An Unsupervised Framework for Detecting Anomalous Messages from Syslog Log Files

Risto Vaarandi, Bernhards Blumbergs
TUT Centre for Digital Forensics and Cyber Security
Tallinn University of Technology
Tallinn, Estonia
firstname.lastname@ttu.ee

Markus Kont
Technology Branch
NATO CCDCOE
Tallinn, Estonia
firstname.lastname@ccdcoe.org

***Abstract*—System logs provide valuable information about the health status of IT systems and computer networks. Therefore, log file monitoring has been identified as an important system and network management technique. While many solutions have been developed for monitoring known log messages, the detection of previously unknown error conditions has remained a difficult problem. In this paper, we present a novel data mining based framework for detecting anomalous log messages from syslog-based system log files. We also describe the implementation and performance of the framework in a large organizational network.**

Keywords*—anomaly detection for system logs; pattern mining from log files; LogCluster

I. Introduction

Network faults, service failures, security incidents, and other error conditions often trigger log messages which provide detailed error information to system administrators. Therefore, automated system log monitoring for known error conditions is a widely acknowledged practice. Many existing log monitoring tools like Swatch [1] and LogSurfer [2] are rule-based and assume that a human expert defines patterns (e.g., regular expressions) for log messages that require further attention. However, this approach does not allow to identify previously unknown error conditions. For addressing this issue, various anomaly detection methods have been suggested, including hidden Markov models [3, 4], principal component analysis (PCA) [5], decision trees [6], entropy based algorithms [7, 8], support vector machines (SVM) [9, 10], neural networks [11], and logistic regression [12].

For identifying anomalous messages with rule-based log monitoring tools, system administrators often use the following approach – rules are defined for matching all known log messages that reflect normal system activity, while remaining messages are highlighted as anomalous. Since this task requires a lot of expertise, data mining tools have often been suggested for discovering event patterns from log files. However, such tools assume that human experts interpret detected knowledge and create the rules manually which is time-consuming and expensive. In this paper, we propose a novel data mining based framework for *fully automated* rule discovery for real-time detection of anomalous messages from syslog-based logs. Although the framework does not require human intervention and adapts to changes in the system, the human expert can nevertheless augment the framework with hand-written rules (e.g., our framework implementation employs rules for EWMA based alerting). The remainder of this paper is organized as follows – section II reviews related work, section III presents the framework, section IV describes its implementation and performance, and section V outlines future work.

II. Related Work

Yamanishi and Maruyama [3] have suggested an unsupervised method for network failure prediction from syslog error events. The method divides the log into time slots (sessions) and converts original events into event type symbols. Sessions are modeled with hidden Markov model mixtures, while model parameters are learned in unsupervised fashion. An anomaly score is calculated for each session, with one reported as anomalous if its score exceeds a threshold. Salfner [4] has proposed a supervised failure prediction algorithm which is based on hidden semi-Markov models and takes numerical error event type sequences for its input. The approach employs Levenshtein distance function for grouping similar events under the same event type ID. According to experiments conducted with telecommunication log data, the method compares favorably to other approaches. Xu, Huang, Fox, Patterson and Jordan [5] have suggested an unsupervised method which detects anomalous event sequences using PCA. The method employs source code analysis for detecting event types and event type sequences. Sequences are then converted into vectors, where each vector attribute reflects the number of events of some type in the sequence. During detection process, vectors containing frequently occurring patterns are filtered out, since they are highly likely to correspond to normal event sequences. For detecting anomalous sequences, PCA is applied for remaining vectors. Reidemeister, Jiang and Ward [6] have proposed a supervised method which harnesses two-stage clustering algorithm for mining event type patterns from labeled log files that contain error messages. Detected patterns are then converted into bit vectors that are used for building decision trees with the C4.5 algorithm. Finally, decision trees are employed for detecting recurrent fault conditions from log files. Oliner, Aiken and Stearley [7] have developed an unsupervised algorithm called Nodeinfo which considers events from previous *n* days for anomaly detection. Nodeinfo divides events from each network node into hourly windows (nodehours), and calculates Shannon information entropy

This work has been supported by Estonian IT Academy (StudyITin.ee).

based anomaly score for them. If the nodehour contains log file words that have appeared only for few nodes, the nodehour will get a high anomaly score. According to experiments on supercomputer logs, Nodeinfo performs particularly well for groups of similar nodes that produce similar log messages. Makanju, Zincir-Heywood, Milios and Latzel [8] have developed the STAD framework which also uses the concept of nodehour. STAD employs information content clustering for dividing the set of nodehours into clusters, so that nodehours containing similar alert types are assigned to the same cluster. Rule-based anomaly detection is then used for finding clusters that contain nodehours with alert messages. Kimura, Watanabe, Toyono and Ishibashi [9] have proposed a supervised fault prediction method which extracts log message templates (message types) during its first step. Extracted information is used for building log feature vectors that characterize message frequency, periodicity, burstiness, and correlation with maintenance and failures. Vectors are used for training SVM with Gaussian kernel for future fault prediction. Featherstun and Fulp [10] have suggested the use of spectrum-kernel SVM for predicting disk failures on Linux platform from syslog message sequences. The method extracts facility, severity, and specific fault message substrings from each syslog message, and converts them into numerals that are provided to SVM. The method is able to predict hard disk failures one day in advance with an accuracy of 80%. Du, Li, Zheng and Srikumar [11] have proposed the DeepLog algorithm that uses LSTM neural networks for detecting anomalies in event type sequences, predicting the probability of an event type from previous types in the sequence. In addition, DeepLog implements anomaly detection for event type parameters (such as IP addresses). The algorithm also accepts human feedback about false positives for improving its future accuracy. He, Zhu, He and Lyu [12] have compared the anomaly detection performance of logistic regression, decision tree, SVM, clustering, PCA, and invariants mining, applying the methods to event log data in numerical format. During the experiments on two publicly available data sets, supervised methods were found to be superior to unsupervised algorithms.

III. Detecting Anomalous Event Log Messages

A. *Overview of the Framework*

Existing methods described in the previous section have several drawbacks. Firstly, a number of methods are supervised and rely on labeled training data sets [4, 6, 9, 10, 11] which are expensive to produce. Also, such methods need retraining if system changes introduce new log messages. Some methods from section II assume that event logs contain error messages only [3, 4], while in production environments most messages reflect normal system activity. Some methods are designed for specific fields only like disk fault prediction [10], or rely on mining message patterns from source code [5] which might not be always available. Finally, several methods do not report individual anomalous messages, but rather entire time slots that contain such messages [7, 8].

In this section, we present an unsupervised framework for detecting anomalous messages from syslog log files that addresses aforementioned shortcomings. The framework is data mining based and relies on the following assumption – in a well-maintained IT system, most log messages reflect normal system activity, while messages corresponding to system faults, security incidents, and other error conditions appear infrequently (similar assumption has been made in other research papers, e.g., [5]). Therefore, frequently occurring message patterns naturally represent a baseline of normal system activity. Our framework has been designed for logs collected with a widely used syslog protocol [13], and it is assumed that each message has the following attributes – timestamp, sender hostname, facility (type of the sender, e.g., *daemon*), severity, program name, and free-form message text. We also assume that each log file line fully represents some syslog message. Fig. 1 depicts the implementation of the proposed framework. For detecting frequent message patterns, the framework employs the LogCluster algorithm [14] and its Perl-based implementation [15]. We have selected LogCluster, since according to our recent experiments it compares favorably to publicly available implementations of other log mining algorithms [14]. LogCluster based mining module runs daily, in order to keep the database of frequent patterns up to date with changes in the surrounding environment. Patterns from the last N days and W weeks (so called *mining windows*) are then used for creating Simple Event Correlator (SEC) [16] rules that match messages reflecting normal system activity. Any message which does not match these rules is classified as anomalous. Since the mining windows are sliding, the framework is able to adapt to system changes and new log message types.

The rule mining module does not attempt to discover rules for each host separately, since this involves several challenges. Firstly, some hosts do not produce many log messages which complicates the detection of frequent message patterns. Secondly, when a new host appears in the network, it is unclear what filtering rules should be applied to it. For addressing these issues, the mining is conducted for groups of similar hosts which ensures that sufficient amount of past log data is available (the Nodeinfo algorithm uses a similar approach [7]). For the sake of simplicity, the remainder of this section assumes that all hosts belong to one group.

LogCluster is a data clustering algorithm which detects line patterns from textual log file of n lines for the user-given *support threshold* s (*relative* support threshold r means support threshold $r * n / 100$). Each log file line is split into words by user-given separator (default is whitespace). If a word appears in l lines, its *support* is defined as l and *relative* support as $l / n * 100$. LogCluster begins its work with a pass over the log file for finding *frequent words* – words with the support of at least s. During the second data pass, LogCluster splits the log file into clusters that contain at least s lines. All lines from the same cluster match the same line pattern of frequent words and wildcards. Each wildcard has the form **{m,n}* and matches at least m and at most n words ($m \leq n$). Finally, each line pattern that represents a cluster is reported to the user, for example, *sshd: Connection from *{1,1} port *{1,1}*. As discussed in [5], a detected pattern is meaningful if its words represent constant parts of the log message, while wildcards capture variable parts (e.g., IP addresses). The support of the pattern is defined as the number of lines in the cluster it represents.

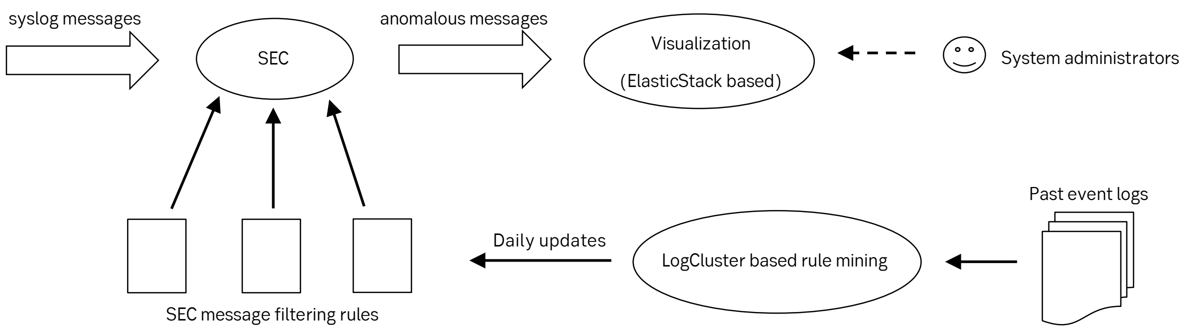

Fig. 1. A framework for detecting anomalous syslog messages.

The latest version of the LogCluster tool [15] can decouple individual phases of the algorithm which allows for parallel execution and pipelining. For example, LogCluster can be configured to detect frequent words only and store them into a *dictionary file* with absolute and relative supports. Also, frequent words can be loaded from dictionary for detecting clusters with a single data pass. Finally, clusters can be stored to *dump files*, so that repeated post-processing of detected patterns can be accomplished without re-executing the entire mining process. However, LogCluster and other support threshold based log mining algorithms have several drawbacks [14, 17] which complicate automated rule discovery. Firstly, they have a single thread of control which makes them less scalable to very large data sets. Also, with higher support thresholds *overgeneralized* patterns could be detected (e.g., **{2,5} for *{1,10}*) that can mistakenly match many anomalous messages. Furthermore, low support thresholds often lead to *overfitting*, where meaningful clusters are split into subclusters with too specific patterns (e.g., *sshd: Closing connection to 10.1.1.1*). Obviously, such patterns do not cover all messages representing normal system activity. Finally, a rare fault condition can trigger a large amount of similar messages within a short time [1], and a mining module might detect a frequent pattern from them. The rest of this section discusses how the framework addresses these challenges.

For increasing the scalability of the framework, daily rule mining is split into independent tasks which can be executed in parallel. Log file messages are first divided by syslog facility and then by program name. Fig. 2 summarizes the rule mining procedure (it assumes that days are numbered in a consecutive order). For each facility, the procedure first identifies the names of *frequent programs* that have produced at least T_{prog} messages during the previous day. If a frequent program has been frequent during at least $K * N$ days in the N day mining window, the framework executes pattern discovery procedure for the message text field of that program. Parameter K is called *daily relevance threshold* ($0 \leq K \leq 1$), and setting it to higher value prevents learning filtering rules for programs with rare but intensive logging activity. The pattern discovery procedure has been summarized in Fig. 3 and will be described in subsection IIIB. Since log messages from infrequent programs might also contain long-term patterns that reflect normal system activity, a separate pattern discovery procedure with the mining window of W weeks and *weekly relevance threshold* L ($0 \leq L \leq 1$) is executed for such messages. This procedure will be discussed in subsection IIIC.

B. Pattern Discovery from Program Message Texts

For mitigating overgeneralization and overfitting, we first attempted to find a method for selecting a single support threshold value that would generate all required meaningful patterns. We evaluated several methods, most notably head/tail breaks hierarchical clustering algorithm [18]. The algorithm iteratively divides data points with heavy-tailed distribution into head and tail by average or mean. We used variants of this algorithm for clustering words by their occurrence time (word occurrence times are known to have a heavy-tailed distribution [19]). Our aim was to identify a support threshold for capturing proper amount of frequent words for creating suitable patterns, but experiments did not yield acceptable results. Thus, the framework selects a number of support thresholds heuristically, starting from a higher value which is iteratively lowered until a stop condition evaluates true. In the framework implementation, the following relative support thresholds have been used: $s_1 = 5$, $s_2 = 2.5$, $s_3 = 1$, $s_i = s_{i-3} / 10$ for $i > 3$, so that $s_i * n / 100 > 100$ (n is the number of lines in input data set).

```
find_rules(N, K, W, L, $T_{prog}$, $T_{weight}$, $T_{weak}$, $T_{prec}$)

D := # of the current day
for each f in { facilities } do
  $A_{f,D-1}$ := { names of programs that produced at least
             $T_{prog}$ messages for facility f at day D-1 }
  $O_{f,D-1}$ := { log messages for facility f at day D-1 that
             did not originate from programs in $A_{f,D-1}$ }
  for each P in $A_{f,D-1}$ do
    n := |{ d | D-N ≤ d ≤ D-1, P ∈ $A_{f,d}$ }|
    if n ≥ K * N then
      $F_{f,P}$ := discover_patterns(P, N, K, $T_{weight}$, $T_{weak}$, $T_{prec}$)
      build filtering rules for facility f and program P,
        using patterns from $F_{f,P}$
    fi
  do
  $I_f$ := discover long term patterns from log messages in
         $O_{f,D-1}$ ∪ … ∪ $O_{f,D-7*W}$ with relevance threshold L
  build filtering rules for facility f,
    using patterns from $I_f$
do
```

Fig. 2. Rule mining procedure.

If S denotes the set of selected support thresholds in relative notation (i.e., $\forall s \in S$, $0 < s \leq 100$) and *min* is the smallest threshold from S, the set of frequent words for *min* contains all frequent words for higher thresholds. Also, we have found that frequent words appearing during few days only are related to overfitting or bursts of anomalous messages. These observations have motivated the following mining procedure. First, LogCluster is used for creating a dictionary file of frequent words for support threshold *min*. After that, dictionary is used for mining patterns in fast single-pass mode with support thresholds from S. For each threshold $s \in S$, frequent word is selected from dictionary if it appears in dictionaries of at least $K * N$ days in the mining window, having support of at least s (in Fig. 3, $s_{d,w}$ denotes relative support of word w at day d). For easing further post-processing, patterns detected for each $s \in S$ are stored into a dump file (in Fig. 3, $B_{d,s}$ denotes patterns for support threshold s and day d).

In the following, we describe the algorithm that selects patterns for building message classification rules, with F denoting the set of selected patterns. The algorithm begins with initializing F to all patterns from sets $B_{d,s}$ in the mining window. For mitigating overfitting, LogCluster supports a pattern joining heuristic which merges too specific patterns based on word weight [14]. The word weight falls to interval *(0..1]* for each word in the pattern, reflecting how strongly the word is associated with other words in this pattern. If a word weight remains below the word weight threshold, it is replaced with a wildcard and similar patterns are merged. For example, if IP addresses are weakly associated with other words in patterns *Closing connection to 10.1.1.1* and *Closing connection to 10.1.1.2*, merging produces the pattern *Closing connection to *{1,1}*. According to experiments, word weight thresholds *0.5...0.8* produce best results for mitigating overfitting [14]. The pattern joining heuristic is applied to each $B_{d,s}$ with word weight threshold T_{weight}, and resulting patterns $R_{d,s}$ are joined to the set of selected patterns F (original patterns are not discarded at this point, since heuristic might accidentally create too generic patterns which will be pruned at further steps).

For excluding overgeneralized patterns from F, the framework employs several techniques. First, according to our observation, the pattern has a high likelihood of being too generic if it contains one word and is detected only for one support threshold during daily pattern mining. More formally, we call the pattern a *weak pattern* at day d if it was detected only for one support threshold from log messages of day d. If pattern x has been detected during n days in the mining window and x has been weak during m days ($m \leq n \leq N$), x is excluded from F if it has one word and $m / n \geq T_{weak}$.

For measuring the degree of generality of a pattern from F, the framework calculates its *precision*, with an overgeneralized pattern receiving a low precision score. If pattern x consists of k elements (words or wildcards), precision of ith element $prec(x_i)$ is defined as follows: $prec(x_i) = 1$ if x_i is a word; $prec(x_i) = m / n$ if x_i is a wildcard $*\{m,n\}$. For finding the precision of pattern x, we have used the following functions: $prec_j(x) = \sum_{i=1}^{k} prec(x_i) / l_j$, where $j=1,2,3$. Parameters l_1 and l_2 denote the maximum and minimum number of words pattern x can match, respectively, while l_3 is defined as $max(l_2, k)$. Since $\sum_{i=1}^{k} prec(x_i) \leq l_j$ for $j=1,2,3$, then $0 < prec_j(x) \leq 1$. For example, suppose pattern x is **{1,6} error *{0,4}*. Then $k = 3$, $prec(x_1) = 1/6$, $prec(x_2) = 1$ and $prec(x_3) = 0$. Also, $l_1 = 11$, $l_2 = 2$ and $l_3 = 3$, and therefore $prec_1(x) \approx 0.11$, $prec_2(x) \approx 0.58$ and $prec_3(x) \approx 0.39$. In other words, any wildcard not matching exactly one word lowers the precision score, with a wildcard matching word sequences with a wide variety of lengths having greater impact (such wildcards are called *generic wildcards*). Therefore, lower precision indicates that pattern has a generic nature, and the framework excludes a pattern from F if its precision is smaller than T_{prec}. The $prec_3()$ function has been used for measuring precision in framework implementation, since unlike $prec_1()$, it does not penalize patterns with many words and few generic wildcards, and unlike $prec_2()$, it does not favor patterns with generic wildcards $*\{0,n\}$.

```
discover_patterns(P, N, K, $T_{weight}$, $T_{weak}$, $T_{prec}$)

D := # of the current day
S := { support thresholds for program P at day D-1 }
W := { frequent words for program P and
       support threshold min(S) at day D-1 }

for each s in S do
  V := ∅
  for each w in W do
    n := |{d | D-N ≤ d ≤ D-1, $s_{d,w}$ ≥ s}|
    if n ≥ K * N then V := V ∪ { w } fi
  done
  $B_{D-1,s}$ := { message text patterns mined with LogCluster
             for program P at day D-1 with support
             threshold s and frequent words from V }
done
F := ∅
for each d in (D-N,…,D-1) do
  S := { support thresholds for program P at day d }
  $C_d$ := ∅
  for each s in S do
    $R_{d,s}$ := { merged patterns for $B_{d,s}$ with word
              weight threshold set to $T_{weight}$ }
    F := F ∪ $B_{d,s}$ ∪ $R_{d,s}$
    $C_d$ := $C_d$ ∪ $B_{d,s}$ ∪ $R_{d,s}$
  done
done
E := ∅
for each x in F do
  n := |{d | D-N ≤ d ≤ D-1, x ∈ $C_d$}|
  m := |{d | D-N ≤ d ≤ D-1, x ∈ $C_d$, x is weak at day d}|
  if (m / n ≥ $T_{weak}$ AND x has only one word)
    then E := E ∪ { x } fi
done
F := F \ E; E := ∅
for each x in F do
  if precision(x) < $T_{prec}$ then E := E ∪ { x } fi
done
F := F \ E; E := ∅
for each x in F do
  n := |{d | D-N ≤ d ≤ D-1, x ∈ $C_d$}|
  if n < K * N then E := E ∪ { x } fi
done
F := F \ E; E := ∅
for each x in F do
  if (∃y ∈ F, x ≠ y, x ~ y) then E := E ∪ { x } fi
done
return F \ E
```

Fig. 3. Pattern discovery procedure for message text field of a program.

After that, the pattern selection algorithm excludes all patterns from F that have been detected for less than $K * N$ days in the mining window. This step will ensure that only repeatedly occurring frequent patterns will be kept in *F*, and learning filtering rules from accidental bursts of anomalous messages is avoided. Also, if pattern *y* matches all events that are matched by pattern *x*, we say that *y* is more general than *x* and denote it as $x \sim y$. During its final step, pattern selection procedure excludes all patterns from *F* that have more general patterns in *F*, since excluded patterns are redundant for deriving filtering rules.

```
# A rule for suppressing messages for successful
# SSH logins

type=Suppress
ptype=RegExp
pattern=sshd(?:\[\d+\])?: Accepted(?:\s+)(?:(?:\S+)
(?:\s+)){1,1}for(?:\s+)(?:(?:\S+)(?:\s+)){1,1}from
(?:\s+)(?:(?:\S+)(?:\s+)){1,1}port(?:\s+)(?:(?:\S+)
(?:(?:\s+)|(?:\s+)?$)){1,1}(?:\s+)?$
desc=Accepted *{1,1} for *{1,1} from *{1,1} port *{1,1}

# A rule for suppressing SNMP daemon messages for
# incoming queries

type=Suppress
ptype=RegExp
pattern=snmpd(?:\[\d+\])?: Connection(?:\s+)from(?:\s+)
UDP\:(?:\s+)(?:(?:\S+)(?:(?:\s+)|(?:\s+)?$)){1,1}
(?:\s+)?$
desc=Connection from UDP: *{1,1}

# A rule for suppressing anacron messages for upcoming
# job executions

type=Suppress
ptype=RegExp
pattern=anacron(?:\[\d+\])?: Will(?:\s+)run(?:\s+)job
(?:\s+)(?:(?:\S+)(?:\s+)){1,1}in(?:\s+)(?:(?:\S+)
(?:\s+)){1,1}min\.(?:\s+)?$
desc=Will run job *{1,1} in *{1,1} min.
```

Fig. 4. Sample automatically created SEC rules for matching messages that reflect normal system activity.

C. *Building Filtering Rules*

Discovery of long-term frequent message patterns for infrequent programs is conducted similarly to the algorithm in Fig. 3, except that the mining window size is *W* weeks and the window is divided into weekly slots. For example, if $W = 4$ and weekly relevance threshold $L = 0.75$, patterns are mined from the logs of last 4 weeks, and patterns detected for less than 3 weekly slots are dropped. Also, LogCluster is executed for messages of *all* infrequent programs of a given syslog facility, and patterns are mined from the concatenation of program name and message text fields. Finally, an additional heuristic is used for excluding overgeneralized patterns – if the pattern has a wildcard for the program name or entire message text (e.g., *sshd: *{1,20}* or **{1,1}: daemon stopped*), the pattern will be excluded from further consideration.

After discovering patterns for frequent and infrequent programs (sets $F_{f,P}$ and I_f from Fig. 2, respectively), the rule mining module derives regular expressions from detected patterns and uses them for creating SEC *Suppress* rules. Fig. 4 depicts some automatically created rules for *sshd*, *snmpd*, and *anacron* that originate from our framework implementation. For efficient processing of incoming messages, the SEC rule base is arranged hierarchically [16] by facility and program name. The rule base also contains dedicated rule files for custom message processing rules written by human experts.

IV. Framework Implementation and Performance

For measuring the performance of the framework, we have evaluated its implementation during 3 months (92 days) in a large EU organization. The framework was running with the following parameters: $N = 10$, $K = 0.5$, $W = 4$, $L = 0.75$, $T_{prog} = 1000$, $T_{weight} = T_{weak} = T_{prec} = 0.5$. During evaluation, the framework processed OS level syslog messages for *auth*, *authpriv*, *cron*, *daemon*, *kern*, and *mail* facilities from 543 Linux servers with standardized OS configuration. Since the servers generated similar log messages for aforementioned six syslog facilities, we used one host group for them. Altogether, 296,699,550 messages were processed by the framework, with 1,879,209 messages (≈0.63%) passing SEC filtering rules and classified as anomalous. During 92 days, 483-551 SEC rules were automatically created by daily rule mining procedure (412-469 rules were created for frequent programs). Table I depicts message classification data for different facilities.

TABLE I. Message Classification by Syslog Facility.

Facility	# of all messages	# of anomalous messages	# of servers generating anomalous messages
auth	3,672,497	34,904	224
authpriv	59,667,935	234,645	465
cron	22,249,354	9,000	200
daemon	198,129,740	494,545	508
kern	8,466,606	907,186	342
mail	4,513,418	198,929	153

Many anomalous messages represented previously unknown fault conditions (e.g., disk issues) that would have remained unnoticed in organizational monitoring system. As can be seen from Table I, unusually large fraction (10.7%) of kernel messages with *kern* facility were classified as anomalous, and they constituted almost half (48.3%) of all anomalous messages. 54.2% of kernel messages were various SELinux warnings, and 22.6% were generated during system reboots (most reboots were part of regular system maintenance, although reboots can also be triggered by system crashes, e.g., executed by hypervisors after virtualization system failures). From remaining kernel messages, most represented serious system errors like file system corruption and out-of-memory conditions. Majority of these error conditions triggered hundreds or thousands messages – for example, during one system crash 115,197 messages were logged (12.7% of all anomalous kernel messages). We also observed similar error message bursts for other facilities – for example, 75.8% of anomalous messages for *mail* facility were triggered by file-system-full condition on a single server. Anomalous messages for *daemon* facility represented various fault conditions or configuration errors, like temporary network outages or connectivity issues with remote services, failures to restart a daemon due to a syntax error in the configuration file, etc. Most anomalous messages for *auth* and *authpriv* facilities reflected authentication errors, while for *cron* facility execution errors for scheduled jobs were reported. We also discovered that some anomalous messages represented rare but normal system activity. Such activities included creation of new user accounts (*authpriv* facility), modification of scheduled jobs (*cron* facility), and system service restarts (*daemon* facility).

For evaluating the classification precision and recall of the framework, we reviewed classification results for log messages that were triggered by vulnerability scanning of the internal network during four non-contiguous days. 1454 log messages were triggered on 231 servers, with 779 messages representing normal system activity (e.g., incoming connection to an SSH server) and 675 messages reflecting malicious actions (e.g., SSH password probing for non-existing user accounts). The framework classified 683 messages as anomalous, while 656 of them were malicious messages, yielding the precision of 96.0% and recall of 97.1%. During another 10 hour experiment, we used the Bbuzz protocol fuzzing framework [20] for pen-testing Linux kernel and UDP services of a test server that was connected to the framework. Protocol fuzzing triggered 46,370 and 7,957 error messages from *snmpd* and kernel, respectively, and all messages were classified as anomalous.

After detection, the framework sends anomalous messages to ElasticStack for searching and visualization purposes. Although ElasticStack provides system administrators with an efficient interface for investigating anomalies and fault conditions, it does not allow for distinguishing critical errors (e.g., a system crash) from events of lower priority (e.g., an accidental login failure). As discussed before, critical faults often trigger a large number of anomalous messages within a short time frame. For identifying such faults in a timely fashion, we have added SEC post-processing rules into the framework for EWMA based anomaly detection. With this approach, moving average μ_t of a time series $\{X_1, X_2, ...\}$ is calculated as $\mu_1 = X_1$, and $\mu_t = \alpha * X_t + (1 - \alpha) * \mu_{t-1}$ for $t > 1$. If the value of α is close to 1, only recent values of a time series influence μ_t, while values close to 0 distribute weight more evenly. In our setup, the stream of anomalous messages is divided into 5 minute time slots for each *(host, facility)* tuple, and moving average μ_t and standard deviation σ_t are calculated for the number of messages (i.e., X_t is the number of messages in time slot t). The framework raises an alarm for time slot t and *(host, facility)* tuple if $|X_t - \mu_t| > m * \sigma_t$ and $X_t \geq n$ (i.e., a burst of at least n anomalous messages has been observed during the last 5 minutes, where the number of messages is more than m standard deviations away from average). For post-processing anomalous messages, we have used the following settings: $\alpha = 0.05$, $m = 3$ and $n = 100$. During 92 day experiment, 214 alarms were generated, with more than half of them associated with system reboots. Alarms were also triggered by out-of-memory and file-system-full conditions, file system corruption events, and other major system faults.

V. Future Work

As for future work, we plan to augment our framework with additional time series analysis methods. Our other research goals include further development of the pattern selection algorithm for automated classification, and studying methods for assigning anomaly scores to individual anomalous messages.

Acknowledgment

The authors express their gratitude to Dr. Rain Ottis and Prof. Olaf Maennel for supporting this work.

References

[1] Stephen E. Hansen and E. Todd Atkins, "Automated System Monitoring and Notification With Swatch," in Proceedings of 1993 USENIX Large Installation System Administration Conference, pp. 145-152.

[2] http://logsurfer.sourceforge.net

[3] Kenji Yamanishi and Yuko Maruyama, "Dynamic Syslog Mining for Network Failure Monitoring," in Proceedings of 2005 ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp. 499-508.

[4] Felix Salfner, "Event-based Failure Prediction: An Extended Hidden Markov Model Approach," PhD Thesis, Humboldt-Universität zu Berlin, 2008.

[5] Wei Xu, Ling Huang, Armando Fox, David Patterson and Michael I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in Proceedings of 2010 International Conference on Machine Learning, pp. 37-46.

[6] Thomas Reidemeister, Miao Jiang and Paul A.S. Ward, "Mining Unstructured Log Files for Recurrent Fault Diagnosis," in Proceedings of 2011 IEEE/IFIP International Symposium on Integrated Network Management, pp. 377-384.

[7] Adam Oliner, Alex Aiken and Jon Stearley, "Alert Detection in System Logs," in Proceedings of 2008 IEEE International Conference on Data Mining, pp. 959-964.

[8] Adetokunbo Makanju, A. Nur Zincir-Heywood, Evangelos E. Milios and Markus Latzel, "Spatio-Temporal Decomposition, Clustering and Identification for Alert Detection in System Logs," in Proceedings of 2012 ACM Symposium on Applied Computing, pp. 621-628.

[9] Tatsuaki Kimura, Akio Watanabe, Tsuyoshi Toyono and Keisuke Ishibashi, "Proactive Failure Detection Learning Generation Patterns of Large-scale Network Logs," in Proceedings of 2015 International Conference on Network and Service Management, pp. 8-14.

[10] R. Wesley Featherstun and Errin W. Fulp, "Using Syslog Message Sequences for Predicting Disk Failures," in Proceedings of 2010 USENIX Large Installation System Administration Conference.

[11] Min Du, Feifei Li, Guineng Zheng and Vivek Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in Proceedings of 2017 ACM Conference on Computer and Communications Security, pp. 1285-1298.

[12] Shilin He, Jieming Zhu, Pinjia He and Michael R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection," in Proceedings of 2016 IEEE International Symposium on Software Reliability Engineering, pp. 207-218.

[13] C. Lonvick, "The BSD syslog Protocol," RFC 3164, 2001.

[14] Risto Vaarandi and Mauno Pihelgas, "LogCluster – A Data Clustering and Pattern Mining Algorithm for Event Logs," in Proceedings of 2015 International Conference on Network and Service Management, pp. 1-7.

[15] https://ristov.github.io/logcluster/

[16] Risto Vaarandi, Bernhards Blumbergs and Emin Çalışkan, "Simple Event Correlator – Best Practices for Creating Scalable Configurations," in Proceedings of 2015 IEEE CogSIMA Conference, pp. 96-100.

[17] Adetokunbo Makanju, "Exploring Event Log Analysis With Minimum Apriori Information," PhD Thesis, University of Dalhousie, 2012.

[18] Bin Jiang, "Head/tail Breaks: A New Classification Scheme for Data with a Heavy-tailed Distribution," The Professional Geographer, 65(3), pp. 482-494.

[19] Risto Vaarandi, "A Data Clustering Algorithm for Mining Patterns From Event Logs," in Proceedings of 2003 IEEE Workshop on IP Operations and Management, pp. 119-126.

[20] Bernhards Blumbergs and Risto Vaarandi, "Bbuzz: A Bit-aware Fuzzing Framework for Network Protocol Systematic Reverse Engineering and Analysis," in Proceedings of 2017 IEEE Military Communications Conference, pp. 707-712.