Dealing with Security Alert Flooding: Using Machine Learning for Domain-independent Alert Aggregation

MAX LANDAUER, FLORIAN SKOPIK, and MARKUS WURZENBERGER,
Austrian Institute of Technology
ANDREAS RAUBER, Vienna University of Technology

Intrusion Detection Systems (IDS) secure all kinds of IT infrastructures through automatic detection of malicious activities. Unfortunately, they are known to produce large numbers of alerts that often become overwhelming for manual analysis. Therefore, aggregation methods have been developed for filtering, grouping, and correlating alerts. However, existing techniques either rely on manually defined attack scenarios or require specific alert formats, such as IDMEF that include IP addresses. This makes the application of existing aggregation methods infeasible for alerts from host-based or anomaly-based IDSs that frequently lack such network-related data. In this paper, we therefore present a domain-independent alert aggregation technique. We introduce similarity measures and merging strategies for arbitrary semi-structured alerts and alert groups. Based on these metrics and techniques we propose an incremental procedure for the generation of abstract alert patterns that enable continuous classification of incoming alerts. Evaluations show that our approach is capable of reducing the number of alert groups for human review by around 80% and assigning attack classifiers to the groups with true positive rates of 80% and false positive rates lower than 5%.

CCS Concepts: • **Security and privacy** → **Intrusion detection systems**; • **Computing methodologies** → **Motif discovery**; *Transfer learning*; *Online learning settings;*

Additional Key Words and Phrases: Alert aggregation, intrusion detection, log data analysis

ACM Reference format:
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. Priv. Secur.* 25, 3, Article 18 (April 2022), 36 pages.
https://doi.org/10.1145/3510581

1 INTRODUCTION

Cyber attacks pose a constant threat for IT infrastructures. As a consequence, **Intrusion Detection Systems (IDS)** have been developed to monitor a wide range of activities within systems and analyze system events and interactions for suspicious and possibly malicious behavior, in which

This work was partly funded by the FFG projects INDICAETING (868306) and DECEPT (873980), and the EU H2020 project GUARD (833456).
Authors' addresses: M. Landauer, F. Skopik, and M. Wurzenberger, Austrian Institute of Technology, Giefinggasse 4, Vienna, Austria, 1210; emails: {max.landauer, florian.skopik, markus.wurzenberger}@ait.ac.at; A. Rauber, Vienna University of Technology, Favoritenstrasse 9-11, Vienna, Austria, 1040; email: rauber@ifs.tuwien.ac.at.

2471-2566/2022/04-ART18 $15.00
https://doi.org/10.1145/3510581

case they generate alerts that are subsequently reported to administrators or **Security Information and Event Management (SIEM)** systems. The main advantage of IDSs is that they are capable of processing massive amounts of data in a largely autonomous operation and are therefore usually deployed as network-based IDSs that analyze network traffic or host-based IDSs that also analyze system logs.

One of the main issues with IDSs is that they often produce large amounts of alerts that easily become overwhelming for analysts, a situation that is commonly referred to as alert flooding [14]. The number of produced alerts depends on the deployed IDS as well as the type of attack, for example, attacks that result in many alerts include denial-of-service attacks that access machines with high intensity, brute-force attacks that repeatedly attempt to log into accounts with random passwords, and automatic scripts that search for vulnerabilities [10]. These attacks produce high loads on the network and consequently cause the generation of many events in the monitored logs, of which a large part is reported by signature-based IDSs that search for patterns corresponding to such common attacks. On the other hand, anomaly-based IDSs that learn a baseline of normal system behavior and report alerts for statistical deviations are known to suffer from high false positive rates, i.e., they frequently report alerts during normal operation. Independent from their origin, alerts that occur in large frequencies are problematic, because they are difficult to categorize and may cause that analysts oversee other relevant alerts that occur less frequently [8, 14]. To alleviate this issue, alerts should be filtered or aggregated before being presented to human analysts.

Alert aggregation techniques usually rely on automatic correlation or manual linking of alert attributes [24]. However, organizations frequently deploy heterogeneous IDSs to enable broad and comprehensive protection against a wide variety of threats, causing that generated alerts have different formats and thus require normalization [32]. Most commonly, attributes of alerts are thereby reduced to timestamps, source and destination IP addresses and ports, and IDS-specific classifications, which are considered the most relevant features of alerts [2]. Unfortunately, alerts from host-based IDSs do not necessarily contain network information and alerts from anomaly-based IDSs do not involve alert types, which renders them unsuitable for existing aggregation techniques. In their survey, Navarro et al. [24] therefore recommend to develop alert aggregation techniques that operate on general events rather than well-formatted alerts to avoid loss of context information. The authors also found that most existing approaches rely on predefined knowledge for linking alerts, which impedes detection of unknown attack scenarios. In addition, modern infrastructures consist of decentralized networks and container-based virtualization that prevent IP-based correlation [10]. There is therefore a need for an automatic and domain-independent alert aggregation technique that operates on arbitrary formatted alerts and is capable of generating representative attack patterns independent from any pre-existing knowledge about attack scenarios.

IDSs generate streams of individual alerts. Aggregating these alerts means to group them so that all alerts in each group are related to the same root cause, i.e., a specific malicious action or attack. Unfortunately, finding such a mapping between alerts and attacks is difficult for a number of reasons. First, attack executions usually trigger the generation of multiple alerts [29], because IDSs are set up to monitor various parts of a system and any malicious activity frequently affects multiple monitored services at the same time. This implies that it is necessary to map a set of alerts to a specific attack execution, not just a single alert instance. Second, it is possible that the same or similar alerts are generated as part of multiple different attacks, which implies that there is no unique mapping from alerts to attacks. This is caused by the fact that IDSs are usually configured for a very broad detection and do not only consist of precise rules that are specific to particular attacks. Third, repeated executions of the same attack do not necessarily manifest themselves in the same way, but rather involve different amounts of alerts and changes of their attributes. This

effect is even more drastic when parameters of the attack are varied, their executions take place on different system environments, or alerts are obtained from differently configured IDSs. Fourth, randomly occurring false positives that make up a considerable part of all alerts [14] as well as interleaving attacks complicate a correct separation of alerts that relate to the same root cause.

In addition, alert sequences should be aggregated to higher-level alert patterns to enable the classification of other alerts relating to the same root cause. In the following, we refer to these patterns as *meta-alerts*. The aforementioned problems are insufficiently solved by existing approaches, which usually rely on models built on pre-existing domain knowledge, manually crafted links between alerts, and exploitation of well-structured alert formats.

This paper thus presents a framework for automatic and domain-independent alert aggregation. The approach consists of an algorithm that groups alerts by their occurrence times, clusters these groups by similarity, and extracts commonalities to model meta-alerts. This is achieved without merging all considered alerts into a single common format. Our implementations as well as data used for evaluation are available online.[1] We summarize our contributions as follows:

- An approach for the incremental generation of meta-alerts from heterogeneous IDS alerts.
- Similarity-metrics for semi-structured alerts and groups of such alerts.
- Aggregation mechanisms for semi-structured alerts and groups of such alerts.
- An evaluation of the proposed approach based on alerts from real-world systems.

The remainder of this paper is structured as follows. Section 2 provides an overview of existing approaches for alert aggregation. Section 3 outlines important concepts of our approach, including alerts, alert groups, and meta-alerts. Section 4 describes the overall procedure of the framework. Section 5 explains the realization of the concepts with the aid of pseudo code. We present the evaluation of our approach in Section 6 and discuss the results. Finally, Section 7 concludes the paper.

2 RELATED WORK

Alert aggregation has been an active field of research for many years. This section therefore reviews the state-of-the-art of alert aggregation in scientific literature. We first outline a number of requirements and then evaluate how each of these requirements is met by existing approaches.

2.1 Requirements

The development of alert aggregation techniques is usually motivated by specific problems at hand. Accordingly, existing approaches are based on different assumptions regarding available data, tolerated manual effort, etc. To compare existing approaches with respect to the issues outlined in Section 1, we define the following list of requirements for domain-independent alert aggregation techniques.

(1) Automatic. Manually crafting attack scenarios is time-consuming and subject to human errors [24]. Therefore, unsupervised methods should be employed that enable the generation of patterns and meta-alerts relating to unknown attacks without manual interference.

(2) Grouping. Attacks should be represented by more than a single alert (cf. Section 1). This grouping is usually based on timing (T), common attributes (A), or a combination of both (C).

(3) Format-independent. Alerts occur in diverse formats [24]. Methods should utilize all available information and not require specific attributes, such as IP addresses.

[1] https://github.com/ait-aecid/aecid-alert-aggregation.

Table 1. Fulfillment of Requirements for Existing Alert Aggregation Approaches

	Requirement				
Approach	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**
Al-Mamory et al. [1]	✓		~	✓	E
Alserhani et al. [3]		A			S
Alhaj et al. [2]	✓		✓		
Bateni et al. [4]		A			S
Cuppens et al. [6]		A	✓		E
De Alvarenga et al. [7]	✓	A			S
Haas et al. [10]	✓	A			S
Hofmann et al. [11]	✓		✓	✓	E
Husák et al. [12]	✓	A		✓	E
Husák et al. [13]	✓	A		✓	S
Julisch [14]	✓		~		E
Landauer et al. [15]	✓	C	~	✓	C
Liang et al. [17]	✓		~		
Long et al. [18]	✓		✓		
Man et al. [19]	✓		~		E
Moskal et al. [21]	✓	A			S

	Requirement				
Approach	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**
Navarro et al. [23]		T	✓		S
Ning et al. [25]		A	✓		S
Patton et al. [26]	✓		✓		
Pei et al. [27]	✓	A			S
Ramaki et al. [28]	✓	A			C
Ren et al. [29]		C	~		S
Saad et al. [30]	✓		~		E
Sadoddin et al. [31]	✓	A		✓	S
Shittu et al. [33]		A			S
Spathoulas et al. [34]	✓	C		✓	S
Sun et al. [35]	✓		✓	✓	
Vaarandi et al. [36]	✓		~		E
Valdes et al. [37]		A	✓		S
Valeur et al. [38]	✓	A		✓	E
Wang et al. [39]	✓	A			S
Zheng et al. [40]	✓		~	✓	E

(4) Incremental. IDSs generate alerts in streams. Alert aggregation methods should therefore be designed to derive attack scenarios and classify alerts in incremental operation.

(5) Meta-alerts. Aggregated alerts should be expressed by human-understandable meta-alerts that also enable automatic detection [8]. Thereby, generated patterns are usually based on single events (E), sequences (S), or a combination (C) of both.

In the next section, we present the state-of-the-art of alert aggregation techniques. Thereby, we determine which of the aforementioned requirements are met by the reviewed approaches.

2.2 Literature Analysis

This section provides an in-depth analysis of existing alert aggregation techniques with respect to the requirements stated in the previous section. Table 1 shows the fulfillment of these requirements, where ✓ or a letter corresponding to the requirement mark a sufficient fulfillment, ~ marks partial fulfillment, and no symbol means that the requirement is not addressed in the respective paper.

Alert aggregation techniques are divided into similarity-based methods that cluster alerts based on common attributes, sequential-based methods that model causal relationships between alerts as conditions, and case-based methods that employ predefined expert rules for correlation [32]. Clearly, case-based methods rely on human attack specification and therefore do not fulfill requirement (1). Sequential-based methods on the other hand are more flexible regarding the detection of attacks, for example, the LAMBDA framework proposed by Cuppens et al. [6] models attack scenarios using pre- and post-conditions based on alert properties. Ning et al. [25] present a similar mechanism with a higher focus on representing attack scenarios as graphs. Alserhani et al. [3] build upon these ideas and create reduced graphs that act as meta-alerts.

Unfortunately, sequential-based methods have limited ability to extract unknown attack scenarios [24] and thus do not fulfill requirement (1). The same applies to approaches that rely on supervised learning, such as algorithms for artificial immune systems used by Bateni et al. [4], or ant colony optimization through reinforcement learning used by Navarro et al. [23]. Our literature analysis shows that only similarity-based methods are capable of fulfilling requirement (1) [24]. Accordingly, the approach we propose in this paper is not based on manually coded conditions or

expert knowledge, but instead uses unsupervised machine learning to derive patterns from raw alert data.

Some approaches are designed for aggregation of single alerts only and therefore do not fulfill requirement (2). These approaches mainly address alert filtering and the generation of alert templates. For example, Julisch [14] propose one of the first well-known approaches for alert aggregation, which computes similarities between alert attributes based on generalization hierarchies for specific attribute types, e.g., IP addresses and ports. The approach by Al-Mamory et al. [1] builds upon these concepts and uses generalization hierarchies to compute alert cluster representatives that are then used for comparison. One of the issues with these hierarchies is that they are defined manually and therefore require mapping to specific attributes. This is solved by Long et al. [18], who propose an XML-based similarity metric for alerts in **Intrusion Detection Message Exchange Format (IDMEF)**. Similarly, Zheng et al. [40] present type-dependent similarity metrics for pre-selected attributes and use the mean, mode, and set unions of specific attribute values to generate meta-alerts.

To enable attack classification rather than alert filtering, the context of alerts needs to be included in the analysis. One possibility to achieve this is to arrange alerts in sequences by their occurrence time [23]. This is based on the idea that alerts relating to the same root cause likely occur in short time intervals [32]. The most common approach however is to link alerts with coinciding values in particular attributes, e.g., De Alvarenga et al. [7] and Husák et al. [12] group alerts by source or destination IP, Shittu et al. [33] use fuzzy combinations of IP address parts and ports, Moskal et al. [21] use sequences of alerts with corresponding destination IP and attack category, and Pei et al. [27] use a total of 29 comparisons of IP addresses, ports, process IDs, host names, etc.

Some approaches consider both timing and attribute correspondence as relevant for alert group formation. In particular, Spathoulas et al. [34] group alerts by their occurrence time and use a weighted similarity metric specifically designed to compare time differences and parts of IP addresses. The purpose of their approach is to display attacks as clusters on IP ranges for visual analysis. Ren et al. [29] also create groups for alerts that occur in close temporal proximity and have coinciding attribute values, which are represented as generalization hierarchies. In our earlier work [15], we first determine alert types by attribute similarity and then create sequences of these alert types based on their interarrival times. The problem with this strategy is that alert types are generated without considering their context of occurrence. Our approach proposed in this paper alleviates this problem by introducing similarity metrics for groups of alerts that allow to cluster only those groups that relate to the same root cause. Since both timing and attribute similarity are leveraged, our approach implements a combined strategy for requirement (2).

Reviewing existing literature with respect to requirement (3) shows that many approaches require that alerts are available in IDMEF format or involve at least attributes for source and destination IP addresses, ports, and type. For example, Husák et al. [13] propose the AIDA framework, which implements an IDMEF alert processing pipeline that removes duplicate alerts, groups the remaining alerts by source IP, and learns common alert sequence patterns. Haas et al. [10] propose to derive graphs from alerts that represent communication between hosts and allow similarity computation. Even though their approach generates so-called motif graphs that represent abstract attacks and thus do not contain IP addresses and ports, their approach requires IP information for graph generation and thus focuses on alerts from network-based IDSs. Approaches with partial fulfillment of requirement (3) employ generalization hierarchies [14] or similarity functions [30, 40] for specific attribute values that are generally valid, but require manual mapping to attributes.

A different approach is proposed by Hoffmann et al. [11], who assume statistical distributions for values of arbitrary attributes and therefore fulfill requirement (3). Patton et al. [26] convert raw text of IDS alerts into vector space models to apply hierarchical clustering. Alhaj et al. [2] select

features from all alert attributes based on their respective information gain entropy. Sun et al. [35] propose a generalized attribute weighting scheme based on rough sets. Our approach also makes use of all alert attributes and is not restricted to specific formats. However, none of the reviewed techniques include similarity metrics for groups of such alerts, which is solved by our approach.

Incremental clustering as described by requirement (4) is essential for the application in real-world scenarios that involve continuously generated alerts. Sadoddin et al. [31] propose one of the few approaches specifically designed for incremental alert processing. In particular, they mine frequent patterns from graphs representing correlated alerts. In addition, Husák et al. [13] and Landauer et al. [15] implement their concepts as pipelines for continuous alert processing. Most approaches however rely on hierarchical clustering [2, 7, 26] or other techniques that do not support incremental processing, or require computation of a correlation matrix prior to alert aggregation [29, 39]. The main issues are that such algorithms only support offline analysis and require repeated training phases that involve manual supervision. To alleviate such problems and fulfill requirement (4), our approach is designed to process incoming alerts and generate meta-alerts on the fly.

The final requirement (5) concerns the generation of meta-alerts. Most commonly, meta-alerts are represented as graphs of event sequences, for example, Wang et al. [39] create a graph with transition probabilities for correlated alert types and Haas et al. [10] generate graphs that represent abstract communication patterns. Other approaches generate patterns for single alerts, for example, Saad et al. [30] use a similarity metric to iteratively refine similar alerts by repeated aggregation, where attributes are replaced by common concepts defined in the generalization hierarchies. Vaarandi et al. [36] use frequent itemset mining to obtain patterns of static and variable alert attributes. Valdes et al. [37] fuse alerts to meta-alerts by creating supersets of values for shared attributes. Valeur et al. [38] arrange their meta-alerts in hierarchical structures and apply breadth-first search when merging new alerts. Similarly, we create abstract alert objects in Landauer et al. [15] and extract sequences of their occurrences. The approach proposed in this paper also combines alert and event sequence information for meta-alert generation, but ensures to merge only those alerts that occur in a specific sequence position to improve precision of the resulting meta-alerts.

Overall, most of the existing approaches only focus on particular aspects of alert aggregation, e.g., focus on individual alerts rather than groups, rely on domain-specific input in the correlation procedure, or impose strict requirements such as a specific format, in particular, IDMEF. Accordingly, none of the approaches fulfill all of our outlined requirements on a domain-independent alert aggregation technique sufficiently. In the following sections we will therefore propose an approach that addresses aforementioned issues with existing approaches and fulfills requirements (1)–(5).

3 ENTITIES & OPERATIONS

This section presents relevant concepts of our alert aggregation approach. We first provide an overview of the entities and their relationships. We then discuss our notion of alerts, outline how alerts are clustered into groups, and introduce a meta-alert model based on aggregated alert groups.

3.1 Overview

Our approach transforms alerts generated by IDS into higher-level meta-alerts that represent specific attack patterns. Figure 1 shows an overview of the involved concepts. The top of the figure represents alerts occurring as sequences of events on two timelines, which represent different IDSs deployed in the same network infrastructure or even separate system environments. Another possibility is that events are retrieved from historic alert logs and used for forensic attack analysis.

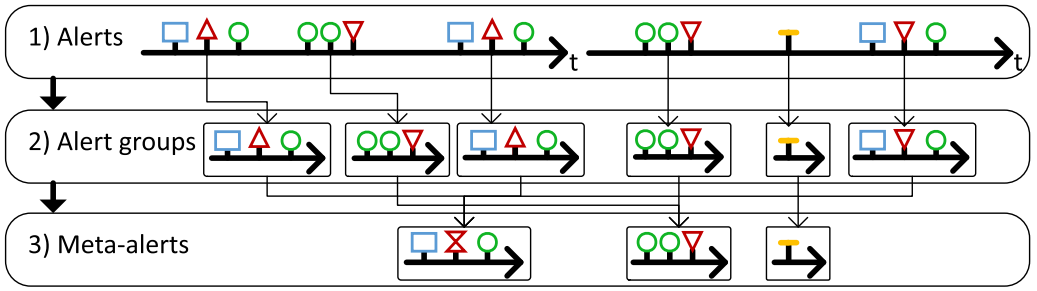


Fig. 1. Overview of the relationships between concepts. Alerts (top) occurring on timelines (t) are grouped by temporal proximity (center) and then aggregated to meta-alerts by similarity (bottom).

Alert occurrences are marked with symbols and colors that represent their types. Thereby, two alerts could be of the same type if they share the same structure, were generated by the same rule in the IDS, or have coinciding classifications. We differentiate between square (□), triangle (△, ▽, ◁, ▷), circle (○), and dash (−) symbols, which are marked blue, red, green, and yellow, respectively. For the examples presented throughout this paper, we consider alerts represented by one of {△, ▽, ◁, ▷} as variations of the same alert type, i.e., these alerts have sufficiently many commonalities such as matching attributes and are thus similar to each other. In general, each alert represents a unique event that occurs only at one specific point in time. However, alerts of the same type, e.g., alerts that are generated by the same violation of a predefined rule or alerts reported by the same IDS, may occur multiple times. We mark these alerts accordingly with the same color.

As outlined in Section 1, automatic mapping of alerts to higher-level meta-alerts is non-trivial. In the simple example shown in Figure 1, it is easy to see that the alert sequence (□, △, ○) and the similar sequence (□, ▽, ○) occur a total of three times, and that the pattern (○, ○, ▽) occurs two times over the two timelines. This is intuitively visible, because these alerts occur close together. Accordingly, it is reasonable to allocate alerts to groups that reflect this characteristic.

The center part of the figure shows groups of alerts based on their respective positions on the timelines. Note that grouping by alert type instead of temporal proximity would result in a loss of information, because alerts would be allocated to groups independent of their contexts, i.e., other alerts that are generated by the same root cause. For example, grouping all alerts of type ○ would have neglected the fact that this type actually occurs in the patterns (□, △, ○) as well as (○, ○, ▽) and may thus not be a good indicator for a particular attack execution on its own.

Computing similarities between groups means measuring the differences of orders, frequencies, and attributes of their contained alerts. Alert groups that yield a high similarity are likely related to the same root cause and should thus be aggregated into a condensed form that reflects a typical instance of that group, i.e., a meta-alert. The bottom of the figure shows the generation of meta-alerts from similar groups. Thereby, orders, frequencies, and attributes of meta-alerts are created in a way to represent all allocated alert groups as accurate as possible. The figure shows that this is accomplished by merging the second alert in the patterns (□, △, ○) and (□, ▽, ○) into alert ⧗, which combines attributes and values of △ and ▽ so that both are adequately represented. In practice, this could mean that two different values of the same attribute in both alerts are combined into a set.

The second meta-alert with alert sequence (○, ○, ▽) is formed from two identical groups and thus does not involve changes to merged alerts. If meta-alert generation was based on similarity of alerts rather than groups, all occurrences of similar alerts △ and ▽ would be replaced with ⧗, thereby decreasing the specificity of the second meta-alert. This suggests that forming groups of

logically related alerts is an essential step for meta-alert generation. Finally, the third meta-alert contains a single alert that only occurred once and is the only alert in its group. Since alerts form the basis of the presented approach, the following section will discuss their compositions in more detail.

3.2 Alerts

IDSs are designed to transmit as much useful information as possible to the person or system that receives, interprets, and acts upon the generated alerts. This includes data derived from the event that triggered the alert, e.g., IP addresses present in the monitored data, as well as information on the context of detection, e.g., detection rule identifiers. As outlined in Section 2, most approaches omit a lot of this information and only focus on specific predefined attributes. Our approach, however, utilizes all available data to generate meta-alerts without imposing any domain-specific restrictions.

To organize all data conveyed with each alert in an intuitive form, alerts are frequently represented as semi-structured objects, e.g., XML-formatted alerts as defined by the IDMEF[2] or JSON-formatted alerts generated by Wazuh[3] IDS. Even though such standards exist, different IDSs produce alerts with data fields specific to their detection techniques. For example, a signature-based detection approach usually provides information on the rule that triggered the alert, while anomaly-based IDSs only indicate suspicious event occurrences without any semantic interpretation of the observed activity. In addition, some IDSs do not provide all attributes required by standards such as IDMEF, e.g., host-based IDSs analyze system logs that do not necessarily contain network and IP information.

Figure 2 shows such an alert that was caused by a failed user login attempt generated by Wazuh. Note that it does not support IP-based correlation, since only "`srcip`" that points to localhost is available. The alert contains semi-structured elements, i.e., key-value pairs (e.g., "`timestamp`"), lists (e.g., "`groups`"), and nested objects (e.g., "`rule`"). In alignment with this observation, we model alerts as abstract objects with arbitrary numbers of attributes. Formally, given a set of alerts $\mathcal{A}$, an alert $a \in \mathcal{A}$ holds one or more attributes κ_a, where each attribute $a.k$ is defined as in Equation (1).

$$a.k = v_1, v_2, \ldots, v_n \quad \forall k \in \kappa_a, n \in \mathbb{N} \tag{1}$$

Note that Equation (1) also holds for nested attributes, i.e., $a.k.j, \forall j \in \kappa_{a.k}$, and that v_i is an arbitrary value, such as a number or character sequence. In the following we assume that the timestamp of the alert is stored in key $t \in \kappa_a, \forall a \in \mathcal{A}$, e.g., $a.t = 1$ for alert a that occurs at time step 1. These alert attributes are suitable to compare alerts and measure their similarities, e.g., alerts that share a high number of keys and additionally have many coinciding values for each common key should yield a high similarity, because they are likely related to the same suspicious event. This also means that values such as IPs are not ignored, but matched by common keys like all other attributes. We define a function $alert\_sim$ in Equation (2) that computes the similarity of alerts $a, b \in \mathcal{A}$.

$$alert\_sim : a, b \in \mathcal{A} \rightarrow [0, 1] \tag{2}$$

Thereby, the similarity between any non-empty alert and itself is 1 and the similarity to an empty object is 0. Furthermore, the function is symmetric, which is intuitively reasonable when comparing alerts on the same level of abstraction. On the other hand, the function implicitly computes how well one alert is represented by another more abstract alert as we will outline in Section 5.1.

[2]Intrusion Detection Message Exchange Format (IDMEF) is defined in IETF RFC 4765 (https://tools.ietf.org/html/rfc4765).
[3]https://wazuh.com/.

```
{
        "timestamp": "2020-03-04T19:26:05.000000+0000",
        "rule": {
                "level": 5,
                "description": "PAM: User login failed.",
                "id": "5503",
                "firedtimes": 28,
                "groups": [
                        "pam",
                        "syslog",
                        "authentication_failed"
                ]
        },
        "full_log": "Mar  4 19:26:05 mail auth: pam_unix(dovecot:auth): authentication failure; logname= uid=0
                euid=0 tty=dovecot ruser=daryl rhost=127.0.0.1  user=daryl",
        "data": {
                "srcip": "127.0.0.1",
                "srcuser": "daryl",
                "dstuser": "daryl",
                "uid": "0",
                "euid": "0",
                "tty": "dovecot"
        },
        "location": "/var/log/forensic/auth.log"
}
```


Fig. 2. Simplified sample alert documenting a failed user login.

We summarize the properties of the function in Equations (3)–(5).

$$alert\_sim(a, a) = 1 \tag{3}$$

$$alert\_sim(a, \emptyset) = 0, \quad a \neq \emptyset \tag{4}$$

$$alert\_sim(a, b) = alert\_sim(b, a) \tag{5}$$

As mentioned, we do not make any restrictions on the attributes of alerts and only consider the timestamp $a.t$ of alert a as mandatory, which is not a limitation since the time of detection is always known by the IDS or can be extracted from the monitored data. In the next section, this timestamp will be used to allocate alerts that occur in close temporal proximity to groups.

3.3 Alert Groups

Alerts generated by an arbitrary number of deployed IDSs result in a sequence of heterogeneous events. Since attacks typically manifest themselves in multiple mutually dependent alerts rather than singular events, it is beneficial to find groups of alerts that were generated by the same root cause as shown in Section 3.1. In the following, we describe our strategies for formation and representation of alert groups that enable group similarity computation.

3.3.1 Formation. Depending on the type of IDS, alerts may already contain some kind of classification provided by their detection rules. For example, the message "`PAM: User login failed.`" contained in the alert shown in Figure 2 could be used to classify and group every event caused by invalid logins. While existing approaches commonly perform clustering on such pre-classifications of IDSs, single alerts are usually not sufficient to differentiate between specific types of attacks or accurately filter out false positives (cf. Section 1). To alleviate this problem, we identify multiple alerts that are generated in close temporal proximity and whose combined occurrence is a better indicator for a specific attack execution. For example, a large number of alerts classified as failed user login attempts that occur in a short period of time and in combination with a suspicious user agent could be an indicator for a brute-force password guessing attack executed through a particular tool. Such a reasoning would not be possible when all alerts are analyzed individually, because

single failed logins may be false positives and the specific user agent could also be part of other attack scenarios.

The problem of insufficient classification is even more drastic when alerts are received from anomaly-based IDS, because they mainly disclose unknown attacks. Accordingly, an approach that relies on clustering by alert classification attributes would require human analysts who interpret the root causes and assign a classifier to each alert. Temporal grouping on the other hand is always possible for sequentially incoming alerts and does not rely on the presence of alert attributes.

Our strategy for alert group formation is based on the interval times between alerts. In particular, two alerts $a, b \in \mathcal{A}$ that occur at times $a.t, b.t$ have an interval time $|a.t - b.t|$ and are allocated to the same group if $|a.t - b.t| \leq \delta$, where $\delta \in \mathbb{R}^+$. This is achieved through single-linkage clustering [9]. In particular, all alerts are initially contained in their own sets, i.e., $s_{\delta,i} = \{a_i\}, \forall a_i \in \mathcal{A}$. Then, clusters are iteratively formed by repeatedly merging the two sets with the shortest interval time $d = \min\left(\left|a_i.t - a_j.t\right|\right), \forall a_i \in s_{\delta,i}, \forall a_j \in s_{\delta,j}$. This agglomerative clustering procedure is stopped when $d > \delta$, which results in a number of sets $s_{\delta,i}$. Each set is transformed into a group $g_{\delta,i}$ that holds all alerts of set $s_{\delta,i}$ as a sequence sorted by their occurrence time stamps as in Equation (6).

$$g_{\delta,i} = \{(a_1, a_2, \ldots, a_n), \forall a_i \in s_{\delta,i} : a_1.t \leq a_2.t \leq \cdots \leq a_n.t\} \tag{6}$$

Equation (7) defines the set of all groups for a specific δ as their union.

$$\mathcal{G}_\delta = \bigcup_{i \in \mathbb{N}} g_{\delta,i} \tag{7}$$

This group formation strategy is exemplarily visualized in Figure 3. The figure shows alert occurrences of types $\{\square, \triangle, \circ, -\}$ in specific patterns duplicated over four timelines with different δ. The sequence $(\square, \triangle, \circ)$ at the beginning of the timelines occurs with short alert interval times and that a similar sequence $(\square, \triangledown, \circ)$ occurs at the end, but involves $\triangledown$ instead of its variant $\triangle$ and has an increased interval time between $\triangledown$ and $\circ$. Nevertheless, due to the similar compositions of these two alert sequences, it is reasonable to assume that they are two manifestations of the same root cause.

In this example, each tick in the figure marks a time span of 1 unit. In timeline (d), all alerts end up in separate groups, because no two alerts yield a sufficiently small interval time lower than $\delta = 0.5$, i.e., $\mathcal{G}_{0.5} = \{(\square), (\triangle), (\circ), (-), (\square), (\triangledown), (\circ)\}$. In timeline (c) where alerts are grouped using $\delta = 1.5$, two groups that contain more than a single alert are formed, because the grouped alerts occur within sufficiently close temporal proximity, i.e., $\mathcal{G}_{1.5} = \{(\square, \triangle, \circ), (-), (\square, \triangledown), (\circ)\}$. Considering the results for $\mathcal{G}_{2.5} = \{(\square, \triangle, \circ), (-), (\square, \triangledown, \circ)\}$ in timeline (b) shows that the aforementioned repeating pattern $(\square, \triangle, \circ)$ and its variant end up in two distinct groups. This is the optimal case, since subsequent steps for group analysis could determine that both groups are similar and thus merge them into a meta-alert as shown in Section 3.1. A larger value for delta, e.g., $\delta = 3.5$ that yields $\mathcal{G}_{3.5} = \{(\square, \triangle, \circ), (-, \square, \triangledown, \circ)\}$ in timeline (a), adds alert of type $-$ to form group $(-, \square, \triangledown, \circ)$, which is not desirable since this decreases its similarity to group $(\square, \triangle, \circ)$. This example thus shows the importance for an appropriate selection of the interval threshold for subsequent analyses.

Note that this strategy for temporal grouping has several advantages over sliding time windows. First, instead of time window size and step width, only a single parameter that specifies the maximum delta time between alerts is required, which reduces complexity of parameter selection. Second, it ensures that alerts with close temporal proximity remain in the same group given any delta larger than their interval times, while intervals of sliding time windows possibly break up groups by chance. Third, related sequences with variable delays result in complete groups as long as there is no gap between any two alerts that exceeds δ, e.g., two groups with similar alerts but

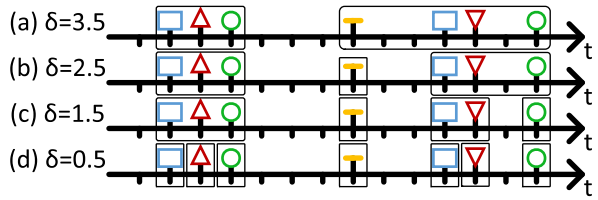


Fig. 3. Alert occurrences duplicated over four parallel timelines show the formation of alert groups based on alert interval times. Larger intervals (top) allow more elapsed time between alerts and thus lead to fewer and larger groups compared to smaller intervals (bottom).

varying delays are found for $\delta = 2.5$ in Figure 3. However, time window sizes must exceed the duration of the longest sequence to yield complete groups, which is more difficult to specify in general.

Despite these benefits, pure time-based grouping suffers from some drawbacks compared to knowledge-based clustering methods, e.g., grouping by classification messages. As seen in the example from Figure 3, the quality of the resulting grouping is highly dependent on a selection of the parameter δ that fits the typical time interval of the events to be grouped. Another issue is that randomly occurring alerts, e.g., false positives, are incorrectly allocated to groups if they occur in close proximity to one of the grouped alerts. Even worse, these alerts could connect two or more groups into a single large group if they happen to occur in between and in sufficiently high amount or close proximity to both groups. As we will outline in the following sections, our approach mitigates these problems by finding groups using several values for δ in parallel.

3.3.2 Similarity Computation. Other than clustering based on predefined alert types, time-based grouping only acts as a preparatory step for subsequent analyses. In particular, a similarity measure for alert groups is required that allows to determine which groups of alerts are likely generated from the same root cause. Only then it is possible to cluster groups by their similarities and in turn generate meta-alerts by merging alert groups that end up in the same clusters. We therefore define function *group_sim* in Equation (8) that computes the similarity of any two groups $g, h \in \mathcal{G}_\delta$.

$$group\_sim : g, h \in \mathcal{G}_\delta \rightarrow [0, 1] \tag{8}$$

Analogous to alert similarity computation (cf. Section 3.2), the similarity between any non-empty group $g \in \mathcal{G}_\delta$ and itself is 1 and the similarity to an empty object is 0. However, we do not impose symmetry on the function, since it can be of interest to measure whether one group is contained in another possibly more abstract group, such as a meta-alert. Details on such a similarity function are discussed in Section 5.2. In the following section, we first explain the representation of meta-alerts and then introduce matching strategies for similarity computations between groups.

3.4 Meta-Alerts

We generate meta-alerts by merging groups, which relies on merging alerts within these groups. In the following, we first introduce features that support the representation of merged alerts and then outline group merging strategies for similarity computations and meta-alert generation.

3.4.1 Alert Merges. As outlined in Section 3.2, alerts are semi-structured objects, i.e., data structures that contain key-value pairs, and are suitable for similarity computation. However, aggregating similar alerts into a merged object that is representative for all allocated alerts is non-trivial, because single alert objects may have different keys or values that need to be taken into account.

For example, the failed login alert in Figure 2 contains the attribute "srcuser" with value "daryl" in the "data" object. Since a large number of users may trigger such alerts, this event type occurs with many different values for attribute "srcuser" over time. An aggregated alert optimally abstracts over such attributes to represent a general failed login alert that does not contain any information specific to a particular event. The computed similarity between such an aggregated alert and any specific alert instance is independent of attributes that are known to vary, i.e., only the presence of the attribute "srcuser" contributes to similarity computations, but not its value. Note that this assumes that keys across alerts have the same semantic meaning or that keys with different names are correctly mapped if alert formats are inconsistent, e.g., keys "src_user" and "srcuser".

We incorporate merging of alerts by introducing two new types of values. First, a *wildcard* value type that indicates that the specific value of the corresponding key is not expressive for that type of alert, i.e., any value of that field will yield a perfect match just like two coinciding values. Typical candidates for values replaced by wildcards are user names, domain names, IP addresses, counts, and timestamps. Second, a *mergelist* value type that comprises a finite set of values observed in several alerts that are all regarded as valid values, i.e., a single matching value from the mergelist is sufficient to yield a perfect match for this attribute present in two compared alerts. The mergelist type is useful for discrete values that occur in variations, e.g., commands or parameters derived from events. Deciding whether an attribute should be represented as a wildcard or mergelist is therefore based on the total number of unique values observed for that attribute (see Sect. 5.3).

We define that each attribute key $k \in \kappa_a$ of an aggregated alert a that is the result of a merge of alerts $A \subseteq \mathcal{A}$ is represented as either a wildcard or mergelist as in Equation (9).

$$a.k = \begin{cases} wildcard\,() \\ mergelist\,(\bigcup_{b \in A} b.k) \end{cases} \tag{9}$$

Note that Equation (9) also applies for nested keys, i.e., values within nested objects stored in the alerts. Since our approach is independent of any domain-specific reasoning, a manual selection of attributes for the replacement with wildcards and mergelists is infeasible. The function *alert_merge* thus automatically counts the number of unique values for each attribute from alerts $A \subseteq \mathcal{A}$ passed as a parameter, selects and replaces them with the appropriate representations, and returns a new alert object a that represents a merged alert that is added to all alerts $\mathcal{A}$ as shown in Equation (10)–(11).

$$a = alert\_merge(A), \quad A \subseteq \mathcal{A} \tag{10}$$

$$\mathcal{A} \Leftarrow a \tag{11}$$

Note that we use the operation $\Leftarrow$ to indicate set extensions, i.e., $\mathcal{A} \Leftarrow a \iff \mathcal{A}' = \mathcal{A} \cup \{a\}$. We drop the prime of sets like $\mathcal{A}'$ in the following for simplicity and assume that after extension only the new sets will be used. The extension of $\mathcal{A}$ implies that merged alerts are also suitable for similarity computation and merging with other alerts or merged alerts. We will elaborate on the details of the alert merging procedure in Section 5.3. The next section will outline the role of alert merging when groups are merged for meta-alert generation.

3.4.2 Group Merges. Similar to merging of alerts that was discussed in the previous section, a merged group should represent a condensed abstraction of all groups used for its generation. Since each group should ideally comprise a similar sequence of alerts, it may be tempting to merge groups by forming a sequence of merged alerts, where the first alert is merged from the first alerts in all groups, the second alert is merged from the second alerts in all groups, and so on. Unfortunately, this is infeasible in practice, because alert sequences are not necessarily ordered,

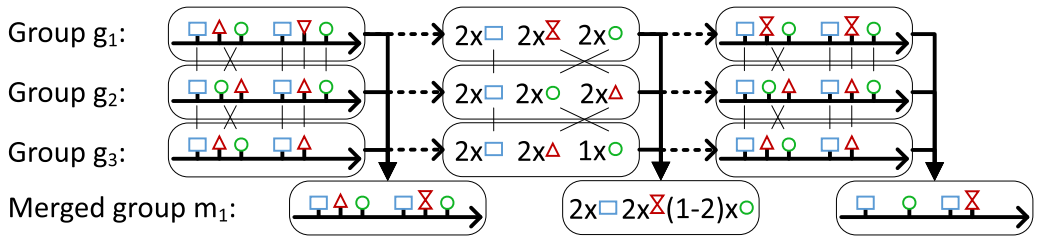


Fig. 4. Merging strategies for alert groups. Left: Finding exact matches between alert pairs. Center: Matching representatives using a bag-of-alerts model. Right: Matching using alert sequence alignment.

involve optional alerts, or are affected by false positives causing that alert positions in sequences are shifted. To alleviate this issue, it is necessary to find matches between the alerts of all groups to be merged. In the following, we describe three matching strategies used in our approach that are suitable for group similarity computation as well as meta-alert generation.

Exact matching. This strategy finds for each alert in one group the most similar alert in another group and uses these pairs to determine which alerts to merge. The idea of finding these matches is depicted in the left side of Figure 4, where lines across groups g_1, g_2, g_3 indicate which alerts were identified as the most similar. As expected, alerts of the same type are matched, because they share several common attributes and values that are specific to their respective types. The figure also shows that correct alerts are matched even though the second and third alert in g_2 are in a different order than in g_1 and g_3. In addition, note that the alert of type $\triangledown$ in g_1 is correctly matched to the related alert type $\triangle$ in g_2 and that the merged group thus contains the merged alert type $\bowtie$ at that position. In addition, there is a missing alert of type $\circ$ in g_3 that leads to an incomplete match. Nevertheless, the alert of type $\circ$ ends up in the merged group, because it occurs in the majority of all merged groups and is therefore considered to be representative for this root cause manifestation.

When only two groups are considered, this matching method is also suitable for measuring their similarity. In particular, this is achieved by computing the average similarity of all matched alerts, where non-matching alerts count as total mismatches. The similarity score is further enhanced by incorporating an edit distance [22] that measures the amount of inserts, removes, and substitutions of alerts, i.e., misalignments such as the occurrence of $(\circ, \triangle)$ instead of $(\triangle, \circ)$ in g_2.

While the exact matching strategy yields accurate group similarities, it is rather inefficient for large groups. The reason for this is that computing the pairwise similarities of all alerts requires quadratic runtime with respect to group sizes. We therefore only use this strategy when the number of required comparisons for groups g, h does not exceed a limit $l_{bag} \in \mathbb{N}$, i.e., $|g| \cdot |h| \leq l_{bag}$, where $|g|$ denotes the size of group g. In the following, we outline an alternative strategy for larger groups.

Bag-of-alerts matching. For this strategy, we transform the alert sequences of all groups into a bag-of-alerts model following the well-known bag-of-words model [20]. This is accomplished by incrementally clustering the alerts within each group using a certain similarity threshold $\theta_{alert} \in [0, 1]$. Thereby, each alert a that is sufficiently similar to one of the initially empty sets of cluster representatives R, i.e., $alert\_sim(r, a) \geq \theta_{alert}, \forall r \in R$, is added to the list C_r that stores all alerts of that cluster, i.e., $C_r \Leftarrow a$, or forms a new cluster with itself as a representative otherwise, i.e., $R \Leftarrow a$. Once all alerts of a group are processed, the bag-of-alerts model for that group is generated by merging all alerts in each cluster, i.e., $alert\_merge(C_r), \forall r \in R$.

The matching procedure then finds the pairs of these merged alerts that yield the highest similarities across groups and aggregates them by identifying lower and upper limits of their corresponding cluster sizes $|C_r|$ in each group. The advantage in comparison to the exact matching strategy is that the number of necessary similarity computations is reduced to the product of the number of clusters per group, which is controllable through θ_{alert}. Note that the speedup stems from the fact that the computation of the bag-of-alerts model only has to be carried out once for each group, but then enables fast matching with all other groups.

The center part of Figure 4 shows bag-of-alert models for sample groups, where alerts of types $\triangle$ and $\triangledown$ in g_1 are merged to ⧗, which is then matched to $\triangle$ in g_2 and g_3 before they are once again merged for the generation of the meta-alert. Since alert type $\circ$ occurs twice in g_1 and g_2, but only once in g_3, the meta-alert uses a range with minimum limit $l_{min} = 1$ and maximum limit $l_{max} = 2$ to describe the occurrence frequency of this alert type.

This strategy also supports measuring the similarity of two groups g, h by averaging the relative differences of occurrence counts, which yields the highest possible similarity of 1 if the respective counts coincide or their intervals overlap, and $min(l_{max,g}, l_{max,h})/max(l_{min,g}, l_{min,h})$ otherwise. Alerts without a match are considered as total mismatches and contribute the lowest possible similarity score of 0 to the average. We favored this similarity metric over existing measures such as cosine similarity [20], because it allows a more intuitive representation of lower and upper occurrence limits which supports human interpretation of meta-alerts.

The downside of the bag-of-alerts strategy is that information on the order of the alerts is lost. However, it is possible to resolve this issue by combining the original alert sequence with the bag-of-alerts model. In the following, we outline this addition to the bag-of-alerts matching.

Alignment-based matching. To incorporate alignment information for large clusters that are not suited for the exact matching strategy, it is necessary to store the original sequence position of all clustered alerts during generation of the bag-of-alerts model of each group. This information enables to generate a sequence of cluster representatives. For example, the right side of Figure 4 shows that group g_1 has sequence (□, ⧗, ○, □, ⧗, ○), because the occurrences of $\triangle$ and $\triangledown$ have been replaced by their cluster representative ⧗ that was generated in the bag-of-alerts model. Note that this strategy is much faster for large groups than the exact matching strategy, because it enables to reuse the matching information of representative alerts from the bag-of-alerts model instead of finding matches between all alerts. Since the corresponding sequence elements across groups are known, it is simple to use sequence alignment algorithms for merging and similarity computation.

We decided to merge the sequences using **longest common sequence (LCS)** [22], because it enables to retrieve the common alert pattern present in all groups and thereby omit randomly occurring false positive alerts [15]. The example in Figure 4 shows that this results in a sequence of representatives (□, ○, □, ⧗) that occurs in the same order in all groups. Using the LCS also enables to compute the sequence similarity of two groups g, h by $|LCS(g,h)|\,/min(|g|\,,|h|)$, which we use to improve the bag-of-alerts similarity by incorporating it as a weighted term after averaging.

Equation (12) defines a function that takes a set of groups $G \subseteq \mathcal{G}_\delta$ and automatically performs all aforementioned merging strategies to generate a new group g.

$$g = group\_merge(G), \quad G \subseteq \mathcal{G}_\delta \tag{12}$$

$$\mathcal{G}_\delta \Leftarrow g \tag{13}$$

Analogous to merges of single alerts, Equation (13) indicates that merges of alert groups have the same properties as normal groups and therefore support similarity computation and merging. In the previous sections, we defined several functions required for meta-alert generation. The following section will embed all aforementioned concepts in an overall procedure.

4 FRAMEWORK

This section outlines a procedure for meta-alert generation based on the aforementioned concepts and functions. We first describe the overall approach and then present its steps in two scenarios.

4.1 Overview

Our procedure reads in a sequence of alerts from one or multiple IDSs. The first step is to form groups from these incoming alerts as outlined in Section 3.3.1. Unfortunately, manually specifying δ as the maximum allowed interval time between alerts is non-trivial, because it requires a high amount of knowledge about alert interactions and expected attack pattern structures. Even worse, different alert patterns may require specific settings for δ that are incompatible with each other. To resolve this issue, we carry out group formation in parallel for several values $\delta \in \Delta$ similar to Figure 3, where Δ is the set of all values for δ. This increases the chance that valid and usable meta-alerts are found for various types of attacks. In addition, it forms a hierarchical structure of alert patterns, where small δ values generate groups that contain mainly technically linked alerts, e.g., a failed login alert that occurs simultaneously with a frequency alert for such events, and groups generated by large δ values that contain sequentially executed attack steps [15]. For simplicity, we only use δ in the following and implicitly assume that all computations are carried out for all $\delta \in \Delta$ analogously.

We define a set of meta-alerts $\mathcal{M}_\delta$ that holds merged groups. Note that the index δ indicates that meta-alerts are generated for all δ values separately, i.e., groups formed by different δ values are not merged together. The reason for this is that merging groups that were partially formed from the same alert occurrences may lead to overly generalized meta-alerts and thus loss of information. For example, consider the groups from Figure 3, where group $(\square, \triangledown, \circ) \in \mathcal{G}_{2.5}$ and group $(-, \square, \triangledown, \circ) \in \mathcal{G}_{3.5}$ contain three identical alerts and may thus be considered similar enough for merging. This is not desirable, since the resulting merge will involve the alert type $-$, which is not part of the actual attack pattern $(\square, \text{I}, \circ)$ that is the result of merging $(\square, \triangle, \circ) \in \mathcal{G}_{2.5}$ and $(\square, \triangledown, \circ) \in \mathcal{G}_{2.5}$. Such cascading merges occurring over different δ values could mostly be prevented by prohibiting merges of groups that contain identical alert instances. However, to avoid this issue altogether and to enable a thorough evaluation for each δ value, we process all meta-alerts sets $\mathcal{M}_\delta$ isolated.

A new group $g \in \mathcal{G}_\delta$ is incrementally added to the set of meta-alerts $\mathcal{M}_\delta$ by finding the meta-alert $m \in \mathcal{M}_\delta$ with the highest similarity, i.e., $sim = max_{m \in \mathcal{M}_\delta}(group\_sim(g, m))$. If the similarity is higher than a predefined threshold $\theta_{group} \in [0, 1]$, i.e., $sim \geq \theta_{group}$, the group is added to the most similar meta-alert m, otherwise a new meta-alert is generated for this group.

It is not recommended to merge group g and meta-alert m directly, i.e., $m = group\_merge(\{g, m\})$, because this causes that meta-alerts over-generalize over time. The reason for this is that a single incorrect allocation of a group to a meta-alert extends mergelists of attributes or introduces wildcards, which will increase the similarity of the meta-alert to all other groups and thus make it more susceptible to incorrect allocations in a self-enforcing loop. As a solution, we store allocated groups for each meta-alert in a so-called *knowledge base* $\mathcal{K}_\delta$, where $K_m \subseteq \mathcal{K}_\delta$ is the set of all groups allocated to meta-alert m. For group g and meta-alert m where $sim \geq \theta_{group}$, we therefore update the knowledge base $K_m \Leftarrow g$ and regenerate meta-alert $m = group\_merge(K_m)$ from all groups. The advantage of this strategy is that it allows to generate meta-alerts from more than two groups at the same time, which is more robust against single group misallocations since attribute merging can be based on majority decision or predefined minimum occurrences. In addition, it allows to adapt group allocations in the knowledge base, e.g., reallocate individual groups that turn out to be incorrectly classified without the need to remove the meta-alert, split one meta-alert m into multiple meta-alerts by extracting subsets of K_m, or merge meta-alerts by unifying their groups.

Storing all identified groups in the knowledge base is usually infeasible in practice due to limited available memory as well as increasing runtime for merging larger amounts of groups. We therefore use a queue to enable the following strategies for storing groups in each $K_m \subseteq \mathcal{K}_\delta$:

- **Unlimited storage.** This strategy implies that queue sizes grow indefinitely. Such a strategy is useful for forensic analyses, where the total number of groups is limited and known to be sufficiently small, and it is thus possible to store all groups.
- **Linear storage.** With this strategy, the size of the queues is limited. Once the queue is full, adding a new group will cause the oldest group in the queue to be removed.
- **Logarithmic storage.** First, the queue is filled to its maximum size. Then, any newly added group will replace the last group with probability 1/2, move the last group one position lower with probability 1/4, move each of the last two groups one position lower with probability 1/8, etc. This ensures that groups at the beginning of the queue remain in the queue for a longer time span and that the groups stored in the queue collectively represent a more diverse set. This strategy is therefore especially useful when related alerts are expected to occur over long time intervals, e.g., when they are collected from different environments.

In the next section, we show the individual steps of the procedure by two application cases. For simplicity, we assume that the unlimited storage strategy is used and thus treat each K_m as a set.

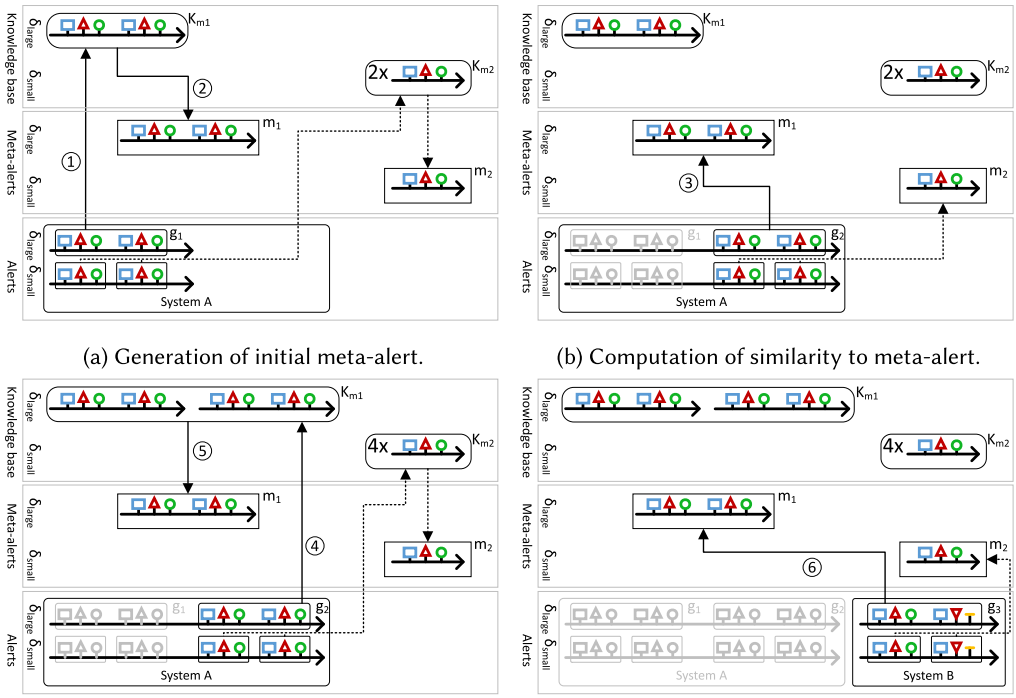
4.2 Scenarios

We select two scenarios to explain the approach for meta-alert generation in the following. The first scenario is displayed in Figures 5(a)–5(d) and deals with reusing meta-alerts for classification of alerts occurring on other systems. Thereby, each of the figures depicts the state of the incremental alert aggregation framework at a specific point in time. Moreover, we constructed the figures to show the alert occurrences $\mathcal{A}$ and the formed groups $\mathcal{G}_\delta$ in the bottom, the generated meta-alerts $\mathcal{M}_\delta$ in the center, and the knowledge base $\mathcal{K}_\delta$ in the top. Note that in each of these blocks we display two sections, one for a δ_{large} value (top) and one for a δ_{small} value (bottom), where $\delta_{large} > \delta_{small}$. For simplicity, we focus only on groups generated by the δ_{large} value in the first scenario.

Figure 5(a) depicts the state of the framework after one group $g_1 = (\square, \triangle, \circ, \square, \triangle, \circ)$ was formed for δ_{large}, i.e., the time passed after the last alert occurrence exceeds δ_{large}. Since no meta-alerts exist at this point, a new meta-alert m_1 is created by instantiating group $K_{m_1} \Leftarrow g_1$ so that $K_{m_1} = \{g_1\}$ in the knowledge-base as indicated by step (1), and generating meta-alert $m_1 = group\_merge(K_{m_1})$ as indicated by step (2). Note that meta-alert m_1 involves the same alert sequence with identical attributes as group g_1, but all values are represented as mergelists as outlined in Section 3.4.1.

Figure 5(b) depicts the occurrence of another group $g_2 = (\square, \triangle, \circ, \square, \triangle, \circ)$ on system A. Step (3) shows that the similarity between g_2 and each $m \in \mathcal{M}_\delta$ is computed, in particular, only the similarity $sim_{g_2} = group\_sim(g_2, m_1)$ is computed since only $m_1 \in \mathcal{M}_\delta$ exists. Due to the fact that both groups g_1, g_2 involve the same alert sequence, we assume that their similarity exceeds a predefined threshold θ_{group}, i.e., $sim_{g_2} = group\_sim(g_2, m_1) \geq \theta_{group}$, indicating that g_1 relates to the same root cause as m_1 and should therefore be aggregated. Figure 5(c) shows that this is achieved by adding group g_2 to the knowledge base storing the groups allocated to m_1, i.e., $K_{m_1} \Leftarrow g_2$ so that $K_{m_1} = \{g_1, g_2\}$, as indicated by step (4). Adding a group to K_{m_1} triggers a regeneration of meta-alert m_1 as indicated by step (5), i.e., $m_1 = group\_merge(K_{m_1})$. Assuming that all alerts in groups g_1, g_2 have the same attributes and values, the resulting meta-alert m_1 remains unchanged.

Figure 5(d) displays group $g_3 = (\square, \triangle, \circ, \square, \triangledown, -)$ occurring in system B at some point after m_1 is generated from alerts on system A. Step (6) depicts the similarity computation $sim_{g_3} = group\_sim(g_3, m_1)$. Note that only the first four out of six alerts in m_1 and g_3 are identical, while the fifth alert $\triangledown$ of g_3 is a variation of alert $\triangle$ in m_1 and the sixth alert is of a different type. If



(a) Generation of initial meta-alert.

(b) Computation of similarity to meta-alert.

(c) Knowledge base update and merging.

(d) Detection of meta-alert on system B.

Fig. 5. Scenario for cross-system alert recognition. Steps (1)–(5) show the meta-alert generation procedure using alerts from system A. Steps (6) indicates the detection of a similar alert group on system B.

the similarity is sufficiently high, i.e., $sim_{g_3} \geq \theta_{group}$, the occurrence of the group is interpreted as a detection of the attack represented by m_1. Otherwise, the group is assumed to depict a new unknown attack, causing that a new meta-alert is generated from group g_3 similar to steps (1)–(2).

The procedure for groups identified for δ_{small} is indicated by dashed arrows and works analogously. Figures 5(a)–5(c) show that four groups occurring on system A are iteratively added to the knowledge base K_{m_2} and are merged to a single meta-alert $m_2 = (\square, \triangle, \circ)$. Figure 5(d) shows that two groups are identified for δ_{small}, of which one comprises the same alert sequence as meta-alert m_2 and is thus similar enough to yield a successful detection of the same attack pattern, while the other group is rather dissimilar and could therefore lead to the generation of a new meta-alert.

The second scenario is visualized in Figures 6(a)–6(d) and focuses on merging alert groups across systems. For simplicity, the following description focuses on groups generated by the δ_{small} value. Similar to the first scenario, steps (1) and (2) in Figure 6(a) indicate the generation of meta-alert m_1 from the first group $g_1 = (\circ, \triangle, \circ, \triangle)$ on system A, so that $K_{m_1} = \{g_1\}$. As shown in Figure 6(b), the difference to the first scenario is that group $g_2 = (\circ, \triangledown, \circ, \triangleright)$ in system A has variations of alert type $\triangle$ occurring in the second and fourth alert. For the sake of example, we consider alert types $\{\triangle, \triangledown, \triangleright, \triangleleft\}$ to be similar alerts with the same set of attributes but different values in one specific attribute, e.g., a different user name (cf. Section 3.4.1). Despite these variations, group g_2 involves similar alert types and therefore yields a sufficiently high similarity, i.e., $sim_{g_2} = group\_sim(g_2, m_1) \geq \theta_{group}$. As a consequence, group g_2 is added to the knowledge base of meta-alert m_1 in step (3), i.e., $K_{m_1} \Leftarrow g_2$ so that $K_{m_1} = \{g_1, g_2\}$, which is in turn used to update

meta-alert $m_1 = group\_merge(K_{m_1})$ as indicated by step (4). Since the resulting meta-alert m_1 is a merge of groups g_1, g_2, its second alert is a merge of alert types $\{\triangle, \triangledown\}$ and its fourth alert is a merge of alert types $\{\triangle, \triangleright\}$.

Different to the first scenario, alert groups from system B are used to generate a cross-system meta-alert. Figure 6(c) shows group $g_3 = (\circ, \triangle, \circ, \triangle)$, which involves alerts $\triangle$ on the second and fourth positions. Since $\triangle$ is part of the aggregated alerts of meta-alert m_1, similarity $sim_{g_3} = group\_sim(g_3, m_1) \geq \theta_{group}$ is high and the group is thus added to K_{m_1}. While also the second alert $\triangledown$ of group $g_4 = (\circ, \triangledown, \circ, \triangleleft)$ yields a perfect match with the second alert ⧗ of meta-alert m_1, the fourth alert $\triangleleft$ of group g_4 is not part of m_1 and thus slightly decreases similarity $sim_{g_4} = group\_sim(g_4, m_1)$, which is nonetheless assumed to exceed θ_{group} since all other alerts match. Therefore, $K_{m_1} \Leftarrow g_4$ so that $K_{m_1} = \{g_1, g_2, g_3, g_4\}$ as indicated by step (5). Note that in all four groups, the second alert is one of $\{\triangle, \triangledown\}$, and the fourth alert is one of $\{\triangle, \triangleright, \triangleleft\}$. When generating m_1 after updating K_{m_1} in step (6), the affected attribute of the fourth alert is therefore replaced with a wildcard so that $m_1 = (\circ, ⧗, \circ, *)$. Since the wildcard matches all values, both groups g_5, g_6 displayed in Figure 6(d) yield perfect matches with m_1, even though alert $\triangledown$ in g_6 does not occur in any group of K_{m_1}.

Inspecting meta-alert m_2 in Figure 6(d) that was generated by groups of system A and system B using δ_{large} shows that the sequence of merged alerts differs from m_1, e.g., alert types $\triangle$ and $\triangledown$ occur instead of alert type ⧗. Since this scenario depicts just an exemplary demonstration that is not based on real alerts, it is not possible to determine which of the meta-alerts m_1, m_2 is better suited for detection. However, both scenarios suggest that it is reasonable to consider multiple values for δ to generate several different meta-alerts that cover a large variety of attack manifestations.

5 IMPLEMENTATION OF THE FRAMEWORK

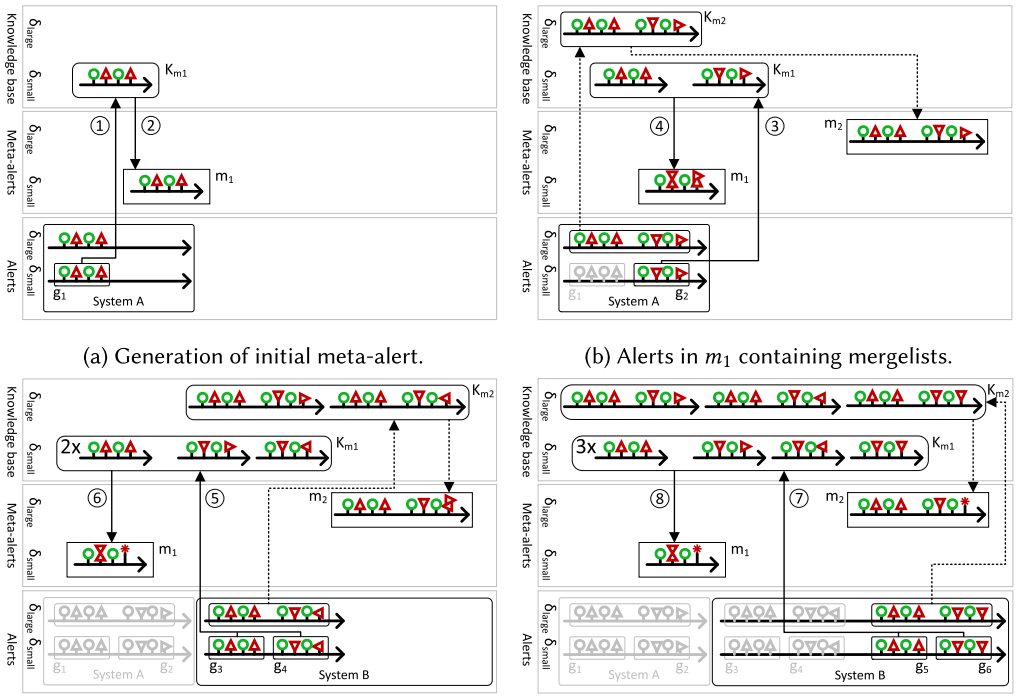
The previous sections provided a theoretical overview of alerts, alert groups, and meta-alerts. Thereby, we defined abstract functions for similarity computation and merging of these concepts to introduce a procedure for automatic meta-alert generation. In this section, we will go into more detail about these functions and discuss their properties with the aid of pseudo code.

5.1 Alert Similarity

This section outlines the alert similarity function from Section 3.2. Since alert objects contain nested dictionaries, recursions are used for similarity computation. Algorithm 1 shows the recursion start in Line 2 of procedure *alert_sim* with parameters $a, b \in \mathcal{A}$. The recursion returns scores for matching and mismatching attributes, which are normalized to a single similarity (Line 3).

The recursive function is realized by iterating over all common attributes of alerts a, b (Line 7). For each of these attributes, the function adds up the achieved match and mismatch scores by comparing the types of the respective values to ensure suitable comparison. This includes (i) dictionary types (Lines 9–10) that call the recursive function with the nested dictionaries, (ii) wildcard (WC) types (Lines 11–12) that always count as matches, (iii) mergelist (ML) types (Lines 13–15) that only count as a mismatch if no two elements are the same in both mergelists and yield larger match scores for more identical elements present in both mergelists corresponding to the overlap metric [20], (iv) standard list types (Lines 16–20) that measure the ratio of common elements with respect to all elements in both lists, and (v) standard value types (Lines 21–25) match if the values of the same attribute in both alerts are identical, and count as mismatches otherwise. We place the check of standard value types at the end to ensure comparison is not carried out on different data types. Finally, the number of keys that are present in one of alerts a, b but not the other contributes to the mismatch score (Line 29), where $\ominus$ is the operator for symmetric difference.

The function fulfills all requirements specified in Section 3.2. The normalization in Line 3 ensures that the resulting similarity scores lie within the interval $[0, 1]$, where 1 indicates that all keys and



(a) Generation of initial meta-alert.

(b) Alerts in m_1 containing mergelists.

(c) Wildcard occurring in fourth alert of m_1.

(d) Additional groups do not further change m_1.

Fig. 6. Scenario for cross-system alert merging. Steps (1)–(8) alternate between knowledge base extensions and repeated meta-alert generation.

values match, and 0 indicates that none of the keys and values match. Moreover, the similarity function is symmetric, since all operations on alerts a, b are symmetric. Due to the incorporation of wildcard and mergelist types, the similarity function implicitly measures how well one alert is represented by another merged alert when comparing alerts on different levels of abstraction.

The pseudo code contains the most important type comparisons, however, the presented checks are not exhaustive. For example, match and mismatch scores of specific type combinations such as list and mergelist are neglected for brevity. Furthermore, the pseudo code does not incorporate weights, which allow to steer the contribution of attributes to the similarity. For example, attribute "timestamp" should have weight 0, because the occurrence times of alerts are not expected to match and should not prevent perfect matches. We refer to our implementation for more details (see Section 1).

5.2 Group Similarity

This section defines a similarity measure for groups that fulfills the requirements defined in Section 3.3.2. In addition, we present the pseudo code for the comparison method that we refer to as the exact matching strategy in Section 3.4.2. We select this strategy, because it establishes the basis for the other matching strategies that involve additional steps such as clustering of alerts within groups.

Algorithm 2 shows the pseudo code for the matching procedure *get_matching* as well as the group similarity function *group_sim*. The matching procedure takes two groups $g, h \in \mathcal{G}_\delta$ as

ALGORITHM 1: Alert similarity computation.

```
1: procedure ALERT_SIM(a, b)
2:     match, mismatch ← alert_sim_rec(a, b)
3:     return match/(match + mismatch)
4: end procedure
5: procedure ALERT_SIM_REC(a, b)
6:     match ← 0, mismatch ← 0
7:     for all k ∈ {κ_a ∩ κ_b} do
8:         mat ← 0, mis ← 0
9:         if type(a.k) = dict and type(b.k) = dict then
10:            mat, mis ← alert_sim_rec(a.k, b.k)
11:        else if type(a.k) = WC or type(b.k) = WC then
12:            mat ← 1
13:        else if type(a.k) = ML or type(b.k) = ML then
14:            mat ← |a.k ∩ b.k| / min(|a.k|, |b.k|)
15:            mis ← (mat > 0) ? 1 : 0
16:        else if type(a.k) = list or type(b.k) = list then
17:            com ← |a.k ∩ b.k|
18:            dif ←max(|(a.k \ b.k|, |b.k \ a.k|)
19:            mat ← com/(com + dif)
20:            mis ← dif/(com + dif)
21:        else if a.k = b.k then
22:            mat ← 1
23:        else
24:            mis ← 1
25:        end if
26:        match ← match + mat
27:        mismatch ← mismatch + mis
28:    end for
29:    mismatch ← mismatch + |κ_a ⊖ κ_b|
30:    return match, mismatch
31: end procedure
```

ALGORITHM 2: Group similarity computation.

```
1: procedure GET_MATCHING(g, h)
2:     pairs ← list(), sims ← list()
3:     for all a ∈ g do
4:         for all b ∈ h do
5:             pairs.append((a, b))
6:             sims.append(alert_sim(a, b))
7:         end for
8:     end for
9:     sort(pairs, by=sims, order=descending)
10:    return pairs
11: end procedure
12: procedure GROUP_SIM(g, h)
13:    sim ← 0
14:    used_a ← ∅
15:    used_b ← ∅
16:    for all a, b ∈ get_matching(g, h) do
17:        if a ∉ used_a and b ∉ used_b then
18:            used_a ⇐ a
19:            used_b ⇐ b
20:            sim ← sim + alert_sim(a, b)
21:        end if
22:    end for
23:    return sim/max(|g|, |h|)
24: end procedure
```

parameters and computes the pairwise similarities between the alerts of each group (Lines 3–8) and stores both the alert pairs and their similarities in respective lists. Then, the pairs are sorted in decreasing order so that the most similar pairs of alerts are at the beginning of the list (Line 9).

As mentioned in Section 3.4.2, computing the alert similarity between all alerts may cause a loss of performance when large groups are compared. The bag-of-alerts matching strategy solves this issue by calling the function with representative alerts for each group, which largely reduces the number of required alert similarity computations. More details on the similarity computation in the bag-of-alerts model is stated in Section 3.4.2 and provided in our implementation.

The function *group_sim* also takes two groups $g, h \in \mathcal{G}_\delta$ as parameters and computes an aggregated similarity of all contained alerts. For this, it first finds the matching between the alerts of the groups and then iterates over all returned pairs (Line 16). Lines 17–19 ensure that each alert is only considered at most once, i.e., alert pairs where one of the alerts was already used for similarity computation are skipped. Line 20 shows that the aggregated group similarity is the sum of all individual pair similarities. Finally, the resulting similarity is normalized in Line 23 to lie in the interval $[0, 1]$. This line also ensures that alerts without matches due to different group sizes decrease the overall group similarity score. Moreover, the function yields a similarity of 1 if groups g, h are identical and a similarity of 0 when all alert pairs achieve a similarity of 0 as required in

Section 2. Also note that the function is symmetric, but can easily be adapted to measure how well one group g fits to another group h, in particular, by replacing the division in Line 23 with $sim/|g|$.

Note that we neglected alignments [22] in the pseudo code for brevity. This could be achieved by counting mismatching alerts as well as misalignments of matched alerts. We refer to our implementation, where we enhance the final similarity by incorporating such an alignment score.

5.3 Alert Merging

In this section, we outline a function for alert merging as specified in Sectioin 3.4.1. The function generates a new alert that comprises wildcards and mergelists, which are represented by the two classes WC and ML that have already been used for alert similarity computation in Algorithm 1.

Algorithm 3 shows the pseudo code for generating a merged alert. Parameters are a set of alerts $A \subseteq \mathcal{A}$ to be merged, a ratio $k_{min} \in [0, 1]$ specifying the minimal relative occurrence frequency of an attribute to be included in the merged alert, a ratio $v_{min} \in [0, 1]$ specifying the minimal relative occurrence frequency of a value to be included in an attribute of the merged alert, and a number $v_{max} \in \mathbb{N}_0$ specifying the maximum amount of values before mergelists are replaced by wildcards.

The procedure first extracts list *keys* that holds all attributes present in the alerts, where identical keys are stored multiple times (Lines 2–5). Line 6 computes the occurrence frequencies of these keys using a count function, which yields dictionary *keys_count* that holds all keys and their respective frequencies. Lines 6–11 then remove all keys with relative occurrence frequencies smaller than k_{min}. This step ensures that rare keys that do not occur in sufficiently many alerts of A are omitted.

An initially empty object c for the alert merge is defined in Line 12. Every remaining attribute key k is then added iteratively to that object in Lines 13–35. Thereby, the values of each key from all alerts are stored in a list (Lines 14–19). Note that a value may also be a list or mergelist, in which case the list *vals* is extended with all values from that list. In case that all values of a particular key are dictionaries, the function *alert_merge* is called recursively for that attribute (Lines 20–21). Otherwise, the values in the list are counted to remove all values that occur with relative frequencies lower than v_{min} (Lines 23–28). In case that no values remain, the number of values exceeds v_{max}, or one of the values is a wildcard, an attribute with key k and a wildcard as value is added to the merged alert c (Lines 29–30). Otherwise, all disclosed values are added to an attribute holding a mergelist (Line 32). Line 36 returns the merged alert c after processing all keys.

As required in Section 3.4.1, every generated alert c is a possibly nested semi-structured object that only holds wildcards and mergelists in its attributes. Accordingly, it is possible to treat it like any other alert $a \in \mathcal{A}$, which includes similarity computations and merging.

5.4 Group Merging

This section discusses the group merging function introduced in Section 3.4.2 that is capable of generating meta-alerts, i.e., aggregated alert groups. The parameters of the function outlined in Algorithm 4 are a set of groups $G \subseteq \mathcal{G}_\delta$, a similarity threshold for alerts $\theta_{alert} \in [0, 1]$, and the values $k_{min}, v_{min}, v_{max}$ required for the *alert_merge* function discussed in Section 5.3.

We introduced the bag-of-alerts model to alleviate performance issues that arise from determining alert matches between two large groups to compute their similarity. Unfortunately, group merging introduces a new problem, since not just two, but arbitrary numbers of groups can be merged at the same time. The main issue with that scenario is that finding alert matches between all pairs of groups is highly resource-intensive and should therefore be avoided. In the following, we solve this problem by merging groups incrementally, i.e., use one group as a representative that all other groups are merged to. This is also represented in Figure 4, where alerts of both groups g_1, g_3 are matched with alerts of group g_2, but there is no alert matching taking place between groups g_1, g_2 themselves. In that scenario, group g_2 acts as the representative group.

ALGORITHM 3: Alert merge computation.

```
1: procedure ALERT_MERGE(A, k_min, v_min, v_max)
2:     keys ← list()
3:     for all a ∈ A do
4:         keys.extend(κ_a)
5:     end for
6:     keys_count ← count(keys)
7:     for all k, freq ∈ keys_count do
8:         if freq/|A| < k_min then
9:             keys.remove(k)
10:        end if
11:    end for
12:    c ← dict()
13:    for all k ∈ keys do
14:        vals ← list()
15:        for all a ∈ A do
16:            if k ∈ κ_a then
17:                vals.append(a.k)
18:            end if
19:        end for
20:        if ∀type(v ∈ vals) = dict then
21:            c_k ← alert_merge(
                   vals, k_min, v_min, v_max)
22:        else
23:            vals_count ← count(vals)
24:            for all v, freq ∈ vals_count do
25:                if freq/|values| < v_min and
                       type(v) ≠ WC then
26:                    vals.remove(v)
27:                end if
28:            end for
29:            if |vals| = 0 or |vals| > v_max or
                   ∃type(v ∈ vals) = WC then
30:                c_k = WC()
31:            else
32:                c_k = ML(vals)
33:            end if
34:        end if
35:    end for
36:    return c
37: end procedure
```

ALGORITHM 4: Group merge computation.

```
1: procedure GROUP_MERGE(G, θ_alert, k_min, v_min,
       v_max)
2:     largest_group ← list()
3:     for all g ∈ G do
4:         if |g| > |largest_group| then
5:             largest_group ← g
6:         end if
7:     end for
8:     d ←dict()
9:     for all a ∈ largest_group do
10:        d[a] ← {a}
11:    end for
12:    for all g ∈ (G \ largest_group) do
13:        used_a ← ∅
14:        used_b ← ∅
15:        for all a, b ∈ get_matching(g, d.keys()) do
16:            if alert_sim(a, b) < θ_alert then
17:                break
18:            else if a ∉ used_a and
                       b ∉ used_b then
19:                used_a ⇐ a
20:                used_b ⇐ b
21:                d[b] ⇐ a
22:            end if
23:        end for
24:        for all missing ∈ (g \ used_a) do
25:            d[missing] ⇐ missing
26:        end for
27:    end for
28:    h ←list()
29:    for all A ∈ d.values() do
30:        c ← alert_merge(A, k_min, v_min, v_max)
31:        h.append(c)
32:    end for
33:    return h
34: end procedure
```

Lines 2–7 in Algorithm 4 show that we select the largest group in the set of groups G as the representative group, because it contains the most alerts and is thus the most likely to yield many alert matches with all other groups to be merged. We then define a dictionary d in Line 8 that holds lists of alerts to be merged in its values. For this, we first initialize it by adding all alerts of the largest group as keys and each alert in a list as their values. Lines 12–27 then append all matching alerts of other groups to these lists by iterating over all remaining groups. For each group, the alert matching is computed in Line 15. We iterate over all alert pairs ordered by their achieved similarity (cf. Algorithm 2) and add the alerts to the best matching key of dictionary d. The iteration stops when the minimum similarity θ_{alert} is reached (Line 16). This check is necessary to avoid that alerts with low similarity are incorrectly merged with each other, resulting in over-generalized alerts in the merged group. In case that not all alerts of the currently processed group could be

ALGORITHM 5: Incremental merging.

```
1: procedure ADD_GROUP(K_δ, g, θ_group, θ_alert, k_min, v_min, v_max)
2:     sim_max ← −1
3:     for all K_m ∈ K_δ do
4:         sim ←group_sim(g, m)
5:         if sim > sim_max then
6:             sim_max ← sim
7:             best ← m
8:             if sim = 1 then
9:                 break
10:            end if
11:        end if
12:    end for
13:    if sim_max < θ_group then
14:        best ←Meta_Alert()
15:    end if
16:    K_best ⇐ g
17:    K_best ←group_merge(K_best, θ_alert, k_min, v_min, v_max)
18: end procedure
```

ALGORITHM 6: Meta-alert generation.

```
1: procedure GENERATE_META_ALERTS(A, Δ, θ_group, θ_alert, k_min, v_min, v_max)
2:     for all δ ∈ Δ do
3:         K_δ ← ∅
4:         for all A ⊆ A, ∀a ∈ A : ∃b ∈ A : |a.t − b.t| < δ do
5:             g ← A : a_1.t ≤ a_2.t ≤ ··· ≤ a_n.t, ∀a_i ∈ A
6:             G_δ ⇐ g
7:         end for
8:     end for
9:     for all g ∈ G_δ do
10:        add_group(K_δ, g, θ_group, θ_alert, k_min, v_min, v_max)
11:    end for
12: end procedure
```

matched, e.g., if the achieved similarity to any alert in the largest group is lower than the minimum matching similarity θ_{alert}, the alerts are added as new keys in d (Lines 24–26) for finding matches in other groups.

After processing all groups, the algorithm iterates over all values of d, merges the alert lists using function *alert_merge*, and stores each of the generated merged alerts in the initially empty list h (Lines 28–32). List h thus contains a sequence of alerts merged from all groups $G \subseteq \mathcal{G}_\delta$, which means that list h is a meta-alert that has all properties of a group as required in Section 3.4.2. Finally, group h is returned by the function in Line 33.

We do not provide the pseudo code for the generation of merged groups using the bag-of-alerts model for brevity. Similar to the group similarity algorithm from Section 5.2, the main difference is that alert representatives instead of the actual alerts are used for matching. In addition, intervals for occurrence counts of alerts are adjusted during merging so that all merged groups are appropriately represented (cf. Section 5.4). The alignment of alerts in the bag-of-alerts model is computed by repeatedly applying the LCS procedure to the individual alignments of alert representatives of all groups. For more details on the realization of these methods, we refer to our implementation.

5.5 Meta-alert Generation

The *group_merge* function presented in the previous section allows to generate meta-alerts from sets of similar groups. To select these groups, we outline procedure *add_group* in Algorithm 5 that produces meta-alerts using the knowledge base as proposed in Section 4.1. In particular, the procedure iterates over all meta-alerts stored in the knowledge base (Line 3) and computes their similarities to the currently processed group g (Line 4) to find the meta-alert *best* that yields the highest similarity (Lines 5–11). In case that a comparison yields a perfect similarity of 1, there is no need to check all other meta-alerts and the loop stops prematurely to improve performance (Lines 8–10).

After the loop is completed, Line 13 checks whether the highest similarity between group g and any meta-alert is lower than threshold θ_{group} or no meta-alerts are available. In this case, a new meta-alert is generated by replacing *best* with an object of class *Meta_Alert* in Line 14, otherwise

best is an adequate match for group g. Either way, group g is added to the knowledge base of meta-alert *best* (Line 16) and the corresponding meta-alert is subsequently updated (Line 10).

Function *generate_meta_alerts* in Algorithm 6 runs the overall framework. The parameters involve all alerts $\mathcal{A}$, a group similarity threshold $\theta_{group} \in [0, 1]$, and the parameters $\theta_{alert}, k_{min}, v_{min}, v_{max}$ that are already known from Sections 5.3 and 5.4. The function iterates over all $\delta \in \Delta$ values specified by the analyst (Lines 2–8) and initializes a knowledge base for every δ as an empty set (Line 3). Lines 4–7 represent the group formation phase for each δ value. Note that instead of the agglomerative clustering algorithm, we only display the requirements on the groups for simplicity, i.e., alerts in groups must occur in sufficiently close temporal proximity (Line 4) and be sorted by timestamp (Line 5). Finally, the function iterates over all groups and calls function *add_group* repeatedly (Lines 9–11). Note that this procedure was designed for an offline setting where all groups are known in advance, however, we argue that it is easy to adapt the code for online analysis.

The pseudo codes in this and the previous section leave out several aspects of our implementation that were omitted for brevity. This includes handling of multiple δ values by running function *generate_meta_alerts* in parallel, queuing strategies, and heuristics that allow to prematurely stop alert and group matching procedures for groups with low similarity to improve performance. Once more we refer to our implementation that provides more details on such aspects.

6 EVALUATION

This section outlines our evaluation of the proposed alert aggregation approach. We first describe the methodology of our evaluation and introduce the data that we used to generate meta-alerts before showing and discussing the results. We point out that the code and data used in this evaluation are available open-source (see Section 1) and thus all results are reproducible.

6.1 Methodology

The purpose of our evaluation is to validate the approach presented in this paper with respect to well-known metrics that are relevant in machine learning and alert aggregation. Thereby, we use a a publicly available real-world data set presented in Section 6.2 that centers around an illustrative attack scenario. The evaluation aims to demonstrate the capability of our framework to extract meta-alerts that represent attack manifestations present in the data.

For a better overview, we evaluate the introduced concepts and operations of our framework stepwise in alignment with Section 3. First, Section 6.3 provides empirical results from the group formation strategy and is used for the selection of appropriate δ values for the remainder of the evaluation. Section 6.4 contains a plot of the group similarities that allows to visually examine whether the proposed similarity functions are suitable for allocating alert groups to attacks. The hierarchical clustering shown in Section 6.5 allows to draw similar conclusions, but additionally visualizes whether the outcomes of the proposed merging functions fit the overall picture. This hierarchical clustering also allows to estimate appropriate values for θ_{group}, which is used in the following sections.

The remaining evaluations focus on quantitatively measuring the performance of the overall approach rather than visual validation and parameter estimation. In Section 6.6, we execute the incremental alert aggregation procedure in a fully unsupervised setting and measure the clustering accuracy as well as the reduction rate. Section 6.7 on the other hand first generates meta-alerts in a supervised way and then measures the accuracy of classification of unknown sample alerts. We argue that both unsupervised and supervised evaluations are necessary to evaluate the capability of generating meta-alerts as well as using these meta-alerts for detection of similar alerts, respectively.

We measure reduction rates for several δ values and thresholds $\theta_{group}, \theta_{alert}$ in Section 6.8. Since real systems are affected by false positive alerts that impair the performance of alert aggregation, we evaluate the robustness of our approach in Section 6.9. Finally, we discuss the results of the evaluation with respect to the requirements from Section 6.11. All evaluations are carried out on a 64-bit Windows 10 machine with an Intel i7-6600U CPU at 2.60 GHz and 16 GB RAM running Python 3.6.8.

6.2 Data

We use the publicly available data set *AIT-LDSv1.1*[4] [16] for our evaluation. The advantage of this data set is that it comprises diverse log files collected from four different web servers that are targeted by the same attack scenario, which is a multi-step attack that involves (i) an Nmap[5] scan, (ii) a vulnerability scan using Nikto[6], (iii) an enumeration of user accounts using the `vrfy` command of the smtp-user-enum tool[7], (iv) a brute-force login attack using Hydra[8], (v) an exploit of a webmail client for webshell upload (CVE-2019-9858), and (vi) an exploit of Exim for privilege escalation (CVE-2019-10149). The parameters of some attack steps are thereby varied so that their manifestations in the log data appear different on each system.

We used two open-source host-based intrusion detection systems, Wazuh[9] and AMiner[10], to process the logs and generate alerts. Wazuh is a signature-based detection engine that comes with a predefined set of rules and was used in its standard configuration. The sample alert from Figure 2 is one of the alerts generated by Wazuh from the logs. AMiner on the other hand is an anomaly-based IDS that was configured to report unknown events as well as new values and combinations of values that occur in predefined positions of the log events. Since the exact execution times of each of the six aforementioned attack steps are provided in the data set, we were able to label all alerts for our evaluation accordingly. Note that the generated alerts are not in IDMEF format or involve useful IP information and thus cannot be appropriately handled by existing approaches.

Figure 7 shows the alerts on timelines for each of the four web servers named *cup*, *spiral*, *onion*, and *insect*. The total number of alerts on all systems is 57,766. We display the type of the alerts (attribute "`AnalysisComponentName`" in AMiner alerts and attribute "`description`" in Wazuh alerts) on the vertical axis, where (1) marks AMiner anomaly types and (2) marks Wazuh rules. The alerts are displayed as circles, where larger sizes indicate more co-occurring alerts. These groups were formed by our grouping approach from Section 3.3.1 with $\delta = 1$ second. Note that the attack using the smtp-user-enum tool (green) was not executed on the *onion* web server and that there is a false positive alert caused by an update after the Exim exploit (purple) on the *insect* web server.

Comparing the graphs for the four web servers shows that attack parameter variations cause highly different alert patterns, in particular, the duration and amount of alerts generated by the Nikto scan (blue) and the user account enumeration (pink) varies greatly. On the other hand, close inspection of the plots reveal that some attack steps that are less affected by variations, e.g., the Exim exploit (purple), show the appearance of the same types of alerts with similar frequencies and timings on all systems. This suggests that it possible to derive meta-alerts across infrastructures that comprise attack patterns suitable for the detection of the same attack on other systems.

[4] https://zenodo.org/record/4264796.
[5] https://nmap.org/.
[6] https://cirt.net/Nikto2.
[7] https://tools.kali.org/information-gathering/smtp-user-enum.
[8] https://tools.kali.org/password-attacks/hydra.
[9] https://wazuh.com/.
[10] https://github.com/ait-aecid/logdata-anomaly-miner.

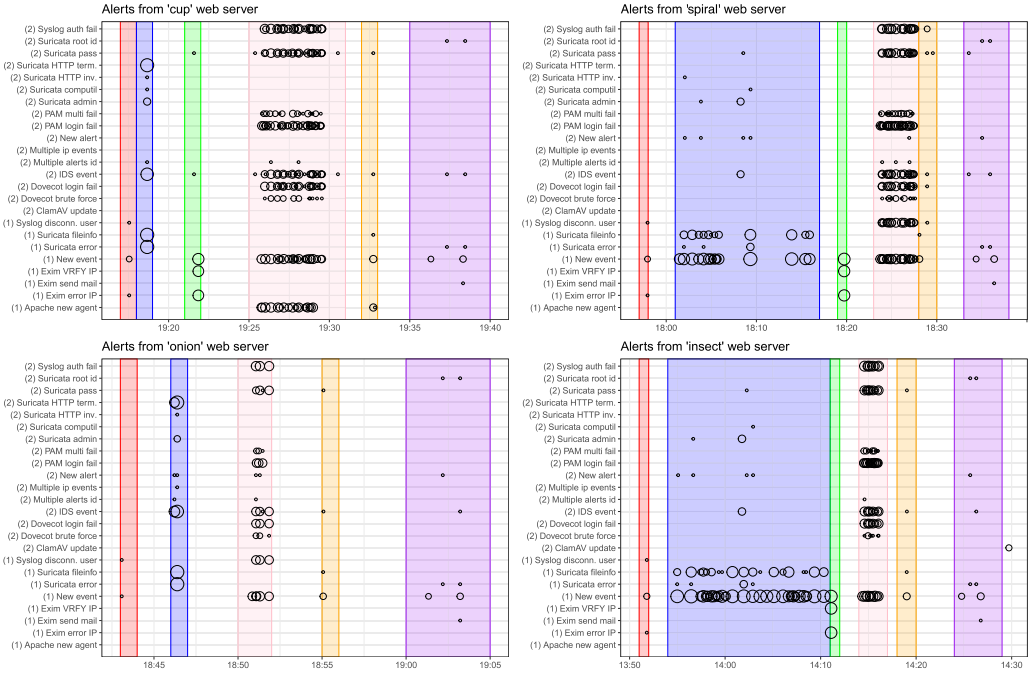


Fig. 7. Alerts from AMiner (1) and Wazuh (2) IDS on four web servers. From left to right, the attacks are Nmap scans (red), Nikto vulnerability scans (blue), user enumerations (green), Hydra brute-force login attempts (pink), webmail exploits for webshell uploads (orange), and Exim exploits for privilege escalation (purple).

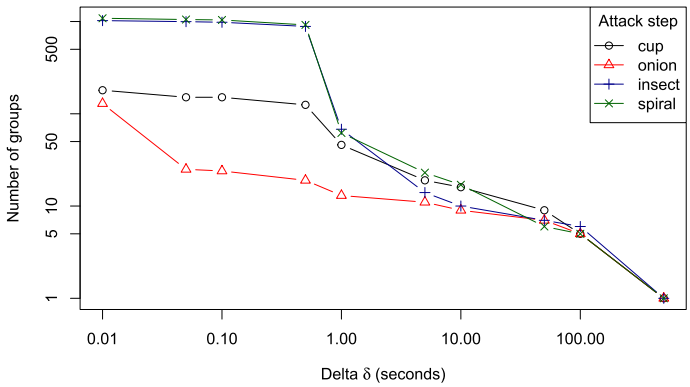


Fig. 8. Influence of δ value on the number of generated alert groups on four systems.

6.3 Group Formation

We discussed the importance of δ values for our grouping method in Section 3.3.1 and proposed to use multiple δ values in parallel to overcome issues with group formation in Section 4.1. To support this decision with real-world evidence, we plot the number of groups from four systems for different δ values in Figure 8. Note that other than the plots in Figure 7, groups are formed across all alert types.

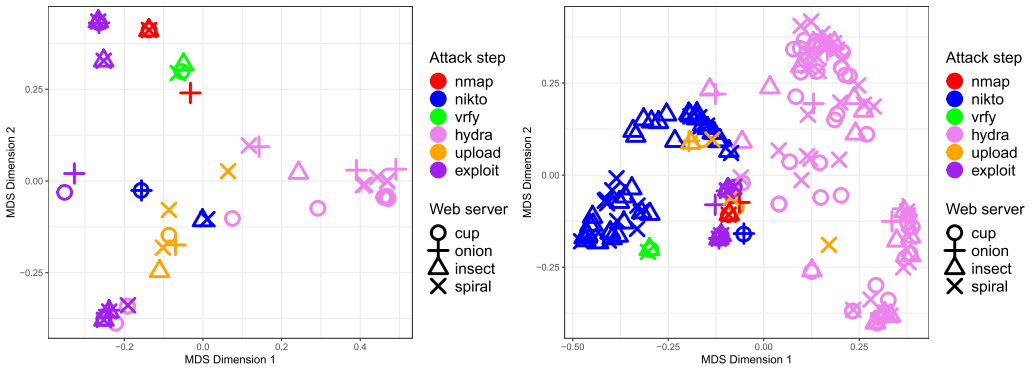


Fig. 9. Multi-dimensional scaling of pairwise group similarities using δ values of 10 seconds (left) and 1 second (right) shows that groups of same attack steps are frequently located close together.

As expected, fewer and larger groups are formed for increasingly larger values of δ and vice versa. In particular, the lowest selected δ value of 0.01 seconds yields a total of 2,417 groups on all systems, while a δ value of 500 seconds causes that only a single group on each system is generated, comprising all attack phases. This large range confirms that the usage of multiple δ values is reasonable. Furthermore, the figure suggests that δ values should be selected on a logarithmic range to avoid that the same or very similar groups are formed multiple times on different δ levels. Accordingly, we will only consider logarithmically distributed δ values in the following.

6.4 Group Similarities

Due to the fact that our meta-alert generation approach is based on group similarities, it is necessary that our similarity functions are capable of clustering related groups with high accuracy. We therefore compute a pairwise similarity matrix of all groups formed using a specific δ value. For this, we make use of the *group_sim* function (cf. Section 5.2) that relies on the alert similarity function *alert_sim* (cf. Section 5.1). As outlined in Section 3.4.2, quadratic runtime complexity of the exact matching strategy makes it necessary to switch to the bag-of-alerts strategy for large groups. We empirically determined that $l_{bag} = 2000$ keeps processing times for most groups below 0.05 seconds, which we consider acceptable. Furthermore, we set $k_{min} = 0.1$, $v_{min} = 0.1$, and $v_{max} = 10$ to ensure that meta-alerts do not contain alert characteristics that occur in less than 10% of merged groups. Finally, we set the weight of alignment information to 0.1 and the weight of attribute "timestamp" to 0. These parameters are used in all following evaluations unless stated otherwise.

We then apply multi-dimensional scaling [5], which is a dimensionality-reduction technique that allows to represent groups as points that largely retain their original similarities derived from the pairwise similarity matrix. Figure 9 shows the groups formed with δ values of 10 seconds (left) and 1 second (right) plotted on the first two dimensions. Groups are marked with the same symbol if they originate from the same system and have the same color if they occurred in the same attack phase. Since groups that relate to the same attack should be similar independent from the system infrastructure, we expect to see groups with the same color and different symbols close together.

The left figure shows that several groups correctly form clusters of related attack steps. For example, the groups related to the "vrfy" attack are placed close together and are relatively isolated. Other groups, such as the ones belonging to the "hydra" attack, appear more spread out. The groups belonging to "nikto" result in two distinct clusters, which corresponds to Figure 7 that shows that

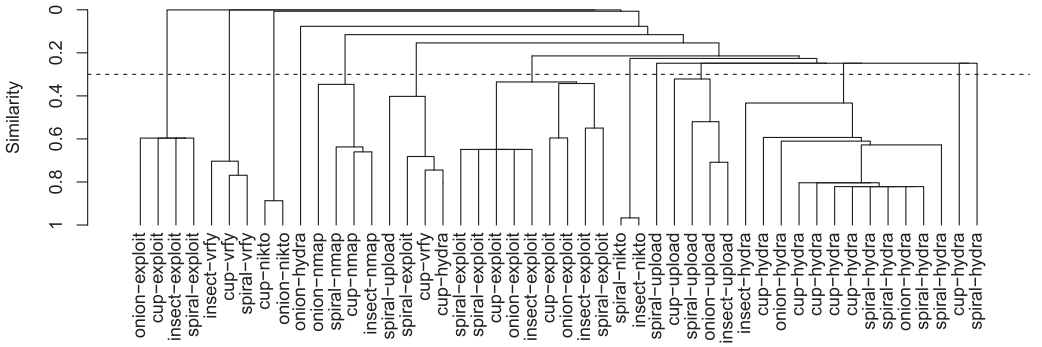


Fig. 10. Dendrogram of group similarities, where each node represents a merged group. The dashed line shows a reasonable cutoff threshold at a similarity of 0.3 that yields meta-alerts with few incorrect allocations.

this attack step lasted over a long time on *spiral* and *insect* systems, but only a short time on *cup* and *onion*. Similarly, the "exploit" attack step forms four separate clusters. The reason for this is that this attack step actually consists of several smaller steps that are sequentially executed and disclosed as separate groups at this δ level, but comprise rather different alert types and frequencies.

The right plot of Figure 9 shows that much more groups are generated for the "nikto" and "hydra" attack when δ is set to a lower value. Since groups that belong to one of these two attacks dominate the variance of the data, it is difficult to reason about the correct clustering of groups that belong to other attacks. However, the fact that groups of "nikto" and "hydra" attacks are clearly separated suggests that similarity-based group clustering is reasonable even for small δ values.

6.5 Hierarchical Aggregation

We evaluate *alert_merge* (cf. Section 5.3) and *group_merge* (cf. Section 5.4) by first computing the pairwise similarity matrix and then merging the two groups that yield the highest similarity. These groups are then removed from the matrix and instead the resulting merged group is added by computing its similarity to all remaining groups. This is repeated until the matrix only contains one group. We then construct a rooted dendrogram with branches that connect at the height of the similarity of the merged groups. AMiner and Wazuh alerts are thereby only grouped but not merged, as they have no common attributes and thus always achieve an alert similarity of $0 < \theta_{alert}$.

Figure 10 displays a dendrogram for groups formed with $\delta = 10$ seconds. The original groups are placed at the leaves and labeled by their corresponding systems and attack steps. Each node represents a merged group, where the height displayed on the vertical axis describes the similarity of the merged groups. The dendrogram shows that groups that are part of the same attack phase are frequently merged with relatively high similarity, while groups that relate to different attacks only merge with low similarity. For example, each of the first four "exploit" groups from the left occurred on different systems and were merged with a similarity of 0.6. The similarity of the resulting merged group to all other groups was 0, indicating that meta-alerts do not tend to overgeneralize.

To select the similarity threshold θ_{group}, we plot cluster purity [20] against the number of clusters in Figure 11. The similarity threshold is selected so that purity is large, i.e., clusters contain mostly groups belonging to the same attack phase, and the number of clusters approximates the true number of attack steps, which is 6 in our case. The figure shows that purity drops for thresholds lower than 0.25, while the number of clusters continuously increases for larger thresholds.

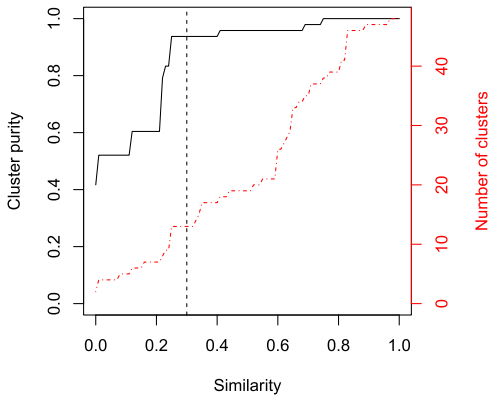


Fig. 11. Cluster purity and the total number of clusters suggests 0.3 as a similarity cutoff threshold.

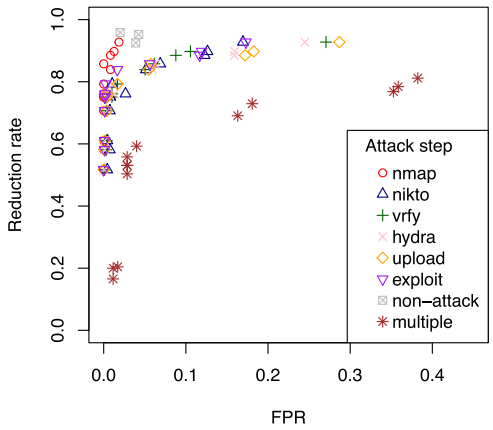


Fig. 12. Reduction rate plotted against false positive rate for several threshold values.

Accordingly, we select $\theta_{group} = 0.3$ as a reasonable trade-off that yields a purity of 0.94 and 13 clusters.

The reason why the number of clusters is larger than the number of attack phases is due to the fact that the labels of the data are not sufficiently fine-grained, i.e., some attack phases actually comprise sequences of sub-steps that should be labeled differently from each other. We decided against manually altering the ground truth data to fit our needs and we will therefore mainly focus on the correct separation of groups into homogeneous meta-alerts in the next section.

6.6 Meta-Alert Generation

The previous sections evaluated the similarity and merging functions. In this section, we evaluate the procedure using these functions for incremental meta-alert generation (cf. Section 5.5). For this, we use the logarithmic storage strategy with a maximum queue size of 25. We then create groups for intervals $\Delta = \{0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500\}$ seconds and iteratively generate meta-alerts through continuous similarity computations and merging. To analyze the influence of the thresholds specifying minimum similarity for group merging and alert matching, we conduct the experiment multiple times and vary $\theta_{group}, \theta_{alert}$ equally in the range of $[0.1, 0.7]$ and a step width of 0.05.

We obtain a set of meta-alerts $\mathcal{M}_\delta$ for each $\delta \in \Delta$ once all groups are processed. To evaluate the quality of the meta-alert generation procedure, we measure the homogeneity of all meta-alerts. In particular, we count (i) two groups with the same attack phase label in the same meta-alert as true positives (TP), (ii) two groups with different labels in the same meta-alert as false positives (FP), (iii) two groups of the same label in different meta-alerts as false negatives (FN), and (iv) two groups with different labels in different meta-alerts as true negatives (TN). As mentioned in previous sections, groups which belong to the same attack phases frequently end up in separate meta-alerts due to sub-steps in attack executions, and the true positive rate $TPR = TP/(TP + FN)$ is therefore not expressive. For this reason, we plot the false positive rate $FPR = FP/(FP + TN)$ against the reduction rate, i.e., the ratio between the number of meta-alerts and groups computed as $r_{group} = |\mathcal{M}_\delta| \,/\, |\mathcal{G}_\delta|$. To deal with groups that only contain false positive alerts and groups that span over several attack phases, we introduce the labels *non-attack* and *multiple*.

Figure 12 shows the results of aforementioned calculations carried out for each attack phase separately. Each point shows the average FPR and reduction rate over all δ values achieved for

a particular threshold for both θ_{group} and θ_{alert}. In general, larger thresholds yield more meta-alerts that lead to lower *FPR* and reduction rates, i.e., points closer to the bottom-left of the plot, while smaller thresholds cause that groups are more easily merged to meta-alerts, which increases reduction rate and *FPR*, i.e., result in points closer to the top-right of the plot. There is thus a trade-off between the reduction rate and accuracy. Thresholds in the range $[0.2, 0.4]$ yield the best results for all attack phases and achieve average reduction rates of around 80% and average *FPR* of less than 5% for all attack types. These results correspond to a rule of thumb from Husák et al. [12], who state that up to 85% of alerts can reasonably be aggregated on average.

6.7 Cross-System Classification

The previous section focused on the evaluation of the incremental meta-alert generation procedure in an unsupervised way. To evaluate whether the generated meta-alerts are suitable for classification of attack executions on other systems, we carry out a supervised evaluation. In particular, we use alerts from three out of the four systems to form groups and generate meta-alerts in a training phase, where groups are only allocated and merged with meta-alerts that belong to the same attack phase. We then use the alerts from the fourth system as test data to generate groups, determine the most similar meta-alert, and measure the accuracy of this classification. We also require that the similarity to the best-matching meta-alert exceeds 0.1, otherwise the group is assigned to the non-attack class. Thereby, we count (i) a true positive (*TP*) for an attack if a group belonging to that attack is correctly allocated to a meta-alert of the same attack, (ii) a false positive (*FP*) for the meta-alert's attack and (iii) a false negative (*FN*) for the group's attack if the group is incorrectly allocated to a meta-alert with a different attack, and (iv) a true negative (*TN*) for all attacks that the group is correctly not assigned to. To obtain better estimations for model performance through cross-validation, we repeat this procedure so that alerts from every system are used as test data and average the results. As before, we also use a range of δ values and compute all rates as averages.

The left side of Figure 13 shows *TPR* plotted against *FPR* for all attacks, where each point represents the results achieved using a specific threshold. The graph shows that each of the six original attack classes achieve a low *FPR* of less than 5%. The *TPR* appears to depend on the attack type, since several points that refer to the same attack label are relatively close together and form groups. The attacks achieve *TPR* in the range $[0.75, 0.95]$ for most threshold settings, except for the "upload" attack, which performs comparatively bad with a *TPR* of only 0.5. The reason for this is that this attack only caused similar alert sequences in *onion* and *insect*, but involved additional alerts in *cup* and *spiral* due to differences in the infrastructure setup (cf. Figure 7).

The right side of Figure 13 shows the confusion matrix for $\theta_{group} = \theta_{alert} = 0.3$. Note that other than the plot on the left side, the confusion matrix represents total numbers of TP, FP, FN, and TN rather than averages over all δ values. We normalized the matrix column-wise to obtain the relative frequencies of class allocations so that *TPR* is visible in the main diagonal. This allows to obtain a better overview of the misclassifications, e.g., around 50% of alert groups belonging to "upload" are incorrectly classified as one of "non-attack", "vrfy", "hydra", or "nikto" for aforementioned reasons.

We also visualize the performance of our classifier with respect to δ. For this, we use the F1-score computed by $F1 = TP/(TP + 0.5 * (FN + FP))$, since it provides a single measure that is large when both *FN* and *FP* are low. Figure 14 shows box plots of the F1-score for several δ values, split up by attack steps and thresholds. As visible by the height of the boxes, the variance of the F1-score with respect to δ values is relatively small. In addition, four out of six attacks have at least one setting for the threshold and δ value so that the highest possible F1-score is reached. The plot also confirms that the performance is mostly dependent on the type of attack, since similar F1-scores are reached for most threshold values. There are some exceptions to this observations,

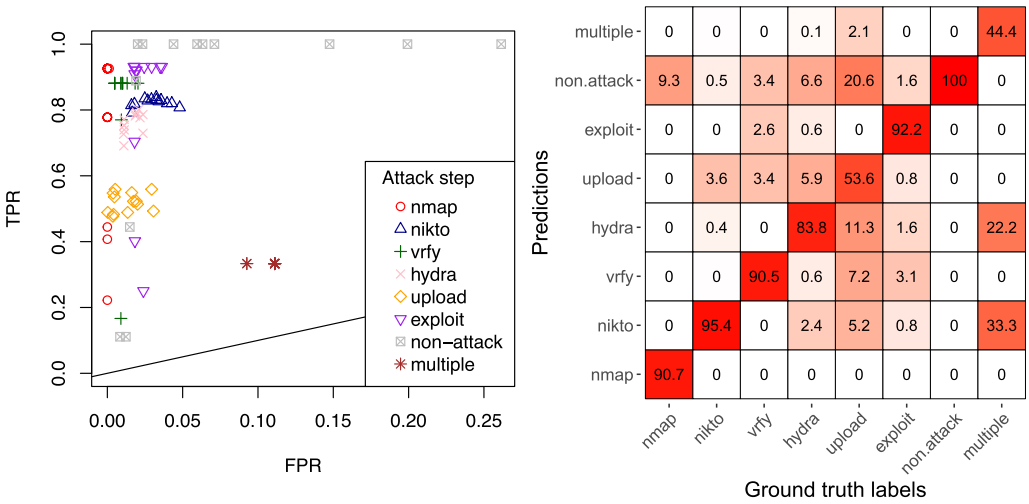


Fig. 13. Left: True positive rate plotted against false positive rate of cross-system attack classifications for several threshold values. Right: Confusion matrix of attack classifications using 0.3 as threshold.

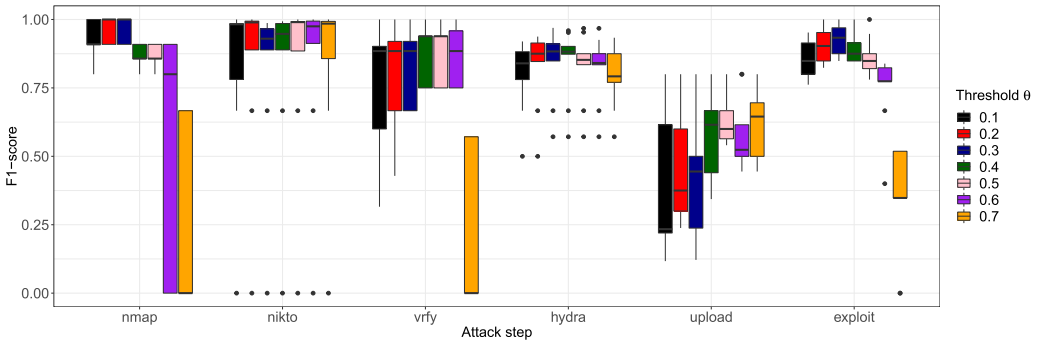


Fig. 14. F1-score boxplots of cross-system classification for several δ values, categorized by attack and θ_{group}.

in particular, the performance declines for a threshold of 0.7 for "vrfy" and for thresholds larger than 0.6 for "exploit" and "nmap". As visible in Figure 13, this is due to a decrease of TPR.

6.8 Reduction

The ability to reduce alert groups to meta-alerts is a key feature of our approach, because a high reduction rate indicates that many groups were merged to few meta-alerts. As shown in Figure 12, there is a limit to the reduction rate at which accuracy starts to decline. To improve understanding of parameter influence on the reduction rate, we therefore visualize the reduction rate for several combinations of δ values and thresholds $\theta_{group}, \theta_{alert}$ in the following. Note that we compute the reduction rate on alerts of only four systems. Using more data, i.e., obtaining alerts of the same attack scenario from additional systems, would likely increase the reduction rates.

The left plot in Figure 15 shows the group reduction rates, i.e., the ratio between the number of meta-alerts and the number of groups computed as $r_{group} = 1 - |\mathcal{M}_\delta| \,/\, |\mathcal{G}_\delta|$. The plot shows that reduction rates decrease for increasing thresholds, because larger thresholds mean that it is less likely that groups reach the minimum required similarity to be allocated to meta-alerts.

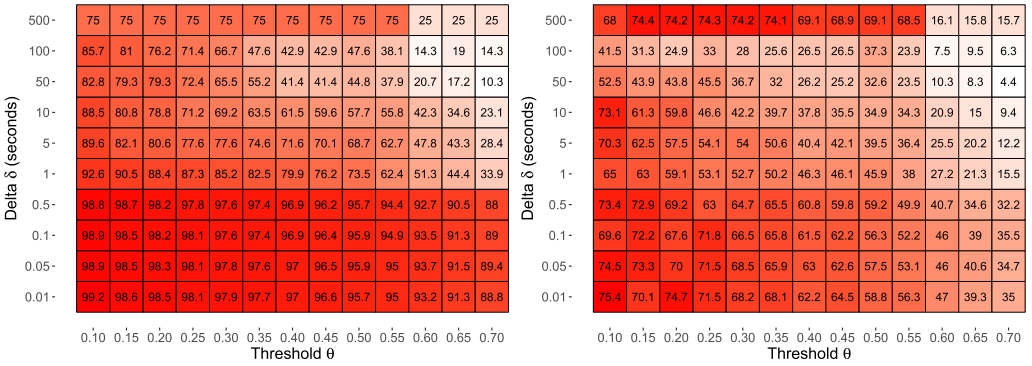


Fig. 15. Influence of δ and θ_{group} on reduction rates. Left: Group reduction rates. Right: Alert reduction rates.

Moreover, reduction rates decrease for increasing δ values, because smaller δ values cause that more groups are generated, making it easier to find similar groups. For example, the "hydra" brute-force attack repeats the same action multiple times in short intervals, and smaller δ values break up the generated alert patterns into shorter sequences that are then suitable to be merged with each other. A cutoff appears around $\delta = 0.5$ seconds that reaches reduction rates of around 88% to 99% for all thresholds, while $\delta = 1$ second yields reduction rates down to 33% for high thresholds.

The right plot in Figure 15 shows the reduction rates of alerts rather than groups. The value is computed as the average reduction rate of all meta-alerts, i.e., $r_{alert} = (1/|\mathcal{M}_\delta|) \Sigma_{m \in \mathcal{M}_\delta} (1 - |m| / (|K_m|))$. Overall, the alert reduction rates also decrease for increasing δ values and thresholds, however, to a less effect compared to the group reduction rates.

6.9 Robustness

Our approach relies on the assumption that adequate alert groups are formed in the first stage of our procedure (cf. Section 3.3.1). Despite using several δ values in parallel, the grouping phase is susceptible to intervening alerts that are not part of attacks. In particular, such noise alerts form new groups, change the composition of existing groups, or cause that groups are combined.

To evaluate the robustness of our approach with respect to noise alerts, we randomly duplicate alerts and uniformly distribute them over the input data. Adjusting the total number of alerts added in this way allows us to set the noise intensity. The plots in Figure 16 show the influence of noise alerts on the number of generated groups and classification performance on each system, where the noise intensity on the horizontal axis is displayed as the average amount of alerts inserted per minute. As visible in the plot on the left side, the total number of groups increases, since random alerts that occur with a temporal distance larger than δ to other alerts form new groups. The curve peaks when approximately 10 noise alerts are inserted, followed by a rapid decline caused by group merges. This is reasonable, since 10 noise alerts per minute mean that an alert is inserted every 6 seconds on average, which corresponds to the used δ values of 5 seconds. The plot on the right side shows that the average TPR and F1-score decline with increasing noise intensity, while the FPR remains constant at a low level. In particular, TPR and F1-score rapidly decrease from around 0.6 to 0 approximately when 10 noise alerts per minute are inserted, corresponding to the peak of the number of groups. We therefore conclude that δ functions as a breakdown point for our approach and that δ values exceeding the average noise intensity should be avoided.

To overcome this issue, we recommend to reconfigure the deployed IDSs. Randomly and repeatedly occurring alerts indicate that some sensors are too sensitive and therefore report normal

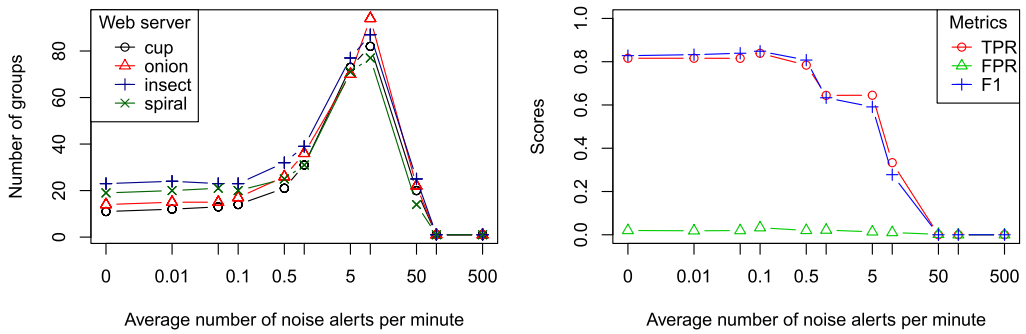


Fig. 16. Robustness to randomly generated noise alerts for $\delta = 5$ seconds and $\theta_{group} = \theta_{alert} = 0.3$. Left: Influence on number of alert groups. Right: Influence on *TPR*, *FPR*, and F1-score.

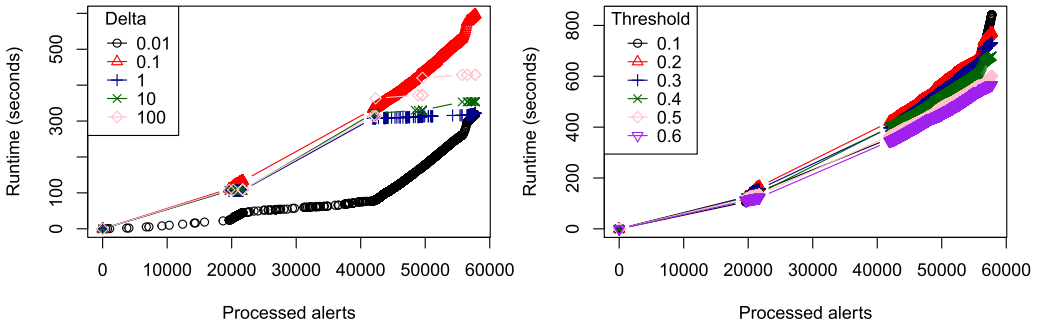


Fig. 17. Runtime required to process alerts for different values for different δ (left) and thresholds (right).

behavior as malicious, which is not desirable for manual or automatic analysis and should be fixed anyway. Alternatively, it is also possible to set up a filter for particular alert types that are known to report false positive alerts prior to performing alert aggregation. In particular, meta-alerts generated by our approach that have been manually labeled as false positive alerts could be used to design such filter rules. Furthermore, our implementation also provides a non-symmetric variation of the group similarity metric outlined in Section 5.2 that measures how well one alert group is represented by another and thus improves robustness against noise alerts in one of the groups.

6.10 Runtime

We analyze the runtime of our approach by measuring the time it takes to process groups, i.e., compute the similarity between a group and all meta-alerts, find the best matching meta-alert, add the group to the knowledge base, and generate the meta-alert. Since our procedure is incremental, the number of meta-alerts and therefore also the number of necessary similarity computations is increasing over time, causing that the processing time per group is also expected to increase. However, due to the fact that most meta-alerts are generated at the beginning and few new meta-alerts are generated over the long run, the runtime should be approximately linear.

In the following, we considered alerts rather than groups to compensate for the fact that larger groups likely require more time to process than smaller groups. Figure 17 therefore shows the cumulative runtimes it took to process alerts for several δ values at thresholds $\theta_{group} = \theta_{alert} = 0.3$ (left) and different thresholds $\theta_{group}, \theta_{alert}$ using $\delta = 0.1$ seconds (right). Note that for $\delta > 0.01$ two large groups are generated by "nikto" on systems cup and onion, each with approximately 20,000

alerts. The plots show that processing these large groups (approx. alerts 0 to 40,000) as well as large amounts of small groups (approx. alerts 40,000 to 60,000) largely follows linear complexity.

6.11 Discussion

The previous sections present empirical results that give insights into the practical application of the proposed alert similarity functions and aggregation strategies of the overall framework. The results assert the functioning of the approach and confirm the fulfillment of all requirements on a domain-independent alert aggregation approach stated in Section 2.1. Requirement (1) is fulfilled, because our framework automatically extracts meta-alerts representing unknown attack scenarios and classifies new alerts based on manually tagged meta-alerts, e.g., after forensic analysis of previously generated meta-alerts. We recognize that sequential-based and case-based methods that rely on manually coded knowledge are capable of modeling only the most distinct features of attacks and thus achieve higher accuracy when classifying attack executions than similarity-based methods, especially when variations of attacks or IDS configurations are considered. However, we argue that this issue generally applies to all similarity-based methods and is compensated by the ability to detect new attacks. We are confident that meta-alerts generated by our approach could ease the process of manual attack specification, since the merging procedure also reduces attributes and values of alerts as well as alert occurrences to typical properties of the attacks.

Our approach combines time-based and attribute-based grouping strategies by contextualizing alerts through temporal proximity and considering all available attributes for similarity computation. This solves the problem of mapping alerts to attacks (cf. Section 1) in alignment with requirement (2). Due to our format-agnostic similarity metrics, requirement (3) is also fulfilled.

To meet requirement (4), we designed our approach as an incremental clustering procedure and avoided over-generalization of meta-alerts by the use of knowledge bases. Queueing strategies (cf. Section 4.1) thereby ensure that the time required to update meta-alerts is not continuously increasing. We recorded the processing times during our evaluations and ascertained that the overall runtime is approximately linear with respect to the number of alerts processed.

Finally, we see our generated meta-alerts as improvements over state-of-the-art that usually involve graphs of attack steps, because they have the same semi-structured format as incoming alerts and are therefore easy to understand for humans and support machine processing. Regarding requirement (5), our approach is thus implementing a combined strategy of meta-alert representation, since single alerts are enriched with mergelists and wildcards, and embedded in sequences.

7 CONCLUSION

In this paper we introduced a novel approach for meta-alert generation based on automatic alert aggregation. Our method is designed for arbitrary formatted alerts and does not require manually crafted attack scenarios. This enables to process alerts from anomaly-based and host-based IDSs that involve heterogeneous alert formats and lack IP information, which is hardly possible using state-of-the-art methods. We presented a similarity metric for semi-structured alerts and three different strategies for similarity computation of alert groups: exact matching, bag-of-alerts matching, and alignment-based matching. Moreover, we proposed techniques for merging multiple alerts into a single representative alert and multiple alert groups into a meta-alert. We outlined an incremental procedure for continuous generation of meta-alerts using aforementioned metrics and techniques that also enables the classification of incoming alerts in online settings.

For our evaluation, we generated alert data sets by forensically analyzing real log data containing traces of multi-step attacks using a signature-based IDS and an anomaly-based IDS. The evaluation shows that our approach is capable of reducing the number of alert groups by around 80% while maintaining a true positive rate of around 80% and a false positive rate of less than

5%. These results suggest that our framework is suitable for application in real-world scenarios, in particular, situations where a large number of similarly configured machines are available and meta-alerts could be generated and used for detection across systems with high precision.

We foresee a number of extensions for future work. As mentioned in the paper, we implemented a function to measure how well one alert group is contained in another in addition to their similarity. Measuring how well a group is represented by a meta-alert could reduce the problem of noise alerts within groups and could even be used to separate alerts of overlapping attack executions into distinct groups. On the other hand, determining how well meta-alerts are represented by groups could allow to automatically recognize and improve incorrectly formed meta-alerts. We did not carry out any evaluations in this direction, since the used data does not contain such artifacts. Furthermore, we explained that group formation with different δ values enables generation of diverse meta-alerts. However, we do not make use of the fact that this also yields a hierarchical structure of groups. It could be interesting to transfer these relationships between groups to meta-alerts in order to improve their precision. Finally, we plan to evaluate our approach and aforementioned extensions on additional data sets with a higher number of web servers and more attack variations.

REFERENCES

[1] Safaa Al-Mamory and Hongli Zhang. 2009. Intrusion detection alarms reduction using root cause analysis and clustering. *Computer Communications* 32, 2 (2009), 419–430.

[2] Taqwa Ahmed Alhaj, Maheyzah Md Siraj, Anazida Zainal, Huwaida Tagelsir Elshoush, and Fatin Elhaj. 2016. Feature selection using information gain for improved structural-based alert correlation. *PloS One* 11 (2016), 1–18.

[3] Faeiz Alserhani. 2016. Alert correlation and aggregation techniques for reduction of security alerts and detection of multistage attack. *International Journal of Advanced Studies in Computers, Science and Engineering* 5, 2 (2016), 1.

[4] Mehdi Bateni, Ahmad Baraani, and Ali Ghorbani. 2013. Using artificial immune system and fuzzy logic for alert correlation. *International Journey of Network Security* 15, 3 (2013), 190–204.

[5] Ingwer Borg and Patrick Groenen. 2005. *Modern Multidimensional Scaling: Theory and Applications*. Springer Science & Business Media.

[6] Frédéric Cuppens and Alexandre Miege. 2002. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 202–215.

[7] Sean Carlisto De Alvarenga, Sylvio Barbon Jr., Rodrigo Sanches Miani, Michel Cukier, and Bruno Bogaz Zarpelão. 2018. Process mining and hierarchical clustering to help intrusion alert visualization. *Computers & Security* 73 (2018), 474–491.

[8] Huwaida Tagelsir Elshoush and Izzeldin Mohamed Osman. 2011. Alert correlation in collaborative intelligent intrusion detection systems - A survey. *Applied Soft Computing* 11, 7 (2011), 4349–4365.

[9] Brian S. Everitt, Sabine Landau, Morven Leese, and Daniel Stahl. 2011. Cluster Analysis 5th ed.

[10] Steffen Haas, Florian Wilkens, and Mathias Fischer. 2019. Efficient attack correlation and identification of attack scenarios based on network-motifs. In *Proceedings of the 38th International Performance Computing and Communications Conference*. IEEE, 1–11.

[11] Alexander Hofmann and Bernhard Sick. 2009. Online intrusion alert aggregation with generative data stream modeling. *IEEE Transactions on Dependable and Secure Computing* 8, 2 (2009), 282–294.

[12] Martin Husák, Milan Čermák, Martin Laštovička, and Jan Vykopal. 2017. Exchanging security events: Which and how many alerts can we aggregate?. In *Proceedings of the Symposium on Integrated Network and Service Management*. IEEE, 604–607.

[13] Martin Husák and Jaroslav Kašpar. 2019. AIDA framework: Real-time correlation and prediction of intrusion detection alerts. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*. 1–8.

[14] Klaus Julisch. 2003. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security* 6, 4 (2003), 443–471.

[15] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. 2019. A framework for cyber threat intelligence extraction from raw log data. In *Proceedings of the International Conference on Big Data*. IEEE, 3200–3209.

[16] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. 2021. Have it your way: Generating customized log data sets with a model-driven simulation testbed. *IEEE Transactions on Reliability* (2021).

[17] Wei Liang, Zuo Chen, Ya Wen, and Weidong Xiao. 2016. An alert fusion method based on grey relation and attribute similarity correlation. *International Journal of Online and Biomedical Engineering* 12, 08 (2016), 25–30.
[18] Jidong Long, Daniel Schwartz, and Sara Stoecklin. 2006. Distinguishing false from true alerts in Snort by data mining patterns of alerts. In *Proceedings of the Defense and Security Symposium*, Vol. 6241. International Society for Optics and Photonics, 99–108.
[19] Dapeng Man, Wu Yang, Wei Wang, and Shichang Xuan. 2012. An alert aggregation algorithm based on iterative self-organization. *Procedia Engineering* 29 (2012), 3033–3038.
[20] Christopher Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*. Cambridge University Press.
[21] Stephen Moskal, Shanchieh Jay Yang, and Michael Kuhl. 2018. Extracting and evaluating similar and unique cyber attack strategies from intrusion alerts. In *Proceedings of the International Conference on Intelligence and Security Informatics*. IEEE, 49–54.
[22] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Computing Surveys* 33, 1 (2001), 31–88.
[23] Julio Navarro, Aline Deruyver, and Pierre Parrend. 2016. Morwilog: An ACO-based system for outlining multi-step attacks. In *Proceedings of the Symposium Series on Computational Intelligence*. IEEE, 1–8.
[24] Julio Navarro, Aline Deruyver, and Pierre Parrend. 2018. A systematic survey on multi-step attack detection. *Computers & Security* 76 (2018), 214–249.
[25] Peng Ning, Yun Cui, and Douglas Reeves. 2002. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 245–254.
[26] Robert Patton, Justin Beaver, Chad Steed, Thomas Potok, and Jim Treadwell. 2011. Hierarchical clustering and visualization of aggregate cyber data. In *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference*. IEEE, 1287–1291.
[27] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 583–595.
[28] Ali Ahmadian Ramaki, Morteza Amini, and Reza Ebrahimi Atani. 2015. RTECA: Real time episode correlation algorithm for multi-step attack scenarios detection. *Computers & Security* 49 (2015), 206–219.
[29] Hanli Ren, Natalia Stakhanova, and Ali Ghorbani. 2010. An online adaptive approach to alert correlation. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 153–172.
[30] Sherif Saad and Issa Traore. 2012. Heterogeneous multi-sensor IDS alerts aggregation using semantic analysis. *Journal of Information Assurance and Security* 7, 2 (2012), 79–88.
[31] Reza Sadoddin and Ali Ghorbani. 2009. An incremental frequent structure mining framework for real-time alert correlation. *Computers & Security* 28, 3–4 (2009), 153–173.
[32] Saeed Salah, Gabriel Maciá-Fernández, and Jesús Díaz-Verdejo. 2013. A model-based survey of alert correlation techniques. *Computer Networks* 57, 5 (2013), 1289–1317.
[33] Riyanat Shittu, Alex Healing, Robert Ghanea-Hercock, Robin Bloomfield, and Muttukrishnan Rajarajan. 2015. Intrusion alert prioritisation and attack detection using post-correlation analysis. *Computers & Security* 50 (2015), 1–15.
[34] Georgios Spathoulas and Sokratis Katsikas. 2013. Enhancing IDS performance through comprehensive alert post-processing. *Computers & Security* 37 (2013), 176–196.
[35] Jiaxuan Sun, Lize Gu, et al. 2020. An efficient alert aggregation method based on conditional rough entropy and knowledge granularity. *Entropy* 22, 3 (2020), 1–23.
[36] Risto Vaarandi and Kārlis Podiņš. 2010. Network IDS alert classification with frequent itemset mining and data clustering. In *Proceedings of the International Conference on Network and Service Management*. IEEE, 451–456.
[37] Alfonso Valdes and Keith Skinner. 2001. Probabilistic alert correlation. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 54–68.
[38] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. 2004. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on Dependable and Secure Computing* 1, 3 (2004), 146–169.
[39] Chih-Hung Wang and Ye-Chen Chiou. 2016. Alert correlation system with automatic extraction of attack strategies by using dynamic feature weights. *International Journal of Computer and Communication Engineering* 5, 1 (2016), 1.
[40] Qiu Hua Zheng, Yi Guang Xuan, and Wei Hua Hu. 2011. An IDS alert aggregation method based on clustering. *Advanced Materials Research* 219 (2011), 156–159.

Received December 2020; revised August 2021; accepted January 2022