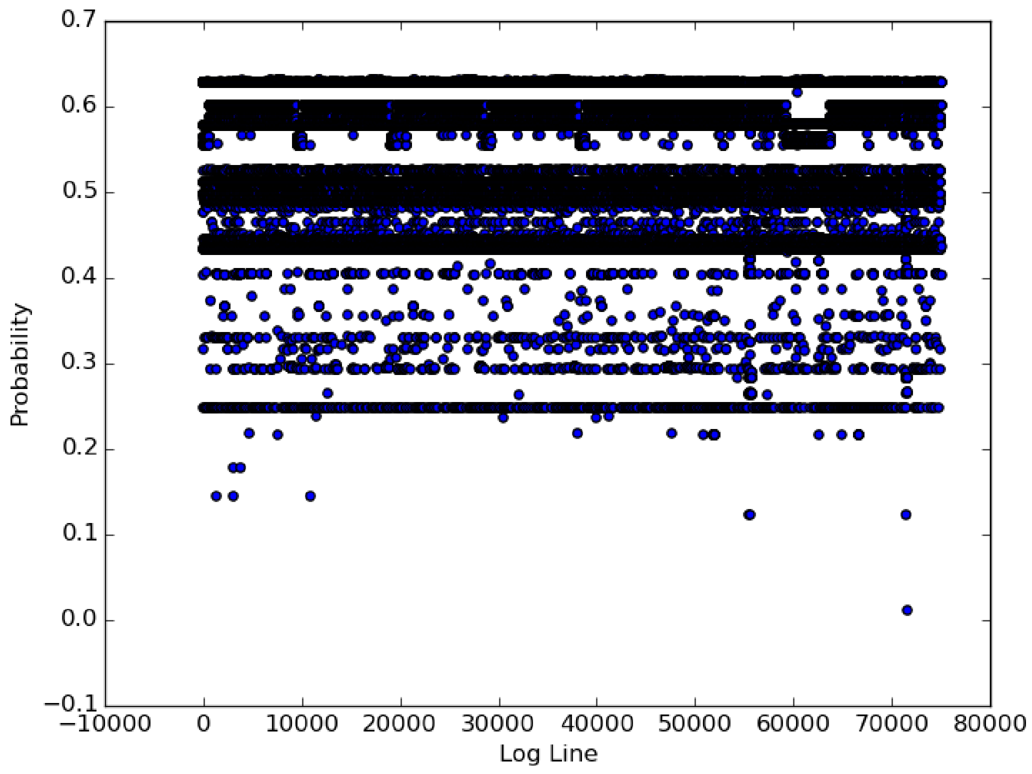

Fig. 11 Apache access log, 75,000 lines, deconstructed URI, one feature removed

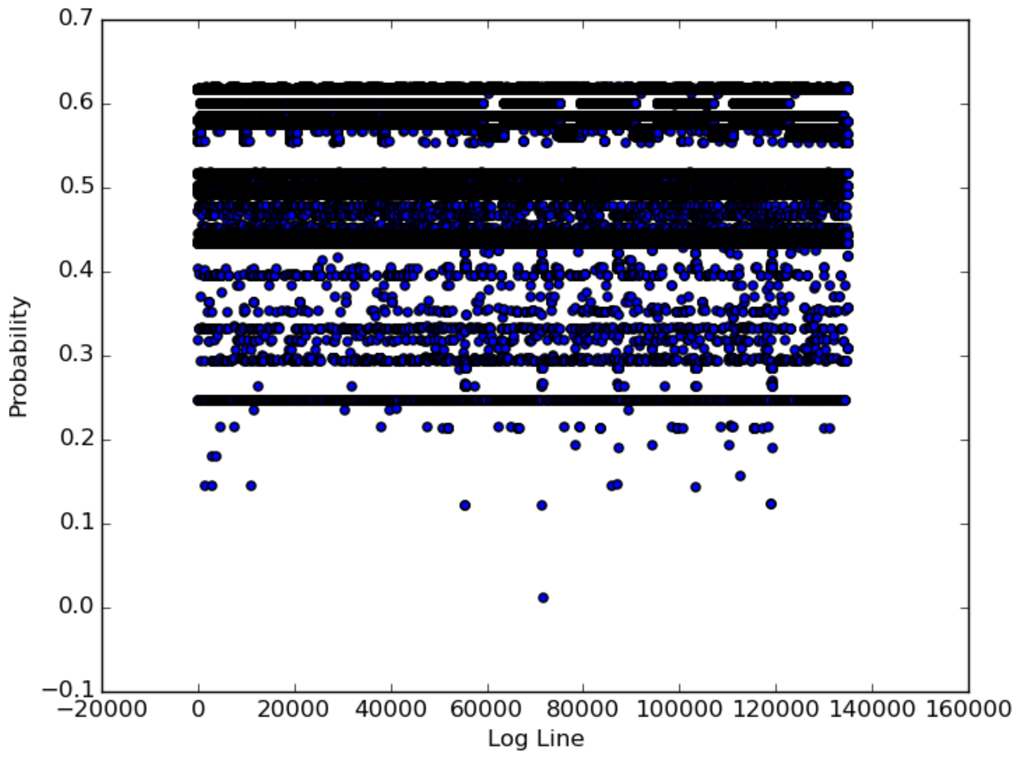

Fig. 12 Apache access log, 135,000 lines, deconstructed URI, one feature removed

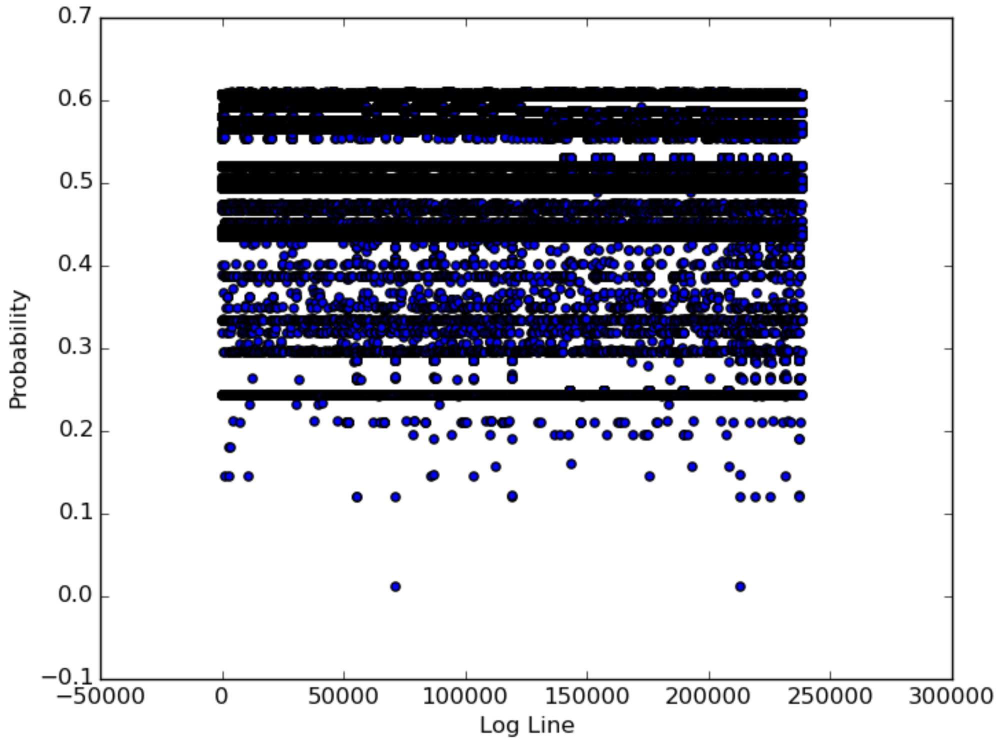

Fig. 13 Apache access log, 238,000 lines, deconstructed URI, one feature removed

When the graphs are compared between the original experiment (all features intact), the second experiment (URI is deconstructed) and this experiment, several changes can be seen. The range for calculated probabilities for each record has expanded from being between 0.1 and 0.5 to between 0.1 and 0.7. This shift has further refined the banding into more discernable, distinct bands and redefines the outliers as can be seen in the number of outliers increasing below the lowest distinct band formed between 0.2 and 0.3.

No or little change was seen for flagging potential true-positives with one feature removed. Flagging of potential false-positives, however, increased significantly in comparison to the prior experiment where the URI was deconstructed—sometimes even greater than the results in the first experiment where all unmodified features were used.

No changes were seen in the performance of proper identification of Nessus scan probes with one feature removed. All performance results remained exactly the same as the prior experiment where the URI was split.

5.1.4 Deconstructed URI, Two Features Removed

For the next experiment, a second feature was removed to determine if there would be a different result than the previous experiment where only one feature was removed (Figs. 14–17). In addition to removing the HTTP protocol feature removed in the previous experiment, the feature containing the HTTP request method (“`GET`”, “`PUT`”, “`HEAD`”, …) was removed due to lack of sufficient variation in values and the perceived limited value in using this feature to identify interesting or potentially malicious behavior.

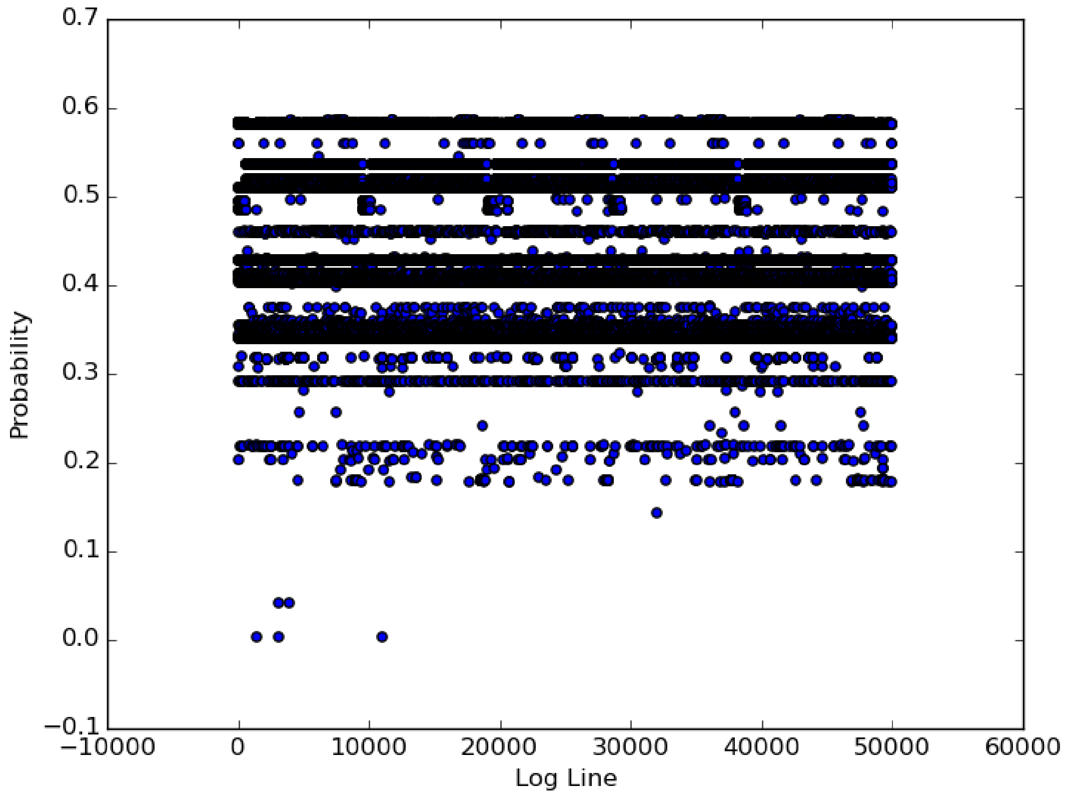

Fig. 14 Apache access log, 50,000 lines, deconstructed URI, two features removed

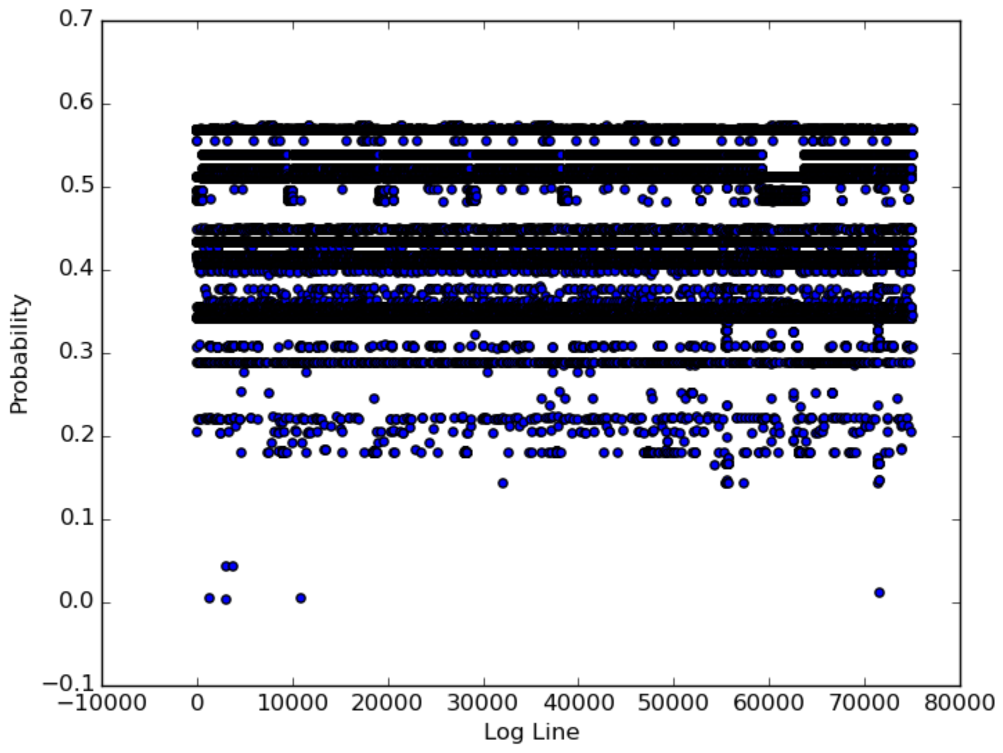

Fig. 15 Apache access log, 75,000 lines, deconstructed URI, two features removed

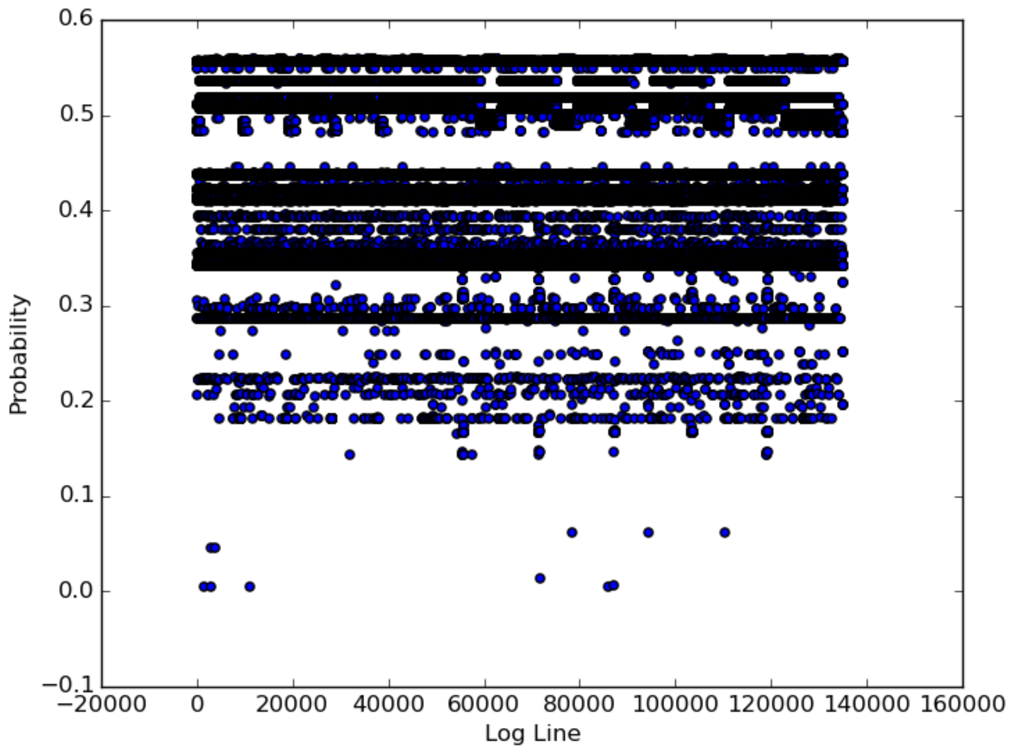

Fig. 16 Apache access log, 135,000 lines, deconstructed URI, two features removed

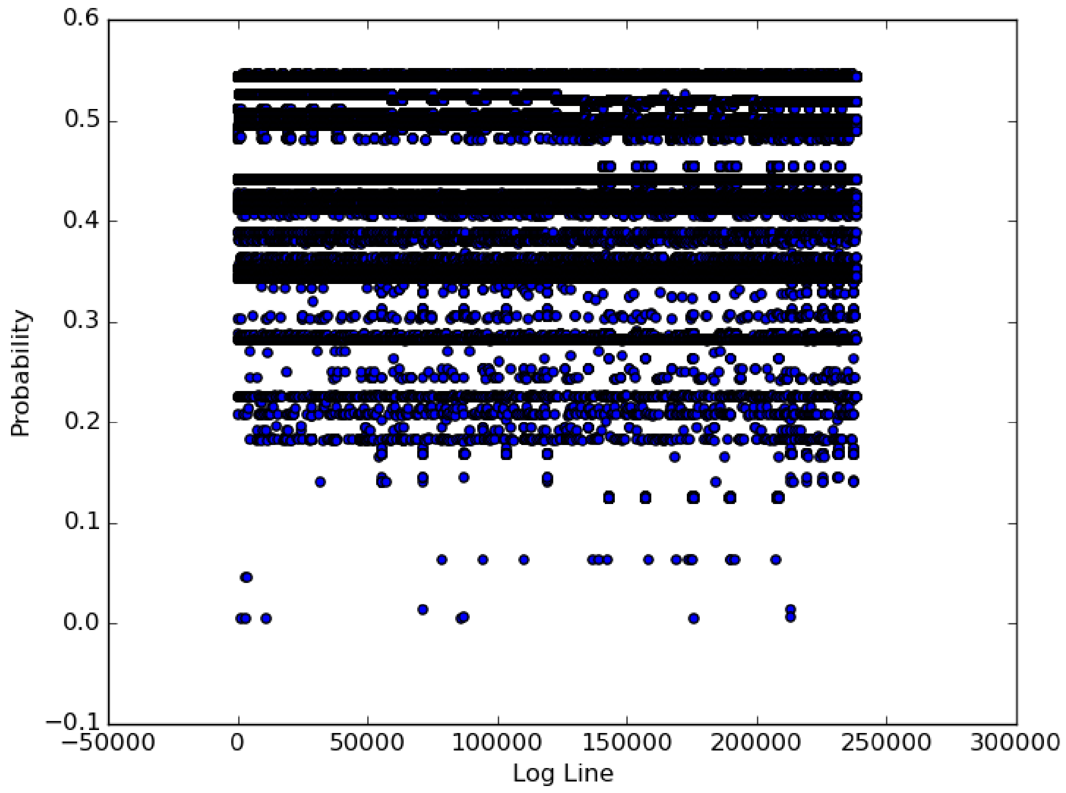

Fig. 17 Apache access log, 238,000 lines, deconstructed URI, two features removed

The results from removing the second feature and deconstructing the URL improved the flagging of potential true-positives by as much as 19 times versus the first experiment where all features were included and kept intact. In comparison to the prior experiment where only one feature was removed, the flagging of potential true-positives improved by sizeable amounts. However, flagging of potential false-positives deteriorated when compared to any of the previous experiments. While the calculated percentage of potential false-positives contained in the output file remained roughly similar to prior experiments, the overall number of flagged log lines increased along with the number of potential true-positives, keeping the ratios roughly the same.

With the second feature removed, another increase in the performance for proper identification of Nessus scan probes occurred. While the 75,000-line log file performance remained at 100%, the larger log files both increased from 62.5% to 100%.

Comparing graphs through the various experiments, the lowest distinct band shifted upward, which raised the question that if the threshold could be raised higher, would it result in further improvement of the flagging of potential true-positives. Additional experiments using the 238,000-line log file and raising the threshold further (0.32, then 0.33) resulted in a significantly increased flagging of potential true-positives to 83.45% and 89.77% accuracy, respectively. While the percentage of flagged potential false-positives decreased very slightly (<1%), there was a significant increase in the number flagged due to the overall number of log lines being flagged. While initial experiments saw a reduction of the original file down to 2% or less of the original file size, these last experiments using higher thresholds showed a reduction down to approximately 37% of the original file size, which is still a significant reduction in the amount of data that would need to be fed into another process, but a significant increase over the original results that also showed significantly less potential true-positive flagging.

5.2 Linux Messages Log Results

After seeing successful results using the technique with the Apache HTTPD access log file, the next question to be answered was: Will similar results be seen with a different type of log file; in other words, how transportable will the technique be? For the next set of experiments, a Linux `messages` log file was used as the input data.

To support this set of experiments, a new transformation plugin was written to process the input file and extract the following features:

1) Severity: A numeric value indicating the severity level[19] of the log entry.
2) Source: Hostname of the system logging the message.
3) Program: Name of the process/service logging the message.
4) Message: The text of the message logged.

Note: Due to the number of "`audit warning: expired`" and "`audit warning: closefile`" messages, these were consolidated to "`audit warning`" during transformation. All other log messages were kept intact.

Other features provided in the file (timestamp, PID, and site) were removed from the feature set during transformation. The values were either of little value (timestamp, PID) or contained a single value that would not contribute to determining if a log line contained something interesting or an indicator of malicious behavior. This decision also factored in the results of removing several features during the experiments executed with the Apache HTTPD access log files that showed an increase in potential true-positive performance when a second feature that had little variation was removed from the feature set.

Note: The data for the Linux `messages` log file used for this report were provided in JavaScript Object Notation (JSON) format from another system storing the collected log data in that format. Under normal circumstances, this log file is plain text. Although in JSON format, the actual contents (values) of the features were exactly the same as they would have been if the original plain text log file had been available. The only impact on the experiment was the coding of the transformation plugin to handle JSON formatted file instead of a text-based log file.

Two different log files were provided, which were referred to as messages Log A and messages Log B. Log A provided a maximum 238,000 lines and Log B provided a maximum of 22,155,000 log lines. Similar to the experiments executed with the Apache HTTPD access log file, each file was experimented with using increasing numbers of log lines until the full log file was used (Figs. 18–21).

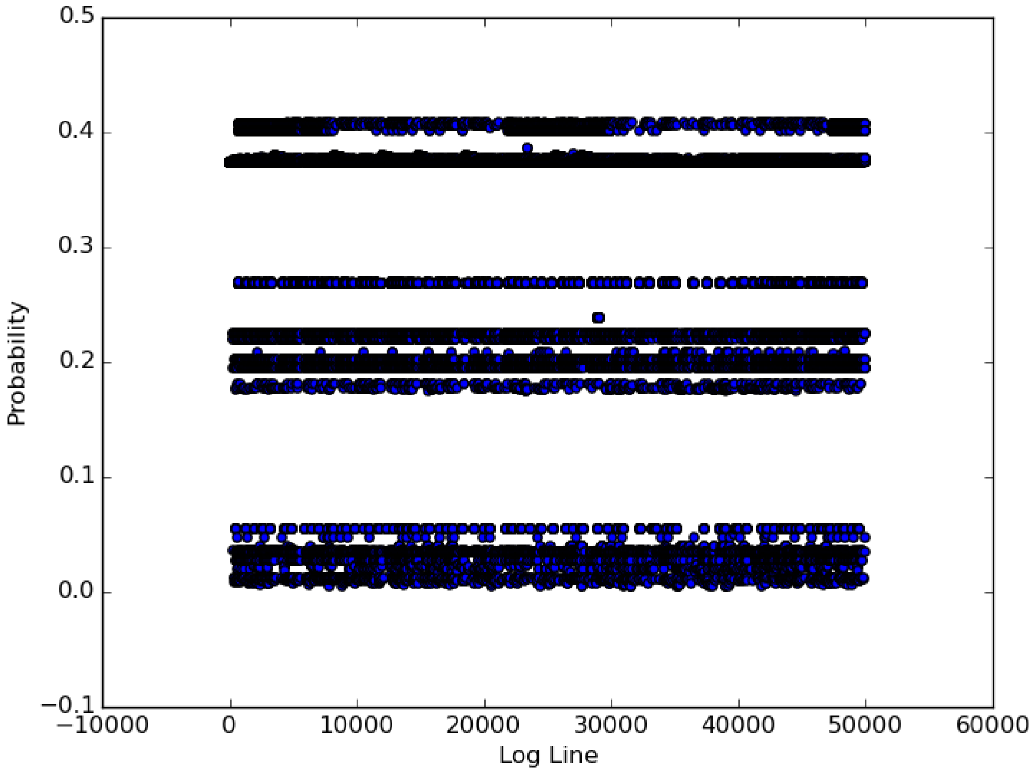

Fig. 18 Linux messages Log A, 50,000 lines

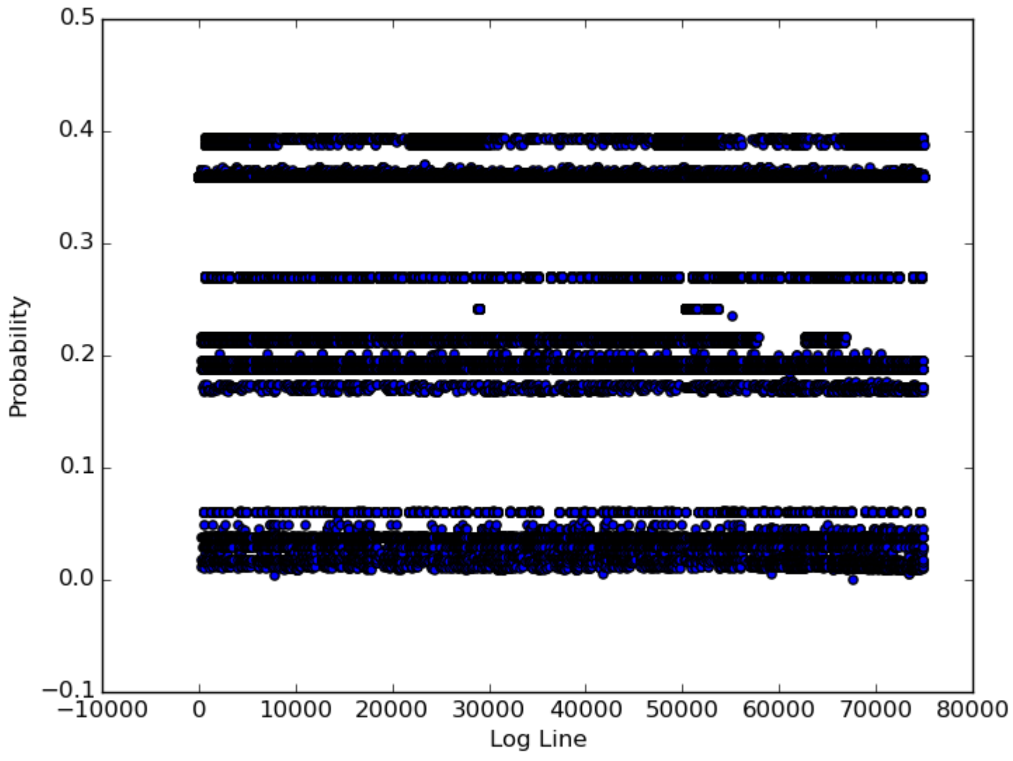

Fig. 19 Linux messages Log A, 75,000 lines

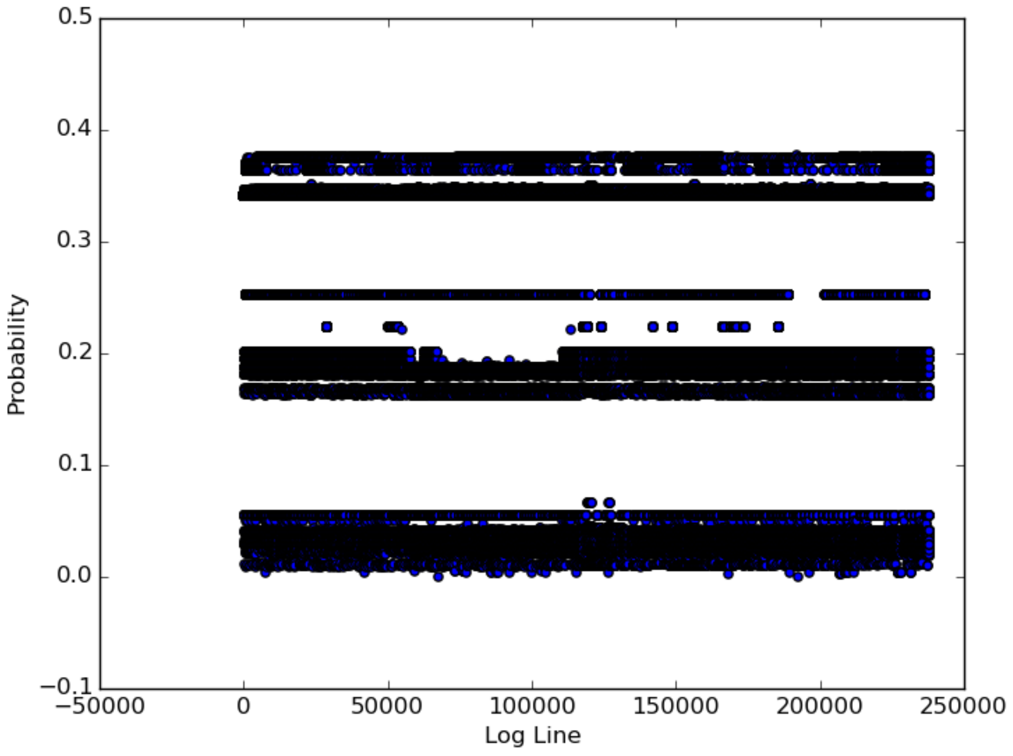

Fig. 20 Linux messages Log A, 238,000 lines

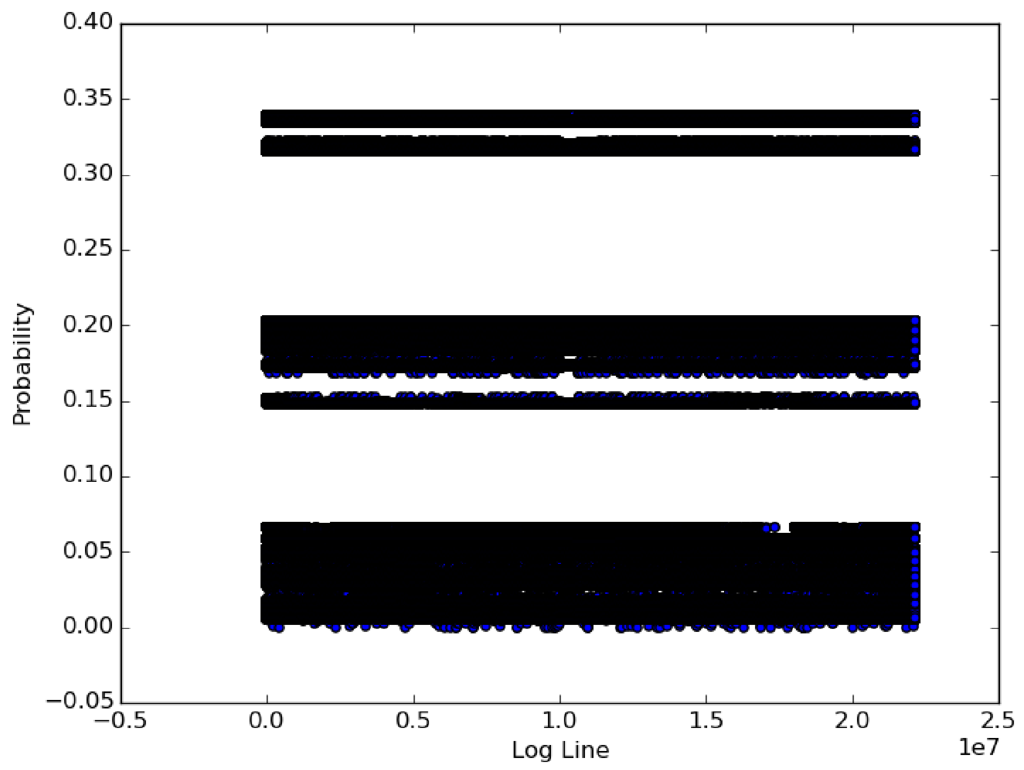

Fig. 21 Linux messages Log B, 22 million lines

Similar to the Apache HTTPD access log file, distinct banding is visible. For the Linux `messages` log file, the banding occurs at a much lower value necessitating the use of a much smaller threshold value of 0.01. With this threshold setting the resulting files were 1.62% or less than the size of the original file. If the threshold value was set higher, the resulting output file would see a considerable increase in size indicating the threshold had gone above the point where potentially interesting or malicious activity was logged and into where routine or highly repetitive log entries start occurring.

Due to the highly repetitive nature of the Linux `messages` log files, it is not possible to generate a reasonable methodology to roughly identify potentially interesting or malicious activity. Therefore, we were unable to generate rough statistics for this set of log files in regard to performance in flagging potential true-positive and potential false-positive log lines. A quick, manual review of the results falling below the threshold was performed, and while nothing overtly indicating potentially malicious activity was found, there were log lines that appeared potentially interesting.

While successful in significantly reducing the size of the original log files to a much more manageable size, it was not possible to conclusively determine how successful the technique was for including malicious or interesting behavior in the reduced data set. Further experimentation would be required and additional feature engineering, similar to that performed on the URI feature for HTTPD log files, may be necessary.

6. Conclusions

The primary goal when undertaking this research was to determine if the application of a machine learning technique could be used for effective processing and reduction of voluminous text-based log files generated naturally by operating systems or the processes and services running on them. When using a PCA algorithm to recreate results from a research paper using log files from our systems, it was discovered this technique, in addition to being resource heavy (see Table A-2 in the Appendix, where time to execute each step is summarized), made it more challenging to review outliers as the number of plotted points increased. We then pivoted to using a Naïve Bayes-based algorithm as the starting point for experimentation. Based upon the results of this approach in our research, the answer is yes, the approach is viable. While there are ample avenues yet to be explored (see Section 7) that may increase performance and reduce resource utilization further, the results showed significant log file size reductions (summarized in Table A-1 of the Appendix) can be obtained with a reasonable level of assurance that enough true-positives (summarized in Table A-3 of the Appendix) are included during the reduction to justify further research efforts. Based on the few known log entries that would be considered malicious, significant improvement in categorization was seen with minor feature engineering improvements.

7. Future work

There are several areas for future work that may be undertaken beyond the research performed for this report. The first area is further refinement and experimentation with feature engineering. While some exploration was performed, such as filtering features included in the log files that provide little value (timestamp for each logged line, destination IP address, etc.), more research can be performed in further intelligently deconstructing some features into smaller subcomponents based on log line context. By further refinement of the decomposition of the URI, it may be possible to detect interesting or potentially malicious accesses based on specific paths/subcomponents within a web application. This may be possible by altering the original technique of calculating probabilities that treat the log file as a single class, but instead, automating class identification during training based on the perceived web application extracted from the URI. This would narrow the probability being calculated for a given log line to "does the request to this application appear within normal ranges compared to other requests" instead of the generalized "does this request look normal for this log file."

Additional improvements for some log types, such as Linux system/messages log files, may be possible through grouping related log lines into single events as detailed in a research paper by Xu et al.[3] This would require researching methodologies for better deconstructing features into subcomponents, coupled with research on how to intelligently link log lines together to form events.

Leveraging virtual reality displays to allow dynamic interaction (zooming in/out, altering viewing angle, etc.) with graphed results needs to be explored. For this paper, the Naïve Bayes-based technique facilitated easy viewing and interpretation due to the 2-D nature of the results. For other algorithms such as PCA, which has been successfully used in similar research,[3] the ability to zoom in and view outliers will greatly aid in determining "is that a single plotted point in the outlier or are there hundreds that are tightly grouped and look like a single point?" Additionally, if metadata can be tied to a plotted point, by zooming in through virtual reality the researcher or user can more quickly determine the nature of the log line (benign, interesting, malicious, etc.). The methodology presented in this report used a threshold value that, while effective, may likely require constant tuning and review. By using the graph as the display, a researcher or user can quickly determine visually, without hesitation, where the line between routine noise and interesting resides. Initial research for alternative graphical representation of IDS data has been undertaken,[20] and has continued to evolve from basic research to exploring the capabilities virtual reality headsets such as the Oculus Rift[21] provide. Those techniques may be highly applicable for this use case when using graphed data.

Lastly, the results from the technique presented in this report may be used as a filtering mechanism to greatly reduce the initial volume of log lines down to a more manageable set that is then fed into other machine learning algorithms or processes. For heavily used systems, the volume of logged data is daunting and having an automated mechanism to quickly and efficiently cull through the data and perform automated data reduction is highly desirable. Logged data for today's systems are well beyond the capability, both time- and cost-wise, to have humans reading and interpreting the data. Subsequent research along this line of thought would entail the application of additional machine learning algorithms or other techniques to identify log lines containing indicators of malicious intent.

8. References

1. Ponemon Institute LLC. Global encryption trends study. Traverse City (MI): Ponemon Institute LLC; 2017 Apr.

2. Goh VT, Zimmermann J, Looi M. Towards intrusion detection for encrypted networks. ARES '09. 4th International Conference on Availability, Reliability and Security; 2009 Mar 16–19; Fukuoka, Japan. Washington (DC): IEEE Computer Society; c2009. p. 540–545.

3. Xu W, Huang L, Fox A, Patterson D, Jordan M. Detecting large-scale system problems by mining console logs. SOSP '09. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles; 2009 Oct 11–14; Big Sky, MT. New York (NY): Association for Computing Machinery (ACM); c2009. p. 117–132.

4. Li W. Automatic log analysis using machine learning: awesome automatic log analysis version 2.0. Uppsala, Sweden: Uppsala University, Department of Information Technology; 2013 Nov.

5. Aharon M, Barash G, Cohen I, Mordechai E. One graph is worth a thousand logs: uncovering hidden structures in massive system event logs. In: Buntine W, Grobelnik M, Mladenić D, Shawe-Taylor J, editors. ECML PKDD 2009. Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases: Part I; 2009 Sep 7–11; Bled, Slovenia. Lecture notes in computer science. Berlin Heidelberg (Germany): Springer-Verlag; c2009; vol 5781. p. 227–243.

6. Sipola T, Juvonen A, Lehtonen J. Anomaly detection from network logs using diffusion maps. In: Iliadis L, Jayne C, editors. EANN/AIAI 2011. Proceedings Part 1: Engineering Applications of Neural Networks–IFIP International Conference on Artificial Intelligence Applications and Innovations; 2011 Sep 15–18; Corfu, Greece. IFIP Advances in information and communication technology. Boston (MA): Springer; c2011; vol 363. p. 172–181.

7. He S, Zhu J, He P, Lyu MR. Experience report: system log analysis for anomaly detection. Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE); 2016 Oct 23–27; Ottawa, Ontario, Canada. Los Alamitos (CA): IEEE Computer Society; c2016. p. 207–218.

8. Khan SS, Madden MG. One-class classification: taxonomy of study and review of techniques. Knowl Eng Rev. 2014;29(3):345–374.

9. Balakrishnan N, Childs A. Outlier. In: Encyclopedia of mathematics [accessed 2018 May 22]. http://www.encyclopediaofmath.org /index.php?title=Outlier&oldid=17990.

10. NumPy developers. NumPy [accessed 2017 Dec 11]. http://www.numpy.org.

11. scikit-learn developers. scikit-learn: machine learning in Python [accessed 2017 Dec 11]. http://scikit-learn.org.

12. Matplotlib development team. Matplotlib: Python plotting–matplotlib 3.0.2 documentation [accessed 2017 Dec 11]. https://matplotlib.org.

13. Ritchey RP, Parker TW. Simple plugin methodology in Python. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2014 Aug. Report No.: ARL-CR-0743.

14. Apache Software Foundation. Apache HTTP server project [accessed 2017 Dec 11]. http://httpd.apache.org.

15. WorldWideWeb (W3) Project. Universal resource identifiers in WWW [accessed 2018 Jan 22]. https://www.w3.org/Addressing/URL/uri-spec.html.

16. US Department of Defense. RFC 1413 identification protocol. 1993 Feb [accessed 2018 Jan 22]. https://www.ietf.org/rfc/rfc1413.txt.

17. Apache Software Foundation. Log files—Apache HTTP server version 2.4 [accessed 2017 Dec 17]. https://httpd.apache.org/docs/2.4/logs.html.

18. Ghorbani AA, Lu W, Tavallaee M. Network intrusion detection and prevention: concepts and techniques. New York (NY): Springer; 2010. p. 77–78.

19. Wikipedia. Syslog [accessed 2017 Dec 15]. https://en.wikipedia.org/wiki/Syslog#Severity_level.

20. Zage DM, Zage WM. Intrusion detection system visualization of network alerts. Muncie (IN): Ball State University; 2010 July. doi.org/10.21236/ada532723.

21. Oculus VR, LLC. Oculus [accessed 2017 Dec 13]. https://www.oculus.com.

Appendix. Log File Reduction Experiment Performance Tables

Table A-1 Log file reduction

Log file	Machine learning plugin used	Variation	Threshold	Total number of log lines	Flagged log lines	Reduced to
access_log_arl50k	pr_naivebayes	All features, unmodified	0.28	50000	921	1.84%
access_log_arl50k	pr_naivebayes	URI split	0.28	50000	449	0.90%
access_log_arl50k	pr_naivebayes	URI split, one feature removed	0.28	50000	919	1.84%
access_log_arl50k	pr_naivebayes	URI split, two features removed	0.28	50000	9916	19.83%
access_log_arl75k	pr_naivebayes	All features, unmodified	0.289	75000	1392	1.86%
access_log_arl75k	pr_naivebayes	URI split	0.289	75000	791	1.05%
access_log_arl75k	pr_naivebayes	URI split, one feature removed	0.289	75000	1573	2.10%
access_log_arl75k	pr_naivebayes	URI split, two features removed	0.289	75000	16002	21.34%
access_log_arl135k	pr_naivebayes	All features, unmodified	0.291	135000	2581	1.91%
access_log_arl135k	pr_naivebayes	URI split	0.291	135000	1580	1.17%
access_log_arl135k	pr_naivebayes	URI split, one feature removed	0.291	135000	2964	2.20%
access_log_arl135k	pr_naivebayes	URI split, two features removed	0.291	135000	30122	22.31%
access_log_arl238k	pr_naivebayes	All features, unmodified	0.29	238468	4933	2.07%
access_log_arl238k	pr_naivebayes	URI split	0.29	238468	3182	1.33%
access_log_arl238k	pr_naivebayes	URI split, one feature removed	0.29	238468	4723	1.98%
access_log_arl238k	pr_naivebayes	URI split, two features removed	0.29	238468	56901	23.86%
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	0.32	238468	87930	36.87%
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	0.33	238468	88292	37.02%
linux_json_messages_log_a50k	pr_naivebayes	All features, unmodified	0.01	50000	380	0.76%
linux_json_messages_log_a75k	pr_naivebayes	All features, unmodified	0.01	75000	306	0.41%
linux_json_messages_log_a238k	pr_naivebayes	All features, unmodified	0.02	238000	1860	0.78%
linux_json_messages_log_b235k	pr_naivebayes	All features, unmodified	0.01	235000	3767	1.60%
linux_json_messages_log_b238k	pr_naivebayes	All features, unmodified	0.01	238000	3846	1.62%
linux_json_messages_log_b22m	pr_naivebayes	All features, unmodified	0.01	22155000	252215	1.14%
access_log_arl50k	scikit_pca3d	All features, unmodified	N/A	50000	Unknown	Unknown
access_log_arl75k	scikit_pca3d	All features, unmodified	N/A	75000	Unknown	Unknown
access_log_arl135k	scikit_pca3d	All features, unmodified	N/A	135000	Unknown	Unknown
access_log_arl238k	scikit_pca3d	All features, unmodified	N/A	238468	Unknown	Unknown

Table A-2   Experiment execution times

Log file	Machine learning (ML) plugin used	Variation	Transform time	ML time	Graph time
access_log_arl50k	pr_naivebayes	All features, unmodified	0.582324	1.179033	0.594666
access_log_arl50k	pr_naivebayes	URI split	0.884504	1.33897	0.593581
access_log_arl50k	pr_naivebayes	URI split, one feature removed	0.889563	1.324833	0.589569
access_log_arl50k	pr_naivebayes	URI split, two features removed	0.843764	2.424957	0.618201
access_log_arl75k	pr_naivebayes	All features, unmodified	0.909166	1.83868	0.701342
access_log_arl75k	pr_naivebayes	URI split	1.338222	1.942471	0.703313
access_log_arl75k	pr_naivebayes	URI split, one feature removed	1.28529	2.064837	0.72904
access_log_arl75k	pr_naivebayes	URI split, two features removed	1.321554	3.786124	0.727776
access_log_arl135k	pr_naivebayes	All features, unmodified	1.656418	3.27457	0.905734
access_log_arl135k	pr_naivebayes	URI split	2.491095	3.522524	0.915193
access_log_arl135k	pr_naivebayes	URI split, one feature removed	2.480453	3.739586	0.94742
access_log_arl135k	pr_naivebayes	URI split, two features removed	2.373569	7.042017	0.928869
access_log_arl238k	pr_naivebayes	All features, unmodified	2.861895	6.073885	1.258091
access_log_arl238k	pr_naivebayes	URI split	4.437634	6.42249	1.257771
access_log_arl238k	pr_naivebayes	URI split, one feature removed	4.330635	6.572117	1.25948
access_log_arl238k	pr_naivebayes	URI split, two features removed	4.292044	12.842495	1.265883
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	5.259213	36.044498	1.300434
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	4.642547	35.524674	1.271213
linux_json_messages_log_a50k	pr_naivebayes	All features, unmodified	0.980014	0.996318	0.614873
linux_json_messages_log_a75k	pr_naivebayes	All features, unmodified	1.515785	1.504827	0.775141
linux_json_messages_log_a238k	pr_naivebayes	All features, unmodified	4.89925	4.84755	1.280357
linux_json_messages_log_b235k	pr_naivebayes	All features, unmodified	4.956596	5.112881	1.283089
linux_json_messages_log_b238k	pr_naivebayes	All features, unmodified	4.846727	5.194599	1.282377
linux_json_messages_log_b22m	pr_naivebayes	All features, unmodified	473.470269	477.361412	83.691511
access_log_arl50k	scikit_pca3d	All features, unmodified	23.928283	867.472425	6.264164
access_log_arl75k	scikit_pca3d	All features, unmodified	44.544439	2962.699099	9.48431
access_log_arl135k	scikit_pca3d	All features, unmodified	250.092163	102978.4837	17.947807
access_log_arl238k	scikit_pca3d	All features, unmodified	out of memory	out of memory	out of memory

Table A-3 Potential true/false-positive performance

Log file	Machine learning plugin used	Variation	Flagged log lines	Actual potential true-positives	Potential flagged true-positives	% Potential true-positives flagged	Flagged potential false-positives	% Potential false-positives
access_log_arl50k	pr_naivebayes	All features, unmodified	921	60	8	13.33%	13.33%	99.13%
access_log_arl50k	pr_naivebayes	URI split	449	60	6	10.00%	10.00%	98.66%
access_log_arl50k	pr_naivebayes	URI split, one feature removed	919	60	6	10.00%	10.00%	99.35%
access_log_arl50k	pr_naivebayes	URI split, two features removed	9916	60	19	31.67%	31.67%	99.81%
access_log_arl75k	pr_naivebayes	All features, unmodified	1392	558	32	5.73%	5.73%	97.70%
access_log_arl75k	pr_naivebayes	URI split	791	558	76	13.62%	13.62%	90.39%
access_log_arl75k	pr_naivebayes	URI split, one feature removed	1573	558	76	13.62%	13.62%	95.17%
access_log_arl75k	pr_naivebayes	URI split, two features removed	16002	558	362	64.87%	64.87%	97.74%
access_log_arl135k	pr_naivebayes	All features, unmodified	2581	1583	59	3.73%	3.73%	97.71%
access_log_arl135k	pr_naivebayes	URI split	1580	1583	185	11.69%	11.69%	88.29%
access_log_arl135k	pr_naivebayes	URI split, one feature removed	2964	1583	187	11.81%	11.81%	93.69%
access_log_arl135k	pr_naivebayes	URI split, two features removed	30122	1583	888	56.10%	56.10%	97.05%
access_log_arl238k	pr_naivebayes	All features, unmodified	4933	5349	453	8.47%	8.47%	90.82%
access_log_arl238k	pr_naivebayes	URI split	3182	5349	600	11.22%	11.22%	81.14%
access_log_arl238k	pr_naivebayes	URI split, one feature removed	4723	5349	686	12.82%	12.82%	85.48%
access_log_arl238k	pr_naivebayes	URI split, two features removed	56901	5349	2778	51.93%	51.93%	95.12%
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	87930	5349	4464	83.45%	83.45%	99.13%
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	88292	5349	4802	89.77%	89.77%	99.13%

Table A-4 Identification of Nessus scan

Log File	Machine learning plugin used	Variation	Total Nessus lines	Identified as true-positives	Identified as false-positives	Accuracy (true-positives)
access_log_arl50k	pr_naivebayes	All features, unmodified	0	N/A	N/A	N/A
access_log_arl50k	pr_naivebayes	URI split	0	N/A	N/A	N/A
access_log_arl50k	pr_naivebayes	URI split, one feature removed	0	N/A	N/A	N/A
access_log_arl50k	pr_naivebayes	URI split, two features removed	0	N/A	N/A	N/A
access_log_arl75k	pr_naivebayes	All features, unmodified	6	3	3	50.00%
access_log_arl75k	pr_naivebayes	URI split	6	6	0	100.00%
access_log_arl75k	pr_naivebayes	URI split, one feature removed	6	6	0	100.00%
access_log_arl75k	pr_naivebayes	URI split, two features removed	6	6	0	100.00%
access_log_arl135k	pr_naivebayes	All features, unmodified	16	5	11	31.25%
access_log_arl135k	pr_naivebayes	URI split	16	10	6	62.50%
access_log_arl135k	pr_naivebayes	URI split, one feature removed	16	10	6	62.50%
access_log_arl135k	pr_naivebayes	URI split, two features removed	16	16	0	100.00%
access_log_arl238k	pr_naivebayes	All features, unmodified	32	10	22	31.25%
access_log_arl238k	pr_naivebayes	URI split	32	20	12	62.50%
access_log_arl238k	pr_naivebayes	URI split, one feature removed	32	20	12	62.50%
access_log_arl238k	pr_naivebayes	URI split, two features removed	32	32	0	100.00%
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	32	32	0	100.00%
access_log_arl238k	pr_naivebayes	URI split, two features removed, higher threshold	32	32	0	100.00%

List of Symbols, Abbreviations, and Acronyms

2-D	two-dimensional
3-D	three-dimensional
AALA	Awesome Automatic Log Analysis
API	application programming interface
BG/L	Blue Gene/L
CPU	central processing unit
HTTP	Hypertext Transfer Protocol
HTTPD	Apache HTTP server daemon
IDS	intrusion detection systems
IP	Internet Protocol
JSON	JavaScript Object Notation
OCC	one-class classification
OS	operating system
PARIS	Principal Atoms Recognition In Sets
PCA	principal component analysis
PID	process identification
URI	uniform resource identifier
URL	uniform resource locator

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
3 (PDF)	ARL RDRL CIN S R P RITCHEY G SHEARER K RENARD