Detecting Unknown Cyber Security Attacks Through System Behavior Analysis

Florian Skopik(✉), Markus Wurzenberger, and Max Landauer

AIT Austrian Institute of Technology, Vienna, Austria
{florian.skopik,markus.wurzenberger,max.landauer}@ait.ac.at

Abstract. For many years signature-based intrusion detection has been applied to discover known malware and attack vectors. However, with the advent of malware toolboxes, obfuscation techniques and the rapid discovery of new vulnerabilities, novel approaches for intrusion detection are required. System behavior analysis is a cornerstone to recognizing adversarial actions on endpoints in computer networks that are not known in advance. Logs are incrementally produced textual data that reflect events and their impact on technical systems. Their efficient analysis is key for operational cyber security. 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. The AMiner is an open source implementation of a pipeline that implements many machine learning algorithms that are feasible for deeper analysis of system behavior, recognizing deviations from learned models and thus spotting a wide variety of even unknown attacks.

Keywords: Cyber security · Anomaly detection · Unknown attacks · Log data analysis · System behavior monitoring · Intrusion detection · Machine learning · Artificial intelligence

1 Introduction

Log files capture information about almost all events that take place in a system. Depending on the set log level, the deployed logging infrastructure automatically collects, aggregates and stores the logs that are continuously produced by many components and devices, e.g., web servers, databases, or firewalls. The textual log messages are usually human-readable and attached to a timestamp that specifies the point in time when the log entry was generated. Especially for large organizations and enterprises, the benefits of having access to long-term log data are manifold: Historic logs enable forensic analysis of past events. Most prominently applied after faults occurred in the system, forensic analysis gives system administrators the possibility to trace back the roots of observed problems. Moreover, the logs may help to recover the system to a non-faulty state, reset incorrect

An adapted version of this article has been published in [11].

J. Kołodziej et al. (Eds.): Cybersecurity of Digital Service Chains, LNCS 13300, pp. 103–119, 2022.
https://doi.org/10.1007/978-3-031-04036-8_5

transactions, restore data, prevent losses of information, and replicate scenarios that lead to erroneous states during testing. Finally, logs also allow administrators to validate the performance of processes and discover design bottlenecks. In addition to these functional advantages, storing logs is typically inexpensive since log files can effectively be compressed due to a high number of repeating lines. Consequently, capturing and storing logs is rather straight forward compared to their analysis. Eventually, we do not face a big data problem, but rather a big analysis problem.

A major issue with forensic log analysis is that problems are only detected in hindsight. Furthermore, it is a time- and resource-consuming task that requires domain knowledge about the system at hand. For these reasons, modern approaches in cyber security shift from a purely forensic to a proactive analysis [6]. Thereby, real-time fault detection is enabled by constantly monitoring system logs in an online manner, i.e., as soon as they are generated. This allows timely responses and in turn reduces the costs caused by incidents and cyber attacks [10]. On top of that, indicators for upcoming erroneous system behavior can frequently be observed in advance. Detecting such indicators early enough and initiating appropriate countermeasures can help to prevent certain faults altogether.

Unfortunately, this task is hardly possible for humans since log data is generated in immense volumes and fast rates. When considering large enterprise systems, it is not uncommon that the number of daily produced log lines is up in the millions, for example, publicly available Hadoop Distributed File System (HDFS) logs comprise more than 4 million log lines per day [17] and small organizations are expected to deal with peaks of 22,000 events per second [2]. 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 a condensed form. Literally hundreds of different machine learning approaches have been proposed over the last decades. However, when it comes to processing log data online (i.e., when they are 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:

- Single log lines cannot easily be categorized as good or bad, but their classification often relies on the surrounding context.
- Most machine learning approaches were designed for numerical data, 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 approaches must be able to process data online, i.e., when they are produced. As a consequence, approaches need to work in a “single pass” manner and process data in streams in an efficient way.

– Since the monitored environment may change rapidly, the usually separated training- and detection-phases of machine learning approaches may overlap and disturb each other. It is not acceptable that a single change triggers the need to learn a complex model from scratch, but models should be rather adaptable.

Bearing these challenges in mind, in recent years, we have worked on several concepts to extend the State of the Art, specifically to move from token-based to character-based parsing and analysis and enable much more fine-grained analysis at comparable speed. To accomplish this, we went away from linear lists of search patterns to tree-based parsers which at the same time tremendously simplify structured access to unstructured log data. Eventually, this is an important prerequisite for advanced analysis, such as time series analysis, correlation analysis or sequence analysis.

The Open Source Software AMiner [1] implements these concepts. We take a closer look into challenges of log data analysis for security purposes, the design methodology of a modern machine learning based solution, and provide insights into its application with some practical examples.

2 Log Data Analysis for Security Purposes

A wide variety of security solutions have been proposed in recent years to cope with increasing security challenges. While some solutions could effectively address upcoming cyber security problems, research on intrusion detection systems is still one of the main topics of computer security science. Signature-based approaches (using blocklists) are still the de-facto standard applied today for some good reasons: they are simple to configure, can be centrally managed, i.e., do not need much customization for specific networks, yield an efficient, robust and reliable detection and provide low false positive rates. While these are significant benefits for their application in today's enterprise environments, there are, nevertheless, solid arguments to work on more sophisticated anomaly-based detection mechanisms.

For instance, technical zero-day vulnerabilities are not detectable by blocklisting approaches; in fact, there are no signatures to describe the indicators of an unknown exploit. Furthermore, attackers can easily circumvent conventional intrusion detection, once indicators are widely distributed. Re-compiling a malware with small modifications will change hash sums, names, IP addresses of command and control servers, rendering previous indicators useless. The interconnection of previously isolated infrastructures entails new entrance points to ICT infrastructures. Especially in infrastructures comprising legacy systems and devices with small market shares, signature-based approaches are mostly inapplicable, because of their lack of (long-term) vendor support and often poor documentation. Eventually, sophisticated attacks use social engineering as an initial intrusion vector. Here, no technical vulnerabilities are exploited, hence, no concise blocklist indicators for the protocol level can appropriately describe erratic and malicious behavior.

Especially the latter aspect requires smart anomaly detection approaches to reliably discover deviations from a desired or previously observed system's behavior because of an unusual utilization through an illegitimate user. This is the usual case when an adversary manages to steal user credentials, e.g., by phishing, and uses these legitimate credentials to illegitimately access a system. However, an attacker will eventually utilize the system differently from the legitimate user, for instance running scans, searching shared directories and trying to extend their presence to surrounding systems. These activities will be executed at either unusual speed, or at unusual times, taking unusual routes in the network, issuing actions with unusual frequency, or causing unusual data transfers at unusual bandwidth. This will generate a series of events, visible in log data and identifiable by anomaly-based detection approaches.

Blocklisting approaches can be effective in some of these cases. For instance, detecting access attempts outside business hours is a standard case, which every well-configured IDS can handle. Nevertheless, using blocklisting only, the security personnel must think upfront of all the potential attack cases, and how they could manifest in the network. This is not only a cumbersome and tedious task, but also extremely error-prone. In contrast to that, the application of anomaly-based approaches seems promising: one needs to describe the "normal and desired system behavior" (this means to create an allowlist of what is known good) and everything that differs from this description is classified as potentially hostile. The effort is comparatively lower, and remarkably demonstrates the advantage of an anomaly-based approach [4]. However, these advantages come with a price. While signature-based approaches tend to generate false negatives, i.e., not detected attacks, anomaly-based approaches usually are prone to high false positive rates. Complex behavior models and potentially error-prone training phases are just some of the drawbacks to consider. Deploying, configuring and effectively operating an anomaly detection system that ingests log data, is a complicated task. We employ the setting described in Example 1.1 as a guiding scenario for the step-wise introduction of our log data analysis solution.

Example: An internal web server hosts numerous services and sensitive resources. Legitimate users may retrieve these resources and modify them via Web-based forms. The security operators collect access logs, using client IP, user agent, requested resource name, and request method to build a system model, consisting of expected event types and values, used as a baseline for anomaly detection. The log data look like this:

```
[...]
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 short and simplified snippet, we can already observe a number of properties feasible for model building. For instance, we can observe different client IPs accessing different types of Web resources, although not all IPs access the same resources. We can observe similarities in paths (there is projectX and projectY), we can see that only one user 10.0.0.130 edits documents (using HTTP POST requests), while the others mainly retrieve data (using GET). We learn that all users utilize the same user agent, presumably the browser of the companys software standard. Looking closer, we observe even certain sequences per IP address, for instance, whenever there is a POST request, the same client retrieves the changed document again via a consecutive GET request. Furthermore, doc2 consists of two parts, which are both always retrieved together, assuming doc2 includes doc2p1 the browser automatically fetches both in two consecutive requests. These are all behavioral properties of using the Web based systems which can be observed and captured using machine learning.

Example 1.1. Illustrative scenario

3 Incremental Character-Based Event Processing

Every analysis starts with dissecting the input data to make it fit for further processing. Commonly, predefined parsers are used to break up log data according to known event structures. This is also the right phase in the analysis process to get rid of excessive raw data and keep only those parts with high entropy, i.e., that contribute most knowledge in the further analysis process. For instance, an access event on a web server, as illustrated in Example 1.1, could be associated with a representative event class id and enriched with the most important parameters (e.g., client IP, user agent and accessed resource name), while the raw log text could be dumped. This way, fixed parts of log lines that add little to the further analysis and variable parts that carry the interesting information can be distinguished. Moreover, progressing from word-based tokenizing to the character level and considering similarities between words at specific positions within a log line, e.g., similarities of IP addresses, resource names and file paths, adds additional value and simplifies the construction of a baseline. Character-based event processing and templating aims to achieve exactly that.

Clustering is an effective technique to describe the computer system and network behavior by grouping similar log lines in clusters [7]. Furthermore, it allows to periodically review rare events (outliers) and checking frequent events by comparing cluster sizes over time (e.g., trends in the number of requests to certain resources) [8]. Hence, clustering supports organizations to have a thorough understanding about what is going on in their network infrastructures, to review log data and to find anomalous events in log data. Existing tools are not fully suitable to cover all these requirements, as they still show some essential deficits. Most of them, such as SLCT [13] implement token-based matching of log entries, i.e., they split up log lines at whitespaces or other defined delimiters. They treat terms and words with multiple spellings or differences in one character, such as "php-admin" and "phpadmin", or similar URLs, as entirely different. This motivated our research on character-based matching algorithms with comparable runtime performance to token-based matching. Furthermore, existing traditional tools applied for log line clustering are usually not able to process log data with high throughput and therefore are only applicable for forensic analysis, but not for online anomaly detection.

Our incremental clustering approach [16] that implements density and character-based clustering, applies a single pass clustering algorithm that processes data in streams or 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: (i) process data timely, i.e., when it is generated, (ii) adopt the cluster map promptly (Notice, cluster map refers to the structure of the clustering, i.e., the clusters and their identifiers, which can be, for example, a template or a representative for each cluster.), and (iii) deal with large amounts of data. Nevertheless, existing clustering approaches that usually process all data at once, such as SLCT [13], suffer from three major drawbacks, which make them unsuitable for online anomaly detection in log data: (i) Static cluster maps: Adapting/updating a cluster map is time consuming and computationally expensive. If new data points occur that account for new clusters, the whole cluster map has to be recalculated. (ii) Memory expensive: Distance-based clustering approaches are limited by the available memory, because large distance matrices have to be stored – depending on the applied distance, n^2 or $n^2/2$ elements have to be stored. (iii) Computationally expensive: Log data is stored as text data. Therefore, string metrics are applied to calculate the distance (similarity) between log lines. Their computation is usually expensive and time consuming.

In order to overcome these challenges, we introduced an incremental clustering approach that processes log data sequentially in streams to enable online anomaly detection in ICT networks. We proposed a concept, applied in Example 1.2, that comprises a number of novel features [16] as follows:

– The processing time of incremental clustering grows linearly with the rate of input log lines, and there is no re-arrangement of the cluster map required. 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 semi-supervised approach based on self-learning reduces the configuration and maintenance effort for a system administrator.
– The modularity of our approach allows the application of different existing metrics to build the cluster map and carry out anomaly detection.
– Our approach enables detection of point anomalies, i.e., single anomalous log lines, by outlier detection. Collective anomalies, i.e., anomalous number of occurrences of normal log lines that represent a change in the system behavior, are detected through time series analysis.

Example: Applying the introduced incremental clustering to the log data of Example 1.1, the following clusters emerge (assuming we blind out the timestamp 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 from the others. Also, the one request to /projY is different 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 becomes already 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"
```

Example 1.2. Clustered log lines

4 Creating Cluster Templates

Most clustering algorithms [7], when applied to our problem scope, provide no or only inaccurate and insufficient information on the content of a log line cluster.

Specifically, tokens that differ in only single characters are seen as entirely different; furthermore, optional tokens in a log line disturb the sequence of tokens and lead to wrong comparisons. Thus, we developed template generators [14] that create meaningful cluster descriptions, a prerequisite for feature selection used by machine learning solutions as well as generating log parsers [15]. Furthermore, templates can be applied for log classification in general, for log reduction through filtering, and for event counting. A template is basically a string that consists of substrings which 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 wildcards, which represent variable parts of the log lines, such as usernames, IP addresses, and identifiers (IDs). Furthermore, a template matches all log lines of the corresponding cluster.

A sequence alignment is the result of an algorithm that arranges two strings, so that the least number of operations (i.e., insertions, deletions, or 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 [16], i.e., generating a multi-line alignment [9]. In contrast to token-based template generators, character-based approaches do not rely on predefined building blocks in the form of tokens. These approaches recognize static and variable parts of log lines independently from predefined delimiters. Example 1.3 provides an example of generated templates.

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 or biologic sequences to calculate pair-wise and multi-line alignments, which however require knowledge about the evolution of nucleotides and are therefore not suitable for log data [9]. Algorithms to align multiple sequences of any characters with no genetic context are challenging. The main reason is the difficulty to overcome 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 needs to handle each line only once. The resulting template (cf. Example 1.3) has a high similarity to the optimal template on pre-clustered data [14].

Example: Based on the four clusters of Example 1.2, our template generation approach would come up with the following four templates. Notice, we use only a very limited amount of log lines to demonstrate the approach. In a productive setting, we would record access logs over several days, if not weeks, and create the templates out of a much larger number, resulting in much more generic templates. We further skipped the processing of the timestamp and manually set it to be variable, reflected by an asterisk symbol.

```
[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"
```

Example 1.3. Log line templates

Eventually, we have now a method to dissect single log lines, analyze their content character-wise, and identify regions of similarities. These are all prerequisites to learn data structures and automatically create parsers.

5 Learning Data Structures and Creating Tree-Based Parsers

Advanced data analysis does much more than clustering and outlier detection. To enable further analysis of e.g., trends, correlations or 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 to identify which parts carry important information (i.e., help us to characterize the type of event and its unique parameters). Log parsers enable us to do that.

Currently, most log parsers simply apply a set of regular expressions to process log data. The set describes all possible log events and log messages, when the monitored system or service runs in a normal state. Each regular expression looks for static and variable parts that are usually separated by whitespaces, and describes one type of log event or log message. Regular expressions applied in parsers can be depicted as templates. For example, in the template "Connection from * to *", "Connection", "from" and "to" are static and "*" are variable. Those templates are generated applying clustering and template generator, as just discussed. Subsequently, to parse log data, all of these regular expressions are applied in the same order to each log line separately until the line matches a regular expression. This procedure is inefficient, with a complexity of O(n) per log line, where n is the number of regular expressions.

A tree-based parser approach [15] aims at reducing the complexity of parsing and therefore increasing the performance. Since there are no commonly accepted standards that dictate the overall log syntax, developers may freely choose the structure of log lines produced by their services or applications. For example, the syslog standard defines that each log line has to start with a timestamp followed by the host name. However, the remainder of the syntax can be chosen without any 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 towards the end of the lines. Consequently, modeling a parser as a tree, leads to a parser tree that comprises a common trunk and branches towards the leaves, see Example 1.5. The parser tree represents a graph theoretical rooted out-tree. This means, during parsing, it processes log lines token-wise from left to right and only parts of the parser tree that are relevant for the log line at hand are reached. Hence, this type of parser avoids parsing over the same log line more than once as it 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, i.e., branch, of the parser tree.

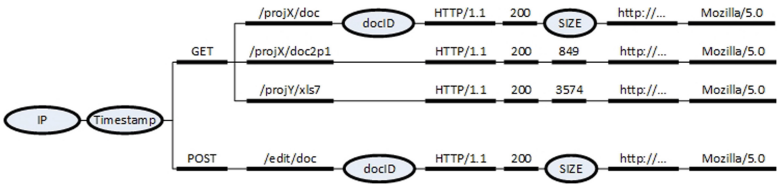
Example 1.4 visualizes a part of a parser tree for Web server access logs. This example demonstrates that the tree-based parser consists of three main building blocks. The nodes with bold lines represent tokens with static text patterns. This means that in all corresponding log lines, a token with this text pattern has to occur at the position of the node in the tree. Oval nodes represent nodes that allow variable text until the next separator or static pattern along the path in the tree occurs. The third building block is a branch element. The parser tree branches, when in a certain position only a small number of different tokens with static text occur.

In a nutshell, applying a tree-like parser model provides the following advantages, regarding performance and quality of log analysis:

- In opposite to an approach that applies distinct regular expressions, a tree-based parser avoids to parse over the same 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.
- Because of the tree-like structure, the parser model could be seen as a single, very large regular expression that models a system's log data. Therefore, the computational complexity for log line parsing is more like O(log(n)) than O(n) when handling data with separate regular expressions.
- The tree-like structure allows to reference 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 just pick out the data they need without searching the tree again. Furthermore, it allows to quickly apply anomaly detection algorithms to the different tokens and to correlate the information of different tokens within a single line and across lines.

Example: Using the generated templates given in Example 1.3, we steer the creation of a tree-like parser by applying the described parser generation methodology. Notice that the resulting tree may differ depending on the selected configuration parameters that influence whether different values result in a variable node,

denoted by a circle below, or a branching point. For instance, the request size would naturally be considered a variable value, however, if only two or three different sizes are recorded in the log data, it could also be modeled as three parallel branches each consisting of a static but different value.
Taking the cluster templates above, a resulting parser tree might look like this, whereas the blue bubbles reflect a variable node that may assume any 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 response code, url path, and user agent, the model becomes generally applicable. The table below 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 below.

```
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"
```

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

Example 1.4. Log line parsers

With the tree-like parser, we have now the means to match incoming log lines to observed structures and can thus categorize events. Furthermore, we are able to distinguish static from variable parts – a vital 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,

long short-term memory), feature selection is a mandatory prerequisite for analysis and anomaly detection.

6 System Behavior Modeling and ML-Based Anomaly Detection

Anomaly detection (AD) approaches [4] are more flexible than signature-based approaches and can detect novel and previously unknown attacks. They apply machine learning to determine a system's normal behavior through observation. The authors of [3] differentiate between six classes of AD algorithms. Statistical AD is a semi-supervised method: A model defines the expected behavior of the system and data deviating from this model are marked as anomalies. Statistical AD uses simple algorithm that may be challenged by complex attacks. In Classification based AD a classifier is trained on two or more categories of data, typically on benign and attack samples. Optionally, attacks could be sub-categorized, e.g., DoS attacks, intrusion and malicious software infections. In production mode the system signals samples categorized other than benign. Classification is supervised, depending on correct categorization of all training samples. Clustering based AD is an unsupervised anomaly detection method. Clustering is the process of assigning samples with common or similar properties – the so-called features – to distinct clusters. We discussed clustering earlier in this paper. Main challenges for clustering include the identification of anomalous clusters, as well as the definition of their bounds (i.e., finding the optimum boundary between benign samples and anomalous samples). Knowledge based AD is included for the sake of completeness. It utilizes a list of known attacks and for each data sample it compares whether it matches a known attack pattern. This can be done using regular expression on log data. Combination learning based AD (also known as ensemble methods) combine several methods for their decision. For example, one can include five different classifiers and use majority voting to decide whether a datum should be considered an anomaly. Although [12] lists Machine learning based AD as a distinct category, we argue that ML denotes an ensemble of methods and technologies that are typically used for classification and clustering.

Most "classic" machine learning approaches suffer from several drawbacks when applied for anomaly detection on log data as discussed in the introduction. Specifically, complex "monolithic" models are of limited use in an environment that undergoes frequent changes, such as triggered by updates in computer systems. Fine-granular, explainable models that may be adapted to new situations are required.

The AMiner [1] implements a wide variety of machine learning based anomaly detection approaches, ranging from rather simple (outlier) detection mechanisms that signal new types of events or new values at certain paths, to more complex sequence violation and time-series analysis. Figure 1 depicts the whole AMiner pipeline that provides interfaces to ingest and parse log data with the mechanisms described in this article, forwards the pre-processed data to the analysis

components and enables output on various channels, including console, message queues, syslog and e-mail. For the sake of brevity, we do not describe all mechanisms here in detail, but highlight the results of their application main in context of the example that we started earlier in this article.

Example 1.5 provides insights on the capabilities of some machine learning approaches in the area of simple detection, time series analysis, correlation analysis, and sequence detection. With these few approaches it becomes already rather hard for an attacker to mount a successful attack unnoticed. For more details on the concrete algorithms, we refer the reader to the scientific literature [7,14–16]; regarding the AMiner specifically to [12].

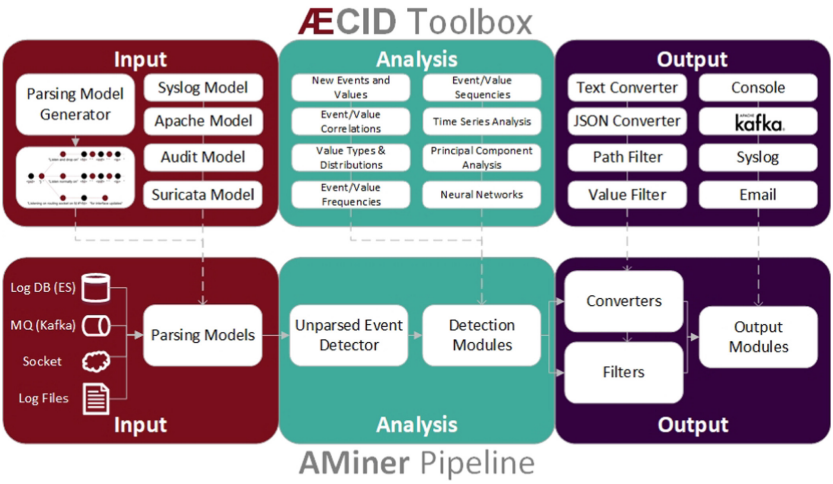


Fig. 1. The AMiner pipeline implements a wide variety of anomaly detection approaches applicable to log data.

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 that below. 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 in the first place. She, however, attempts to access a resource from a directory which was never retrieved by the client that she owns, e.g. "GET /projY/xls7". Since the IP "10.0.0.130" only occurred with resources "/projX/doc1", "/edit/doc2", "/projX/doc2", "/projX/doc2p1", and "/edit/doc1" up to this point, this triggers a new value combination of values at paths "/accesslog/hostip" and "/accesslog/request" that was never observed before. Notice, here the advantage of character-based templates comes into play. If a huge amount of documents reside within "/projX/" we could generally consider accesses to documents therein as normal, but may still alert on access to any documents in "/projY/". In addition, it would be possible to use even more paths to increase the granularity of value combination analysis. Specifically, adding the request method at path "/accesslog/method" to aforementioned paths does not only allow to analyze which resources are accessed by which users, but also include how they are accessed. Be aware that there is usually a limit to adding paths, since all value combinations must be observed at least once in the training phase to enable appropriate detection, i.e., 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 legitimate user agent string, however since she is in a hurry does that in bursts, i.e., downloads numerous resources in short time intervals.

Detection of frequency anomalies: The SARIMA [5] model predicts how many events of a specific type or of a certain source are considered normal, based on the history of observations, and enables alerting on any significant deviation. If, let's say 10-20 requests of documents per hour have been observed in the past few observation cycles for user 10.0.0.130, Mallory's attempt to retrieve documents in bulks (say, 100 requests made in one hour) will be detected.

CORRELATION ANALYSIS:

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

Detection of divergent correlations: Going back to what we consider normal behavior, the log data listing shows that 10.0.0.130 triggered 7 HTTP request, specifically 5 GET and 2 POST requests. After observing requests for a longer time span, 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, she will issue over time lots of GET request, but no POST request and therefore disturbs this established ratio. The Variable Correlation Detector aims to establish a baseline (i.e., an expected value that is considered normal) and alerts on any significant deviations from that. For example, using aforementioned data the learned model could describe that a reasonably sized sample of events have IP "10.0.0.130" occur with a GET request in 70% and a POST request in 30% of all cases, while for all other IP addresses this ratio is around 95% and 5%. Any

deviations reported by statistical tests on sufficiently large sections of the data, e.g., an increase of GET requests made by "10.0.0.130" to 90%, are then detected and reported as anomalies.

SEQUENCE DETECTION:
Stealthy attack: Mallory expanded 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 single 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 sub requests to fetch linked content (we assume here that caching is disabled on client side to make this example easier comprehensible, i.e., all linked resources are fetched every time a resource is requested). In the example, whenever doc2 is fetched, doc2p1 is fetched too – the detector thus learns the sequence "/projX/doc2" followed by "/projX/doc2p1" as normal behavior. Since Mallory attempts to crawl the page with a command line tool and not the browser, she fetches linked content only once and leaves out e.g., linked images, such as a site logo, which is embedded on every page. For the aforementioned example, this means that any new pair of subsequently requested resources is detected as an anomaly, e.g., the sequence "/projY/xls7" followed by "/projX/doc2". Also, session management might break, depending on the deployed Web technology stacks. 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 example above allows efficient learning, but has lower model granularity than larger sequence lengths that require long training phases and tend to overfit the data more easily.

Example 1.5. Application of log anomaly detection methods

7 Conclusion

Log data analysis and anomaly detection in computer networks need to cope with some significant challenges, such as frequent changes of the observed systems (which is not the case for other machine learning domains), a certain degree of adaptability of learned models (which is not the case for most classic machine learning approaches) and a high amount of produced complex data that need to be processed as streams (in contrast to offline multi-pass 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 able to discover malicious behavior. The disadvantage of using complex detectors however are that (i) they are much more complex to configure and maintain, (ii) it takes longer for them to learn, and (iii) they are prone to high false positive

rates. The art is to find the sweet spot between detecting enough to act in time and not drowning in false alerts.

The concepts for anomaly detection described in this chapter are deployed in the GUARD[1] cybersecurity framework. The GUARD framework specifically makes use of the tree-like parser approach and the light-weight AMiner agent that enables log-based online anomaly detection on host and service level.

For a more comprehensive view on the material covered in this article, please refer to [12].

Acknowledgements. This work was supported in part by the European Union H2020 project GUARD under grant 833456.

References

1. Aminer project on github. https://github.com/ait-aecid/logdata-anomaly-miner
2. Allen, R., Richardson, B.: Neural network, that's the tech; to free your staff from, bad regex (2019)
3. Bhuyan, M.H., Bhattacharyya, D.K., Kalita, J.K.: Network anomaly detection: methods, systems and tools. IEEE Commun. Surv. Tutor. **16**(1), 303–336 (2013)
4. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: a survey. ACM Comput. Surv. (CSUR) **41**(3), 1–58 (2009)
5. Cryer, J.D.: Time Series Analysis, vol. 286. Springer, Heidelberg (1986)
6. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: an online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 33–40. IEEE (2017)
7. Landauer, M., Skopik, F., Wurzenberger, M., Rauber, A.: System log clustering approaches for cyber security applications: a survey. Comput. Secur. **92**, 101739 (2020)
8. Landauer, M., Wurzenberger, M., Skopik, F., Settanni, G., Filzmoser, P.: Dynamic log file analysis: an unsupervised cluster evolution approach for anomaly detection. Comput. Secur. **79**, 94–116 (2018)
9. Notredame, C.: Recent evolutions of multiple sequence alignment algorithms. PLoS Comput. Biol. **3**(8), e123 (2007)
10. Skopik, F.: Collaborative Cyber Threat Intelligence: Detecting and Responding to Advanced Cyber Attacks at the National Level. CRC Press, Boca Raton (2017)
11. Skopik, F., Landauer, M., Wurzenberger, M.: Online log data analysis with efficient machine learning: a review. IEEE Secur. Priv. https://doi.org/10.1109/MSEC.2021.3113275. https://www.computer.org/csdl/magazine/sp/5555/01/09563044/1xvtlDhkcz6
12. Skopik, F., Wurzenberger, M., Landauer, M.: Smart Log Data Analytics: Techniques for Advanced Security Analysis. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-74450-2

[1] GUARD is a project co-funded within Horizon 2020 Funding Programme (833456). The project aims to provide a cybersecurity framework to guarantee reliability and trust for digital service chains. More information can be found on the project website: https://guard-project.eu/.

13. Vaarandi, R.: A data clustering algorithm for mining patterns from event logs. In: Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003) (IEEE Cat. No. 03EX764), pp. 119–126. IEEE (2003)
14. Wurzenberger, M., Höld, G., Landauer, M., Skopik, F., Kastner, W.: Creating character-based templates for log data to enable security event classification. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 141–152 (2020)
15. Wurzenberger, M., Landauer, M., Skopik, F., Kastner, W.: AECID-PG: a tree-based log parser generator to enable log analysis. In: 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 7–12. IEEE (2019)
16. Wurzenberger, M., Skopik, F., Landauer, M., Greitbauer, P., Fiedler, R., Kastner, W.: Incremental clustering for semi-supervised anomaly detection applied on log data. In: Proceedings of the 12th International Conference on Availability, Reliability and Security, pp. 1–6 (2017)
17. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 117–132 (2009)