Boot Log Anomaly Detection with K-Seen-Before

Johan Garcia
Department of Mathematics and Computer Science
Karlstad University, Karlstad, Sweden
Email: johan.garcia@kau.se

Tobias Vehkajarvi
Department of Mathematics and Computer Science
Karlstad University, Karlstad, Sweden
Email: tobias.vehkajarvi@kau.se

***Abstract*—Software development for embedded systems, in particular code which interacts with boot-up procedures, can pose considerable challenges. In this work we propose the K-Seen-Before (KSB) approach to detect and highlight anomalous boot log messages, thus relieving developers from repeatedly having to manually examine boot log files of 1000+ lines. We describe the KSB instance based anomaly detection system and its relation to KNN. An industrial data set related to development of high-speed networking equipment is utilized to examine the effects of the KSB parameters on the amount of detected anomalies. The obtained results highlight the utility of KSB and provide indications of suitable KSB parameter settings for obtaining an appropriate trade-off for the cognitive workload of the developer with regards to log file analysis.**

I. Introduction and related work

Development of embedded software is often challenging as it may require more explicit knowledge of the underlying hardware aspects of the target platform. In particular, developing software that interacts directly with the hardware during the system boot up phase provides additional obstacles since debugging is not as easily performed in such settings. Development of specialized network equipment that needs to handle very high link rates is one use case where boot log messages can be crucial during the software development process. Other application areas where boot logs can be of considerable interest are IoT software development, as well as various containerization solutions. During development, it is cumbersome and error-prone for developers to manually inspect 1000+ lines of boot log messages for signs of unexpected or erroneous behavior. Thus, we here examine the use of anomaly detection as a tool to aid in the examination of boot log files.

Anomaly detection is a well studied area with applicability in a range of application domains [1]. There exist a range of different anomaly detection approaches, which vary in the degree to which they are generic or tailored to a particular application area. In this work we propose and study the K-Seen-Before (KSB) anomaly detection approach which utilizes instance-based learning. Instance based learning approaches do not construct an explicit model, instead they retain all or a subset of the instances in the training set. For each new observation to be classified they compute some distance metric between the new observation and the stored training instances. A decision function is applied to the computed metric to decide whether or not the new observation is an anomaly. As a contrast, model based anomaly detection algorithms such as one class SVM [2] or isolation forest [3] uses the training data to construct a model of the pertinent characteristics of the training data, and then checks a new observation against the model to decide if it is an anomaly or not.

While considerable work has been performed on examining operational logs to infer knowledge about system operation [4], [5] and security aspects [6], very limited work is available on boot log anomaly analysis and its ability to provide insights to developers. Here, we present KSB and explore the anomaly detection characteristics over 753 distinct boot log files with over one million lines while varying the threshold settings of the different KSB components. We also consider the UI aspects of providing anomaly information to the developer.

There are a wide variety of anomaly detection approaches as discussed in surveys [1], [7]. The KNN anomaly detection approach [8] is one common approach which computes the distance to the Kth nearest neighbor and uses a decision function to choose an appropriate threshold for classifying an observation as an anomaly. The KNNW approach [9] is similar, but instead considers the sum of distances to all K nearest neighbors. Several more recent variations on the KNN approach has been proposed, for example K-point nearest neighbor graph approaches [10]. An evaluation of the performance a wide variety of anomaly detection methods in [11] however find that the seminal approaches kNN, kNNW, and also the Local Outlier Factor [12], remain state of the art as no more recent methods were found to offer any comprehensive improvement over these methods. For the case of KSB, a subset of its mechanisms can be seen as an ensemble of per-dimension kNNW with the maximum distance set to zero.

Earlier work on log analysis is often centered around log analysis and correlation in distributed environments such as supercomputers / High performance computing [4], [13]. The problem of handling a mixture of text and numerical data as is done by KSB has also been explored in [14] for the problem of anomaly detection in syslog data from a supercomputer, which is not directly related to boot log files. Handling log messages that are split into multiple lines can be challenging, as is discussed in [5]. However, in that work there are explicit line numbers to guide the reconstruction into a single line. The importance on tokenization, as done in KSB is highlighted in [15]. A notable difference to all the above work is that the present work considers finite length boot logs, rather than the time continuous operational logs considered above.

978-1-7281-7303-0/20/$31.00 ©2020 IEEE
DOI 10.1109/COMPSAC48688.2020.0-140

```
1  SYSLOG: <174>Jan 23 08:03:51 pl2 PLOS-0: plosloader: Copying 50265444 bytes of code
2  SERIAL: SMP-loader[0]: [0] F
3  SYSLOG: <174>Jan 23 08:03:51 pl2 PLOS-0: plosloader: Sending start signal
4  SERIAL: ound cpu with APIC ID: 2
5  SYSLOG: <158>Jan 23 08:03:51 pl2 pld: [Packetlogicd:Info] - Init - Lib
6  SYSLOG: <157>Jan 23 08:03:51 pl2 pld: [Memory:Notice] CACHE 'List Cache': init: numitems: 0, itemsize: 32
```

Figure 1: Boot log excerpt

```
SYSLOG: <[num#P0]>[ts] pl[num#P1] PLOS-[num#P2]: plosloader: Copying [num#P3] bytes of code
SERIAL: SMP-loader[[num#P0]]: [[num#P1]] Found cpu with APIC ID: [num#P2]
SYSLOG: <[num#P0]>[ts] pl[num#P1] PLOS-[num#P2]: plosloader: Sending start signal
SYSLOG: <[num#P0]>[ts] pl[num#P1] pld: [Packetlogicd:Info] - Init - Lib
SYSLOG: <[num#P0]>[ts] pl[num#P1] pld: [Memory:Notice] CACHE 'List Cache': init: numitems: [num#P2], ...
```

Figure 2: Templates corresponding to Figure 1.

II. Boot log anomaly detection

Anomaly detection can be subdivided into supervised and unsupervised methods. In essence, a supervised method has access to ground truth labels which indicate which observations are anomalies and which are not. This ground truth data is then used both to create an appropriate model, and to set an appropriate decision threshold to maximize some metric of interest, often using suitable cross-validation approaches on the training data set. It can be noted that supervised anomaly detection can be viewed as regular supervised binary classification task, albeit on a highly imbalanced data set.

In the use case considered here, developer support in the context of embedded system development, there are however no ground truth labels available. It thus becomes imperative to learn the behavior of the anomaly detection algorithm in relation to the decision function so that the system can be configured to best suit the detection task at hand. In this work we thus explore the effect of varying the settings of the anomaly detection algorithm in order to determine what a suitable configuration can be for the considered use case.

A. Boot log file structure and normalization

This work concerns the detection of anomalies in boot log files, and an example excerpt of one such file is shown in Figure 1. A notable aspect of these log lines is the presence of time stamps in most log lines, which makes it impossible to consider a simple diff between different log files. In many log lines there are also numerical values that may naturally change to a larger or smaller degree between different boot ups of the system. The mixture of text and numerical data is one complicating factor for an anomaly detection system as purely text based methods do not capture the numerical variation well, and many generic anomaly detection approaches would require a very large number of dimensions to capture the textual aspects.

As a preparation for further KSB processing, normalization processing is performed on the log files. Due to the asynchronous operation of the different logging processes it is possible for logging messages to become divided into multiple lines. This is shown in lines 2 and 4 in Figure 1. A line merging procedure is applied to construct a single line before further processing takes place. The line merging utilizes the difference in line endings i.e `\n` versus `\r\n` that exists for splitted and non-splitted lines in the source data set.

Type	Regular expression
Timestamp	[A-Za-z]{3}\s*[0-9]+\s*[0-9]*\s+[0-9]+:[0-9]+:[0-9]+
IPv4	[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
Port	:[0-9]+
Hexdec	0x[0-9a-fA-F]+|[0-9a-fA-F]{8,16}
Numeric	[0-9]+
MAC-address	([0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2})

Table I: Regular expressions used to decompose log lines

Further, all time stamp information in the log lines is replaced by relative time only, as the actual point in time when the system booted is deemed insignificant. The log line with the earliest given time is used as the zero reference for the relative time calculations.

B. Templates and parameters

The K-Seen-Before detection approach is based on learning the occurrence numbers of particular instances of various types of constructs that make up the log file. An initial step of the approach is thus to subdivide each log line into the different constructs that it is composed of. For the various constructs, different threshold values can then be applied during the anomaly detection phase. The main constructs of interest are the timestamp, the line template, and the collection of parameter values. These constructs are separated out with the help of regular expressions. The utilized regular expressions are shown in Table I. The resulting decomposition of the example lines from Figure 1 is shown in Figure 2. Here it can be observed that the timestamp and the numerical values have been replaced by tokens. The resulting line is now a *template* which describes the generic appearance of a particular log line without consideration for aspects that may change between different boot up sessions.

Algorithm 1 KSB instance learning algorithm

```
Require: set of log files $\mathbf{F}$
 1: $k \leftarrow 0$;  $\mathbf{T} \leftarrow [\ ]$; $t^T, c^T, c^P, u^P \leftarrow \{\}, \{\}, \{\}, \{\}$
 2: for file $F_i$ in $\mathbf{F}$ do
 3:     joinSplittedLines($F_i$)
 4:     for line $L_j$ in $F_i$ do
 5:         $\widehat{T}, t, \widehat{\mathbf{P}} \leftarrow$ extractTemplateParams($L_j$)
 6:         $t^T[\widehat{T}] \leftarrow$ updateRunningAverage($t^T[\widehat{T}], t$)
 7:         if not $\widehat{T}$ in $\mathbf{T}$ then
 8:             $k \leftarrow k + 1$
 9:             $\mathbf{T}_k \leftarrow \widehat{T}$
10:         end if
11:         $c^T[\widehat{T}] \leftarrow c^T[\widehat{T}] + 1$
12:         for parameter value $P_l$ in $\widehat{\mathbf{P}}$ do
13:             $c^P[getk(\widehat{T}), l, P_l] \leftarrow c^P[getk(\widehat{T}), l, P_l] + 1$
14:         end for
15:     end for
16: end for
17: for all $P_{kl}$ do
18:     $u^P[P_{kl}] \leftarrow |c^P[k, l, *]| \div c^T[\mathbf{T}_k]$
19: end for
```

C. K-Seen-Before training phase

We now turn to the description of the KSB algorithm. More formally we consider the task of detecting anomalous entries based on a training set of n boot log files. Each log file $F_i | i = 1...n$ has some number of lines m_i. Each F_i consists of log lines $L_{ij} | j = 1...m_i$. Each line can be represented as a line template T_k where k goes from 1 to the number of unique templates present in the data set. Each template T_k may further have numerical parameters P_{kl} present where l indexes the parameter order of template T_k. Within each log line there may also be a timestamp t. A log file can thus be represented as a sequence of log line descriptors, i.e. $F_i = ((T, t, (P_{..}, P_{..}, ...)), (T, t, (P_{..}, P_{..}, ...)), ...$

The flow of KSB is shown in Algorithm 1. KSB takes a set of training log files $\mathbf{F}$ as input, and begins by initializing the template counter k, the template list $\mathbf{T}$, the timestamp per template dictionary t^T, the template and parameter count dictionaries c^T, c^P and the parameter value uniqueness fraction dictionary u^P (line 1). Each file F_i is then processed in turn. The per file processing consists of joining splitted lines (line 3) and extracting a template $\widehat{T}$, timestamp t and parameter list $\widehat{\mathbf{P}}$ for each log line in the file (line 5). Then updates are performed to the average template timestamp (line 6), the template list and occurrence count dictionary (lines 7-11). Each parameter on the line is then processed to update the parameter occurrence value (lines 12-14), and finally the parameter uniqueness fraction is computed over all observed parameter values (lines 17-19).

D. K-Seen-Before anomaly detection phase

The algorithm for the KSB anomaly detection phase is shown in Algorithm 2. Here, the required inputs are the log file

Algorithm 2 KSB anomaly detection algorithm

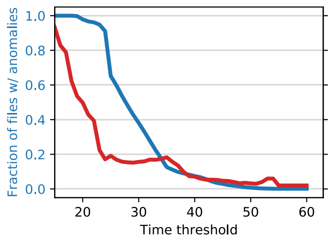
```
Require: $\widehat{F}, \mathbf{T}, t^T, c^T, c^P, u^P, t_{thresh}, T^c_{thresh}, P^c_{thresh}, P^u_{thresh}$
 1: joinSplittedLines($\widehat{F}$)
 2: for line $L_j$ in $\widehat{F}$ do
 3:     $\widehat{T}, t, \widehat{\mathbf{P}} \leftarrow$ extractTemplateParams($L_j$)
 4:     if $t > t^T[\widehat{T}] + t_{thresh}$ or $t < t^T[\widehat{T}] - t_{thresh}$ then
 5:         $reportTimeAnomly(\widehat{T}, t, t^T[\widehat{T}], t_{thresh})$
 6:     end if
 7:     if $c^T[\widehat{T}] < T^c_{thresh}$ then
 8:         $reportTemplateAnomaly(\widehat{T}, c^T[\widehat{T}], T^c_{thresh})$
 9:     end if
10:     for parameter value $P_l$ in $\widehat{\mathbf{P}}$ do
11:         if $u^P[P_{kl}] < P^u_{thresh}$ then
12:             if $c^P[P_{getk(\widehat{T}),l}] < P^c_{thresh}$ then
13:                 $repParamAnom(\widehat{T}, c^P[P_{getk(\widehat{T}),l}], P^c_{thresh})$
14:             end if
15:         end if
16:     end for
17: end for
```

$\widehat{F}$ that is to be analyzed, the data structures created during the training phase (i.e. $\mathbf{T}, t^T, c^T, c^P, u^P$) and a set of thresholding settings. Those settings are the relative timestamp deviation threshold t_{thresh}, the template count threshold T^c_{thresh}, the parameter count threshold P^c_{thresh}, and the parameter uniqueness threshold P^u_{thresh}. The anomaly detection phase starts by the necessary joining of any splitted lines, and then examines each line in turn. The per line processing starts identically as the learning phase with the extraction of a template $\widehat{T}$, timestamp t and parameter list $\widehat{\mathbf{P}}$ (line 3). The algorithm then proceeds to check for time anomalies (lines 4-6), template anomalies (lines 7-9) and parameter anomalies (lines 10-16). In each anomaly detection step there is one or more threshold settings that control the sensitivity of the algorithm. For the template count and parameter count settings the threshold value corresponds directly to the number of zero distance neighbors that needs to be present in the training data set.

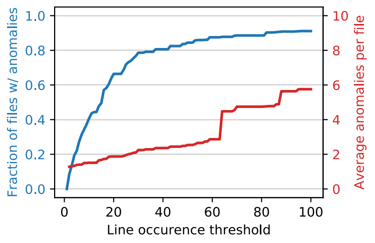
III. Examination of anomaly detection behavior

When using an anomaly detection system there is an inherent trade-off between false positives and false negatives. A false positive is an observation which the system reports to be anomalous while in fact it is normal/uninteresting. A false negative is an observation that is anomalous but the system fails to identify it as such. The KSB approach has several threshold settings which can be used to control the trade-off between false positives and false negatives.

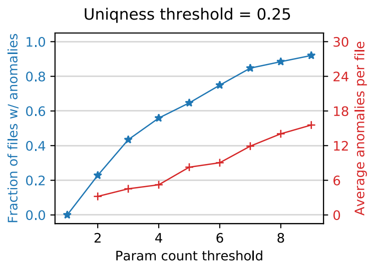
This examination is performed on a data set obtained from a company involved with embedded software development for specialized high-speed network equipment. During the development process boot log files are automatically captured and stored in a repository, and the results here are from 753 such files. As this examination is done on a data set without ground truth labels no exact performance metrics can be obtained for



(a) over t_{thresh} time interval threshold



(b) over T^c_{thresh} template count threshold



(c) over P^c_{thresh} parameter count threshold

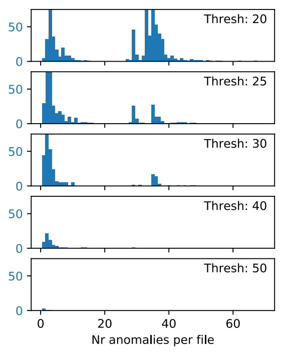
Figure 3: Fraction of log files with anomalies (blue), and the average number of anomalies (for files with anomalies, red)

the anomaly detection system. The examination here instead focuses on examining and characterizing the trade-off between threshold settings and the number of generated anomalies from the perspectives of 1) the fraction of examined log files which has at least one anomaly reported for them, and 2) the average number of anomalies present in files with a non-zero anomaly count. These metrics couple to the cognitive workload of the developer, that is 1) the probability that a log file will need to be looked at, and 2) the average number of anomalous log lines that needs to be checked within an anomalous log file. The examination is done by varying the threshold settings of the anomaly detection phase as discussed in Section II-D, and provides vital information to a practitioner who wishes to find appropriate threshold setting for their particular use case. We note that each deployment of KSB needs may need to be tuned in accordance with the specific source of boot log files, and the relative tolerance of false positives versus false negatives of the organization. To perform the examination a leave-one-out cross validation approach is used. It can be noted that in order to speed up the examination the training phase is not redone in each cross-validation fold, instead the occurrence threshold values are simply increased by one to obtain the same effect.

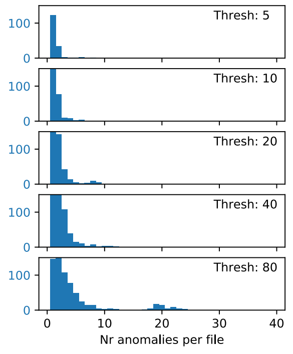
A. *Log line relative time anomalies*

Particular boot log messages often occur within some time interval in relation to the initial startup of the system. As many of the log lines have the current time included in them, this can be used used to compute timestamps relative to the earliest observed timed log line. Time-based anomaly detection uses a configurable threshold in order to determine if the relative time of a particular log line is within the normal range, defined as an interval around the mean time of all observed relative times of that log line template, or if the timing should be considered as anomalous. The value of this threshold influences how many anomalies that will be detected, and it is thus of interest to explore the detection behavior when this setting is varied.

The results when varying the time threshold is shown in Figure 3a. From the results it can be inferred that there appears to be a fairly wide range of timestamps observed in the underlying data, at least for a subset of the log lines. We note that the right-hand side red axis shows the average number of anomalies per file *for files which have anomalies*. Thus,



(a) time interval threshold



(b) template count threshold

Figure 4: Distribution of number of anomalies per log file for different values of the T^c_{thresh} template count threshold.

as the number of files with anomalies decrease as the time threshold increases, the average number of anomalies in the remaining anomalous files may increase. This effect can be observed in Figure 3a at around the 52 seconds threshold, and these observed results are not counter-intuitive as might be presumed at first glance.

In addition to the average number of anomalies per log file, the distribution of the number of anomalies per log file can also be relevant and is shown in Figure 4a. Clearly, the average which is shown in Figure 3a is for threshold values of 20 to 30 coming from a bimodal underlying distribution. This implies that a subset of log files will require the check of a large number of anomalies, and another subset a small number, for the same configuration setting.

B. *Template count anomalies*

Template count anomalies occur when an examined log file contains log lines for which their template has occurred fewer than a threshold value number of times in the training data. As this threshold value increases, a larger fraction of the relatively rare log lines are considered to be anomalous, and the fraction of log files with any anomaly trend towards one as seen in Figure 3b. However, even for large template count threshold values the average number of detected anomalies

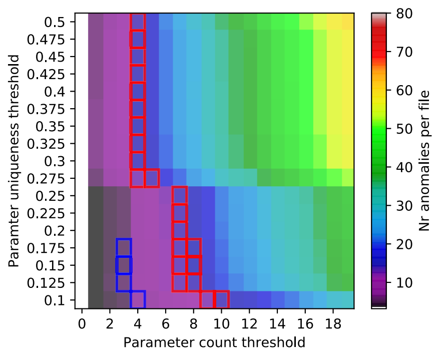


Figure 5: Average number of anomalies (for files with anomalies), over a range of threshold values for the P^c_{thresh} parameter count and P^u_{thresh} parameter uniqueness. Red boxes indicate settings resulting in an average number of anomalies >12 and <15. Blue boxes indicate settings resulting in average fraction of files with anomalies >0.45 and <0.55.

is low when compared to the timing anomalies. From a distribution standpoint Figure 4b shows that only a very weak bimodal tendency is present for the highest threshold.

C. *Parameter value anomalies*

Parameters value anomalies are triggered by anomalous values of the numerical parts of a log line. In some cases, such as outgoing IP port numbers, the number generation process has a degree of randomness in it resulting in a large variation of values for that particular parameter. Only applying an occurrence count threshold as done for the template count threshold would be problematic for these types of values. Consequently, the parameter value anomaly detection has a second threshold, the value uniqueness fraction. This threshold controls how large fraction of all observed parameter values for a given parameter that needs to be unique in order to consider that parameter highly variable and thus not apply the parameter count threshold test in order to detect a possible anomalous value. Evaluating this anomaly type is complicated by the dependence on two threshold values. With the parameter uniqueness threshold fixed, the behavior of the parameter count threshold is shown in Figure 3c.

The effect of different parameter setting combinations are explored in Figure 5, which additionally marks setting combinations that achieve two particular goals of developer cognitive workload. This illustrates that the KSB method, although it being unsupervised, can use the training data to explore appropriate settings for a given development context where an organization wishes to achieve some particular balance between risk of missed anomalies against the additional developer workload of checking more anomaly indications.

IV. Discussion

When considering anomaly detection as a tool to improve developer productivity it is also relevant to consider how developers interact with the anomaly detection system. A UI front-end was developed in order to display the outcome from the KSB anomaly detector. An example of how the UI can display the output from an examination run is shown in Figure 6. The UI marks the lines with identified anomalies by using red text and each such line is also marked with a letter to the left of the line number. This letter indicates what type of anomaly has been detected with T for time, P for parameter, and U for unseen template. To provide context, there are also three lines before and after each anomalous line included in the display. For each line it is possible to see which template the particular line matched to by pressing the right hand 'Show' button.

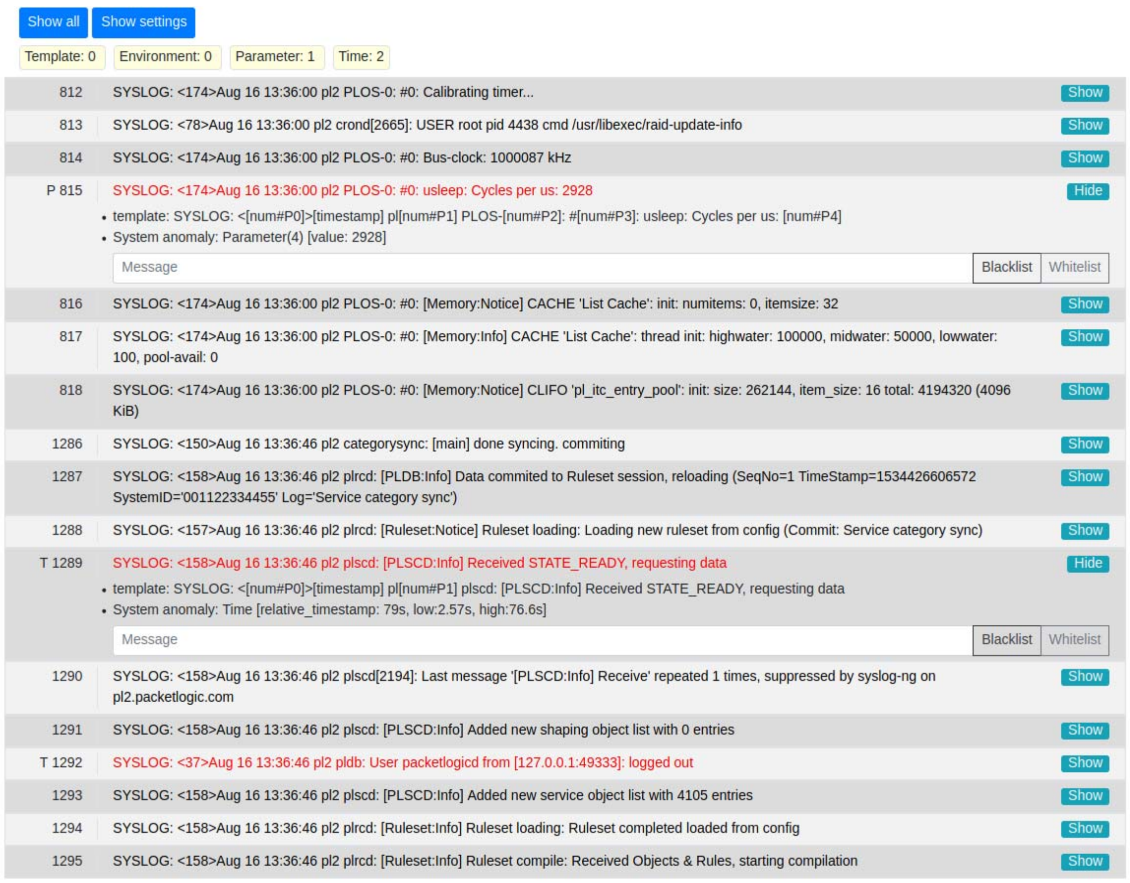
For each anomalous line extra information is provided to give more insight in the underlying cause for the particular detected anomaly. Thus, the anomaly on line 815 is due to the fourth parameter value being sufficiently different from what has previously been observed, as determined by the parameter count setting, while not being sufficiently random to be captured by the parameter uniqueness test. The anomaly on line 1289 is caused by the relative timestamp for this particular line being outside the configured time interval boundaries.

The UI further allows individual templates to be whitelisted or blacklisted, and to record the rationale for such a decision. Here, whitelisting means that any log line matching this particular template will not be flagged as an anomaly regardless of how many of the anomaly detection components mark it as anomalous. Blacklisting instead signifies that the particular template will always be flagged as an anomaly. As the white- and black-lists become populated, the system will over time reduce both the false positive and false negative rates.

If the user wishes to view the complete log file of around 1350 lines this is achieved by pressing the 'Show all' button. It is also possible to examine the effects of varying the anomaly detection thresholds and other settings directly in the UI by employing the 'Show settings' button. This allows a developer to override the configured settings to adapt to individual requirements on the trade-off between false negatives and workload.

V. Conclusions

For the problem of boot log anomaly detection we introduce the K-Seen-Before (KSB) approach. The high-dimensional problem of mixed text and numerical value anomaly detection is attacked by KSB mainly with consecutive 1-dimensional K-Nearest Neighbor classifiers with zero distance threshold, after appropriate line template search and tokenization has been performed. The sensitivity of the algorithm to settings of the threshold levels is explored over a wide setting space using an industrial boot log data set of 753 log files with over one million lines. It is shown how settings can be identified which provide an appropriate trade-off for the cognitive workload experienced by the developer when examining log files. Future work include the extension of the current work to more fine-grained handling of different numerical data, alternate time anomaly approaches, and handling combinations of parameter values in addition to evaluating each parameter in isolation.



```
812     SYSLOG: <174>Aug 16 13:36:00 pl2 PLOS-0: #0: Calibrating timer...
813     SYSLOG: <78>Aug 16 13:36:00 pl2 crond[2665]: USER root pid 4438 cmd /usr/libexec/raid-update-info
814     SYSLOG: <174>Aug 16 13:36:00 pl2 PLOS-0: #0: Bus-clock: 1000087 kHz
P 815   SYSLOG: <174>Aug 16 13:36:00 pl2 PLOS-0: #0: usleep: Cycles per us: 2928
        • template: SYSLOG: <[num#P0]>[timestamp] pl[num#P1] PLOS-[num#P2]: #[num#P3]: usleep: Cycles per us: [num#P4]
        • System anomaly: Parameter(4) [value: 2928]
816     SYSLOG: <174>Aug 16 13:36:00 pl2 PLOS-0: #0: [Memory:Notice] CACHE 'List Cache': init: numitems: 0, itemsize: 32
817     SYSLOG: <174>Aug 16 13:36:00 pl2 PLOS-0: #0: [Memory:Info] CACHE 'List Cache': thread init: highwater: 100000, midwater: 50000, lowwater:
        100, pool-avail: 0
818     SYSLOG: <174>Aug 16 13:36:00 pl2 PLOS-0: #0: [Memory:Notice] CLIFO 'pl_itc_entry_pool': init: size: 262144, item_size: 16 total: 4194320 (4096
        KiB)
1286    SYSLOG: <150>Aug 16 13:36:46 pl2 categorysync: [main] done syncing. commiting
1287    SYSLOG: <158>Aug 16 13:36:46 pl2 plrcd: [PLDB:Info] Data commited to Ruleset session, reloading (SeqNo=1 TimeStamp=1534426606572
        SystemID='001122334455' Log='Service category sync')
1288    SYSLOG: <157>Aug 16 13:36:46 pl2 plrcd: [Ruleset:Notice] Ruleset loading: Loading new ruleset from config (Commit: Service category sync)
T 1289  SYSLOG: <158>Aug 16 13:36:46 pl2 plscd: [PLSCD:Info] Received STATE_READY, requesting data
        • template: SYSLOG: <[num#P0]>[timestamp] pl[num#P1] plscd: [PLSCD:Info] Received STATE_READY, requesting data
        • System anomaly: Time [relative_timestamp: 79s, low:2.57s, high:76.6s]
1290    SYSLOG: <158>Aug 16 13:36:46 pl2 plscd[2194]: Last message '[PLSCD:Info] Receive' repeated 1 times, suppressed by syslog-ng on
        pl2.packetlogic.com
1291    SYSLOG: <158>Aug 16 13:36:46 pl2 plscd: [PLSCD:Info] Added new shaping object list with 0 entries
T 1292  SYSLOG: <37>Aug 16 13:36:46 pl2 pldb: User packetlogicd from [127.0.0.1:49333]: logged out
1293    SYSLOG: <158>Aug 16 13:36:46 pl2 plscd: [PLSCD:Info] Added new service object list with 4105 entries
1294    SYSLOG: <158>Aug 16 13:36:46 pl2 plrcd: [Ruleset:Info] Ruleset loading: Ruleset completed loaded from config
1295    SYSLOG: <158>Aug 16 13:36:46 pl2 plrcd: [Ruleset:Info] Ruleset compile: Received Objects & Rules, starting compilation
```

Figure 6: UI for displaying the detected anomalies (red lines) and their environment

ACKNOWLEDGMENTS

The authors wish to thank Daniel Larsson for discussions on boot log anomaly detection and an initial implementation of KSB, Anders Waldenborg and Sandvine for providing problem background and log files. Funding for this study was provided by the HITS project grant from the Swedish Knowledge Foundation.

REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[2] M. Amer, M. Goldstein, and S. Abdennadher, "Enhancing one-class support vector machines for unsupervised anomaly detection," in *ACM SIGKDD Workshop on Outlier Detection and Description*, 2013, pp. 8–15.

[3] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *IEEE International Conference on Data Mining (ICDM)*, 2008, pp. 413–422.

[4] Z. Li, M. Davidson, S. Fu, S. Blanchard, and M. Lang, "Converting unstructured system logs into structured event list for anomaly detection," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, pp. 1–10.

[5] F. Zulkernine, P. Martin, W. Powley, S. Soltani, S. Mankovskii, and M. Addleman, "Capri: a tool for mining complex line patterns in large log data," in *International Workshop on Big Data, Streams and Heterogeneous Source Mining*, 2013, pp. 47–54.

[6] Z. Liu, T. Qin, X. Guan, H. Jiang, and C. Wang, "An integrated method for anomaly detection from massive system logs," *IEEE Access*, vol. 6, pp. 30 602–30 611, 2018.

[7] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial intelligence review*, vol. 22, no. 2, pp. 85–126, 2004.

[8] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *ACM SIGMOD International Conference on Management of Data*, 2000, pp. 427–438.

[9] F. Angiulli and C. Pizzuti, "Fast outlier detection in high dimensional spaces," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2002, pp. 15–27.

[10] K. Sricharan and A. O. Hero, "Efficient anomaly detection using bipartite k-nn graphs," in *Advances in Neural Information Processing Systems*, 2011, pp. 478–486.

[11] G. O. Campos, A. Zimek, J. Sander, R. J. Campello, B. Micenková, E. Schubert, I. Assent, and M. E. Houle, "On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study," *Data Mining and Knowledge Discovery*, vol. 30, no. 4, pp. 891–927, 2016.

[12] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," in *ACM SIGMOD International Conference on Management of Data*, 2000, pp. 93–104.

[13] N. Taerat, J. Brandt, A. Gentile, M. Wong, and C. Leangsuksun, "Baler: deterministic, lossless log message clustering tool," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 285, 2011.

[14] E. Baseman, S. Blanchard, Z. Li, and S. Fu, "Relational synthesis of text and numeric data for anomaly detection on computing system logs," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2016, pp. 882–885.

[15] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1573–1582.