Online Log Data Analysis With Efficient Machine Learning: A Review

Florian Skopik, Max Landauer, and Markus Wurzenberger | Austrian Institute of Technology

Logs are incrementally produced textual data that reflect events and their impact on technical systems. Their efficient analysis is key for operational cybersecurity. We investigate approaches beyond applying simple regular expressions and provide insights into novel machine learning mechanisms for parsing and analyzing log data for online anomaly detection.

Log files capture information about almost all events that take place in a system. Historic logs enable forensic analysis of past events and give system administrators a means to trace the roots of observed problems. Moreover, logs may help to recover to a nonfaulty state, reset incorrect transactions, restore data, and replicate scenarios that lead to erroneous states. Storing logs is typically inexpensive since the files can be effectively compressed. However, a major issue with forensic log analysis is that problems are detected only in hindsight; thus, modern approaches in cybersecurity shift from purely forensic to proactive analysis.[1] This enables timely responses and, in turn, reduces costs caused by incidents and cyberattacks.

When considering large enterprise systems, it is not uncommon that the number of daily produced log lines is in the millions. Clearly, this makes manual analysis impossible, and it thus stands to reason to employ machine learning algorithms that automatically process the lines and recognize interesting patterns that are then presented to system operators in condensed form. Literally hundreds of different machine learning approaches have been proposed during the past decades. However, when it comes to processing log data online (i.e., when the information is generated), it becomes quite tricky to pick (and possibly adapt) them to the specific requirements of this application area. The reasons for that are manifold, as in the following:

- Single log lines cannot easily be categorized as good or bad, and their classification often relies on the surrounding context.
- Most machine learning approaches were designed for numerical information, e.g., sensor readings, not complex text-based data.
- Log data possess unknown grammar, which means their style, format, and meaning is usually not fully documented and understood by those analyzing them.
- For intrusion detection, near real-time use is preferred. This means methods must be able to process data online, i.e., when the information is produced. As a consequence, approaches need to work in a “single-pass” manner and process data in streams in an efficient way.
- Since a monitored environment may rapidly change, the usually separated training and detection phases of machine learning approaches may overlap and disturb one another. It is not acceptable that a single change triggers a need to learn a complex model from scratch; rather, models should be adaptable.

Bearing these challenges in mind, in this article, we present the concept of a log data analysis pipeline,

Digital Object Identifier 10.1109/MSEC.2021.3113275
Date of current version: 7 October 2021

map existing approaches to the functional blocks of this pipeline, highlight specific challenges, and discuss adaptations and recommendations to improve applicability for online log data analysis. The open source software AMiner (https://github.com/ait-aecid/) implements these concepts. We take a closer look into challenges of log data analysis for security purposes, examine relevant approaches, and provide insights into their application through some practical examples. The main goal of this article is to make important work accessible to a large audience.

Log Data Analysis for Security Purposes

Blocklisting approaches can be effective in several use cases. For instance, detecting access attempts outside business hours is a standard case that every well-configured intrusion detection system can handle. Nevertheless, using blocklisting only, security personnel must think upfront of all potential attack cases and how they could manifest in a network. This is not only a tedious task but also extremely error prone. In contrast, the application of anomaly-based approaches that discover deviations from a defined system's state seems promising: one needs to describe the "normal and desired system behavior" (this means creating an allow list of what is known to be good), and everything else is classified as potentially hostile. The effort is comparatively lower and demonstrates the advantages of an anomaly-based technique.[2] However, these advantages come with a price. While signature-based methods tend to generate false negatives, i.e., undetected attacks, anomaly-based approaches are usually prone to high false positive rates. Complex behavior models and potentially error-prone training phases are just some of the drawbacks to consider.

To keep false positive rates at acceptable levels, it is important to carefully design an anomaly detection procedure. We identify four major building blocks that are necessary to establish a log processing pipeline for anomaly detection. Figure 1 provides an overview. The first step involves clustering log data, with the purpose of generating groups of similar events. Clustering is fully unsupervised; i.e., it is not necessary to manually code any knowledge about the log structures and assign labels to single lines. Instead, approaches based on string similarity and n-gram analysis assign lines to groups of similar events. There are two main outcomes of this step. First, outliers are identified as lines that end up in small clusters. These lines are the most basic form of anomalies, as they indicate rare events. Second, the resulting groups of lines are suitable to be transformed into templates, i.e., patterns that are descriptive for all lines in a specific group. This task is addressed in the second step of the pipeline.

Generating such cluster templates from groups with similar events is a nontrivial task since artifacts in the logs are diverse. For example, events with different numbers and orders of tokens and lengths of these tokens may appear in the same cluster. Sequence alignment is a commonly used method for template generation, but it is mostly applied to pairs of strings rather than groups. It is therefore necessary to continuously merge such templates to figure out which characters in the logs are static or variable. Eventually, the generated templates are utilized by a parser generation module in the third step. Log parsers leverage tree-like structures rather than lists of patterns, such as regular expressions, to ensure that parsing takes place with the highest possible efficiency. Accordingly, log templates are split into tokens by a set of delimiters and then transformed into a single parsing tree. Thereby, tokens that are different in most lines are replaced by variables, while static tokens differentiate event types. Parsers are essential for all subsequent analyses because they map all tokens to semantically meaningful attributes.

Parsed log data are analyzed by anomaly detection techniques as described in the fourth step. Available detection methods are diverse and usually address specific characteristics of artifacts that are commonly subject to change when malicious behavior manifests in the logs. For example, time series analysis detects events that occur with higher or lower frequency than usual,

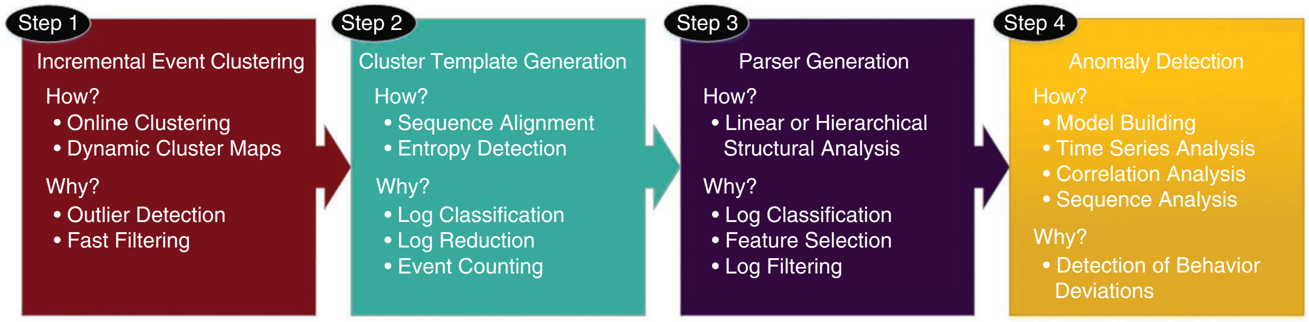

Figure 1. The log data analysis pipeline consisting of four major building blocks that are sequentially executed.

correlation analysis detects events that typically occur together and fail to do so, and sequence analysis detects workflow changes; i.e., known events appear in a new order. All these methods have in common that they are based on a model of normal behavior that is learned on training data and continuously adapted through time, where anomalies may be detected in a semisupervised manner during learning and in a separate detection phase. Deploying, configuring, and effectively operating an anomaly detection system that ingests log data is a complicated task. We employ the setting described in the following as a guiding scenario for the stepwise introduction of our log data analysis solution.

Scenario

An internal web server hosts numerous services and sensitive resources. Legitimate users may retrieve these resources and modify them via web-based forms. Security operators collect access logs, using client Internet Protocol (IP) addresses, user agent, requested resource name, and request methods to build a system model, consisting of expected event types and values employed as a baseline for anomaly detection. The log data look as follows:

```
[…]

10.0.0.130 – – [04/Mar/2021:06:55:35] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://doc
.acme.com/projX/” “Mozilla/5.0”

10.0.0.139 – – [04/Mar/2021:06:55:45] “GET/
projX/doc2 HTTP/1.1” 200 2845 “http://doc
.acme.com/projX/” “Mozilla/5.0”

10.0.0.139 – – [04/Mar/2021:06:55:45] “GET/
projX/doc2p1 HTTP/1.1” 200 849 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.121 – – [04/Mar/2021:06:55:47] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.1.0.137 – – [04/Mar/2021:06:55:48] “GET/
projY/xls7 HTTP/1.1” 200 3574 “http://
doc.acme.com/projY/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:54]
“POST/edit/doc2 HTTP/1.1” 200 3243
“http://doc.acme.com/projX/edit.php?
page=doc2” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:55] “GET/
projX/doc2 HTTP/1.1” 200 3243 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:55] “GET/
projX/doc2p1 HTTP/1.1” 200 849 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:56] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:56:34]
“POST/edit/doc1 HTTP/1.1” 200 4341
“http://doc.acme.com/projX/edit.php?
page=doc1” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:56:36] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

[…].
```

Looking into this rather simplified snippet, we can already observe several properties feasible for model building. For instance, we see different client IPs accessing various types of web resources, although not all IPs access the same ones. We can find similarities in paths (there are project X and project Y), and we can see that only the user 10.0.0.130 edits documents (using HTTP POST requests), while the others mainly retrieve data (using GET). We learn that all users employ the same user agent, presumably the browser of the company’s software standard. Looking closer, we observe even certain sequences per IP address. For instance, whenever there is a POST request, the same client retrieves a changed document again via a consecutive GET request. Furthermore, doc2 consists of two parts, which are always retrieved together; assuming doc2 includes doc2p1, the browser automatically fetches both in two consecutive requests. These are all behavioral properties of using web-based systems that can be observed and captured using machine learning.

Step 1: Incremental Character-Based Event Clustering

Purpose

Clustering is an efficient method for grouping similar events and recognizing rare ones. Thus, it 1) supports reducing the number of events when analyzing large data sets, 2) enables outlier detection, and 3) facilitates time series analysis when observing cluster properties through time, e.g., using evolutions to correlate clusters in different time windows.

Main Challenges

Many clustering approaches, such as distance-based techniques, do not facilitate processing huge data sets because they store large distance matrices, consuming significant amounts of memory. Additionally, many clustering algorithms do not implement single-pass procedures, which makes them inapplicable for online log analysis. Almost all existing log clustering solutions implement token-based approaches and split log lines only when a "space" occurs. Thus, they often do not yield exact results, due to long tokens and because they recognize highly similar but not equal tokens as entirely different, such as terms and words with different encodings, e.g., "can't open file" versus "can\'t open file," and similar uniform resource locators (URLs) and related paths, e.g., "/home/alice/test.txt" versus "~/test.txt." Then again, character-based clustering suffers from highly complex string metrics and therefore poor runtime. Ultimately, log clustering requires high-performance, incremental, character-based approaches that employ smart filters to optimize runtime.

Relevant Works

The simple logfile clustering tool (SLCT)[3] was one of the first clustering algorithms specifically developed for log data. It implements a density-based approach and analyzes token frequencies. Iterative partitioning log mining (IPLoM)[4] applies iterative partitioning, where groups of log lines are recursively split into subgroups according to particular token positions. The type-casted pattern and rule miner (CAPRI)[5] uses a density-based method for clustering and additionally applies statistical analysis to identify contextual relationships among clusters. While SLCT and IPLoM do not implement single-pass procedures and are capable only of processing static, i.e., fixed, data sets, CAPRI enables stream-based analysis after a training phase, although the system model is static. Additionally, none of the tools implements a character-based method.

Our Recommended Approach

The mentioned challenges motivated our research on character-based matching algorithms with runtime performance comparable to token-based matching. Our incremental clustering approach[6] that implements density and character-based clustering applies a single-pass clustering algorithm that processes data in streams as well as line by line instead of batches. This enables online anomaly detection; i.e., log lines are processed at the time they are generated. Clustering approaches that are applied for online anomaly detection have to fulfill some essential requirements: 1) rapidly process data, i.e., when they are generated; 2) promptly adopt the cluster map (note that *cluster map* refers to the structure of the clustering, i.e., the clusters and their identifiers, which can be, for example, a template or representative for each cluster); and 3) deal with large amounts of data.

Nevertheless, existing clustering approaches that usually process all data at once, such as SLCT[3] and IPLoM,[4] suffer from the following three major drawbacks, which make them unsuitable for online anomaly detection in log data:

1. *Static cluster maps*: Adapting/updating a cluster map is time-consuming and computationally expensive. If new data points occur that account for new clusters, a whole cluster map has to be recalculated, as with CAPRI.[5]
2. *Expensive memory*: Distance-based clustering approaches are limited by available memory because large distance matrices have to be stored: depending on the applied distance, this amounts to n^2 or $n^2/2$ elements.
3. *Computational expense*: Log data are stored as text. Therefore, string metrics are applied to calculate the distance (similarity) between log lines. Their computation is usually costly and slow.

We introduced a novel incremental clustering approach[6] with the following features that sequentially processes log data in streams for online anomaly detection:

- The processing time of incremental clustering grows linearly with the rate of input log lines, and there is no required rearrangement of the cluster map. The distances between log lines do not need to be stored.
- Fast filters reduce the number of distance computations that have to be carried out. A semisupervised approach based on self-learning reduces the configuration and maintenance effort for system administrators.
- The modularity of our approach facilitates the application of different metrics to build the cluster map and carry out anomaly detection.
- Our method enables the detection of point anomalies, i.e., single anomalous log lines, by outlier detection. Collective anomalies, i.e., anomalous numbers of occurrences of normal log lines that represent a change in system behavior, are detected through time series analysis.

Example

Applying the introduced incremental clustering to the previously outlined log data, the following clusters emerge (assuming we blind out the time stamp from the similarity calculations). Naturally, the two POST requests are different from all the GET requests. Furthermore, the two requests for doc2p1 are longer and look a bit different than the others. Also, the request to/projY varies from all the others in at least two spots, and the IP address differs, too, from all the others. In this simple example, the advantage of character-level templates already becomes visible: we can account for similarities in paths and IP addresses (e.g., distinguish IP address from different subnets without the need to specify the same):

```
[Cluster 1]
10.0.0.130 – – [04/Mar/2021:06:55:35] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.139 – – [04/Mar/2021:06:55:45] “GET/
projX/doc2 HTTP/1.1” 200 2845 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.121 – – [04/Mar/2021:06:55:47] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:55] “GET/
projX/doc2 HTTP/1.1” 200 3243 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:56] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:56:36] “GET/
projX/doc1 HTTP/1.1” 200 3844 “http://
doc.acme.com/projX/” “Mozilla/5.0”

[Cluster 2]
10.1.0.137 – – [04/Mar/2021:06:55:48] “GET/
projY/xls7 HTTP/1.1” 200 3574 “http://
doc.acme.com/projY/” “Mozilla/5.0”

[Cluster 3]
10.0.0.139 – – [04/Mar/2021:06:55:45] “GET/
projX/doc2p1 HTTP/1.1” 200 849 “http://
doc.acme.com/projX/” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:55:55] “GET/
projX/doc2p1 HTTP/1.1” 200 849 “http://
doc.acme.com/projX/” “Mozilla/5.0”

[Cluster 4]
10.0.0.130 – – [04/Mar/2021:06:55:54]
“POST/edit/doc2 HTTP/1.1” 200 3243
“http://doc.acme.com/projX/edit.php?
page=doc2” “Mozilla/5.0”

10.0.0.130 – – [04/Mar/2021:06:56:34]
“POST/edit/doc1 HTTP/1.1” 200 4341
“http://doc.acme.com/projX/edit.php?
page=doc1” “Mozilla/5.0”.
```

Step 2: Creating Cluster Templates

Purpose

Combining template generation with clustering offers plenty of application possibilities: 1) templates can be transformed into allow list rules, 2) templates provide an accurate description of the content of log lines assigned to the same cluster, and 3) they facilitate parser generation and log line classification, i.e., assigning event types to log lines. The latter enables the application of a large variety of anomaly detection algorithms based on frequency and sequence analysis and thus, for example, employ event count matrices.

Main Challenges

Since clustering and template generation are closely related and clustering algorithms often provide some sort of template, template generation inherits most of its challenges from log line clustering. Again, there exist token- and character-based approaches. While token-based methods benefit from superior performance, character-based templates are much more accurate and handle similar but not equal tokens much better and recognize variable parts more precisely. However, calculating character-based templates is complex and impossible in the time frames required by online analysis. Thus, robust heuristics are required that optimize performance and effectively reduce runtime.

Relevant Works

The parallel log parser (POP)[7] applies the longest common subsequence (LCS) and implements an iterative partitioning approach similar to IPLoM,[4] where templates are iteratively updated by identifying particular token positions in which the log lines differ. LogHound[8] applies the density-based approach of SLCT as well as frequent item set mining to generate templates. LogSig[9] implements some sort of LCS by finding common word, i.e., token, pairs in similar log lines to build representative templates. All the approaches generate meaningful token-based templates but neglect character-based algorithms.

Our Recommended Approach

We developed template generators (six that create meaningful cluster descriptions), a prerequisite for the feature selection used by machine learning solutions as well as generating log parsers. Furthermore, templates can be applied for log classification in general, such as with LogSig,[9] for log reduction through filtering and event counting. A template is basically a string that consists of substrings that occur in every log line of a cluster in a similar location. Those substrings are referred to as *static parts of the log lines of the cluster*. They are separated by wild cards, which represent variable parts of the log lines, such as user names, IP addresses, and identifiers. Furthermore, a template matches all log lines of the corresponding cluster.

The cornerstone of cluster template generation on a character level is an efficient means to determine the degree of similarity of two log lines, i.e., strings. A sequence alignment is the result of an algorithm that arranges two strings so that the smallest number of operations (i.e., insertions, deletions, and replacements of characters) is required to transform one string into the other one; i.e., it assumes the highest possible similarity. We solved the problem of generating a sequence alignment for more than two log lines on a character level,[6] i.e., generating a multiline alignment.[10] In contrast to token-based template generators, such as POP[7] and LogHound,[8] character-based approaches do not rely on predefined building blocks in the form of tokens. They recognize static and variable parts of log lines independently from predefined delimiters.

There exist many efficient algorithms and string metrics, such as the Levenshtein distance and the Needleman–Wunsch algorithm, to achieve an alignment for two character sequences. Furthermore, there are algorithms for genetic and biologic sequences to calculate pairwise and multiline alignments; however, they require knowledge about the evolution of nucleotides and are therefore not suitable for log data.[10] Algorithms to align multiple sequences of any characters with no genetic context are challenging. The main reason is the difficulty of overcoming the high computational complexity of this problem, which is at least $O(n^m)$, where n is the length of the shortest log line and m is the number of lines in a cluster. We proposed a character-based cluster template generator that incrementally processes the lines of a log line cluster and reduces the computational complexity $O(n^m)$ to $O(mn^2)$. The algorithm processes log lines sequentially and thus follows an incremental approach, which must handle each line only once. The resulting template has a high similarity to the optimal template on preclustered data.[6]

Example

Based on the four clusters created previously, our template generation approach would come up with the following four templates. Notice that we use only a very limited number of log lines to demonstrate the technique. In a typical setting, we would record access logs across several days, if not weeks, and create the templates from a much larger number of log lines, resulting in many more generic templates. We further skipped the processing of the time stamp and manually set it to be variable, reflected by an asterisk:

```
[Cluster 1]: 10.0.0.1* - - [*] "GET/
projX/doc* HTTP/1.1" 200 * "http://doc
.acme.com/projX/" "Mozilla/5.0"

[Cluster 2]: 10.1.0.137 - - [*] "GET/
projY/xls7 HTTP/1.1" 200 3574 "http://
doc.acme.com/projY/" "Mozilla/5.0"

[Cluster 3] 10.0.0.13* - - [*] "GET/
projX/doc2p1 HTTP/1.1" 200 849 "http://
doc.acme.com/projX/" "Mozilla/5.0"

[Cluster 4] 10.0.0.130 - - [*] "POST/
edit/doc* HTTP/1.1" 200 * "http://
doc.acme.com/projX/edit.php?page=doc*"
"Mozilla/5.0".
```

We now have a method to dissect single log lines, analyze their content in a characterwise manner, and identify regions of similarities. These are all prerequisites to learn data structures and automatically create parsers.

Step 3: Parser Generation

Purpose

Parsers match logs to the syntaxes of known events and map all values to specific referenceable attributes. This enables a semantic interpretation of the contents of log lines and a subsequent application of detection techniques on the data. Generating parsers manually, however, is a time-consuming task that requires extensive domain knowledge about the numerous distinct log events that can occur. Parser-generating algorithms therefore analyze samples of log data and automatically create log event templates by determining which parts of the logs are variable or static.

Main Challenges

Regular expressions facilitate defining sequences of variables and fixed tokens and thus appear as a logical choice for log parsers. However, they suffer from slow runtime performance in comparison to tree-based parsers that leverage the fact that many log events have similar tokens. Unfortunately, creating dynamically adjusting parser trees when new events appear or the syntax of existing events changes, is nontrivial. Parser generators further rely on many parameters, such as delimiters for tokens and thresholds that balance over- and underfitting, which are difficult to configure in practice.

Relevant Works

Drain[1] uses token similarity to build an event tree. However, its approach assumes that the number of tokens is fixed for each event and thus cannot be applied to logs with optional tokens. The Scalable Handler for Incremental System log (SHISO)[11] also harnesses similarity to find and merge event templates. While both methods are incremental, they do not address the fact that templates tend to overgeneralize through time. Spell,[12] on the other hand, searches for common sequences in log events to determine static parts.

Our Recommended Approach

Advanced data analysis does much more than clustering and outlier detection. To enable further analysis of, e.g., trends, correlations, and value distributions, a first step is to make the single parts of a log line (i.e., the features of log data) easily accessible and then identify which ones carry important information (i.e., they help us characterize the type of event and its unique parameters). Effective log parsers enable us to do that. A tree-based parser approach[6] aims at reducing the complexity of parsing and therefore increasing the performance. Since there are no commonly accepted standards that dictate the syntax of logs, developers may freely choose the structure of log lines produced by their services and applications. For example, the syslog standard states that each log line has to start with a time stamp followed by the host name. However, the remainder of the syntax can be chosen without restrictions.

Applying standards, such as syslog, causes log lines produced by the same service or application to be similar in the beginning but differ more toward the end of the lines. Consequently, modeling a parser as a tree leads to a parser tree that includes a common trunk and branches toward the leaves. A parser tree represents a graph theoretical rooted out-tree. This means that during parsing, it processes log lines tokenwise from left to right, and only parts of the parser tree that are relevant to the log line at hand are reached. Hence, this type of parser avoids parsing across the same log line more than once as would be done when applying distinct regular expressions. As a result, the complexity for parsing reduces from $O(n)$ to $O(\log(n))$. Eventually, each log line relates to one path (branch) of the parser tree.

Figure 2 depicts a part of a parser tree for web server access logs. This example demonstrates that a tree-based parser consists of three main building blocks. The nodes with bold lines represent tokens with static text patterns. This means that in all corresponding log lines, a token with this text pattern has to occur at the position of the node in the tree. Nodes depicted as ovals enable variable text until the next separator or static pattern along the path in the tree occurs. Variables are inserted based on token frequency rather than overall log similarity.[11] Optional nodes are used to define tokens that do not necessarily have to occur.[1] The third building block is a branch element. When the parser tree branches are in a certain position, only a small number of different tokens with static text occurs.

Applying a tree-like parser model provides the following advantages in terms of performance and log analysis quality:

- In contrast to distinct regular expressions, a tree-based parser avoids parsing across a data entity more than once because it follows, for each log line, one path of the parser tree in the graph-theoretical tree that represents the parser, and leaves out irrelevant model parts.
- Therefore, the computational complexity for log line parsing is closer to $O(\log(n))$ than $O(n)$ when handling data with separate regular expressions.
- The tree-like structure facilitates referencing all the single tokens with an exact path; e.g., "/accesslog/hostip" enables access to the requesting client's IP address. Thus, parsed log line parts are quickly accessible so that rule checks can pick out only the data they need without searching the tree again. Furthermore, the structure facilitates quick applications of anomaly detection algorithms to different tokens and correlating information of different tokens within a single line and across lines.

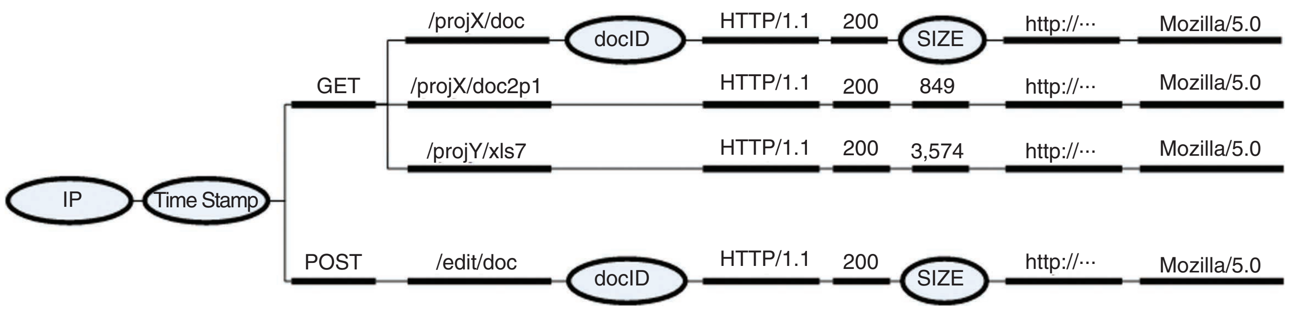

Figure 2. A log line parser tree.

Example

Using the preceding templates, we steer the creation of a tree-like parser by applying the described parser generation methodology. Notice that the resulting tree (see Figure 2) may differ depending on the selected configuration parameters that influence whether different values result in a variable node or branching point. For instance, the request size would naturally be considered a variable value; however, if only a few different sizes are recorded in the log data, it could also be modeled as parallel branches, each consisting of a static but different value. Eventually, we gain a tree structure where each node is referenced by a unique path to retrieve its value. Further generalizing this view, e.g., introducing variable nodes for the response code, URL path, and user agent, the model becomes generally applicable. Table 1 shows the first log line of our example, dissected according to the generalized parser model. In the analysis phase, the different tokens are referenced with the paths given there:

```
10.0.0.130 – – [04/Mar/2021:06:55:35] "GET/
projX/doc1 HTTP/1.1" 200 3844 "http://
doc.acme.com/projX/" "Mozilla/5.0".
```

With the tree-like parser, we have the means to match incoming log lines to observed structures, and, thus, we can categorize events. Furthermore, we are able to distinguish between static and variable parts, which is an important means of feature selection for the machine learning algorithms applied on top of log data. Regardless of whether domain-specific and customized algorithms or general-purpose algorithms are employed (such as neural networks, principal component analysis, and long short-term memory), feature selection is a mandatory prerequisite for analysis and anomaly detection.

Step 4: System Behavior Modeling and Machine Learning-Based Anomaly Detection

Purpose

Log analysis frameworks keep track of all system and user behavior in high detail, enabling precise reasoning about what happened when. Unfortunately, this also means that immense volumes of log data are produced nonstop. While humans are able to dive into specific passages of these logs and interpret recorded activities, the sheer amount of data renders manual monitoring impossible. Therefore, anomaly detection recognizes unusual events and alerts analysts to relevant logs that possibly relate to attacks that were not discovered by rule-based monitors.

Main Challenges

Log data are generated and analyzed in streams, meaning that learning and detection take place continuously and in parallel. This prevents applications of machine learning methods that require multiple passes through data. Moreover, log data often contain traces of erratic user behavior that is difficult to distinguish from malicious intentions and causes high false alarm rates. Finally, logs usually have to be interpreted within their context of occurrence, e.g., the time of day and related events.

Relevant Works

He et al.[13] use event count vectors for frequency-based detection with clustering and principal component analysis. Furthermore, they detect new event sequences with automatically mined invariants. However, these approaches do not adequately address the fact that values also have to be taken into account when detecting anomalies. Deeplog,[14] too, detects anomalies in event sequences but by using neural networks. LogRobust employs semantic word vectors as input to a neural network. Unfortunately, neural networks often suffer from low explainability and are unsuitable for online learning.

Our Recommended Approach

Most machine learning approaches suffer from several drawbacks when applied to online anomaly detection on log data, as discussed previously. Specifically, complex "monolithic" models are of limited use in an environment that undergoes frequent changes, such as updates in computer systems. Fine/granular, explainable models that may be adapted to new situations are required.[14,15]

Table 1. Token paths and values.

Node (parser path)	Token value
/accesslog/hostip	10.0.0.130
/accesslog/time_model	04/Mar/2021:06:55:35
/accesslog/time_model/time	1614837335
/accesslog/time_model/timezone	0
/accesslog/method	GET
/accesslog/request	/projX/doc1
/accesslog/protocol	HTTP
/accesslog/version	1.1
/accesslog/status	200
/accesslog/size	3844
/accesslog/referrer	http://doc.acme.com/projX/
/accesslog/useragent	Mozilla/5.0

In the following, we provide insights into the abilities of some machine learning approaches that are particularly useful for online log data analysis. Other than existing works,[13] we design detectors for the analysis of events and values occurring in log data. With these few methods, it already becomes rather hard for a hacker to mount a successful attack unnoticed. For more details about the concrete algorithms, we refer the reader to Skopik et al.[6]

Attack Scenario

The attacker Mallory has gained remote access to local client 10.0.0.130 and tries to collect as many resources as possible from the web server for later exfiltration.

Simple Detectors

Detection of new values. Mallory simply uses Wget to crawl (parts of) the web server and leaves traces in logs similar to the one in the following. Here, specifically, the user agent can easily be detected as the new value "Wget/1.20.3 (linux-gnu)" in the path "/accesslog/useragent" because, until analyzing this event, the only observed user agent was "Mozilla/5.0":

```
10.0.0.130 - - [14/May/2021:06:06:18] "GET/
projX/doc1 HTTP/1.1" 200 4569 "-" "Wget/
1.20.3 (linux-gnu)".
```

Detection of new value combinations. Mallory changes the user agent to the legitimate standard user agent "Mozilla/5.0" and thus evades detection at first. She, however, attempts to access a resource from a directory that was never retrieved by the client she owns, e.g., "GET/projY/xls7." Since, to this point, the IP "10.0.0.130" occurred only with resources "/projX/doc1," "/edit/doc2," "/projX/doc2," "/projX/doc2p1," and "/edit/doc1," this triggers a new combination of values at paths "/accesslog/hostip" and "/accesslog/request" that was never observed. Notice that the advantage of character-based templates comes into play. If a huge number of documents resides within "/projX/," we could generally consider accesses to documents therein as normal, but we may still alert on access to documents in "/projY/." In addition, it would be possible to use even more paths to increase the granularity of the value combination analysis. Specifically, adding the request method at path "/accesslog/method" to the aforementioned paths enables us to analyze which resources are accessed by which users as well as how they are retrieved. Be aware that training the models takes considerably longer for more complex detector configurations.

Time Series Analysis

Improved attack. Having learned from previous experiences, Mallory accesses only legitimate resources with a valid user agent string. But, since she is in a hurry, she does it in bursts; i.e., she downloads numerous resources in short time intervals.

Detection of frequency anomalies. The seasonal autoregressive integrated moving average (SARIMA)[6] model predicts how many events of a specific type and source are considered normal based on a history of observations, and it enables alerting on any significant deviation. If, let us say, 10–20 document requests per hour have been perceived in the past few observation cycles for user 10.0.0.130, Mallory's attempt to retrieve documents in bulk (say, 100 requests in 1 h) will be detected.

Correlation Analysis

Advanced attack. Mallory again changes her behavior and carries out a much slower moving attack; e.g., she downloads only a couple of resources per hour to evade SARIMA detection.

Detection of divergent correlations. Going back to what we consider normal behavior, the log data listing shows that 10.0.0.130 triggered seven HTTP requests, specifically, five GET and two POST. After observing requests for a longer duration, a certain ratio of GET/POST requests will emerge depending on the user's typical activities. If Mallory polls the web server for new documents, through time, she will issue many GET requests, but no POST requests, and therefore disturb this ratio. A variable correlation detector aims to establish a baseline (i.e., an expected value that is considered normal) and alert on significant deviations from it. For example, using aforementioned data, the learned model could detect that a reasonably sized sample of events with IP "10.0.0.130" occurs with a GET request in 70% and a POST request in 30% of cases, while for all other IP addresses, the ratio is around 95% and 5%. Deviations reported by statistical tests on sufficiently large sections of the data (e.g., GET requests made by "10.0.0.130" increase to 90%) are reported as anomalies.

Sequence Detection

Stealthy attack. Mallory expands her remote access to several internal clients, not just 10.0.0.130, and is now able to collect only small portions of the resources from each client she owns. As a consequence, there are no request bursts from single IPs, nor does the correlation of client IPs to request methods change significantly. This way, Mallory hopes to evade detection once and for all.

Detection of breaking sequences. Usually, a GET request to a single HTML site triggers a set of subrequests to fetch linked content (we assume that caching is disabled on the client side to make this example easier). Whenever doc2 is fetched, doc2p1 is, too. Thus, the detector learns the sequence "/projX/doc2" followed by "/projX/doc2p1" as normal behavior. Since

Mallory attempts to crawl a page with a command line tool and not a browser, she fetches linked content only once and leaves out, e.g., linked images, such as a site logo that is embedded on every page. This breaks previously observed and learned sequences, which is easily detected. Note that detection complexity here mainly depends on the lengths of the analyzed sequences; i.e., a sequence length of two, as in the preceding example, enables efficient learning but has less model granularity than larger sequence lengths that require long training phases and tend to overfit the data more easily.

Log data analysis and anomaly detection in computer networks need to cope with some significant challenges, such as frequent changes to observed systems (which is not the case for other machine learning domains), a certain degree of learned model adaptability (which is not the case for most classic machine learning approaches), and a large amount of complex data that need to be processed in streams (in contrast to offline multipass analysis). A multitude of approaches are available, from rather simple detectors to much more complex analysis solutions that account for the interdependencies of log events, including sequence and time series analysis.

Keep in mind, the more complex detectors we apply, the more likely we are to discover malicious behavior. However, the disadvantages of using complex detectors are 1) they are much more complex to configure and maintain, 2) it takes longer for them to learn, and 3) they are prone to high false positive rates. The art is to find the sweet spot between detecting enough anomalies to act in time and not drowning in false alerts. AMiner extends any existing security solution, such as a Security Information and Event Management solution (SIEM) and Elastic Stack (https://www.elastic.co), for event monitoring. Therefore, it provides in-depth, high-performance online log analysis and anomaly detection, employing various smart and sophisticated detectors far beyond event search and signature-based detection. For a more comprehensive review of the material in this article, please refer to Skopik et al.[6] ■

Acknowledgments

This work was supported in part by the Austrian FFG project DECEPT under grant 873980, and in part by the European Union H2020 project GUARD under grant 833456 and the EDIDP project PANDORA under grant SI2.835928.

References

1. P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, June 2017, pp. 33–40. doi: 10.1109/ICWS.2017.13.
2. V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009. doi: 10.1145/1541880.1541882.
3. R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proc. 3rd IEEE Workshop IP Oper. Manage. (IPOM 2003)(IEEE Cat. No. 03EX764)*, Oct. 2003, pp. 119–126. doi: 10.1109/IPOM.2003.1251233.
4. A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, June 2009, pp. 1255–1264. doi: 10.1145/1557019.1557154.
5. F. Zulkernine, P. Martin, W. Powley, S. Soltani, S. Mankovskii, and M. Addleman, "Capri: A tool for mining complex line patterns in large log data," in *Proc. 2nd Int. Workshop Big Data, Streams Heterogeneous Source Mining, Algorithms, Syst., Programming Models Appl.*, Aug. 2013, pp. 47–54. doi: 10.1145/2501221.2501228.
6. F. Skopik, M. Wurzenberger, and M. Landauer, *Smart Log Data Analytics: Techniques for Advanced Security Analysis*, 1st ed. New York: Springer-Verlag, 2021.
7. P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 6, pp. 931–944, 2017. doi: 10.1109/TDSC.2017.2762673.
8. R. Vaarandi, "A breadth-first algorithm for mining frequent patterns from event logs," in *Proc. Int. Conf. Intell. Commun. Syst.*, Nov. 2004, pp. 293–308.
9. L. Tang, T. Li, and C. S. Perng, "LogSig: Generating system events from raw textual logs," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2011, pp. 785–794. doi: 10.1145/2063576.2063690.
10. C. Notredame, "Recent evolutions of multiple sequence alignment algorithms," *PLoS Comput. Biol.*, vol. 3, no. 8, p. e123, 2007. doi: 10.1371/journal.pcbi.0030123.
11. M. Mizutani, "Incremental mining of system log format," in *Proc. IEEE Int. Conf. Services Comput.*, June 2013, pp. 595–602. doi: 10.1109/SCC.2013.73.
12. M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proc. IEEE 16th Int. Conf. Data Mining (ICDM)*, Dec. 2016, pp. 859–864. doi: 10.1109/ICDM.2016.0103.
13. S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2016, pp. 207–218. doi: 10.1109/ISSRE.2016.21.
14. M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, Oct. 2017, pp. 1285–1298. doi: 10.1145/3133956.3134015.
15. X. Zhang *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 807–817. doi: 10.1145/3338906.3338931.

Florian Skopik is head of the cybersecurity research program at the Austrian Institute of Technology, Vienna, 1210, Austria. His research interests include critical infrastructure protection, intrusion detection, and national cybersecurity. Skopik received a Ph.D. in computer science from Vienna University of Technology. He is a member of various conference program committees (e.g., Symposium on Applied Computing, International Conference on Availability, Reliability and Security, International Conference on Critical Information Infrastructure Security), editorial boards, and standardization groups, such as ETSI TC Cyber, IFIP TC11 WG1, and OASIS CTI. He frequently serves as a reviewer for numerous high-profile journals, including Elsevier's *Computers & Security* and *ACM Computing Surveys*. He is a Senior Member of IEEE. Contact him at florian.skopik@ait.ac.at.

Max Landauer is a scientist at the Austrian Institute of Technology, Vienna, 1210, Austria. His research interests include log data analysis, anomaly detection, and cyberthreat intelligence. Landauer is a Ph.D. candidate in computer science at the Vienna University of Technology. His Ph.D. studies are a cooperative project between the Vienna University of Technology and the Austrian Institute of Technology. Contact him at max.landauer@ait.ac.at.

Markus Wurzenberger is a scientist and project manager at the Austrian Institute of Technology, Vienna, 1210, Austria. His research interests include log data analysis with a focus on anomaly detection and cyberthreat intelligence. Wurzenberger received a Ph.D. in computer science from Vienna University of Technology. Contact him at markus.wurzenberger@ait.ac.at.

Digital Object Identifier 10.1109/MSEC.2022.3171998