A Year of Automated Anomaly Detection in a Datacenter

Rufaida Ahmed
School of Computing
University of Utah
Salt Lake City, UT, USA
rufida94@cs.utah.edu

Joseph Porter
School of Computing
University of Utah
Salt Lake City, UT, USA
jporter@cs.utah.edu

Abubaker Abdelmutalab
School of Computing
University of Utah
Salt Lake City, UT, USA
abubaker@cs.utah.edu

Robert Ricci
School of Computing
University of Utah
Salt Lake City, UT, USA
ricci@cs.utah.edu

***Abstract*—Anomaly detection based on Machine Learning can be a powerful tool for understanding the behavior of large, complex computer systems in the wild. The set of anomalies seen, however, can change over time: as the system evolves, is put to different uses, and encounters different workloads, both its 'typical' behavior and the anomalies that it encounters can change as well. This naturally raises two questions: how effective is automated anomaly detection in this setting, and how much does anomalous behavior change over time?**

In this paper, we examine these question for a dataset taken from a system that manages the lifecycle of servers in datacenters. We look at logs from one year of operation of a datacenter of about 500 servers. Applying state-of-the art techniques for finding anomalous events, we find that there are a 'core' set of anomaly patterns that persist over the entire period studied, but that in to track the evolution of the system, we must re-train the detector periodically. Working with the administrators of this system, we find that, despite these changes in patterns, they still contain actionable insights.

I. Introduction

A key task for administrators of large computer facilities is understanding the steady-state operation of their facilities and reacting to any anomalies that might occur. The sequences of events that actually take place in "normal" operation may or may not align with administrators' intuition about the behavior of the facility and its users; having a full understanding is important to effective system administration. Exceptions to normal sequences may indicate problems with the facility's hardware, software, or configuration and may require administrator attention. Such exceptions could also signal new uses or emergent behaviors that administrators should be aware of.

Understanding "normal" and anomalous behavior is not always straightforward. Events from these types of systems are typically collected in logfiles [8], and simply looking for "errors" in these logfiles is not always informative [4]. Some "errors" may be benign: they might correspond to ways in which the facility is used that were not anticipated by software writers, or they may represent transient states that the system is able to recover from itself. On the flip-side, some sequences that are not explicitly flagged in logs as "errors" may be quite worrying, such as increased frequency of certain operations or cessation of others. The examination of logfiles for anomalies and errors is thus a ripe area for machine learning and data mining [8], [15], [5].

In this paper, we apply the technique of anomaly detection by *invariant mining* [14], [13] to the administration of CloudLab [9], a facility used by thousands of researchers and educators in computer science. CloudLab collects extensive logfiles regarding the provisioning of the servers under its control; as we lay out in more detail in Section III, the dataset used for this paper covers a year of operation of 583 servers, comprising a total of 15,018,235 log entries. Invariant mining looks at the relationships between frequencies of entries in these logfiles, finding patterns that describe typical operation ("invariants") and log sequences that "violate" those invariants and are thus anomalous. Our goal is to look at the following questions to find whether invariant mining is a useful technique to aid administrators in this setting:

1) Does invariant mining successfully create discriminators capable of distinguishing "normal" behavior from anomalous behavior?
2) Do the invariants found provide information that is interpretable by the administrators of these systems?
3) Do the set of invariants change over time, and if so, how much?

In Section II, we cover related work on analyzing system logfiles and mining them for anomalies. In Section III, we describe the facility from which our dataset comes and the dataset itself. Section IV explains how we use invariant mining on this dataset, Section V goes through our findings from this analysis, and Section VI concludes and suggests future work.

II. Related Work

The high risk posed by compromised systems, anomalies, and security threats has led to substantial interest in analyzing system logs to debug system failures and perform root cause analysis.

Moreover, Machine Learning (ML) and data mining techniques have been used widely to monitor large scale systems for the purpose of anomaly detection and system diagnosis. Several statistical and machine-learning models have been proposed to analyze the behavior of systems and detect failures or problem by deeply analyzing systems logs and other sources of data. In this section, we discuss the most closely related efforts in the area of anomaly detection and log file analysis.

A. Anomaly Detection

There are some generic methods that use system logs for anomaly detection. Typically, this is done by using a log parser to parse the unstructured log entries into a structured form [7] that can be analyzed and modeled by different machine learning techniques. These machine learning techniques are divided into *supervised* and *unsupervised* methods.

For supervised methods, labels are required to complete the analysis and perform anomaly detection. The simplest labels for this use would be "normal" and "anomalous". In practice, however, it is usually hard to obtain labeled data in log files: system logs commonly contain hundreds of thousands to millions of entries, making manual labeling by administrators too time-consuming. Additionally, because anomalies are, by definition, rare, it is not practical to use small subsets of system logs for training, since this would risk including too few, or even zero, anomalies.

Related work in the area of anomaly detection in systems goes back several decades. For example, Bates et al. [3] proposed an event definition language that allow programs to generate logs with deep semantics information, such as hierarchical relationships between events. However, this approach requires access to the source code. Some more recent methods [5] perform anomaly detection on log files without requiring hand-crafted features or pre-processing of data. These work on raw text data and output a score for each log entry, which enables the systems administrator to classify the log entry as either anomalous or normal. Baseman [2] proposes a framework that performs anomaly detection by combining graph analysis, relational learning and kernel density estimation. Moreover, it presents a novel event block detection algorithm that extracts related syslog messages from the log files. The proposed methods analyze individual messages rather than event blocks which limit the application scope. Furthermore, Baseman [1] introduced Interpretable and Interactive Classifier-Adjusted Density Estimation with Temporal components (iCADET). This framework utilizes random forest classifiers to explain the labeling of certain points as likely anomalous. This technique is more suitable for smaller scale data.

There are also some open source solutions for log files inspection and anomaly detection. For example, Project Scorpio [1] connects to streaming sources and uses unsupervised machine learning methods to generate a prediction of anomalous log entries.

B. Logfile Analysis

Other research efforts have specifically targeted logfile analysis for anomaly detection. For example, DeepLog [8] proposes a deep neural network model utilizing Long Short-Term Memory (LSTM). This model allows DeepLog to train a model *unsupervised* based on the log pattern and report an anomaly when log patterns deviate from the expected result by the trained model. Several other approaches based in machine learning have been proposed for different systems. Many of these are rule-based approaches, which limits them to specific application and requires domain knowledge. For example, M. Cinque [6], performs a change in the logging mechanism itself, which requires both effort and domain knowledge to implement the change to the logging system first. Other kinds of tool rely on comparing anomalous logs against normal ones, such as [15]. A limitation of such tools such tools is the fact that it is hard to detect new kind of anomalies that the model has not been exposed to before. Because our goal in this work is to study how anomalies change over time, it is important that we be able to find anomalies that were not seen during earlier periods.

Furthermore, some methods were developed to reduce the size of the log files and thus reducing the effort needed for analysis. For example, LSTM, which have been recently used for log analysis purposes in data centers. T. Yang and V. Agrawal. [19] introduced a framework that highlights the messages it deems to be the most important text in the failed log messages, making it less tedious for the human operator or even automated software to analyze the cause behind the failures.

Invariant mining [14] is a general approach that does not rely on the nature of the data or require any significant domain knowledge and unlike rule/keyword based log analysis tools the rules are easier to update when components are upgraded or changed as they usually tend to do. Our work builds on this work, which we give an overview of in Section IV-A. Lou et al. [14] applied invariant mining to two case studies, Hadoop and CloudDB (a structured data storage service developed by Microsoft). The testing environment was setup specifically for the purposes of this research. In contrast, our our work uses real-life data from a time span of one year, giving us the opportunity to gain a better understanding of the nature of anomalies and the benefits of using invariant mining to detect anomalies in real datacenter systems.

III. Dataset

In this section, we describe the dataset. We talk about CloudLab[2], the facility our logfiles come from. In addition to describing the contents of the logs themselves, we also cover the process we used to prepare the data for analysis. The dataset used for this paper is available with DOI 10.5281/zenodo.4073861.

A. CloudLab

CloudLab [9] is a facility that serves the computer systems research community. It operates as an environment in which researchers can build their own clouds: it provisions resources at a "bare metal" level, enabling its users to see, control, and modify portions of the cloud software stack including virtualization, networking, and storage. It has approximately 5,00 users around the world who have, to date, run 150,000 experiments on it. CloudLab has three main clusters; the data

[1]https://github.com/AICoE/log-anomaly-detector

[2]https://cloudlab.us/

that we use for this paper comes from its cluster at the University of Utah [17].

We chose CloudLab for this study because we have access to both its logfiles, which are collected centrally, and its administrators, who can help us interpret our results and evaluate their utility. While CloudLab is a unique facility in terms of the specific features it offers to users, its basic functionality of managing the provisioning of servers, interaction with users via a web interface, etc. has much in common with other facilities and should lead to generalizable results.

In this paper, we focus on CloudLab's *node booting* process: the automated process of booting servers into various operating systems for user experiments, utility tasks (such as re-imaging local hard drives), and general system administration. Though a conceptually simple task, this process involves interactions between firmware and BIOS on the servers themselves, standard network protocols such as DHCP and TFTP, and a number of services that CloudLab runs to track server state [16] and inform servers what their next actions should be. There is substantial emergent complexity in this system and large amounts of parallelism that are difficult to control. As a result, failures to boot are not uncommon, and the CloudLab code includes many measures to detect and automatically recover from common failure modes. Because of the way CloudLab allocates resources, it is common for a server to be part of several experiments in a single day in sequence, and thus to go through this boot process every few hours. It is also common for some servers to be allocated to an individual experiment for long periods of time, meaning that they may not reboot for a period of days or weeks.

B. Data Collection

CloudLab log data is collected, processed and stored using the ELK (Elasticsearch[3], Logstash[4], Kibana[5]) stack. In our configuration, the ElasticSearch cluster is composed of five data nodes and one client node that also serves the Kibana frontend. As is common with the ELK stack, we have Filebeat[6] aggregate and forward logs from the main CloudLab servers to be processed by Logstash and stored in the ElasticSearch cluster.

The logfiles that we collect come from a mix of standard server software, such as Apache, ISC DHCPD, and `tftpd`; and custom software that has been developed for CloudLab and other related testbeds [18], [11]. Overall, we collect on average 350,000 logs entries per hour (though only a subset of those is used in this analysis.)

C. Parsing and Cleaning

To be used for data mining and machine learning, log messages must be individually identified and parameters, etc. parsed out; the relatively *unstructured* text found in the logfiles must be converted into *structured* data. For invariant mining in particular, each log message must be assigned a corresponding event ID, (also called a *log key*) that indicates the message type. These event IDs are matched to specific patterns, where the pattern represents the constant parts of the message and the variable parameters that the message contains. To get this information, we process each message against a set of Grok [10] patterns. While this log parsing method is sometimes automated [7], [20], [12], our initial attempts to use these automated systems did not produce satisfactory results; thus, we created the Grok patterns by hand to ensure accuracy and to further explore our data. Lists of patterns are consumed by a script to automatically generate a LogStash configuration file to process messages, and we *version* these patterns: each entry in Elasticsearch is tagged with the version number of the pattern set, so that when we add or change patterns, we can re-parse all stored log entries.

An interesting aspect of processing logfiles is that sometimes *mapping* is required between different identifiers for the same entity. One way this manifests in the CloudLab data is that in some logfiles, machines are identified by their "node ID", the primary identifier CloudLab uses to track its resources. In others this information is not available. For example, in DHCP logs, initial requests are identified only by their MAC address. As part of our parsing process in Logstash, we use mapping tables to augment records with *all* identifiers for the node to make it easier to relate entries with each other.

With the data processed and stored using our ELK stack, the data must be collected and formed into data files before applying invariant mining. Data files were created using a script that generated ElasticSearch queries based on selected node type, node range, date range and log types. Each entry from the resulting query had its message and event ID written to an output file along with its assigned session ID. The session ID is formed from the node ID and date to delineate chunks of log entries into *sessions*, where each session represents a 24 hour period for a particular node.

To provide clean datasets, some data had to be excluded because of inconsistencies or errors. As a result of abnormal node ID formats and deformed log messages, some log entries were not correctly matched with a pattern and any such entry was excluded from generated data sets. Additionally, each message has two timestamps; one from the machine time which contains the message and another assigned by Logbeat at its collection time. In some cases, the two timestamps differed significantly, with Logbeat retrieving the log months after the machine timestamp. Such occurrences had to be excluded from datasets to ensure that the date used to form session IDs were accurate.

D. Resulting Dataset

For the purposes of this paper, the dataset was formed from logfiles of all CloudLab nodes of the types `m400` and `m510`, and was gathered from January 1 to December 31 of 2019. The resulting dataset contains over 15 million log entries for those 583 nodes and forms 51,375 sessions.

[3]https://github.com/elastic/elasticsearch
[4]https://github.com/elastic/logstash
[5]https://github.com/elastic/kibana
[6]https://github.com/elastic/beats

The dataset we use for this paper is formed from four specific logfiles, each of which has its own set of message patterns. All of these logfiles record events related to the process of *provisioning* and *booting* nodes. Typically, a reboot of a node is initiated by the CloudLab server in response to a user starting or ending an experiment, though users can reboot nodes themselves either intentionally or as a side effect of a kernel crash on the node.

- `reboot` is a log that contains the system's high-level "intent" with respect to rebooting nodes; that is, when a node is intentionally rebooted, an entry is created in this logfile.
- `stated` reports the status of an internal state machine used in some CloudLab processes [16]. Each state (such as `BOOTING`) has a set of expected successor states (such as `DHCP`, `RELOADING`, etc.) and some states have timeouts associated with them. CloudLab uses this state machine to detect and attempt to recover from certain kinds of faults.
- `bootinfo` is a CloudLab-specific daemon that is used to inform nodes of what they should boot next (eg. boot into a special memory-based filesystem used for re-imaging, boot from a partition on the disk, etc.) The first-stage bootloader contacts this service, so it provides information that a node has reached a certain point in the boot process and gives context regarding what the node is booting.
- `dhcpd` records DHCP events from the server's perspective. Because nodes contact the DHCP sever at multiple points during the boot process (from the PXE ROM, OS initialization, etc.), this provides fairly fine-grained information regarding nodes' progress through the boot process.

To parse these log files, we used 48 unique log patterns with `bootinfo` and `stated` having the most unique patterns, with 25 and 15 respectively.

IV. Analysis Methodology

We took the dataset described in Section III and applied invariant mining [14] to find what constitutes "normal" behavior for the CloudLab provisioning process, and to find deviations from this normal. In addition to mining invariants for specific time periods, we also develop a method for examining how they change over time so that we can understand if the steady-state behavior of the facility changes or not.

As mentioned before, manual inspection of log files is infeasible due to the system's large scale and high complexity. Moreover, the software that manages this system is updated frequently, which makes it impractical to rely on rule-based log analysis solutions. Since invariant mining does not utilize constant rules, does not require labels for training, and does not depend on the domain knowledge of system admins, it is more appropriate for use with regularly-revised, large-scale systems.

A. Invariant Mining

The idea behind invariant mining [14] is that what we consider to be normal behavior can be learned by mining the log files to discover the inherent linear characteristics of the program workflow. Any log entry that does not match the workflow will be considered anomalous. This method can be used to automatically define rules for anomalies and thus automatically detect them. The linear invariants reflect the properties of execution path and so a violation of an invariant can often reflect the physical meaning of the system problem which makes it a superior diagnostic tool for human operators.

The input that we provide to the invariant miner is a set of sessions (described in Section III-C), with each session containing a count of how many times each log key occurred during the session. The miner looks for sets of keys that typically occur with linear relations and outputs these ratios. For example, the miner might discover that each message indicating that a server has begun rebooting is typically paired with a message indicating a successful boot. Or, it might find that a message indicating that a server has begun PXE booting typically matches with two DHCP requests: one from the PXE ROM, and another from the OS once the server has booted into the OS.

Each ratio is called an *invariant*, and log sessions that do not follow this relation are said to *violate* the invariant; sessions that contain violations are said to be *anomalous*. Once we have this set of invariants, finding anomalies is straightforward: to check an individual session, we simply count occurrences of log keys and check whether they violate any invariants.

The invariant miner has a simple data model in that it just looks for integer ratios in the counts of event occurrences. Advantages of this approach include the fact that it does not require the semantic information that would be required to truly match up specific events, and that these ratios stay the same (under normal conditions) no matter how many boot cycles are observed in a given window. What it gives up in return is that while it can identify the presence of an anomaly, the invariant miner but itself cannot point to specific log messages that caused this anomaly. For example, if the anomaly detector finds that there are more "started booting" messages in a session than "successfully booted" ones (violating the expected one-to-to ratio), it can flag an anomaly, but finding *which* boot attempt failed requires additional processing or manual inspection.

For this paper, half of the data set is used as a training dataset and the other half is used as a test dataset.

B. Comparison Across Time

To analyze change over time, we divided the data from year 2019 into four quarters (January–March, April–June, etc.) and trained the invariant miner with each quarter's data independently. We then compared the *sets* of invariants found in each quarter. These results were used to study how usable the invariants are (that is, whether administrators found them accurate and actionable), which invariants are persistent over the year, etc. We also compare the number of invariants

violated in each quarter and analyze the reasons behind the difference in invariants violations between the quarters. Since the persistent violations occur in different time periods through the year, we highlight them as the most persistent violations.

C. Implementation

Our implementation is based on `loglizer` [13] by the Logpai team[7]. The original tool's invariant mining output was not sufficient by itself to list all invariants and sessions that violated them, so modified the source code to produce the needed information. This information included the mapping between the numbering of event types and the actual Event IDs in our log files. It also included the number of violations for each invariant, which sessions violated which invariants and which sessions are completely "clean."

We also wrote a post processing program to map the arbitrary session numbering used internally by `loglizer` to the original session IDs from the log files. It also maps the event in the invariants to its original text format to make it easier for human interpretation.

We programmed our data parser to output log files in the same format needed by the invariant miner. The most important feature in the obtained log files is to group log events according to their types. These logs groups are formed into session which contain all log entries for a particular server in a single day. The invariant miner counts the number of occurrences for each type, this count is used to find the ratio for occurrences for multiple event types and thus finding the needed invariants.

V. Findings

We now return to the main motivating questions for this paper: Is invariant mining accurate at finding actual anomalies in this dataset? Are the invariants it finds meaningful to administrators? Are the set of anomalies fairly constant over time, or do they vary? We start by looking at the invariants themselves.

A. Invariants Found

In the invariant mining process, log messages are grouped together according to a set of parameters that correspond to the same event type. The invariant miner then utilizes the event types and their frequencies to produce invariants such as the following:

```
(11, 29): [ 1.0, -1.0]
(17, 18): [ 1.0, -1.0]
(1,  65): [-4.0,  1.0]
```

The first invariant corresponds to two event types, 11 and 29, and the ratio of their occurrences in a normal session is $1 : 1$.[8] When this ratio is not satisfied the invariant is considered violated and an anomaly is reported. The second invariant shows another pair of event types that appear in a one-to-one ratio in normal sessions, and the third reports a one-to-four ratio.

Mapping the event IDs to actual log messages, a set of example invariants are shown in Figure 1. Each shows a pair of log lines that are expected to appear in a one-to-one ratio in a "normal" session.

In our dataset, the miner found an invariant space dimension of 16 in the first quarter, meaning that it found 16 unique invariants. For the second quarter of the year, the invariant miner produced 17 different invariants. For the third quarter, the result was 13 invariants. And for the last quarter of the year, the result was 19 different invariants. Table I shows the total number of sessions and percentage of anomalies for each quarter in both the training data set and test data set.

B. Usefulness and Interpretability

We found that while some invariants were "useful", not all were. Here, we define "useful" by three criteria.

First, they must be *non-trivial* in the sense that it is *possible* to violate them. In some cases, the invariant miner found two types of log entries that are produced by the same function in the same program. In this case, it is nearly impossible to see one message without the other: the program would have to hang or crash within a few lines of code. No violations of this type were found in this dataset. We found six distinct invariants of this type. They are easy to identify, because they are never violated, and do not affect the accuracy of the anomaly detection.

Second, an invariant must be *sensible*. We evaluate this by looking at the expected ratio produced by the miner. While most invariants have ratios such as $1 : 1$ or $2 : 1$ that would be expected from a system of this type, the miner found some "invariants" with ratios as high as $311,785 : 1$. The highest-ratio event that, by manual inspection, appeared sensible was $1 : 7$. This corresponds to the number of times that one of the processes will retry apparent failures before giving up. Over the full year, the miner found 14 "invariants" with ratios of $15 : 1$ or higher. We find it highly likely that "violations" of these represent false identification of anomalies. Fortunately, they are easy to filter out, since there is a large gap between the largest "sensible" ratio $(7 : 1)$ and the smallest "insensible" ratio $(15 : 1)$; we can simply filter out invariants with ratios above $10 : 1$. We speculate that these false invariants were found due to a few highly-anomalous nodes that had behavior that persisted over multiple sessions. For example, one node was stuck in a "boot loop" for months, unnoticed by the operators. This resulted in many thousands of spurious DHCP messages intermingled with a few messages of other types. These sessions tended to be flagged as anomalous due to violating other invariants.

Third, invariants must be *interpretable*, meaning that administrators are able to understand, in a general sense, what the reasons behind a violation are or what the consequences of it might be. This is a much harder criterion to evaluate quantitatively, so we examine it qualitatively. The pattern that

[7] https://github.com/logpai/loglizer

[8] One part of the ratio is always shown as negative, as the miner is solving equations of the form $a \cdot x + b \cdot y = 0$. Which part is positive and which is negative is arbitrary.

```
Invariant 1: Ratio: ['1.0', ' -1.0']
 <DATE> <TIME> <NODE_ID>: in PXEWAIT, sending PXEWAKEUP
 <DATE> <TIME> boss bootinfo[<PID>]: <IP>: SEND: query bootinfo
Invariant 2: Ratio ['1.0', ' -1.0']
 <DATE> <TIME> [<PID>]: <NODE_ID>: RESET done, bootwhat returns NORMALv2
 <DATE> <TIME> [<PID>]: <NODE_ID>: Clearing reload info
Invariant 3: Ratio: ['-1.0', ' 1.0']
 <DATE> <TIME> <NODE_ID>: ssh reboot returned 255
 <DATE> <TIME> <NODE_ID>: waiting 30s for reboot
```

Fig. 1. Example invariants found by the invariant miner

Dataset	Jan-Mar		Apr-Jun		Jul-Sep		Oct-Dec	
	sessions	anomalies	sessions	anomalies	sessions	anomalies	sessions	anomalies
Training dataset	5220	3.8%	5713	4.4%	6154	2.7%	8600	4.7%
Test dataset	5220	3.1%	5713	5.0%	6154	1.9%	8601	4.8%

TABLE I
NUMBER OF SESSIONS AND PERCENTAGE OF ANOMALIES FOUND PER QUARTER IN OUR DATASET.

we find among the most interpretable invariants is that they follow one or more of the following properties:

- They involve entries that appear in more than one logfile. In other words, to detect the invariant, it is necessary to correlate information across logfiles. This provides significant value to administrators, who tend to inspect a single log file at a time.
- There is a clear way to match events. That is, it is possible, either through timing or unique identifiers, to confirm that one event is in some way a response to or consequence of the other. Note that the invariant-based anomaly detector does not produce such a matching itself, but this can generally be done by additional processing or manual inspection.
- They are asynchronous operations: an operation on a node is started by one process; the node performs some actions that are not directly in the logfiles, may take a variable amount of time, and may fail; and the success or failure of those actions are observed by a different process.

An example of invariant that meets all of these criteria would be one that relates a log message indicating that a node is to be rebooted with one that logs a successful DHCP response to the node later, after the node has shut down, made it through BIOS, and the NIC's boot ROM, etc. We found five invariants that we deemed highly interpretable by these criteria: some of these are discussed in more detail in the following subsections.

C. Accuracy of Anomaly Detection

In order to assess the accuracy of anomaly detection using invariant mining, we compared the labels produced by the invariant-based detector with labels assigned by humans. To do this without requiring undue operator effort, we ran the invariant miner on the dataset described in Section III and labeled sessions according to the invariants found. We then created five sets, each containing ten sessions that the invariants had labeled as "normal" and ten that it had labeled as "anomalous". After correcting for a few invariants with "insensible" ratios, the base rate of "normal" sessions in the dataset we gave to administrators was 56%. Each set was given to a different administrator of the CloudLab testbed, who was asked to label each session. The administrators were told that their set contained a mix of normal and anomalous sessions, but were not told how many there were of each, and were not given a definition of "anomalous"; the intention behind this methodology was to see how well the precise ratios found by the invariant mining process match up with human intuition.

In our evaluation, we consider a "normal" label as negative result and an anomalous label as positive results. Therefore, a false negative refers to an incorrect labeling by the classifier for a truly positive results and a false positive refers to an incorrect labeling by the classifier for a truly negative result. The accuracy of the invariant miner was reasonable: it correctly labeled 70% of sessions identified by the administrators as normal, and 73% of the sessions labeled as anomalous. This gives us an overall false positive rate of 30% and false negative rate of 27%. The overall precision obtained is 0.7087, recall is 0.7300 and the F1-score is 0.7192.

There were two other interesting findings from this portion of our study.

First, we found that the administrators made a distinction between behavior that indicated a problem with the system and unusual user behavior. One example of this in our context is that most boot sequences are initiated by the system in response to higher-level user requests, such as the start or termination of experiments. These kinds of sessions have telltale log entries indicating the start of the process. If users shut down or reboot machines themselves (eg. by running `shutdown` on the machine itself), these telltale log entries are absent, and there may be log entries indicating an unexpected shutdown. Most administrators independently came up with

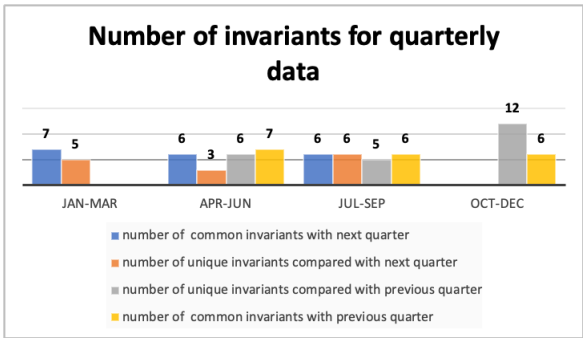

Fig. 2. Comparison of number of invariants throughout one year. Note that the first quarter has no preceding quarter, and the last has no succeeding one.

this third label, which, in terms of frequency, is anomalous, but is likely to not require administrator intervention. For the purposes of the calculations above, we considered these to be anomalous, but it suggests the potential for future work to distinguish known-benign classes of anomalies from those that might require intervention.

Our second finding was that the sessions that were mislabeled by invariants tended to fit very specific patterns. One of the biggest discrepancies was a single server that exhibited the same anomalous behavior (according to the administrator) over three different days, but was labeled as normal according to the invariants. Nearly all the rest of false positives were caused by a single set of three related invariants, that caused sessions identified as normal by the administrators to be flagged as anomalous by the detector. This seems to suggest that relatively simple heuristics could be used to greatly improve the accuracy rates, and suggests an avenue for future work.

D. Evolution of Invariants Over Time

First, we compared the invariants through the four quarters of the year 2019 using the number of unique invariants in the current quarter compared to the previous quarter and next quarter. We also used the number of shared invariants between quarters as a measure for the evolution of invariants. In our study of the invariants, we focus more on the persistent invariants through quarters as they are the most meaningful invariants.

Figure 2 shows the comparison between the number of invariants obtained through the year. From quarter to quarter, we see that approximately half of the invariants change; that is, the number of invariants that each quarter has in common with its neighboring quarters is about half of the total number of invariants for the quarter. This points out the need to periodically re-train the anomaly detector.

When comparing the invariants across all quarters, we find that we have 6 core invariants that persist through the year. This means that in most cases, when Figure 2 shows invariants in common with the previous and/or next quarters, it is referring to this set. As with most useful invariants that we find, these all occur with ratio $1:1$. Two of these invariants have to do with using `ssh` to "gracefully" reboot nodes. (One of these can be seen as Invariant 3 in Figure 1.) These two often cause false positives; they are two of the three associated with false positives (as determined by human administrators) above. Another pair (such as Invariant 2 from Figure 1) have to do with CloudLab's disk imaging process: they show the state transitions that are supposed to occur when the imaging process finishes and the node boots into its new image. The fifth invariant is a trivial one as defined in Section V-B: it documents a node requesting information from the `bootinfo` process and the reply that is sent out.

The sixth (Invariant 1 from Figure 1) is the most interesting: it contains one message from the `reboot` log, and another from `bootinfo`. The former indicates that the CloudLab software has decided to reboot a node, and the latter indicates that the node has reached an important point, several steps into the boot process. This is interesting not only because it crosses multiple log files, but because the point identified comes *after* several other log messages (such as ones from `dhcpd`) would normally be seen. This strongly suggests two things: (1) Nodes that CloudLab decides to reboot do normally come up, which is expected (2) The lack of earlier invariants from the `dhcpd` log suggests that it is *not* terribly uncommon for nodes to fail early in the boot process and require power cycling by the CloudLab control software. If the sequence "reboot, DHCP, PXE, bootinfo" (the 'normal' boot sequence) were dominant, we would expect to see invariants for the "reboot, DHCP" part of the sequence. This lack suggests that CloudLab not infrequently times out waiting for the DHCP message and power cycles the node. The fact that the "reboot, bootinfo" invariant *does* exist implies that when this power cycle occurs, the node does eventually reach the "bootinfo" stage, suggesting that these kinds of failures are transient. This understanding meshes well with our findings in Section V-C, in which we found that there are a significant number of abnormal occurrences that are dealt with automatically by CloudLab and do not require operator intervention.

VI. Conclusion and Future Work

We find that invariant mining is fairly accurate on our real-world dataset, agreeing with the "anomaly" labels assigned by system administrators more than 70% of the time. The patterns of these inaccuracies suggest that simple heuristics, or occasional manual pruning of invariants, may substantially improve accuracy. Applying such heuristics is a topic for future work. We found that, in contrast to the binary nature of classification performed by most anomaly detection, administrators naturally and independently arrived at a *trinary* system of classification. This classification system subdivides anomalous events into those that require human attention, and those that, while rare, are in some sense "expected" and do not require additional attention. Building this distinction into anomaly detection is likely to be another fruitful avenue for future work.

We also found that anomaly rates vary substantially between quarters (from 1.9% to 5%), and that the set of invariants that describes these anomalies varies as well. This points to the

need to periodically re-train anomaly detectors, as they become stale over time. In this paper, we have used fixed time periods for sessions (24 hours) and re-training periods (3 months); as future work, we intend to investigate other values, including using sliding windows for sessions.

ACKNOWLEDGMENTS

We thank the Cloudlab administrators for access to the data and help interpreting it; in particular, we thank Mike Hibler, Gary Wong, David Johnson, and Aleksander Maricq for manually labeling sessions. We also thank Vivek Srikumar for his help selecting and interpreting ML techniques, and to the anonymous MLCS reviewers whose comments helped to improve the paper. We also thank Ali Ibrahim. This material is based upon work supported by the National Science Foundation under Grant Nos. 1743363 and 1801446.

REFERENCES

[1] E. Baseman, S. Blanchard, N. DeBardeleben, A. Bonnie, and A. Morrow. Interpretable anomaly detection for monitoring of high performance computing systems. In *Outlier Definition, Detection, and Description on Demand Workshop at ACM SIGKDD. San Francisco (Aug 2016)*, 2016.

[2] 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)*, pages 882–885. IEEE, 2016.

[3] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of systems and software*, 3(4):255–264, 1983.

[4] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Proceedings of the First workshop on Machine Learning for Computing Systems*, June 2018.

[5] S. Bursic, A. D'Amelio, and V. Cuculo. Anomaly detection from log files using unsupervised deep learning, 09 2019.

[6] M. Cinque, D. Cotroneo, and A. Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6):806–821, 2012.

[7] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *Proceedings of 16th IEEE International Conference on Data Mining (IEEE ICDM)*, Dec. 2016.

[8] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of 24th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2017.

[9] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2019.

[10] Elastic Co. Grok filter plugin (documentation). https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html. Retrieved March 2020.

[11] Flux Research Group, University of Utah. Emulab source code. https://gitlab.flux.utah.edu/emulab/emulab-devel. Retrieved March 2020.

[12] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016.

[13] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *27th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2016.

[14] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, pages 1–14, 2010.

[15] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 353–366, 2012.

[16] M. Newbold. Reliability and state machines in an advanced network testbed. Master's thesis, University of Utah, 2004.

[17] The CloudLab Team. Cloudlab hardware (documentation). https://docs.cloudlab.us/hardware.html#(part._cloudlab-utah). Retrieved March 2020.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*. USENIX, Dec. 2002.

[19] T. Yang and V. Agrawal. Log file anomaly detection. Technical report, Stanford, 2016.

[20] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. Tools and benchmarks for automated log parsing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2019.