2018 IEEE 38th International Conference on Distributed Computing Systems

LogLens: A Real-time Log Analysis System

Biplob Debnath*, Mohiuddin Solaimani†, Muhammad Ali Gulzar†, Nipun Arora†,
Cristian Lumezanu*, Jianwu Xu*, Bo Zong*, Hui Zhang§†, Guofei Jiang§†, and Latifur Khan‡
*NEC Laboratories America, Inc., Princeton, New Jersey, USA
‡CS Department, The University of Texas at Dallas, USA
§ Ant Financial, Hangzhou, China
Email: biplob@nec-labs.com, solaimani.rakib@gmail.com, gulzar@cs.ucla.edu, nipun@dropbox.com
{lume,jianwu,bzong}@nec-labs.com, shengchu.zh@antfin.com, geoff.jiang@yahoo.com, lkhan@utdallas.edu

***Abstract*—Administrators of most user-facing systems depend on periodic log data to get an idea of the health and status of production applications. Logs report information, which is crucial to diagnose the root cause of complex problems. In this paper, we present a real-time log analysis system called *LogLens* that automates the process of anomaly detection from logs with no (or minimal) target system knowledge and user specification. In *LogLens*, we employ unsupervised machine learning based techniques to discover patterns in application logs, and then leverage these patterns along with the real-time log parsing for designing advanced log analytics applications. Compared to the existing systems which are primarily limited to log indexing and search capabilities, *LogLens* presents an extensible system for supporting both stateless and stateful log analysis applications. Currently, *LogLens* is running at the core of a commercial log analysis solution handling millions of logs generated from the large-scale industrial environments and reported up to 12096x man-hours reduction in troubleshooting operational problems compared to the manual approach.**

I. Introduction

Log analysis is the process of transforming raw logs – written records of software systems events – into information that helps operators and administrators to solve problems [1, 2]. Log analysis is used in a variety of domains such as detecting security threats [3, 4, 5], compliance auditing [6], power plant fault detection [7], or data center operations [8, 9, 10, 11, 12]. The ability to analyze logs quickly and accurately is critical to reduce system downtime and to detect operational problems before or while they occur.

A critical aspect of a log that enables fast and accurate analysis is its structure. Recognizing the structure of a log greatly helps in easy extraction of specific system information, such as the type, time of creation, source of a specific event, the value of key performance indicators, etc. Without a known log structure, log analysis becomes a simple keyword-based text search tool. In fact, most commercial log analytics platforms today [13, 14] allow users to directly specify log patterns or to generate models based on domain knowledge. While supervised log analysis can help extracting important insights without ambiguity, it also has several shortcomings: a) it is specific to what the user seeks and focuses on known errors and b) it cannot easily adapt to new data sources and formats. As more new devices and data formats enter the market (Gartner, Inc. forecasts that 20.4 billion IoT units will be in use worldwide by 2020 [15]), it becomes increasingly difficult for the supervised log analysis tools to keep track and adapt to new log structures and identify unknown anomalies.

In this paper, we describe *LogLens*, a log analysis system to automatically detect operational problems from *any* software system logs. Rather than taking the log structure as an input, *LogLens* automatically learns structures from the "correct logs" and generates models that capture normal system behaviors. It subsequently employs these models to analyze production logs generated in real-time and detects anomalies. Here, we define *anomaly* as a log or group of logs that do not match the normal system behavior models. *LogLens* requires no (or minimal) user involvement and adapts automatically to new log formats and patterns as long as users can provide a set of logs for building models against which anomalies are detected.

LogLens classifies anomaly detection algorithms into two major groups: *stateful* and *stateless*. Stateless anomalies arise from analyzing a single log instance, while stateful anomalies appear when a combination of multiple logs does not match the trained model. For example, identifying errors or warnings in operational logs do not require keeping state about each log. In contrast, identifying maximum duration violation of a database transaction requires storing start event time of the transaction so that when an end event of the same transaction comes, anomalies can be detected by calculating the duration of the transaction. *LogLens* presents one exemplary stateless algorithm and one exemplary stateful algorithm. The exemplary stateless algorithm is a log parser, which parses logs using patterns discovered during system normal runs and reports anomalies if streaming logs cannot be parsed using discovered patterns. This stateless parser can parse logs up to 41x faster than the Logstash [16], which is a widely used log parsing tool. The exemplary stateful algorithm discovers relationships among log sequences representing usual operational workflows from the system normal runs and reports anomalies in the streaming logs. This stateful algorithm can handle heterogeneous log streams and can automatically discover ID fields to link multiple logs corresponding to an event.

To analyze massive volumes of logs with zero-downtime,

[†] Work done while worked at NEC Laboratories America, Inc.

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00105

we deploy *LogLens* on top of the Spark [17] framework. While Spark provides a low latency and high throughput data processing platform, our experience in building large scale log analysis system reveals that Spark lacks several key features needed to deploy *LogLens* as a real-time service with zero-downtime. In particular, its immutable broadcasting feature [18] forces us to restart service in order to update the learned models in a running system, thus we can not guarantee zero-downtime. Moreover, stateful algorithms need external stimuli for efficient memory management and timely anomaly detection. As a remedy, *LogLens* introduces a rebroadcasting mechanism. In addition, it proposes to add an external heartbeat controller for efficiently managing open states and for immediately reporting anomalies.

The rest of this paper is organized as follows: Section II describes our *LogLens* architecture. Section III describes our stateless log parsing algorithm. Section IV describes our stateful log sequence anomaly detection algorithm. Section V describes the challenges we faced and solutions adopted for deploying *LogLens* as a service. Section VI shows our experimental results. Section VII describes two case-studies of *LogLens* deployment in solving real-world problems. Finally, Section VIII states the conclusion and lesson learned followed by a bibliography.

II. LogLens Architecture

In this section, we present the system architecture of *LogLens* and the rationale behind our design choices.

A. Design Goals

The design of *LogLens* is driven by the following goals:

- **Handling heterogeneous logs**. Logs may have a variety of formats depending on their sources and what they are trying to convey. An automated log analyzer should be able to handle any log formats irrespective of its origin.
- **Minimizing human involvement**. Ideally, an automated log analyzer should work from scratch without any prior knowledge. For logs from the new sources, it should not mandate any human involvement. To this end, *LogLens* leverages unsupervised machine learning based techniques. Human interaction is limited to providing “training” logs, which captures “correct” behaviors. *LogLens* learns various models from this training dataset and uses them later to detect anomalies. In addition, *LogLens* provides options to the users to incorporate their domain knowledge in order to improve the accuracy of the anomaly detection algorithms.
- **Providing a generic system.** We aim to design a generic system which captures most real-world use cases and challenges. To this end, *LogLens* presents two exemplary anomaly detection algorithms. The first algorithm is stateless, while the second algorithm is stateful. *LogLens* presents a stateless log parser, which is a core component to design any log analysis algorithm. Usually, stateful algorithms are more complex and need quite an effort to implement efficiently – *LogLens* presents a log sequence anomaly detector to demonstrate various real-world challenges.
- **Handling data drift.** System behavior typically evolves over time. Hence, log data characteristics and behavior models may also change. To this end, *LogLens* periodically relearns models to adapt to system behavior change.
- **Expediting stateful anomaly detection.** Real-time anomaly detection algorithms are generally event-driven. Thus, in the absence of logs, some anomalies cannot be detected in time. *LogLens* ensures that all anomalies are reported in time by leveraging an external heartbeat controller which generates dummy messages periodically.
- **Deploying log analysis as a service.** We aim to design a system which can handle high volume and high velocity of the log streams in real-time. However, we want to leverage existing open-source data processing frameworks to minimize implementation and deployment effort. In addition, we want to guarantee zero-downtime (i.e., no service disruption). To this end, *LogLens* chooses Spark [17] big data processing framework because of its maturity, huge echo system and community support, and widespread adoption in the industry and academic realms. However, we find that even Spark (as well as similar frameworks, i.e., Flink [19], Samza [20], etc.) does not have all necessary features to deploy a log analysis service (see Section V). Finally, *LogLens* enhances the Spark framework to satisfy our design goals.

B. Architectural Components

Figure 1 illustrates the architecture of *LogLens*. Now, we briefly describe each component.

Agent is a daemon process which collects heterogeneous logs from multiple sources and sends them to the log manager.

Log Manager receives logs from agents. It controls incoming log rate and identifies log sources. It forwards incoming logs to the parser. It also stores them into the log storage.

Log Storage is the main storage or archival component. It organizes logs based on the log source information. Stored logs can be used for building models during log analysis. They can also be used for future log replaying to perform further analysis, or for post-facto querying when troubleshooting operational problems.

Model Builder generates models for the anomaly detection. It takes a set of training logs assuming that they represent normal behavior and uses unsupervised machine learning based techniques to build models. To adapt to system behavior change, periodically it collects logs from the log storage for relearning models and stores them on the model storage.

Model Storage stores models. All the anomaly detectors read models directly from the model storage.

Model Manager retrieves model information from the model storage and notifies the controller to update a model. *LogLens* supports both automatic and human interaction inside model manager. For example, users can configure *LogLens* to automatically instruct model builder every midnight to rebuild models using the last seven days logs. In addition, model

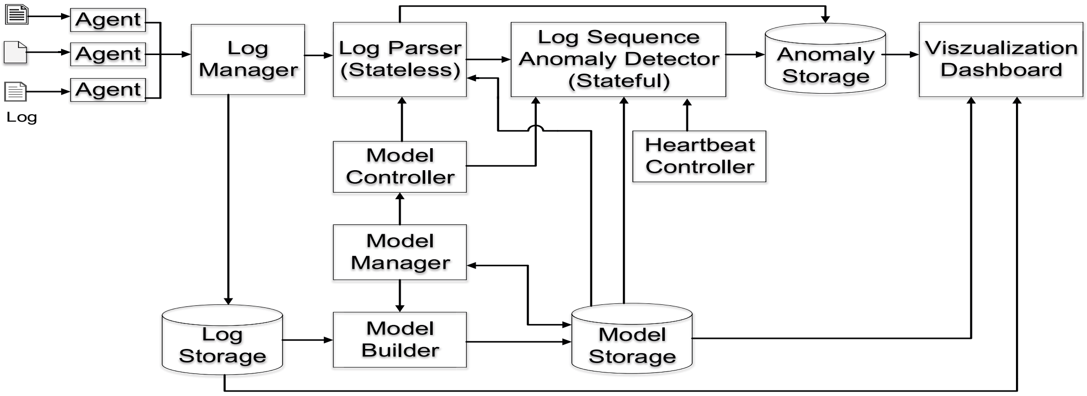


Fig. 1: *LogLens* architecture showing major components and operational workflows.

manager allows human experts to inspect models and edit them to incorporate domain knowledge.

Model Controller gets notifications from the model manager and sends control instructions to the anomaly detectors. Models can be added or updated or deleted, and each operation needs a separate instruction which contains detail information about the steps that need to be executed. Anomaly detectors read control instructions and take action accordingly.

Log Parser takes streaming logs and log-pattern model from the model manager as input. It parses logs using patterns and forwards them to the log sequence anomaly detector. All unparsed logs are reported as anomalies and presented to the user for further review. Log parser is an example implementation of the stateless anomaly detection algorithm. We describe it in detail in Section III.

Log Sequence Anomaly Detector detects anomalous log sequence of an event (or transaction), which consists of a sequence of actions and each action is represented by a log. It is a stateful algorithm which detects malfunctioned events by analyzing abnormal log sequences based on an automata-based model. We describe it in detail in Section IV.

Heartbeat Controller periodically sends heartbeat (i.e., echo or dummy) messages to the log sequence anomaly detector. These messages aid to report anomalies in time and to identify open states in a transaction.

Anomaly Storage stores all anomalies for human validation. Each anomaly has a type, severity, reason, timestamp, associates logs, etc.

Visualization Dashboard provides a graphical user interface and dashboard to the end users. It combines information from log storage, model storage, and anomaly storage to present anomalies to the users. Users can easily view anomalies and take actions to rebuild or edit models. It also allows users to run complex analysis by issuing ad-hoc queries.

Most components described above can be implemented using many different open-source products. *LogLens* uses Spark big data processing framework. It uses Kafka [21] for shipping logs and communicating among different components. For the storage, it uses Elasticsearch [14] a NoSQL database. Elasticsearch provides a very useful query facility that can be used for data exploration. Furthermore, it has close integration with Kibana [22], which provides a tool for building visualization front-ends and writing interactive ad-hoc queries.

Now, we describe our exemplary anomaly detection algorithms in Section III and Section IV, and deployment challenges and solutions in Section V

III. Stateless: Log Parser

For an automated log analysis system, a core step is to parse raw logs and make them structured so that various log analysis tasks could be carried out by leveraging the structured form of the raw logs. *LogLens* parses logs using patterns learned from the systems normal runs. Here, we define *pattern* as a GROK expression [23]. For example, for the log *"Connect DB 127.0.0.1 user abc123"*, one of the matching GROK patterns is *"%{WORD:Action} DB %{IP:Server} user %{NOTSPACE:UserName}"* and after parsing *LogLens* produces {*"Action": "Connect", "Server": "127.0.0.1", "UserName":"abc123"*} as a parsing output in JSON format. Parsed outputs can be used as a building block for designing various log analysis features. For example, our stateful algorithm (see Section IV) uses them to detect log sequence violations.

Challenges. Automatically parsing heterogeneous logs without human involvement is a non-trivial task. *LogLens* parses logs in two phases: 1) it discovers a set of GROK patterns from a set of logs representing system normal runs and 2) it parses logs using these GROK patterns.

Existing log analysis tools either use predefined regular expressions (RegEx) or source-code level information for log parsing [11, 16, 24]. Thus, these tools are supervised and need human involvement – they cannot be used for the first phase. Our earlier work, LogMine [25], shows how to discover patterns without any human involvement by clustering similar logs. LogMine uses tokenized logs and datatypes of the tokens during the similarity computation step. However, identifying some tokens, especially timestamp identification is a very challenging task. In addition, LogMine may fail to meet user needs as it is very hard to automatically infer semantics of a field in the GROK pattern.

In the second phase, we need a tool to parse incoming logs. We can use Logstash [16], an industrial-strength open-source log parsing tool, which can parse logs using GROK patterns. However, we find that Logstash suffers from two severe scalability problems: 1) it cannot handle a large number of patterns and 2) it consumes huge memory (see Section VI-A). Since *LogLens* discovers patterns with no (or minimal) human involvement, it can generate a huge number of patterns which is very problematic for the Logstash.

Solution. *LogLens* provides an efficient solution for identifying timestamps and to meet user expectation it allows users to edit/modify automatically generated GROK patterns. For the fast parsing, *LogLens* transforms both logs and patterns into their underlying datatypes and builds an index for quickly finding the log-to-GROK mapping. Now, we describe log parsing workflow in detail.

A. Model Building

1) Tokenization: LogLens preprocesses a log by splitting it into individual units called *tokens*. Splitting is done based on a set of delimiters. The default delimiter set consists of white space characters (i.e., space, tab, etc.). *LogLens* also allows users to provide delimiters to overwrite the default delimiters in order to meet their needs. In addition, users can provide regular expression (RegEx) based rules to split a single token into multiple sub-tokens. For example, to split the token "123KB" into sub-tokens "123" and "KB", user can provide the following RegEx rule: *"[0-9]+KB"* $\Longrightarrow$ *"[0-9]+ KB"*.

2) Datatype Identification: During this step, for every token *LogLens* identifies its datatype based on the RegEx rules. Table I shows the sample RegEx rules for identifying different datatypes in *LogLens*.

Datatype	Regular Expression (RegEx) Syntax
WORD	[a-zA-Z]+
NUMBER	-?[0-9]+(.[0-9]+)?
IP	[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}
NOTSPACE	\S+
DATETIME	[0-9]{4}/[0-9]{2}/[0-9]{2} [0-9]{2}:[0-9]{2}:[0-9]{2}.[0-9]{3}
ANYDATA	.*

TABLE I: Syntax for various data types. Notation is adopted from the Java Pattern API [26].

Challenge. *LogLens* identifies timestamps and unifies them into a single format *"yyyy/MM/dd HH:mm:ss.SSS"* corresponding to the DATETIME datatype. However, we find that it is a very cumbersome process due to the heterogeneity of timestamp formats used in various logs. For example, timestamp "2016/02/23 09:00:31" can be expressed in "2016/23/02 09:00:31" or "2016/23/02 09:00:31.000" or "Feb 23, 2016 09:00:31" or "2016 Feb 23 09:00:31" or "02/23/2016 09:00:31" or "02-23-2016 09:00:31" and so on. *LogLens* allows users to specify formats to identify timestamp related tokens. It uses Java's SimpleDateFormat [27] notation to specify a timestamp format. However, if users do not specify any format, *LogLens* identifies timestamps based on a set of predefined formats (for example, MM/dd HH:mm:ss, dd/MM HH:mm:ss:SSS, yyyy/MM/dd HH:mm:ss.SSS etc.). Users can also add new formats in the predefined list. The worst case time complexity of identifying timestamp is $O(k)$, where k is the total number predefined formats or the total number of user-specified formats.

Solution. *LogLens* uses the following two optimizations to quickly identify tokens related to the timestamp formats:

- **Caching matched formats.** *LogLens* maintains a cache to track the matched formats. Caching reduces the amortized time complexity to $O(1)$. To identify timestamp related tokens in a log, first, *LogLens* finds if there is a cache hit. In case of a cache miss, *LogLens* checks the non-cached formats and if a match found, then the corresponding format is added to the cache. This simple caching strategy works well in practice as logs from the same (or similar) sources use same formats, and every source uses only a few different formats to record timestamps.
- **Filtering.** *LogLens* maintains a set of keywords based on the most common form of specifying month (i.e., jan-dec, january-december, 01-12, 1-9) , day (i.e., 01-31), and hour (i.e., 00-59), day of the week (i.e., mon-sun, monday-sunday), etc. It uses these keywords to filter out tokens which cannot be related to a timestamp. If a token cannot be filtered, then only *LogLens* checks the predefined formats.

3) Pattern Discovery By Clustering Similar Logs: In this step, *LogLens* clusters preprocessed logs based on a similarity distance using LogMine [25] algorithm. All logs within a cluster are merged together to generate one final pattern in the form of a GROK expression. *LogLens* assigns a field ID for each field. The field ID consists of two parts: 1) the ID of the log pattern that this field belongs to and 2) the sequence number of this field compared to other fields in the same pattern. The log format pattern IDs can be assigned with the integer number *1, 2, 3, ... m* for a log pattern set of size m. The field sequence order can be assigned with the integer number *1, 2, 3, ... k* for a log pattern with k variable fields. For example, for the log "2016/02/23 09:00:31 127.0.0.1 login user1" the corresponding generated GROK pattern would be "%{DATETIME:P1F1} %{IP:P1F2} %{WORD:P1F3} user1".

4) Incorporating Domain Knowledge: LogLens automatically generates patterns, therefore it may not always meet user needs. In addition, users may want to generate patterns from one system and later want to apply them to a different system with some minor modifications. A user may even want to delete some patterns or add new patterns or edit datatypes. To solve these issues, *LogLens* allows users to edit automatically generated patterns. It supports the following editing operations:

- *LogLens* allows users to add the semantic meaning of a field by renaming its generic name. For example, *LogLens* may assign "P1F1" as a generic field name for the "logTime" field, thus it may be difficult for users to interpret the parsed output. By renaming "P1F1" to "logTime", users can easily fix this problem. To ease renaming effort, *LogLens* uses a heuristic based approach to leverage commonly used patterns found in the logs. For example, *LogLens* automatically renames "PDU = %{NUMBER:P1F1}" as "PDU = %{NUMBER:PDU}". If none of the heuristics matches, then only *LogLens* assigns a generic name.

- *LogLens* allows users to specialize a field. For example, a user can specialize "%{IP:P1F2}" by replacing it with the fixed value "127.0.0.1".
- *LogLens* allows users to generalize a specific token value. For example, a user can generalize "user1" to "%{NOTSPACE:userName}" in order to convert it into a variable field.
- *LogLens* allows users to edit datatype definition to include multiple tokens under one field. To support this feature, it introduces the ANYDATA (i.e., wildcard) datatype, which is defined in Table I.

B. Parsing Logs and Anomaly Detection

LogLens uses patterns discovered during modeling stage for parsing logs. If a log does not match with any patterns, then it is reported as an anomaly.

Problem Definition. Log parsing problem using a set of patterns can be formalized as follows: *given a set of m GROK patterns and a set of n logs, find out the log-to-pattern mappings*. A naïve solution scans all m patterns to find a match for every log. This simple algorithm needs on the average $\frac{m}{2}$ comparisons for the matched logs, while for the unmatched logs it incurs m comparisons. So, the overall time complexity is $O(mn)$. *LogLens* aims to reduce the number of comparisons to $O(1)$, thus the overall time complexity reduces to $O(n)$.

Solution Sketch. *LogLens* leverages the fact that logs and patterns have the common underlying datatypes representing their structures, thus it can build an index based on these structures to quickly find the log-to-pattern mapping. *LogLens* maintains an index in order to reduce the number of comparisons by using the following three steps:

1) **Finding candidate-pattern-group.** To parse a log, *LogLens* first generates a *log-signature* by concatenating the datatypes of all its tokens. For example, for the log "2016/02/23 09:00:31.000 127.0.0.1 login user1" the corresponding *log-signature* would be "DATETIME IP WORD NOTSPACE". Next, *LogLens* finds out if there is a *candidate-pattern-group* which can parse the *log-signature*.
2) **Building candidate-pattern-group.** If no group found, first *LogLens* builds a *candidate-pattern-group* by comparing an input logs *log-signature* with all GROK m patterns using their *pattern-signatures* (explained later) to find out all potential candidate patterns and put all candidate patterns in one group. In a group, patterns are sorted in the ascending order of datatype's generality and length (in terms of number of tokens). If no candidate pattern is found, then the *candidate-pattern-group* is set to empty. Next, *LogLens* adds this group in a hash index using *log-signature* as the "key", and *candidate-pattern-group* as the "value". Finally, it follows Step 3.
3) **Scanning the candidate-pattern-group.** If a *candidate-pattern-group* is found, *LogLens* scans all patterns in that group until the input log is parsed. If an input log cannot be parsed or group has no patterns (i.e., empty), then *LogLens* reports it as an anomaly.

Pattern-Signature Generation. *LogLens* generates a *pattern-signature* from each GROK pattern as follows. First, it splits a pattern into various tokens separated by white space characters. Next, it replaces every token by its datatype. For example, the token "%{DATETIME:P1F1}" is replaced by its datatype "DATETIME". If datatype is not present in the token, then *LogLens* finds out the datatype of the token's present value. For example, the token "user1" is replaced by "NOTSPACE" by using the RegEx rule defined in Table I. Thus, the *pattern-signature* of the GROK pattern "%{DATETIME:P1F1} %{IP:P1F2} %{WORD:P1F3} user1" would be "DATETIME IP WORD NOTSPACE".

How to compare log-signature with pattern-signature? If a *log-signature* is parsed by a *pattern-signature*, then corresponding GROK pattern is added to the *candidate-pattern-group*. There are two cases to consider for the *pattern-signature*: without and with the ANYDATA datatype (i.e., wildcard). The first case (i.e., without) is easy to handle, while the second case is challenging due to the variability arising from the presence of wildcard. *LogLens* solves this problem with a dynamic programming algorithm. It can be formally defined as follows: given a *log-signature* of length r tokens, $L =< l_1, l_2, ..., l_r >$ and a *pattern-signature* of length s tokens, $P =< p_1, p_2, ..., p_s >$, we have to find out if L can be matched by P. Let us define $T[i, j]$ to the boolean value indicating whether $< l_1, l_2, ..., l_i >$ is parsed by $< p_1, p_2, ..., p_j >$ or not. This matching problem has optimal substructure, which gives the following recursive formula:

$$T[i,j] = \begin{cases} true & \text{if } i = 0 \text{ and } j = 0 \\ T[i-1, j-1] & \text{if } l_i = p_j \text{ or } isCovered(l_i, p_j) \\ T[i-1, j] \mid\mid T[i, j-1] & \text{if } p_j = * \end{cases}$$

Here, $isCovered(l_i, p_j)$ is a function, which returns true if the RegEx definition corresponding to l_i's datatype is covered by the RegEx definition of the p_j's datatype. For example, isCovered("WORD", "NOTSPACE") returns true. In contrast, isCovered("NOTSPACE", "WORD") returns false. Based on the above formulation, *LogLens* uses dynamic programming to compute the solution in a bottom-up fashion as outlined in Algorithm 1. If $T[r, s]$ is true, then *LogLens* adds the GROK pattern corresponding to P in the *candidate-pattern-group*.

IV. Stateful: Log Sequence Anomaly Detector

Log sequence anomaly detector detects abnormal log sequence in an event (or transaction). Here, we define an event as follows: *an event is an independent operational work unit of*

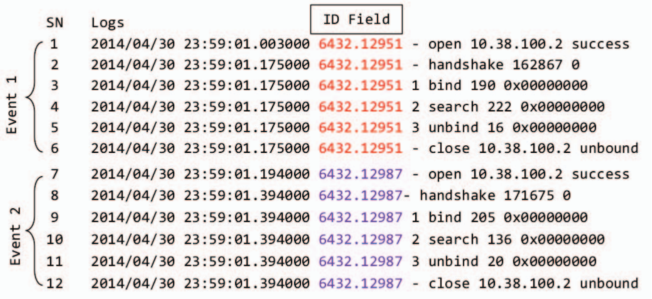
Algorithm 1 Dynamic Programming Algorithm

```
procedure isMatched
    Input: String logSignature, String patternSignature
    Output: boolean (i.e., true/false)
    String L[] = logSignature.split(" ");
    String P[] = patternSignature.split(" ");
    boolean T[][] = new boolean[L.length+1][P.length+1];
    T[0][0] = true;
    for (int i = 1; i < T.length; i++) do
        for (int j = 1; j < T[0].length; j++) do
            if (L[i-1].equals(P[j-1])) then
                T[i][j] = T[i-1][j-1];
            else if (P[j-1].equals("ANYDATA")) then
                                        ▷ Handling wildcard case
                T[i][j] = T[i-1][j] || T[i][j-1];
            else if (isCovered(L[i-1], P[j-1])) then
                              ▷ Is log-token covered by pattern-token?
                T[i][j] = T[i-1][j-1];
            end if
        end for
    end for
     return T[L.length][P.length];
end procedure
```

a business process with a finite action sequence. For example, cloud data center provides users to access, instantiate virtual machines, assign resources, and so on. It executes each of the above tasks by coordinating multiple internal services distributed on different servers. It generates logs for each of the action (i.e., start VMs, contact scheduler, resource manager, hypervisors, etc.) forming an event. Thus, each event has logs from multiple sources following a sequence. The malfunctioning event follows unusual/deviated action sequence, which may lead to system failure. However, traditional stateless anomaly detection techniques dealing with individual logs do not catch these failures because individual logs may not be anomalous although altogether they follow an abnormal sequence. This requires a stateful algorithm to dig abnormal event log sequences by storing all intermediate log sequence information in memory/state.

Challenge. Logs in an event may not be always homogeneous. Thus, detecting anomalous log sequence from incoming logs is a challenging problem as it requires to discover events and to preserve log sequence information (i.e., state) in memory. Most of the log sequence anomaly detectors [11, 28, 29, 30, 31] are supervised as they need human input for discovering events and do not work for heterogeneous logs. Some research works use unsupervised learning [4, 5, 10], but they are not domain-agnostic to handle heterogeneous logs. Here, our goal is to design a generic unsupervised algorithm for handling heterogeneous logs.

Solution. *LogLens* describes a log sequence based anomaly detection algorithm that discovers event automatically using a finite state automaton (FSA) based model. *LogLens* has two phases: learning and detection. During the learning phase, it builds a model that captures the normal behavior of an event. First, it discovers event ID Fields automatically from the heterogeneous logs by leveraging the fact that log parser has already identified the log format and parsed logs into various fields. Next, it builds automata which have rules that represent the normal event log sequences. For example, Figure 2 shows some logs representing two events with event ID Field discovered by *LogLens*. Figure 3 shows corresponding automata with discovered rules, where each state corresponds to a log in an event. During detection phase, *LogLens* uses this automatically discovered automata model to detect anomalies. Now, we briefly describe these two phases.



```
SN  Logs
1   2014/04/30 23:59:01.003000 6432.12951 - open 10.38.100.2 success
2   2014/04/30 23:59:01.175000 6432.12951 - handshake 162867 0
3   2014/04/30 23:59:01.175000 6432.12951 1 bind 190 0x00000000
4   2014/04/30 23:59:01.175000 6432.12951 2 search 222 0x00000000
5   2014/04/30 23:59:01.175000 6432.12951 3 unbind 16 0x00000000
6   2014/04/30 23:59:01.175000 6432.12951 - close 10.38.100.2 unbound
7   2014/04/30 23:59:01.194000 6432.12987 - open 10.38.100.2 success
8   2014/04/30 23:59:01.394000 6432.12987- handshake 171675 0
9   2014/04/30 23:59:01.394000 6432.12987 1 bind 205 0x00000000
10  2014/04/30 23:59:01.394000 6432.12987 2 search 136 0x00000000
11  2014/04/30 23:59:01.394000 6432.12987 3 unbind 20 0x00000000
12  2014/04/30 23:59:01.394000 6432.12987 - close 10.38.100.2 unbound
```

Fig. 2: Sample event trace logs.

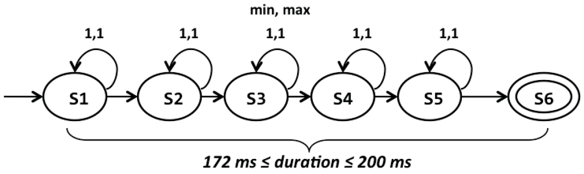


Fig. 3: Sample automaton for an event from the logs of Figure 2. It has the rule of min/max occurrence of each state s and min/max time duration of an event. Each state corresponds to a log in that event.

A. Model Building

1) Automatic Event ID Field Discovery: Log Parser (described in Section III) parses logs and sends them to the log sequence anomaly detector. Each parsed log has a log pattern and a field set. *LogLens* discovers a unique ID Field from these parsed logs in an event by leveraging the fact that ID appears the same in multiple logs in an event. *LogLens* uses a variant of the Apriori based [32] technique, however the key challenge for *LogLens* is to discover events from a large volume of logs with varying formats. *LogLens* has following two main steps:

- **Building a reverse index.** *LogLens* builds a reverse index of log fields based on their field content. First, it extracts all field contents from a parsed log. Next, it builds a reverse index table. Each field content is a key and a list of logs with (log pattern, field) pair as a value in the table.
- **ID Field discovery.** *LogLens* discovers ID Field for all possible log patterns by scanning the reverse index. For each

possible event ID content, it builds a list of (log pattern, field) pair for all logs that have this ID content. This gives multiple lists and *LogLens* builds a set of unique lists. If any list covers all log patterns discovered in the training logs, then *LogLens* assigns it to an event ID Field.

2) Event Automata Modeling: In this step, *LogLens* profiles automata with rules from logs using ID Fields. It scans through each log and extracts its ID Field and content. *LogLens* also keeps track of the log arrival time. For an ID Field content, it keeps a sorted list of log patterns with their fields. Finally, it merges them and builds automata with rules. An automaton is built with states. Each log pattern with its ID Field is a state which stores log arrival time, the number of occurrences, etc. Each automaton has a begin, an end, and multiple intermediate states. *LogLens* also tracks the occurrence of the intermediate states and the duration between the begin and the end state. After building automata, *LogLens* profiles the minimum and maximum of those statistics (min/max duration of an event, min/max occurrence of the intermediate states, etc.) and uses them as rules for detecting anomalies.

B. Anomaly Detection

LogLens collects incoming logs in real-time. First, it extracts log pattern and ID from each log. It groups all logs that have a common Field ID. After that, it sorts logs in each group based on their arrival time – this gives incoming log sequence in an event. Next, it scans logs in each group and validates them against the automata rules discovered during model learning. Logs in a sequence will be flagged as anomalies if they violate any of these rules. Table II shows various anomaly types reported by *LogLens*.

Type	Anomaly
1	Missing begin/end state
2	Missing intermediate states
3	Min/Max occurrence violation of the intermediate stats
4	Min/Max time duration violation in between the begin state and the end state

TABLE II: Sample log sequence anomalies.

V. *LogLens* AS A SERVICE: CHALLENGES AND SOLUTIONS

In this section, we highlight two key real-world deployment challenges that we encountered when implementing *LogLens* as a service using Spark [17]. We believe that these challenges and our proposed generic solutions will offer insights for building similar services in the near future.

A. Supporting Dynamic Model Updates

Challenges. Spark's data parallel execution model uses *broadcast* variables to load models and distribute data to all workers. However, broadcast variables have been designed to be immutable and can only be updated before data stream processing is started. The only possible way to update a model in Spark is to re-initialize and re-broadcast the model data to all workers. Unfortunately, this process can lead to drastic consequences: 1) it introduces a downtime of several seconds, if not minutes, depending on the size of the cluster; 2) restarting the cluster requires rescheduling and redistribution of data and memory leading to significant decrease in the throughput of the cluster; and 3) if a stateful Spark streaming service is terminated, all the state data is lost and losing states can have significant impact on the efficacy of the anomaly detection algorithms. To eliminate any possibility of downtime or loss of state, the model update mechanism should meet at least the following two requirements: 1) service must be up, and running all the time and 2) states must be preserved during model updates.

Solution. In *LogLens*, to update a broadcast variable (BV) at runtime, we modify Spark internals with minimum possible changes. Our solution is capable of rebroadcasting the immutable BVs at runtime without job termination. BVs is a serializable data object that is a virtual data block containing a reference to the actual disk block where the variable resides. When a BV is used in a Spark program, it is shipped to each individual worker. During execution, whenever a worker requests the value of a BV using *getValue()* method, Spark first looks into the local data-block-cache of the worker for the variable. If there is a cache miss, it sends a pull request to the driver (where the variable is initially stored) to get the value over the network. Once this variable is received, it is stored on the local-disk-block-cache of that worker. From now and so on, this cached value of the variable will be used whenever a *getValue()* method is called.

To rebroadcast a BV which already resides in the local-disk-block-cache of individual workers, *LogLens* invalidates all locally cached values. Thus, whenever *getValue()* method is called for that BV, a pull request is made to the driver. At the driver, when a pull request is received rather than handing over the old value, the driver sends the updated value. The worker then receives the updated value and stores it in the local cache. From now and so on, the newly fetched local copy of the BV will be used.

Whenever a new model is issued from the model manager, it is read from the model storage and enrolled into a queue. The scheduler then waits for the current job to finish. *LogLens*'s dynamic model update implementation communicates with the block manager of each worker as well as the driver. It also tracks all BV identifiers to maintain the same ID for the updated BV which is otherwise incremented at each update. This allows workers to retrieve the original BV after cache invalidation. Furthermore, *LogLens*'s implements a thread-safe queuing mechanism to avoid any race conditions due to the extreme parallelization of the Spark jobs.

Spark data processing is a queue-based execution of the data received in every micro-batch. In *LogLens*, model update operation runs between these micro-batches in a serialized lock process. The model data is loaded into memory, and an in-memory copy operation loads the data to the BV. The execution proceeds as normal and whenever the broadcast value is required, workers fetch a fresh copy from the master. The only blocking operation is the in-memory copy operation, and hence the overhead is directly dependent on the size of the model. In practice, we find that this overhead is negligible and it does not incur any slow-down on *LogLens*.

B. Implementing Stateful Algorithms Efficiently

Expedited Anomaly Detection. *LogLens* focuses on the real-time anomaly detection, thus it is essential to report anomalies as soon as they occur. At the same time to allow for scalable and fast execution, *LogLens* uses data-parallel algorithms to distribute the processing workload of incoming logs across worker nodes. Data partitioning logic is only constrained by grouping together logs which have an inherent causal dependency on each other (i.e., same model, source, etc.) – this allows *LogLens* to optimize performance and to avoid performance bottlenecks as much as possible.

In a stateful anomaly detection, each log is independent of the other, thus if a log comes, then anomalies can be reported to the user. However, several real-world issues are potentially problematic especially in the case of stateful anomalies, which depend on the previous states. Two of these issues are:

1) What if a transaction fails and no log comes at all from a source or for a particular key or pattern of the model? Essentially, the saved state is already "anomalous", but cannot be reported since we have no concrete log as evidence. In this case, the anomaly would never be reported.
2) Similarly if logs of certain automata are coming very infrequently (several hours apart). This could be because of overload in the target system. In such a scenario, the anomaly may not be reported immediately for any countermeasures to be taken.

Traditional timeout based approaches cannot be used as they use system time, which can be very different from *"log time"*. The log timestamps may come faster or slower than the actual time progress within the *LogLens* system. Hence only the log rate of embedded timestamps within the logs can be used to predict timestamps in the absence of logs. Furthermore, the key based mapping of states only allows similar keys to access or modify the state. Even if somehow *LogLens* receives an event that informs the program logic to flush the unnecessary states, there is currently no way to access the states without their keys.

Solution. To allow for expedited real-time anomaly detection, *LogLens* uses an external *heartbeat controller*. This controller generates a heartbeat message for every log source and periodically sends it to the anomaly detectors if the corresponding log agent is still active. The heartbeat message is embedded with a timestamp based on the last log observed and the rate of logs from that source. Hence in the absence of logs, the heartbeat message provides the current time of the target systems and allows *LogLens* to proceed with the anomaly detection.

Efficient State Management. To enable efficient memory management of the open states, *LogLens* extends the Spark API (v1.6.1) to expose the reference of the state in a partition to the program logic. Within the program logic, the state-map can be accessed by calling the *getParentStateMap()* method on the state object. This method returns the reference to the state-map object where all the states of that partition are stored. For anomaly detection, this state-map is enumerated to find the states that are open and expired with respect to the current log time. Although *LogLens* does not have the key to an open state, it can still access that state and reports anomalies which would otherwise go entirely undetected. However, because of event-driven nature of the Spark's stream processing, *LogLens* still needs a trigger on all partitions to handle the infrequent log arrival scenario.

Solution. As a remedy, *LogLens* also leverages the external *heartbeat controller* and a custom partitioner. *LogLens* periodically receives heartbeat messages from this controller to trigger the expired state detection procedure. This external message is sent to the same data channel (where logs arrive) with a specific tag to indicate that it is a heartbeat message. If such a message is observed in the program logic, the custom partitioner kicks in and broadcasts the same heartbeat message to all partitions. Whenever a heartbeat message is received, an anomaly detection algorithm iterates over its states to detect anomalies and clean up expired states. This procedure is performed at all the partitions on every worker since the heartbeat message is duplicated and broadcast to each partition on the data channel.

VI. Experimental Results

The goal of this section is to show experimental results to evaluate the functionality and effectiveness of *LogLens*.

Dataset. We use six different datasets covering various data-center operations for evaluation as shown in Table III. In the table, we have proprietary dataset D1 of trace log of a data center (Figure 2 shows sample logs), a synthetic dataset D2, storage server based dataset D3, OpenStack based dataset D4 for infrastructure as a service deployment, PCAP based dataset D5, and proprietary D6 dataset covering network operations. We simulate these datasets as streams in our *LogLens* system.

Dataset	Type	Total logs	
		Training	Testing
D1	Trace log	16,000	16,000
D2	Synthetic	18,000	18,000
D3	Storage Server	792,176	NA
D4	OpenStack [33]	400,000	NA
D5	PCAP [34]	246,500	NA
D6	Network	1,000,000	NA

TABLE III: Evaluation Dataset.

Experimental setup. We perform our tests on a Spark cluster with Spark Streaming. Our cluster has one master and eight worker nodes. We use Spark version 1.6.1 with Kafka version 0.9.0.1. For replaying log data, we have developed an agent, which emulates the log streaming behavior.

A. Log Parser

Fast Timestamp Identification. *LogLens* has 89 predefined timestamp formats in the knowledge-base. From our experiments using datasets in Table III, we find that by combining both caching and filtering, *LogLens* can detect timestamps up to 22x faster than the linear scan-based solution – 19.4x is contributed by caching, and the rest is contributed by filtering.

Fast Log Parsing. We compare *LogLens* against Logstash [16], a popular open-source log parsing tool, to show its efficiency. For these experiments, we use D3, D4, D5, and D6 datasets which use the same set of logs in both training and testing phases for sanity checking – a correct parser does not produce any anomalies for these datasets. Using LogMine [25] algorithm, first, we generate a set of GROK patterns from the training logs; next, we parse testing logs using these patterns; and we expect that every testing log matches a GROK pattern as testing logs are same as the training logs. Table IV shows that *LogLens* runs up to 41x times faster than Logstash (v5.3.0), and handles large number of patterns. Both *LogLens* and Logstash parse all training logs and produce same parsing results. For the D4 and D6 datasets, Logstash did not generate any output even running for more than 48 hours, and eventually we stopped it. The main reason is as follows: D4 and D6 datasets produce 3234 and 2012 patterns, respectively and Logstash is not suitable for handling such large pattern-sets.

Dataset	Total Patterns	Running Time		
		LogLens	Logstash	Improvement
D3	301	109 sec	4550 sec	4074.31%
D4	3234	72 sec	NA	NA
D5	243	34 sec	588 sec	1629.41%
D6	2012	170 sec	NA	NA

TABLE IV: Results:*LogLens* vs. Logstash.

B. Log Sequence Anomaly Detector

The effectiveness of *LogLens* in easing human log analysis burden requires detecting anomaly accurately. We also need to verify that heartbeat controller helps to report anomalies in real-time, and model controller instantly reflects system behavior changes after a model update operation. Since log sequence anomaly detector uses the output from the log parser, consequentially our evaluation also demonstrates log parser's efficacy in building advanced log analytics applications.

Accuracy. We use D1 and D2 to evaluate the accuracy of the log sequence anomaly detector because we have ground truth form them. Figure 4 shows that D1 has originally 21 anomalous sequences, and our detector identifies all of them; D2 has originally 13 anomalous sequences, and our detector identifies all of them (red bar). Thus, for both datasets, we get 100% recall.

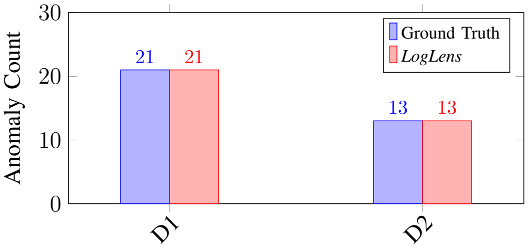


Fig. 4: Log sequence anomaly detector results accuracy.

Heartbeat Controller. In *LogLens*, heartbeat (HB) controller controls open states and helps to report *missing end state anomaly* immediately. With HB controller, we expect to report more anomalies as soon as they occur. Figure 5 shows performance result of our HB controller. For a certain time period, we run our anomaly detector on D1 and D2. If we do not use HB controller, we detect 20 anomalies for D1 and 10 anomalies for D2. However, when with HB controller, we detect 21 anomalies for D1 and 13 anomalies for D2, and all of these extra anomalies are related to the *missing end states*. These results demonstrate that HB controller is effective in immediately reporting anomalies.

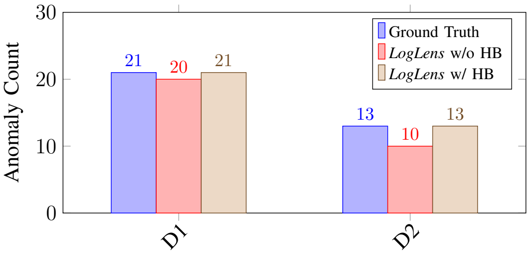


Fig. 5: Anomaly detection with and without heartbeats

Model Controller. *LogLens* provides a key feature like the model update as a service. It can add, update, or delete models without restarting the running service. The goal of this experiment is to show that the number of anomaly changes after the model update in order to verify model controller's functionality. Now, we run two set of experiments. First, we build models using training logs of D1 and D2. D1's model has two automata, while D2 has three automata. Using these models, we detect 21 anomalies for D1, and 13 anomalies for D2. Next, we modify both models by deleting an automaton from them, update models through the model controller without service interruption, and rerun testing. Table V shows that deleting an automaton reduces the number of anomalies from 21 to 13 for D1, and 13 to 9 for D2. This behavior matches with our intuition as the second set of experiments will produce fewer anomalies because they have fewer automata rules. Therefore, these two set of experiments validate the functionality of the model update operation.

VII. Real-world Case Studies

A. Analyzing Custom Application Logs

In this case-study, users want to analyze logs from a custom application. These logs record SQL queries issued in the

Dataset	Total Automata	Anomaly Count	Total Automata (after delete)	Anomaly Count (after delete)
D1	2	21	1	13
D2	3	13	2	9

TABLE V: Anomaly detection using model updates.

system to get various statistics used by other applications. Table VI shows a set of sample logs. These logs are extremely complex, and it took one week for the users to manually generate patterns to parse all logs in order to understand log characteristics. In contrast, *LogLens* generated 367 patterns in 50 seconds, which reduces humans effort by 12096x and aids them to instantly focus on the log analysis tasks.

```
(0):GetFormControl():2[25 21:39:21] SQL SELECT TABLE: tblFormControl WHERE: tblFormControl.oFCID='6a602aaa-9afd-4e2c-95e9-ee900dde4b50'

(0):_GetObjects():2[25 21:39:21] SQL SELECT TABLE: tblContent WHERE: oPID='ad1aa290-01ae-4edd-989c-1cee2ba63707'
AND ((((((oID IN (SELECT oID FROM tblFormData WHERE oFORMINSTID=tblContent.oID AND oFCID='6a602aaa-9afd-4e2c-95e9-ee900dde4b50'
AND ((tblFormData.tValue IS NOT NULL AND tblFormData.tValue ¿ '1799-01-01T00:00:00.000' AND tblFormData.tValue ¡ '2200-01-01T00:00:00.000'
)))))AND (((nType!=15 OR oID IN (SELECT oFORMINSTID FROM tblFormInstance WHERE tblFormInstance.oFORMID=
'3ebee358-2087-43d4-908b-df9ed04e74cc')) AND (nType!=14 OR tblContent.oID='3ebee358-2087-43d4-908b-df9ed04e74cc'))) AND ((oID IN
(SELECT oID FROM tblFormData WHERE oFORMINSTID=tblContent.oID AND oFCID='7e68b547-0869-4a56-a664-26b32d0b5804' AND
((tblFormData.tValue¡='2017-10-26T03:59:59.000' OR tblFormData.tValue IS NULL))))) AND ( (oID IN (SELECT oID
FROM tblFormData WHERE oFORMINSTID=tblContent.oID AND oFCID='e28c6d82-532d-4618-a0a8-d62a15e00731' AND
(tblFormData.sValue=N'dadf4506-2995-42c4-8616-cb43786fa382')))) AND ((oID IN (SELECT oID FROM tblFormData WHERE oFORMINSTID=tblContent.oID
AND oFCID='2a004b8d-16ef-4973-8ec8-be7db392e436' AND ((tblFormData.sValue¡¿N'Y' OR tblFormData.sValue IS NULL)))))))) AND (nType!=15 OR oID IN
(SELECT oFORMINSTID FROM tblFormInstance WHERE oFORMID='3ebee358-2087-43d4-908b-df9ed04e74cc')) AND 1=1 AND (nType=15)) AND(oID IN
(SELECT oID FROM tblPerm WHERE (oGrantID='dadf4506-2995-42c4-8616-cb43786fa382' OR oGrantID='[Authenticated]' OR oGrantID='[Anonymous]'
OR oGrantID IN (SELECT oParent FROM tblMembership WHERE oChild='dadf4506-2995-42c4-8616-cb43786fa382'))
AND fRead=1) ) AND (nSubType!=2 AND nSubType!=1 AND nSubType!=4 AND nSubType!=5) AND (nType!=15 OR nVersion!=0)

(0):_GetObjects():2[25 21:39:21] SQL SELECT TABLE: tblFormData WHERE: oFORMINSTID = '418f38ce-a35e-47db-8e1c-88fc7eb09de3' AND oFCID
IN ('fe53e626-13ae-4206-8bc7-178cbc69b866', '6a602aaa-9afd-4e2c-95e9-ee900dde4b50', '1bfb5785-4f29-488b-8d09-c42faef48fee',
'611e6c07-c8ba-44c5-b745-485e9faddcb4', '3d8dfd3d-2c62-4c19-8cb5-a3bec8bf729b', '7e68b547-0869-4a56-a664-26b32d0b5804')
```

TABLE VI: Custom application logs sample

B. Discovering Security Attacks

In this case-study, we report spoofing attacks [35] discovered by the log sequence anomaly detector of *LogLens* from a set of Signaling System No. 7 (SS7) logs [36]. This dataset contains 2.7 million SS7 logs spanning three hours from 2016/05/09 10:00:00 to 2016/05/09 13:00:00. For the model learning, users take the first two hours of SS7 logs as training data. With this learned model, they discover anomalies in the remaining one hour of SS7 logs.

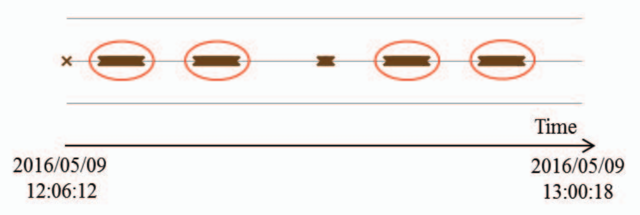
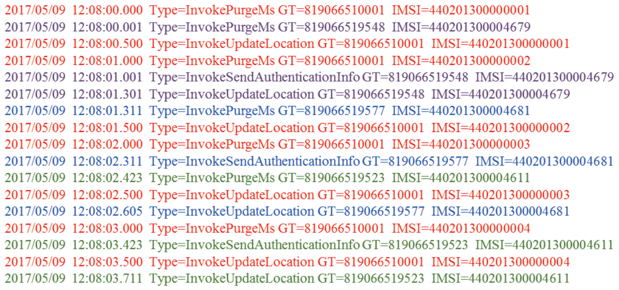


Fig. 6: *LogLens* detects anomalies from SS7 security log data: four clusters of anomalies highlighted by red circles cover logs that indicate spoofing attacks.

As shown in Figure 6, the anomalies form multiple clusters including in total 994 anomalies. In each cluster, its anomalies are temporally close to each other. In practice, such anomaly clusters usually serve as indicators for the significant system events. By manually investigating the logs in the clusters, users find that the abnormal logs are traces left by potential SS7 spoofing attacks.



```
2017/05/09  12:08:00.000  Type=InvokePurgeMs GT=819066510001  IMSI=440201300000001
2017/05/09  12:08:00.001  Type=InvokePurgeMs GT=819066519548  IMSI=440201300004679
2017/05/09  12:08:00.500  Type=InvokeUpdateLocation GT=819066510001  IMSI=440201300000001
2017/05/09  12:08:01.000  Type=InvokePurgeMs GT=819066510001  IMSI=440201300000002
2017/05/09  12:08:01.001  Type=InvokeSendAuthenticationInfo GT=819066519548  IMSI=440201300004679
2017/05/09  12:08:01.301  Type=InvokeUpdateLocation GT=819066519548  IMSI=440201300004679
2017/05/09  12:08:01.311  Type=InvokePurgeMs GT=819066519577  IMSI=440201300004681
2017/05/09  12:08:01.500  Type=InvokeUpdateLocation GT=819066510001  IMSI=440201300000002
2017/05/09  12:08:02.000  Type=InvokePurgeMs GT=819066510001  IMSI=440201300000003
2017/05/09  12:08:02.311  Type=InvokeSendAuthenticationInfo GT=819066519577  IMSI=440201300004681
2017/05/09  12:08:02.423  Type=InvokePurgeMs GT=819066519523  IMSI=440201300004611
2017/05/09  12:08:02.500  Type=InvokeUpdateLocation GT=819066510001  IMSI=440201300000003
2017/05/09  12:08:02.605  Type=InvokeUpdateLocation GT=819066519577  IMSI=440201300004681
2017/05/09  12:08:03.000  Type=InvokePurgeMs GT=819066510001  IMSI=440201300000004
2017/05/09  12:08:03.423  Type=InvokeSendAuthenticationInfo GT=819066519523  IMSI=440201300004611
2017/05/09  12:08:03.500  Type=InvokeUpdateLocation GT=819066510001  IMSI=440201300000004
2017/05/09  12:08:03.711  Type=InvokeUpdateLocation GT=819066519523  IMSI=440201300004611
```

Fig. 7: SS7 attacks revealed by *LogLens* anomalies

Figure 7 shows a few examples of abnormal logs. In particular, the abnormal logs are marked as red, while the logs of other colors are normal ones. As shown in these examples, normal logs follow SS7 protocol with a sequential pattern *"InvokePurgeMs"* → *"InvokeSendAuthenticationInfo"* → *"InvokeUpdateLocation"*. Unlike normal logs, the traces left by spoofing attacks form a different sequence *"InvokePurgeMs"* → *"InvokeSendAuthenticationInfo"* without ending "InvokeUpdateLocation" because attackers only want to guess credentials of the legitimate users without finishing up the full protocol. Anomaly clusters are formed because attackers launched intensive spoofing attacks, which result in a large number of log sequence violations in a short period of time.

It is very difficult even for the domain experts to manually check all 1 million SS7 logs for investigating security attacks. In this empirical study, it took them 2 days to perform such investigation. In contrast, *LogLens* reported abnormal logs that were related to the spoofing attacks in 5 minutes without any domain knowledge, thus it saved 576x of man-hours.

VIII. Conclusion and Lesson Learned

LogLens provides a blueprint for implementing a real-time log analysis system by leveraging unsupervised machine learning based techniques. It can detect anomalies with no (or minimal) human involvement and can easily adapt to the system behavior change. It deploys log analysis as a service by enhancing the Spark [17] big data processing framework. *LogLens* classifies anomaly detection algorithms into stateless and stateful categories and provides a reference implementation for both categories. The stateless log parser runs up to 41x faster than the state-of-the-art Logstash [16] tool and the log analytic applications save up to 12096x man-hours in diagnosing real-world operational problems. Finally, we share one of the key lessons learned from our interaction with the real users: *when designing an automated system instead of no human involvement, we need to focus on the minimization of human involvement*. One of the main problems of learning models from a training dataset (i.e., data-driven approach) is that in many cases this dataset may not cover all the possible use-cases that users want to monitor. Therefore, we have to provide options to users for incorporating their domain knowledge during model building as well as allow them to edit automatically generated models to improve the accuracy of the anomaly detection results.

References

[1] S. Alspaugh, B. Chen, J. Lin, A. Ganapathi, M. Hearst, and R. Katz, “Analyzing log analysis: An empirical study of user log mining,” in *LISA14*, 2014, pp. 62–77.

[2] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy, “The unified logging infrastructure for data analytics at twitter,” *VLDB*, vol. 5, no. 12, pp. 1771–1780, 2012.

[3] LATK, “Log Analysis Tool Kit,” http://www.cert.org/digital-intelligence/tools/latke.cfm, Aug. 2017.

[4] M. Du, F. Li, G. Zheng, and V. Srikumar, “DeepLog: anomaly detection and diagnosis from system logs through deep learning,” in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[5] C. C. Michael and A. Ghosh, “Simple, state-based approaches to program-based anomaly detection,” *ACM Transactions on Information and System Security*, vol. 5, no. 3, Aug. 2002.

[6] E. Analyzer, “An IT Compliance and Log Management Software for SIEM,” https://www.manageengine.com/products/eventlog/, Aug. 2017.

[7] PlantLog, “Operator Rounds Software,” https://plantlog.com/, Aug. 2017.

[8] LogEntries, “Log Analysis for Software-defined Data Centers,” https://blog.logentries.com/2015/02/log-analysis-for-software-defined-data-centers/, Feb. 2015.

[9] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, “Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs,” in *ASPLOS’16*. ACM, 2016, pp. 489–502.

[10] Q. Fu, J. G. Lou, Y. Wang, and J. Li, “Execution anomaly detection in distributed systems through unstructured log analysis,” in *Data Mining, 2009. ICDM’09. Ninth IEEE International Conference on*, 2009.

[11] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *ACM SIGOPS*. ACM, 2009, pp. 117–132.

[12] S. He, J. Zhu, P. He, and M. R. Lyu, “Experience report: System log analysis for anomaly detection,” *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 207–218, 2016.

[13] Splunk, “Turn Machine Data Into Answers,” https://www.splunk.com, Aug. 2017.

[14] ElasticSearch, “Open-Source Log Storage,” Aug. 2017. [Online]. Available: https://www.elastic.co/products/elasticsearch

[15] Gartner, “Iot forecast,” http://www.gartner.com/newsroom/id/3598917, Aug. 2017.

[16] Logstash, “Log Parser,” Aug. 2017. [Online]. Available: https://www.elastic.co/products/logstash

[17] A. Spark, “Lightning-fast cluster computing,” http://spark.apache.org/, Aug. 2017.

[18] F. Yang, J. Li, and J. Cheng, “Husky: Towards a more efficient and expressive distributed computing framework,” *VLDB Endowment*, vol. 9, no. 5, pp. 420–431, 2016.

[19] A. Flink, “Scalable Stream and Batch Data Processing,” https://flink.apache.org/, Aug. 2017.

[20] Samza, “Distributed stream processing framework,” http://samza.apache.org/, Aug. 2017.

[21] Message-Broker, “Apache kafka,” http://kafka.apache.org/, Aug. 2017.

[22] Kibana, “Visualization tool,” Aug. 2017. [Online]. Available: https://www.elastic.co/products/kibana

[23] GROK, “Pattern,” Aug. 2017. [Online]. Available: https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html

[24] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “Sherlog: error diagnosis by connecting clues from run-time logs,” in *ACM SIGARCH*, vol. 38, no. 1. ACM, 2010, pp. 143–154.

[25] H. Hamooni, B. Debnath, H. Xu, Jianwu amd Zhang, G. Jiang, and A. Mueen, “Logmine: Fast pattern recognition for log analytics,” in *CIKM*. ACM, October 2016.

[26] “Java regex,” https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html, Aug. 2017.

[27] RegEx-Format, “Java SimpleDateFormat,” https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html, Aug. 2017.

[28] C. Ezeife and D. Zhang, “Tidfp: mining frequent patterns in different databases with transaction id,” in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2009, pp. 125–137.

[29] “DISCO,” https://fluxicon.com/disco/.

[30] G. Cugola and A. Margara, “Processing flows of information: From data stream to complex event processing,” vol. 44, no. 3. ACM, 2012, p. 15.

[31] A. Margara, G. Cugola, and G. Tamburrelli, “Learning from the past: automated rule generation for complex event processing,” in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 47–58.

[32] I. Tudor, “Association rule mining as a data mining technique,” *Seria Matematic Informatic Fizic Buletin*, vol. 1, pp. 49–56, 2008.

[33] IaSS, “Openstack,” https://en.wikipedia.org/wiki/OpenStack, Aug. 2017.

[34] PCAP, “Packet Capture,” https://en.wikipedia.org/wiki/Pcap, Aug. 2017.

[35] Spoofing-Attack, “Spoofing attack,” https://en.wikipedia.org/wiki/Spoofing_attack, Aug. 2017.

[36] SS7, “Signaling system no. 7,” https://en.wikipedia.org/wiki/Signalling_System_No._7, Aug. 2017.