AMiner: A Modular Log Data Analysis Pipeline for Anomaly-based Intrusion Detection

MAX LANDAUER, MARKUS WURZENBERGER, FLORIAN SKOPIK, WOLFGANG HOTWAGNER, and GEORG HÖLD, Austrian Institute of Technology, Austria

Cyber attacks are omnipresent and their rapid detection is crucial for system security. Signature-based intrusion detection monitors systems for attack indicators and plays an important role in recognizing and preventing such attacks. Unfortunately, it is unable to detect new attack vectors and may be evaded by attack variants. As a solution, anomaly detection employs techniques from machine learning to detect suspicious log events without relying on predefined signatures. While visibility of attacks in network traffic is limited due to encryption of network packets, system log data is available in raw format and thus allows fine-granular analysis. However, system log processing is difficult as it involves different formats and heterogeneous events. To ease log-based anomaly detection, we present the AMiner, an open-source tool in the AECID toolbox that enables fast log parsing, analysis, and alerting. In this article, we outline the AMiner's modular architecture and demonstrate its applicability in three use-cases.

CCS Concepts: • **Security and privacy → Intrusion detection systems**; **Systems security**; • **Computing methodologies → Semi-supervised learning settings;**

Additional Key Words and Phrases: Log data analysis, anomaly detection, intrusion detection systems

ACM Reference format:
Max Landauer, Markus Wurzenberger, Florian Skopik, Wolfgang Hotwagner, and Georg Höld. 2023. AMiner: A Modular Log Data Analysis Pipeline for Anomaly-based Intrusion Detection. *Digit. Threat.: Res. Pract.* 4, 1, Article 12 (March 2023), 16 pages.
https://doi.org/10.1145/3567675

1 INTRODUCTION

Cyber attacks are a permanent threat to computer system security. Recent reports indicate that this situation will only worsen in the future, as adversaries continue to develop new and targeted intrusion methods that bypass existing detection techniques [5]. To counteract such cyber attacks, system operators deploy **intrusion detection systems (IDSs)**. Thereby, the most widespread detection strategy is to monitor systems for so-called signatures, i.e., specific tokens such as hashes or IP addresses that are known to correspond to certain malware.

This project received financial support through the research projects CAIS (Grant No. 832345), CIIS (Grant No. 840842), and CISA (Grant No. 850199) in course of the Austrian KIRAS security research programme, the research projects synERGY (Grant No. 855457) and DECEPT (Grant No. 873980) in course of the ICT of the future programme of the Austrian Research Promotion Agency (FFG), the research project PANDORA (Grant No. SI2.835928) in course of the European Defence Industrial Development Programme (EDIDP), as well as the research projects ECOSSIAN (Grant No. 607577) and GUARD (Grant No. 833456) in course of the European Seventh Framework Programme (FP7) and Horizon 2020.
Authors' address: M. Landauer, M. Wurzenberger, F. Skopik, W. Hotwagner, and G. Höld, Austrian Institute of Technology, Giefinggasse 4, Vienna, Austria, 1210; emails: {max.landauer, markus.wurzenberger, florian.skopik, wolfgang.hotwagner, georg.hoeld}@ait.ac.at.

2576-5337/2023/03-ART12
https://doi.org/10.1145/3567675

For this, signature-based IDSs process high volumes of log data or network traffic and carry out pattern matching. Despite usually achieving high effectiveness in detecting known attacks, signature-based approaches have the strong disadvantages of being unable to detect unknown attacks or variants for which no signatures exist, and requiring frequent updates of their signature databases [15].

Anomaly-based IDSs alleviate these issues by leveraging self-learning techniques that automatically generate models for normal system behavior and detect all activities that diverge from this baseline as potentially malicious [3]. Anomaly detection enables the detection of new attack vectors, however, usually also generates more false positives than signature-based approaches due to the fact that unusual but otherwise benign activities are possibly detected, e.g., the presence of new devices in the monitored network. To counteract this problem, applied algorithms need to continuously adapt their trained models to provide a dynamic baseline for detection.

Anomaly detection has been successfully demonstrated in network traffic analysis [1], however, system log data has become an increasingly important source for detection as network traffic is almost always encrypted in modern networks and thus difficult to analyze [15]. System log data is a rich source of information as it keeps track of almost all events that take place in a system and often serves as the basis for forensic investigations after incidents [4]. Thereby, it is important that logging levels are adequately configured so that both normal as well as anomalous events are captured and that application developers make informed logging decisions to support analysts [30]. However, log data is difficult to analyze as it occurs in many different formats and dependencies of event occurrences as well as event parameters relate to complex underlying workflows of applications. Moreover, the structures of log events are subject to change over time when the software that generates logs is updated [29], which may render (parts of) the trained model useless.

As a consequence, existing approaches for log-based anomaly detection are either optimized for specific types of log sources for which custom parsers are provided or neglect large parts of the logs. Moreover, they only support limited self-learning mechanisms, such as detection of outliers (e.g., a new device) or sudden spikes in event frequencies. Unfortunately, there is no one-fits-all solution as different systems require tailored configurations for data preprocessing and detection. As a solution to these problems, we present the open-source tool AMiner,[1] a modular analysis pipeline for log data that enables fast parsing of events, machine learning techniques for detection, and an interface for anomaly reporting. The AMiner is published as part of the **Automatic Event Correlation for Incident Detection (AECID)**[2] [28] framework that comprises many tools for system log processing and analysis. We summarize the contributions of this article as follows.

- The AMiner architecture constituting a modular pipeline for log data analysis and intrusion detection, and
- a case study of three attack detection scenarios.

The remainder of the article is structured as follows. Section 2 reviews commercial and scientific approaches for anomaly detection in log data. Section 3 describes the architecture of the AMiner. In Section 4, we present three use-cases of the AMiner before discussing real-world deployment strategies in Section 5. Finally, Section 6 concludes the article.

2 RELATED WORK

As log data is a vital source of information for developing and debugging software, it has been around as long as modern computers themselves. However, the sheer volume of logs that is generated nowadays makes manual investigation of logs a cumbersome and error-prone task. Commercial vendors of **Security Incident and Event Management (SIEM)** systems realized that system operators struggle to organize log data and find relevant events, and therefore sell products that fulfill these requirements. One of the best-known tools for log analysis is Splunk,[3] which enables indexing and querying log data and provides a convenient dashboard for analysis.

[1]AMiner GitHub repository, https://github.com/ait-aecid/logdata-anomaly-miner (accessed: 2022-09-09).
[2]AECID GitHub repository, https://github.com/ait-aecid (accessed: 2022-09-09).
[3]Splunk website, https://www.splunk.com/ (accessed: 2022-09-09).

Similarly, insightOps[4] allows analysts to define patterns that are detected as alerts when present in the logs. Papertrail[5] and Mezmo[6] enable flood detection to recognize spikes in event counts, however, rely on manually defined search queries and limits for the maximum allowed logs per second. Datadog[7] and QRadar[8] go one step further and apply models from time-series analysis to address periodic patterns of log frequencies for spike detection. Wazuh[9] is an open-source solution, but only supports detection by predefined rules.

The main problem is that all of these tools require human operators to clearly define detection rules that trigger alerts, but there are hardly any features that leverage self-learning algorithms. Simple learning mechanisms are usually restricted to specific properties and still need manually defined thresholds, e.g., Loggly[10] computes an anomaly score for event counts that deviate from previous frequencies. In addition, commercial solutions are usually closed-source, which means that they are black boxes where detection algorithms are difficult to fully comprehend. These limitations of commercial tools has drawn many researchers to propose scientific approaches for log-based anomaly detection [2].

One of the earliest approaches on self-learning anomaly detection in log data detects new sequences of system operations using sliding windows [8]. More than 20 years later, researchers are still working on new approaches for anomalous sequence detection, but usually employ neural networks. For example, DeepLog [7] utilizes and incrementally updates an LSTM (Long Short-Term Memory) to detect unusual sequences of log events. LogRobust [29] is a similar approach that is specifically developed to handle noise in log data generated by changing logging statements and data collection. An alternative to these approaches is invariants mining that extracts linear relationships between log events [12].

There are also approaches that focus on other properties of the data to derive patterns and detect anomalies, including similarity-based log clustering with string metrics [27] or n-grams [13], principal component analysis of event counts [12], interval times between event occurrences [10], markov chains [11], and attribute properties such as lengths, orders, and presence [16]. Furthermore, some approaches generate user profiles from log data using **User Entity Behavior Analytics (UEBA)** to detect insider threats [23]. While the majority of these scientific works propose detection techniques for one specific feature or metric derived from the data, the AMiner is a generic log processing pipeline including many different types of detectors that come with the default installation. Several of these detectors are also backed up by scientific papers [9, 17, 20, 25]. As a fully open-source tool, the AMiner also supports custom detectors designed by analysts that are integrated as modules.[11] The following sections will provide more information on the available detection mechanisms.

3 ARCHITECTURE

This section first outlines the concept of anomaly-based intrusion detection and then describes the log-processing pipeline and all involved modules of the AMiner.

3.1 Concept

As outlined in Section 2, commercial intrusion detection systems predominantly make use of signature-based detection techniques when analyzing network traffic or system log data. While this is a highly efficient method for detecting known attacks, it is unable to recognize any new or unknown attacks for which no signatures

[4]Rapid7 insightOps website, https://www.rapid7.com/products/insightops/ (accessed: 2022-09-09).
[5]Papertrail website, https://www.papertrail.com/ (accessed: 2022-09-09).
[6]Mezmo website, https://www.mezmo.com/ (accessed: 2022-09-09).
[7]Datadog website, https://www.datadoghq.com/ (accessed: 2022-09-09).
[8]QRadar website, https://www.ibm.com/products/qradar-siem (accessed: 2022-09-09).
[9]Wazuh website, https://wazuh.com/ (accessed: 2022-09-09).
[10]Loggly website, https://www.loggly.com/ (accessed: 2022-09-09).
[11]AMiner custom detectors tutorial, https://github.com/ait-aecid/logdata-anomaly-miner/wiki/HowTo-Create-your-own-SequenceDetector (accessed: 2022-09-09).

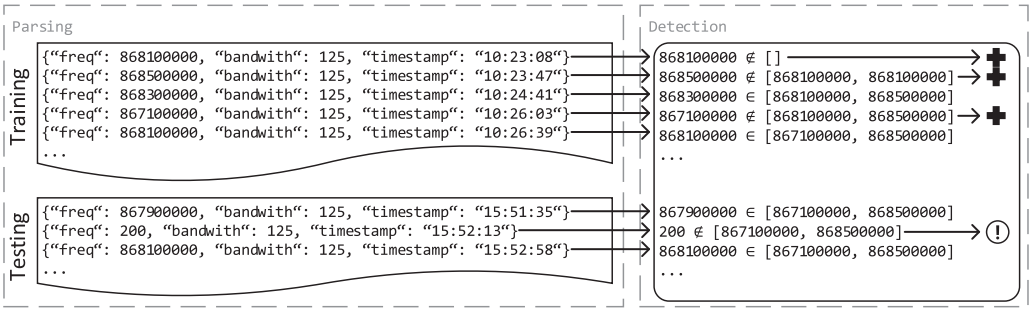

Fig. 1. Detection of anomalies in parsed log data by learning minima and maxima of monitored values. During training, the model is continuously updated whenever a new value lies outside of the learned ranges. In the testing phase, anomalies are reported for such outliers, but ranges remain unchanged.

exist and furthermore requires that the lists of signatures are manually created and maintained [15]. To avoid such problems, the AMiner leverages anomaly-based detection techniques that automatically learn a baseline of normal behavior and then report any divergences of this model as anomalous and potentially dangerous to the system operators. Since this is a different approach to intrusion detection, we do not pursue to replace or even compete with existing signature-based IDSs; instead, the purpose of the AMiner is to provide an additional line of defense in system environments where signature-based IDSs (and possibly other anomaly-based IDSs) are already in place.

Consider Figure 1 as an example for anomaly-based detection as realized by the AMiner. On the left side of the figure, a log source containing events in JSON format is ingested by the parsing component of the AMiner that extracts relevant values. Note that the logs appear in chronological order (as indicated by the timestamp). We differentiate between logs occurring during a training phase that is expected to be free of anomalous behavior and a testing phase that contains in addition to normal activity also events related to attacks or otherwise unusual system utilization that should be detected. The right side of the figure depicts detection based on the numeric ranges of the *freq* value in the logs, which represents the communication frequency of the LoraWAN gateway producing the logs (cf. Section 4.1). In this case, the detector learns the minimum and maximum of the observed values and stores them as the baseline reference model. This model is initially empty and continuously updated whenever a newly observed value falls outside of the current range, e.g., the value 867,100,000 occurring in the fourth line is lower than the minimum value and thus replaced for the subsequent comparisons. While comparison works equally for the testing phase, the model is not updated anymore and all values that fall outside of the range are reported as anomalies.

This principle of continuously checking and updating models is at the core of the AMiner's detection approach. In the next section, we provide a detailed list of detectors and embed them into the overall log-processing pipeline of the AMiner.

3.2 Pipeline

This section introduces a log-processing pipeline that ingests and analyzes log data, detects unusual patterns, and creates reports for anomalies.

3.2.1 Overview. The AMiner is implemented as a generic pipeline for log processing and makes use of several modules and external components to carry out log-based anomaly detection. These modules are developed as part of the AECID[12] [28] toolbox that provides solutions for many log-related issues that go beyond the AMiner.

[12]AECID website, https://aecid.ait.ac.at/ (accessed: 2022-09-09).

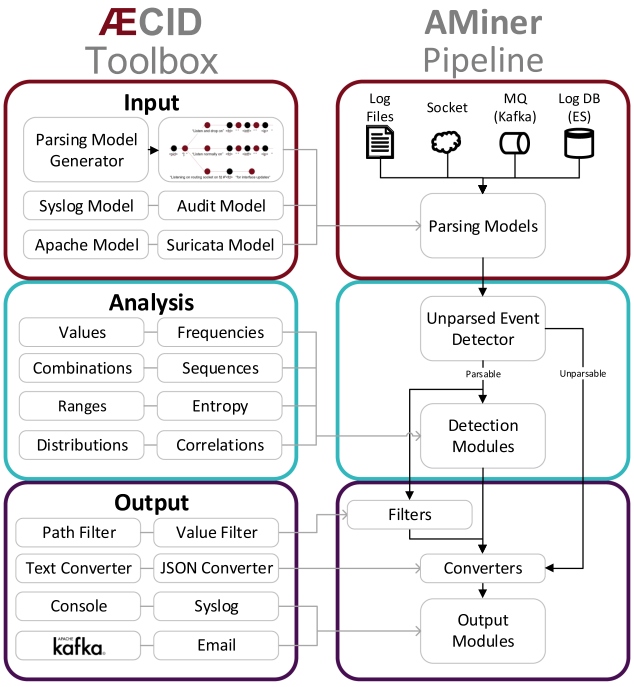

Fig. 2. Usage of AECID modules in the AMiner pipeline.

Figure 2 provides an overview of all involved AECID components and their relation to the AMiner pipeline. As visible in the figure, the pipeline is divided into three main stages. First, log data is ingested from one or more sources and parsed with one of the models provided by the AECID toolbox (*Input* stage). Then, parsed events are forwarded to all active detection modules, which create models for normal behavior and generate anomalies (*Analysis* stage). The figure shows only a general overview of different detector types, in particular, detectors for new values or value combinations, numeric ranges, event frequencies or sequences, and various statistical properties. Note that unparsed events immediately result in anomalies, since they represent unknown events and thus always need to be considered potentially malicious. Finally, all logs and generated anomalies are filtered, converted, and eventually reported to system operators by the respective modules (*Output* stage). In the following, each of the stages is explained in more detail.

3.2.2 Input. The AMiner currently supports four types of log sources. In case that log data is generated on the same host where the AMiner is installed or log files are analyzed forensically, it is simple to point the AMiner directly to the path of the file or socket. Alternatively, add-ons allow to receive log data from a message queue such as Kafka[13] or read them from a log database, in particular, Elasticsearch.[14] In any case, the AMiner will first attempt to process all currently available logs (e.g., start from the top of the log file) and otherwise wait for newly occurring logs.

One of the main difficulties when handling log data is the abundance of different formats and syntaxes, e.g., key-value pairs (audit logs), parameter sequences (apache logs), human-readable text (syslog), JSON format (suricata event logs), and so on. There is therefore a need for log parsers that have two main tasks: (i) classification of log events, where each event follows a specific syntax, and (ii) value extraction, which assigns a unique identifier to

[13] AMiner Kafka interface repository, https://github.com/ait-aecid/aminer-akafka (accessed: 2022-09-09).
[14] AMiner Elastic interface repository, https://github.com/ait-aecid/aminer-aelastic (accessed: 2022-09-09).

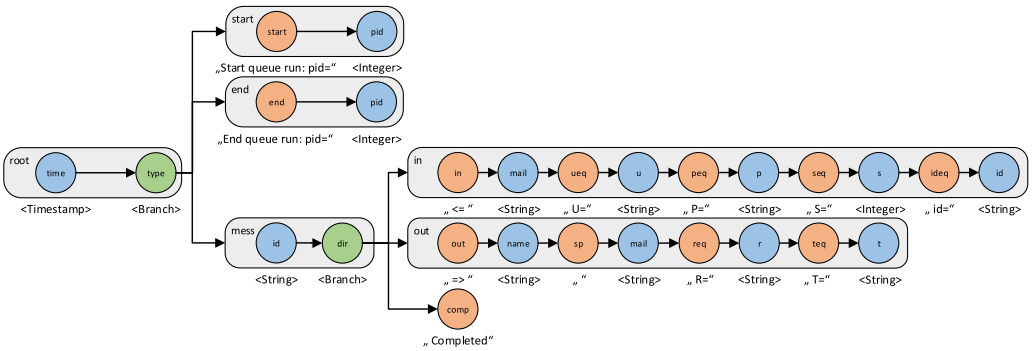

Fig. 3. Parser tree for sample log events.

```
2020-02-29 06:32:13 1j7vfZ-0000Sy-2C <= georgie@mail.cup.com U=www-data P=local S=4162 id=20200229063212.Horde
2020-02-29 06:32:13 1j7vfZ-0000Sy-2C => karri <karri@mail.cup.com> R=local_user T=mail_spool
2020-02-29 06:32:13 1j7vfZ-0000Sy-2C => portia <portia@mail.cup.com> R=local_user T=mail_spool
2020-02-29 06:32:13 1j7vfZ-0000Sy-2C Completed
2020-02-29 06:34:25 Start queue run: pid=1817
2020-02-29 06:34:25 End queue run: pid=1817
```

Fig. 4. Sample log events from the Exim mail service.

each event parameter [18]. Since log data occurs in high volumes, it is essential that log parsers carry out these tasks with high efficiency. Parser trees thereby have a strong advantage over lists of regular expressions with respect to runtime performance, in particular, when many different events are present in the logs [26]. The main reason for this is that each token in any incoming log event only needs to be processed once as all allowed log events are represented in a single data structure. Accordingly, all standard parsers that come with the AMiner leverage tree-structures for parsing. In case that none of these parsers fits the log data at hand, the AECID toolbox also provides a parser generator[15] [26] that automatically analyzes samples of log files and generates a tree-based parser usable by the AMiner.

Figure 3 visualizes an exemplary parser tree for sample log events from the Exim mail service[16] displayed in Figure 4. The parser tree consists of nodes connected to sequences (gray colored nodes) and branches (green colored nodes). For example, the *time* node that represents the timestamp is part of the *root* sequence. Since parameters in log events usually have specific types, we provide different types of nodes to ensure that incorrect data types are immediately detected and potentially necessary conversions are already carried out during parsing. In particular, we provide nodes for static strings (orange colored nodes) that are also used to define different events as branches and variable nodes (blue colored nodes) such as strings with specific character sets, integers, floats, timestamps, hexadecimal values, and JSON objects. For example, the *pid* node only accepts integers and generates anomalies for all other data types. Moreover, there is the possibility to mark nodes as optional, since this is sometimes the case for parameters in event syntaxes. Parser trees fulfill the aforementioned requirements regarding event classification and value extraction. First, we define the event type of a processed log line by the set of nodes traversed when parsing all tokens, i.e., the path from the root node to one of the leave nodes of the parser tree. Second, we accomplish value extraction by treating nodes of parser trees similar to directory structures of operating systems and thus generate unique parser paths for each value. For example, the parser path of the timestamp is */root/time*, where *root* is the first sequence node and thus the root node of the parser tree, and *time* is

[15] AECID parser-generator, https://github.com/ait-aecid/aecid-parsergenerator (accessed: 2022-09-09).
[16] Exim Internet Mailer Website, https://www.exim.org/ (accessed: 2022-09-09).

the node that corresponds to the timestamp and is located within the *root* sequence. Parser paths to nodes deeper down the parser tree leverage sequence and branch elements to differentiate between different event types, for example, the path to the user name (*karri* in the second line of Figure 4) is */root/type/mess/dir/out/name*.

Note that parsers occasionally require updates; in particular, when the software of applications that generate analyzed log data changes across versions [29]. In these cases it is either necessary to map parser nodes according to the new log syntaxes, or re-train affected parts of the model.

3.2.3 Analysis. The first simple detection that is always carried out is closely linked to parsing as described in the previous section. In particular, events with unknown syntax cannot be parsed or analyzed and are thus immediately reported as anomalies, since they are unexpected and thus need attention of system operators. In case that the logs are considered to correspond to normal behavior, it is then necessary to update the parser trees to also cover the reported event, or to rerun the parser generator and add the line to the training logs.

All parsed logs are then forwarded to the detectors present in the AMiner pipeline. The list of active detectors is set in the AMiner configuration file, where each detector and its particular parameterization is specified [25]. Note that it is often useful to add the same detector type multiple times with different parameters, in particular, when distinct event types should be analyzed separately. Table 1 shows a list of all detectors that are currently implemented for the AMiner. Note that except for the *AllowlistViolationDetector* and *MatchFilter* that make use of predefined signatures, all detectors are self-learning, meaning that they automatically generate a model corresponding to normal behavior analogous to the *ValueRangeDetector* as depicted in Figure 1. The detectors cover a wide range of methods, including simple value comparisons, event count matrices for time-series analysis [20] and principal component analysis [12], event sequences [8], statistical tests for distributions [24], and correlation analysis for invariants across event types [9] as well as categorical values within the same event types [17]. All detectors are configurable on a high granularity to enable fine-tuning for specific applications.

As there is usually no training log data file that contains labeled attacks in practice, all detectors leverage either unsupervised learning, i.e., models are generated, adapted, and checked for anomalies indefinitely, or semi-supervised learning, i.e., there is a dedicated configuration parameter that differentiates between training of only normal behavior and detection with mixed data as described in Section 3.1. Moreover, the detectors are designed for incremental learning to adequately handle log data that arrives in continuous streams. In particular, the detectors either employ one-pass algorithms that only need to process each event once, or alternatively analyze chunks of logs occurring in time windows. Finally, efficient detection algorithms are used to keep up with high-volume log data of several thousand events per second.

3.2.4 Output. In addition to anomalies generated by detectors, anomalies may also be raised by filters that select logs based on the type of log event (Path Filter) or specific value in one of the events (Value Filter). This is especially useful for handling warning or error logs. All anomalies (including those related to unparsed logs) are eventually transferred to the converting stage, where the anomaly output is prepared. AMiner anomalies are either in textual or JSON format and involve several mandatory attributes, including the name and type of detector that raised the anomaly, the raw log events that are detected as anomalous, the timestamp of the log event, and a human-readable message, as well as several detector-specific and customizable attributes, e.g., affected paths of the parser tree and corresponding values.

The anomalies are then transferred to the final step of the AMiner pipeline that interfaces other services and notifies human system administrators. The most simple form of reporting is console output, where all anomalies are printed on the same terminal where the AMiner is running. Alternatively, the AMiner is capable of creating new log entries for anomalies (which can be in turn consumed by the AMiner) or automatically sending emails to a given address. Finally, the AMiner is also able to forward anomalies in JSON format to other services such as Elasticsearch via Kafka. This is especially important to import and collect anomalies together with other alerts

Table 1. Detectors in the AECID Toolbox

Detector	Description
AllowlistViolationDetector	Detects violations of rules, e.g., a value is not from a predefined set of values.
CharsetDetector	Detects new characters in values.
EnhancedNewMatchPath-ValueComboDetector	Detects new value combinations, saves combination occurrence times and frequencies, and supports value transformation functions.
EntropyDetector	Detects changes of character distributions.
EventCorrelationDetector	Detects changes of event correlations, e.g., event B follows event A with 95% certainty.
EventCountClusterDetector	Detects changes of occurrence frequencies of values or events through clustering of count vectors.
EventFrequencyDetector	Detects changes of occurrence frequencies of values or events through statistical analysis.
EventSequenceDetector	Detects new sequential occurrences of values or events, and untangles interleaved traces, e.g., by process IDs.
MatchFilter	Detects value matches for predefined lists.
MatchValueAverage-ChangeDetector	Detects deviations of averages and variances of monitored values and reports summaries.
MinimalTransitionTime-Detector	Detects lower deviations of observed time intervals between value occurrences.
MissingMatchPath-ValueDetector	Detects values that fail to occur in expected time windows, e.g., unexpectedly stopped services.
NewMatchPathValueDetector	Detects new values that occur in specific parser paths, e.g., newly started services.
NewMatchPathValue-ComboDetector	Detects new combinations of values, e.g., new combinations of user ID and executable.
NewMatchIdValue-ComboDetector	Detects new combinations of values across linked events, e.g., using process IDs.
PathValueTimeIntervalDetector	Detects value occurrences that violate periodic patterns, e.g., specific times of day.
PCADetector	Detects unusual value occurrence counts with principal component analysis.
PathArimaDetector	Detects unusual value occurrence counts in specific paths with time-series analysis.
TSAArimaDetector	Detects unusual occurrence frequencies of events with time-series analysis.
TimestampsUnsortedDetector	Detects events that do not occur in chronological order, e.g., due to misconfigurations.
ValueRangeDetector	Detects numeric values outside of usual ranges, i.e., minimum and maximum limits.
VariableCorrelationDetector	Detects changes of co-occurrences of categorical values with statistical tests.
VariableTypeDetector	Detects changes of continuous and discrete distributions with statistical tests.

in SIEM solutions such as QRadar [25] that handle alert correlation and provide graphical overviews. There also exists a Kibana dashboard[17] specifically implemented for the AMiner that is publicly accessible.

4 CASE STUDY

This section demonstrates 3 of 23 available detectors for numeric values, string values, and event correlations.

4.1 Value Range Detector

System log data as well as network traffic frequently contain numeric values. For example, netflows involve time durations and numbers of transmitted bytes, system logs state event counts such as how many failed authentication attempts occurred, monitoring logs record system data such as CPU and memory usage, and devices in cyber-physical systems produce logs for all kinds of measurement data such as temperature, humidity, or position. We developed the value range detector specifically to detect anomalies in such numeric data. In a semi-supervised setting, the detector learns the minimum and maximum of observed values during the training phase and then reports anomalies for all values outside of this range during the detection phase (cf. Section 3.1). Note that it is also possible to run the AMiner in unsupervised mode where learning never stops, i.e., values that

[17] AMiner dashboard, https://github.com/ait-aecid/aminer-dashboard (accessed: 2022-09-09).

Fig. 5. Excerpt of Apache access logs containing the consequences of a remote command injection. The event affected by the attack is marked as anomalous, because its request comprises many unusual character combinations.

fall outside of the previously learned range cause that the range is extended accordingly and at the same time generate anomalies. In this setting, anomaly frequency is high in the beginning but gradually declines as the ranges become more stable and accurate to the true distributions. This also means that true positives such as actual attack traces that incorrectly modify the ranges need to be manually or automatically removed (e.g., by removing old values), or the models retrained.

Our demonstration targets a smart mobility use-case where sensors connected to a LoraWAN network are placed in public transport vehicles. The LoraWAN gateways thereby keep track of the connected devices and produce logs in JSON format that contain a number of parameters, including timestamps, frequencies, bandwidths, channel numbers, identifiers, and so on. Figure 1 shows a simplified sample of such log events. We monitored the logs from these gateways and apply the value range detector on the logged frequencies. After training the detector for several hours, the minimum and maximum of the observed frequencies were 867,100,000 Hertz and 868,500,000 Hertz, respectively. We then switched the detector from training to detection mode and launched a malicious process for uplink Denial-of-Service (CVE-2020-28349), which allows the attacker to hijack a gateway and deactivate all others. As part of this attack, events with a frequency of 200 Hertz are generated, which is outside of the learned interval and thus correctly reported as anomalous.

The main idea behind the value range detector is that not all possible benign values need to be observed, which speeds up learning in comparison to treating the frequencies as categorical variables. This means that all other frequencies in the LoraWAN frequency plans[18] within the identified minimum and maximum value are automatically assumed to be benign even though they were not directly observed in the data. Of course, this also comes with the drawback that malicious frequencies located within the range cannot be detected. Such cases could be resolved by more advanced detectors than the simple yet effective value range detector, such as detectors for statistical measures (MatchValueAverageChangeDetector) or distributions (VariableTypeDetector).

4.2 Entropy Detector

Apache access logs keep track of which pages and resources on a web server are requested by users. Among other attacks, these logs also reveal remote command injections that are carried out by exploiting vulnerabilities of installed services and plugins [22], which makes them an interesting source for anomaly detection. Apache access logs have a relatively simple syntax as they only consist of a list of parameters, in particular, timestamp, request method, requested resource, status code, etc. While all other values are categorical, the requested resource is an arbitrary string that is non-trivial to analyze for anomalies as it requires semantic interpretation.

For our demonstration, we set up a web server running WordPress[19] and collect Apache access logs during normal usage. Figure 5 displays a sample of these logs and shows that the requested resources are complex and include repeating strings and parameters, e.g., four resources start with */wp-includes/js/*. For such cases, we apply the entropy detector that generates a model of the usual character distributions and detects those values with

[18]LoraWAN frequency plans, https://github.com/TheThingsNetwork/lorawan-frequency-plans/blob/master/EU_863_870.yml (accessed: 2022-09-09).
[19]WordPress website, https://wordpress.com/ (accessed: 2022-09-09).

unlikely character pairs as anomalies. More specifically, the detector counts how many times each character is followed by any other character and thereby creates a frequency table. Once sufficiently many values have been analyzed, the relative occurrence frequencies of characters stabilize and allow to detect anomalies by aggregating the probabilities of all present character pairs in the new string and declaring it as anomalous if the resulting value is too low. This concept is based on the well-known *freq* tool.[20] Note that we also consider the probabilities of a string starting and ending with specific characters as relevant and therefore form pairs for the (non-existing) 0th and first character as well as the nth and (non-existing) $(n + 1)$th character for a string of length n.

We train the detector on 7,886 access log lines recorded over four days and manually verify the generated frequency table. As expected, a majority of the lines (7,559) start with a slash ("/") character, some (310) with an asterisk ("*"), and 17 lines do not involve a request and are therefore ignored. The slash is most often followed by the character w (9,379), which is reasonable, since most lines start with one of */wp-includes/*, */wp-content/*, */wp-admin/*, and so on, followed by the character j (3,953), which is due to the fact that */js/* is a common substring, and so on. In total, the frequencies of 849 character pairs (with 56 unique characters) are found by the detector.

We then switch to the detection phase and run the detector on a dataset containing attack traces. The probabilities of all character pairs of the observed values is computed, e.g., the probability of a value starting with a slash is 96% and the probability of w following a slash is 34%. The detector computes the average of these values and reports anomalies if the result is lower than a certain threshold that we set to 15%. To put that into perspective, the relatively common benign string */wp-includes/js/wp-embed.min.js?ver=5.8.1* yields a total probability of 37%. However, the request */static/evil.php?cmd=netcat%20-e%20/bin/bash%20192.168.10.238%209951*, which depicts a remote command injection over a web shell only receives a probability of 10% and is therefore correctly detected.

While it is possible to train the detector on any input data, e.g., literature or lists of common websites, it is recommended to train it directly on the data where anomalies should be detected. The reason for this is that the model will fit much better to the peculiarities of the observed data, e.g., in our demonstration most strings start with a slash. We provide an in-depth explanation of this detector, including the data and AMiner configuration used in the demonstration, online.[21]

4.3 Event Correlation Detector

Events in system logs usually form specific sequences that are often recurring. The reason for this is that the logic of the monitored application produces log events in patterns that reflect the underlying workflows. However, since these workflows depend on specific circumstances, e.g., user input, and are subject to delays and concurrent processes, it is not always guaranteed that events occur in precisely the same order, but rather involve swapped, additional, or missing events. Nonetheless, given a sufficiently long time for training, it is possible to extract dependencies between log events that are expected to hold with a certain probability. Violations of these patterns occurring during normal operation represent anomalies that should be reported.

We differentiate between two types of dependencies. First, we define a forward rule as the occurrence of an event A at time t_A that implies that event B subsequently occurs within a certain interval δ, i.e., $t_B \in [t_A, t_A + \delta]$. Second, a backward rule is the occurrence of an event A that implies that event B must have occurred before no longer than δ, i.e., $t_B \in [t_A - \delta, t_A]$. The event correlation detector automatically creates and evaluates such rules and is thus able to detect changes of the workflow. For this, the detector randomly generates a correlation hypothesis of two events that fulfill the requirements for a either a forward or backward rule. The detector then evaluates this hypothesis for a given number of occurrences of event A, say, 100 occurrences. In case that the hypothesis evaluates true sufficiently many times, say, 95%, the dependency is declared stable and transformed into a rule that is subsequently checked, and otherwise discarded. For rules that do not hold with 100% probability,

[20] freq GitHub repository, https://github.com/sans-blue-team/freq.py (accessed: 2022-09-09).
[21] AMiner entropy detector tutorial, https://github.com/ait-aecid/logdata-anomaly-miner/wiki/HowTo-EntropyDetector (accessed: 2022-09-09).

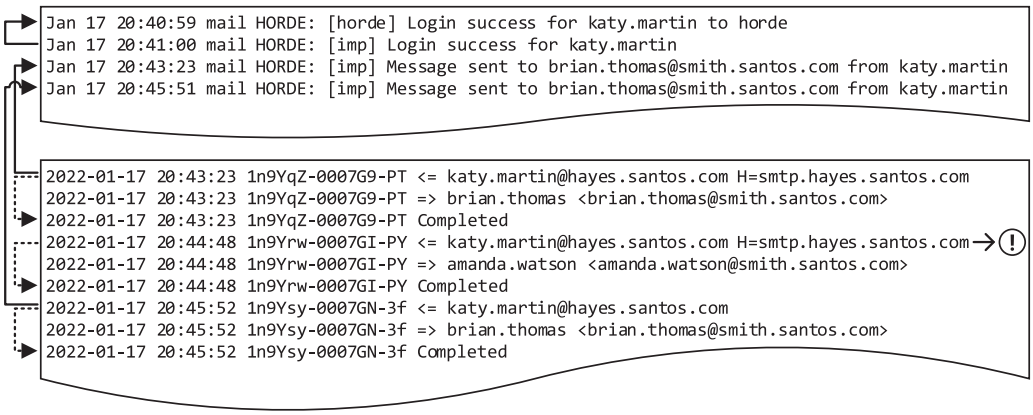

Fig. 6. Event correlations in messages (top) and exim (bottom) logs indicated by arrows, where correlations related to forward and backward rules are displayed as dashed and solid lines, respectively. The marked exim log event is anomalous as the correlating message event that usually occurs prior or simultaneous is absent.

we need to take into consideration that rule violations could randomly accumulate in one of the samples and cause false positives. We therefore use a binomial process to compute the minimum amount of times the rule needs to hold in the sample before an anomaly is reported. For example, a rule that is expected to hold with 95% only needs to evaluate true in 89 times of 100 occurrences to be considered fulfilled at a significance level of $\alpha = 0.01$ [9].

We demonstrate the detector on an actively used Horde[22] mail server. In particular, we use the messages log file that contains events generated by the Horde application as well as the log file of the Exim mail service[23] as both generate different events for sending and receiving mails, among other events. Figure 6 displays samples of these log files. Note that we omit several event parameters in the figure for brevity (cf. sample exim log shown in Section 3.2.2). Since the AMiner processes events from both files in chronologically correct order, we expect that the correlation detector learns the dependency that an event occurrence in the exim logs related to sending mails implies that another event occurs in the messages log file documenting the same action. As the events occur almost instantaneously, we set $\delta = 3$ s and furthermore use a sample size of 300, minimum probability of 99%, and $\alpha = 0.01$. Running the AMiner with the detector on both log files yields a total of 12 back rules and 1 forward rule. Most of these rules relate to event correlations within the same file, e.g., each login of a user generates two events in the messages log file that always follow each other, which is visible as the correlation corresponding to the backward rule (arrow with solid line) in the top of Figure 6.

To test the learned rules, we assume that an attacker intrudes the system and as part of a vulnerability exploit makes use of the mailing service. However, since the attacker is not using the Horde platform as normal users do, a corresponding log event is only generated in the exim log file but not in the messages log file. This is visible in Figure 6, where one of the exim events occurring at 20:44:48 does not have a corresponding event in the messages log file and is thus marked as anomalous. Even though this is just a single missing event, the detector successfully recognizes the rule violation, since the events occurred within each evaluation in the sample size and thus the rule needs to hold in 100% of the cases. More details on this use-case, including all detector configuration parameters as well as the log data, are available online.[24]

[22]Horde webmail, https://www.horde.org/apps/webmail (accessed: 2022-09-09).
[23]Exim Internet Mailer Website, https://www.exim.org/ (accessed: 2022-09-09).
[24]AMiner correlation detector tutorial, https://github.com/ait-aecid/logdata-anomaly-miner/wiki/HowTo-CorrelationDetector (accessed: 2022-09-09).

5 APPLICATION

This section discusses practical applications of the AMiner, in particular, its configuration, execution, and deployment.

5.1 Configuration

The AMiner supports many different application cases and deployment scenarios. Depending on the desired use-case, it is therefore necessary to configure the AMiner accordingly. To make the AMiner's configuration as accessible as possible, we use YAML as configuration language. The configuration file comprises (i) global parameters, such as the overall learn mode and the list of input sources, (ii) the parser tree reflecting the structure of log lines, (iii) input parameters such as the synchronization of multiple log sources to ingest logs in chronologically correct order despite possible delays, (iv) the list of detectors, filters, and analysis components added to the pipeline, and (v) the output modules and converters. More details on the AMiner configuration file and sample configurations are available in the Wiki[25] and documentation.[26]

We already discussed the differences between training and testing phases in Section 3.1 and recognize that it is often difficult to differentiate between them in practical applications for two reasons. First, it is usually not possible to guarantee that no malware is already active during training and incorrectly included in the models, a problem that is usually referred to as poisoning [6]. We argue that this problem is immanent by the definition of anomaly detection and that specific detectors are needed to recognize such cases. Second, it is non-trivial to decide at what point in time the learned models are representative enough so that the training phase should be stopped. Since the number of anomalies detected in the training phase should decrease over time as models become more stable, we recommend to count the number of anomalies in a given time window and stop the training phase when an acceptable limit of false positives per time window has been reached. As many systems exhibit periodic behavior, it is generally recommended to set the duration of the training phase to cover at least one period of normal system behavior, for example, one week when different system utilization is expected on weekends in comparison to weekdays. Alternatively, the training phase could be extended indefinitely and every reported anomaly checked whether it is a true positive, in which case it needs to be removed from the models to ensure that it will be detected again in the future. In general, the problem of many false positives may be alleviated by analysis and aggregation of generated anomalies[27] [19].

5.2 Execution

The AMiner is usually applied either for forensic or online analysis. Forensic analysis means that there is one or more static log datasets (possibly split into training and detection files) at hand that are analyzed. For such applications, the AMiner is conveniently executed on the command line and may be used with the *–offline-mode* flag to stop the AMiner once all lines are processed. This is the standard case for experimentally validating detectors on log datasets and fine-tune configurations before applying them in real environments.

In productive use, the AMiner performs online analysis of logs that arrive in streams. For such applications, it is possible to run the AMiner as a service that is started and stopped via the system. Changes of the configuration can be made by restarting the AMiner without loss of data, since all models are persisted to disk before terminating the program. In addition, the AMiner by default stores the last processed position in the log file on exit so that the same logs are not processed again when restarting the AMiner. To always start from the oldest log event, the *–from-begin* flag is used. Finally, we also point out that the module *ParserCount* that creates messages in regular intervals stating the total number of parsed events and acts as a heartbeat for the AMiner. It

[25] AMiner wiki, https://github.com/ait-aecid/logdata-anomaly-miner/wiki/Getting-started-(tutorial) (accessed: 2022-09-09).
[26] AMiner documentation, https://aeciddocs.ait.ac.at/logdata-anomaly-miner/current/CONFIGURATION.html#configuration-reference (accessed: 2022-09-09).
[27] AECID alert aggregation, https://github.com/ait-aecid/aecid-alert-aggregation (accessed: 2022-09-09).

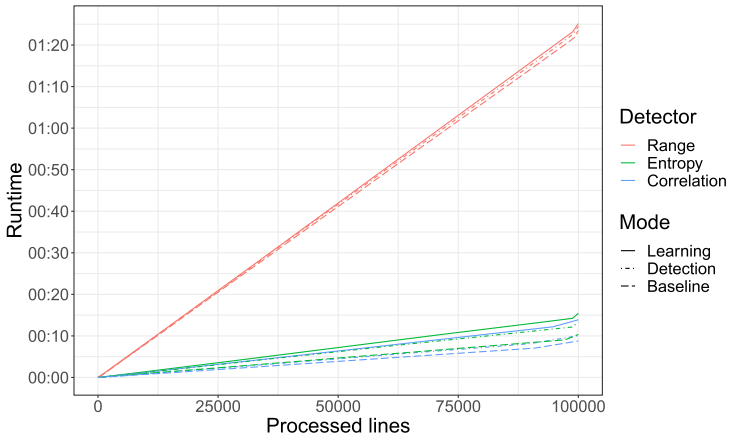

Fig. 7. Runtime required to process JSON logs with the Value Range Detector (red), Apache access logs with the Entropy Detector (green), and Exim/Messages logs with the Event Correlation Detector (blue). The lines depict execution of the AMiner for simply parsing the data (Baseline), training the detectors (Learning), and checking the trained models for anomalies (Detection).

is particularly useful to validate that the AMiner is running (since no other output is generated as long as no anomalies are detected) and verify that log events are processed.

Figure 7 shows runtime measurements of the three use-cases presented in Section 4. To enable comparability, we run the same scenarios on log files with 100,000 events each. The lines marked as *baseline* represent parsing the respective log files without any detectors in place, i.e., the AMiner only dissects the lines according to the parser tree (cf. Section 3.2.2), checks the data types of the parsed tokens, converts values such as timestamps into a common format, and carries out checks for unparsable log events (cf. Section 3.2.3). As visible in the plot, processing the JSON logs from the use-case described in Section 4.1 requires more time than processing Apache access logs (cf. Section 4.2) and Exim/Messages logs (cf. Section 4.3) due to the fact that the JSON objects involve significantly more tokens that may occur in arbitrary order and are thus more expensive to parse. In comparison to the influence of the log type, adding detectors to the pipeline only slightly increases the required runtime. In particular, the simple value range detection approach hardly influences the runtime at all, while the entropy and correlation detectors increase the runtime by around 5 s or 50%. Running the AMiner only in detection rather than training mode (i.e., the previously learned models are not updated) decreases the runtime as expected to end up somewhere in between the respective baseline and training runtime. Independent of the type of log file or applied detector, the plot shows that the runtime scales linearly with the number of processed lines, indicating that the AMiner is suitable for online analysis of log streams.

5.3 Deployment

Deploying the AMiner in productive systems as described in the previous section begs the question where to install it. In most cases, the best choice is to install the AMiner directly on the hosts where the logs to be analyzed are generated [28]. The reason for this is that it avoids the need to transfer large amounts of logs across multiple hosts in the networks and is thus the most performant solution. However, many real-world networks may comprise systems where installation of analysis tools is not foreseen, or devices such as routers that produce log data but rely on external machines for accessing their logs. In these cases, we recommend to collect log data centrally on a single machine dedicated for storing and analyzing the logs.

We also point out that the AMiner is designed to run with minimal consumption of resources and low permissions to allow installation on almost any system. Table 2 shows several measurements that depict the relationship

Table 2. AMiner Performance Measurements for Various Hardware Settings

Mode	Avail. RAM	Avail. CPUs	Lines/s
Baseline	4 GB	4 CPUs	61,635
Baseline	2 GB	2 CPUs	66,393
Baseline	1 GB	1 CPUs	53,638
Baseline	128 MB	1 CPUs	54,557
Baseline	128 MB	0.5 CPUs	35,960
Baseline	64 MB	0.3 CPUs	24,010
Baseline	32 MB	0.1 CPUs	8,972

Mode	Avail. RAM	Avail. CPUs	Lines/s
Training	4 GB	4 CPUs	5,037
Training	2 GB	2 CPUs	5,233
Training	1 GB	1 CPUs	3,618
Training	128 MB	1 CPUs	3,983
Training	128 MB	0.5 CPUs	2,164
Training	64 MB	0.3 CPUs	1,682
Training	32 MB	0.1 CPUs	675

between available computational resources and runtime performance. For this, we set up an Ubuntu virtual machine, specified limits for the available RAM and CPUs as stated in the table, and run the AMiner on syslog events in baseline mode, where only new and unparsed events are detected, as well as training mode, where multiple detectors are added in the analysis pipeline, including detectors for new values, event frequencies, missing events, and correlations. As expected, the amount of processed log events per second decreases when fewer resources are available, however, we argue that the AMiner achieves reasonable log throughput rates even with very limited resources available.

Since only anomalies are forwarded and centrally collected, the AMiner instances may operate as SIEM sensors. This principle of distributed analysis is also referred to as edge computing [14]. Distributed deployments of AMiner instances are also the basis for federated learning [21], where models are shared or aggregated across multiple hosts. As the models learned by the AMiner usually consist of lists of (complex) values as shown in Section 4, it is relatively simple to combine them to form a central model. This could reduce false positives as the resulting models are backed up by more diverse data and furthermore reduce the workload of operators that only need to maintain a single shared configuration for each detector. However, we point out that these concepts also come with downsides, e.g., privacy concerns when sensitive data is shared. We leave a thorough evaluation of usability of the AMiner for federated learning for future work.

In case that there is already a logging infrastructure in place that collects all logs centrally, it is of course also possible to deploy the AMiner in a central position to avoid having to handle multiple AMiner instances. However, we point out that the huge amounts of centrally collected logs could possibly cause that more logs are received per second than the AMiner is capable of parsing. In this case, it is necessary to adjust the configuration of detectors accordingly, e.g., restrict the number of event types that are forwarded to some of the more resource-intensive detectors.

There are three main ways to install the AMiner. First, the AMiner is available in the Debian distribution[28] and is thus easily installed as a package. However, to ensure that the most recent version of the AMiner is used, we instead recommend to install the AMiner from GitHub.[29] For this, we provide an install script and manual how to run it.[30] The third option is to run the AMiner inside a docker container.[31]

6 CONCLUSION

We present the AMiner, an open-source log-processing pipeline for intrusion detection. The AMiner ingests logs from diverse sources and uses custom and automatically generated parser trees to process the data. Moreover, the AMiner provides many detectors that leverage un- and semi-supervised learning, i.e., training of models for normal system behavior and detection of deviations as anomalies. Several interfaces exist that transfer anomalies

[28] AMiner Debian package, https://packages.debian.org/sid/logdata-anomaly-miner (accessed: 2022-09-09).
[29] AMiner GitHub repository, https://github.com/ait-aecid/logdata-anomaly-miner (accessed: 2022-09-09).
[30] AMiner wiki, https://github.com/ait-aecid/logdata-anomaly-miner/wiki/Getting-started-(tutorial) (accessed: 2022-09-09).
[31] AMiner docker setup, https://github.com/ait-aecid/logdata-anomaly-miner/wiki/Deployment-with-Docker (accessed: 2022-09-09).

generated by the AMiner to other services such as SIEMs. The AMiner is designed for forensic and online application as demonstrated in three use-cases. For future work, we foresee developing additional detectors.

REFERENCES

[1] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. 2016. A survey of network anomaly detection techniques. *J. Netw. Comput. Appl.* 60 (2016), 19–31.

[2] Robert A. Bridges, Tarrah R. Glass-Vanderlan, Michael D. Iannacone, Maria S. Vincent, and Qian Chen. 2019. A survey of intrusion detection systems leveraging host data. *ACM Comput. Surv.* 52, 6 (2019), 1–35.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Comput. Surv.* 41 (7 2009).

[4] Anton Chuvakin, Kevin Schmidt, and Chris Phillips. 2012. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management.* Newnes.

[5] CrowdStrike. 2021. Global Threat Report. Retrieved from https://www.crowdstrike.com/resources/reports/global-threat-report/.

[6] Min Du, Ruoxi Jia, and Dawn Song. 2019. Robust anomaly detection and backdoor attack detection via differential privacy. Retrieved from https://arXiv:1911.07116.

[7] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 1285–1298.

[8] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. 1996. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 120–128.

[9] Ivo Friedberg, Florian Skopik, Giuseppe Settanni, and Roman Fiedler. 2015. Combating advanced persistent threats: From network event correlation to incident detection. *Comput. Secur.* 48 (2015), 35–57.

[10] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*. IEEE, 149–158.

[11] Abida Haque, Alexandra DeLucia, and Elisabeth Baseman. 2017. Markov chain modeling for anomaly detection in high performance computing system logs. In *Proceedings of the 4th International Workshop on HPC User Support Tools*. 1–8.

[12] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2016. Experience report: System log analysis for anomaly detection. In *Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE'16)*. IEEE, 207–218.

[13] Antti Juvonen, Tuomo Sipola, and Timo Hämäläinen. 2015. Online anomaly detection using dimensionality reduction techniques for HTTP log analysis. *Comput. Netw.* 91 (2015), 46–56.

[14] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. 2019. Edge computing: A survey. *Future Gen. Comput. Syst.* 97 (2019), 219–235.

[15] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. 2019. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity* 2, 1 (2019), 1–22.

[16] Christopher Kruegel and Giovanni Vigna. 2003. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. 251–261.

[17] Max Landauer, Georg Höld, Markus Wurzenberger, Florian Skopik, and Andreas Rauber. 2021. Iterative selection of categorical variables for log data anomaly detection. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 757–777.

[18] Max Landauer, Florian Skopik, Markus Wurzenberger, and Andreas Rauber. 2020. System log clustering approaches for cyber security applications: A survey. *Comput. Secur.* 92 (2020), 101739.

[19] Max Landauer, Florian Skopik, Markus Wurzenberger, and Andreas Rauber. 2022. Dealing with security alert flooding: using machine learning for domain-independent alert aggregation. *ACM Trans. Privacy Secur.* 25, 3 (2022), 1–36.

[20] Max Landauer, Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Peter Filzmoser. 2018. Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection. *Comput. Secur.* 79 (2018), 94–116.

[21] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. Federated learning: Challenges, methods, and future directions. *IEEE Signal Process. Mag.* 37, 3 (2020), 50–60.

[22] Agathoklis Prodromou. 2019. Using Logs to Investigate—SQL Injection Attack Example. Retrieved from https://www.acunetix.com/blog/articles/using-logs-to-investigate-a-web-application-attack/.

[23] Madhu Shashanka, Min-Yi Shen, and Jisheng Wang. 2016. User and entity behavior analytics for enterprise security. In *Proceedings of the IEEE International Conference on Big Data (Big Data'16)*. IEEE, 1867–1874.

[24] Florian Skopik, Max Landauer, Markus Wurzenberger, Gernot Vormayr, Jelena Milosevic, Joachim Fabini, Wolfgang Prüggler, Oskar Kruschitz, Benjamin Widmann, Kevin Truckenthanner, et al. 2020. synERGY: Cross-correlation of operational and contextual data to timely detect and mitigate attacks to cyber-physical systems. *J. Info. Secur. Appl.* 54 (2020), 102544.

[25] Florian Skopik, Markus Wurzenberger, and Max Landauer. 2021. *Smart Log Data Analytics: Techniques for Advanced Security Analysis*. Springer.

[26] Markus Wurzenberger, Max Landauer, Florian Skopik, and Wolfgang Kastner. 2019. Aecid-pg: A tree-based log parser generator to enable log analysis. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network and Service Management (IM'19)*. IEEE, 7–12.

[27] Markus Wurzenberger, Florian Skopik, Max Landauer, Philipp Greitbauer, Roman Fiedler, and Wolfgang Kastner. 2017. Incremental clustering for semi-supervised anomaly detection applied on log data. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 1–6.

[28] Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Roman Fiedler. 2018. AECID: A self-learning anomaly detection approach based on light-weight log parser models. In *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP'18)*. 386–397.

[29] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.

[30] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 415–425.

Received 9 February 2022; revised 13 September 2022; accepted 6 October 2022