Panacea: Automating Attack Classification for Anomaly-based Network Intrusion Detection Systems *

Damiano Bolzoni[1], Sandro Etalle[1,2], Pieter H. Hartel[1]

[1]University of Twente, Enschede, The Netherlands
[2]Eindhoven Technical University, The Netherlands
{damiano.bolzoni,pieter.hartel}@utwente.nl, s.etalle@tue.nl

Abstract. Anomaly-based intrusion detection systems are usually criticized because they lack a classification of attack, thus security teams have to manually inspect any raised alert to classify it. We present a new approach, Panacea, to automatically and systematically classify attacks detected by an anomaly-based network intrusion detection system.
Keywords: attack classification, anomaly-based intrusion detection systems

Today, security teams aim to automate the management of security events, both to optimize their workload and to improve the chance of detecting malicious activities. However, the automation of the security management tasks poses new challenges.

During the years, Intrusion Detection Systems (IDSs) have been continuously improved to detect the latest threats. However, some events that were once considered dangerous have become "not-relevant" (e.g., port scans). Malicious activities conducted by automatic scanners, BOTnets, and so-called script-kiddies can generate a large number of security alerts. Although true positives when detected by an IDS, these kinds of activities cannot normally be considered a serious threat. Most of them attempt to exploit old vulnerabilities that have already been fixed. The fact that a remote automatic scanner is attempting to replicate a 5-year old attack against a now-secure PHP script on a certain web server is no longer important. As a result, the number of security alerts, consisting of irrelevant true positives and false positives, has increased over the years. The most harmful attacks currently consist of several stages. Ning et al. [1] observe that "most intrusions are not isolated, but related as different stages of attacks, with the early stages preparing for the later ones".

A number of techniques to perform alert correlation have been proposed (Cuppens and Ortalo [2], Debar and Wespi [3], Ning and Xu [4] and Valeur

* This research is supported by the research program Sentinels (http://www.sentinels.nl). Sentinels is being financed by Technology Foundation STW, the Netherlands Organization for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs.

et al. [5]), in order to detect attacks at an early stage, or improve false and non-relevant alert rates.

Nowadays, several "security information management" (SIM) tools are widely used by security teams (e.g., the well-known OSSIM [6]). They are used to ease the management of the security infrastructure, as they integrate with heterogeneous security systems, and can perform a number of tasks automatically. Among those tasks, SIM tools automate the alert filtering and correlation. However, for the tasks to be effective, the attacks that trigger alerts must be classified to provide a good deal of information (apart from the usual IP addresses and TCP ports). In fact, by classifying the attack (e.g., buffer overflow, SQL Injection), it is possible to set in a more precise way an action the system has to execute to handle a certain alert. The alert could trigger automatic countermeasures, e.g., either because an early attack stage has been detected or because the attack class is considered to have a great impact on the security, or being forwarded for manual handling or filtered and stored for later analysis (i.e., correlation) and statistics.

Determining the class of an attack is trivial for an alert generated by a signature-based IDS (SBS). Each signature is the result of an analysis of the corresponding attack conducted by experts: the attack class is manually assigned during the signature development process (i.e., the alert class is included in the signature). Thus, usually security teams do not need to further process the alert to assign a class, and they can set precisely a standard action for the system to execute when such an alert is triggered.

Problem When an anomaly-based IDS (ABS) raises an alert, it cannot associate the alert with an attack class. The system detects an anomaly, but it has too little information (e.g., only source and destination IP addresses and TCP ports) to determine the attack class. No automatic or semi-automatic approach is currently available to classify anomaly-based alerts. Thus, any anomaly-based alert must be manually processed to identify the alert class, increasing the workload of security teams. A solution to automate the classification of anomaly-based alerts is to employ some heuristics (e.g., see Robertson et al. [7]) to analyse the ABS alert for features of well-known attacks. Although this approach could lead to good results, it totally relies on the manual implementation of the heuristics (which could be a labour intensive task), and on the expertise of the operator.

The lack of attack classification affects the overall usability of an ABS, because it makes difficult (if not impossible) for security teams both to employ alert correlation techniques and to activate automatic countermeasures for anomaly-based alerts, and in general to integrate an ABS with a SIM tool.

Contribution In this paper we present Panacea, a simple, yet effective, system that uses machine learning techniques to automatically and systematically classify attacks detected by a payload-based ABS (and consequently the generated alerts as well). The basic idea is the following. Attacks that share some common traits, i.e., some byte sequences in their payloads, are usually in the same class. Thus, by extracting byte sequences from an alert payload (triggered by a certain attack), we can compare those sequences to previously collected data with an

appropriate algorithm, find the most similar alert payload, and then infer the attack class from the matching alert payload class.

To the best of our knowledge, Panacea is the first system proposed that:

- automatically classifies attacks detected by an ABS, without using predetermined heuristics;
- does not need manual assistance to classify attacks (with some exceptions to be described in Section 1.1).

Panacea requires a training phase for its engine to build the attack classifier. Once the training phase is completed, Panacea classifies any attack detected by the ABS automatically.

Limitation of the approach Panacea analyses the generated alert payload to build its classification model. Thus, any alert generated by attacks/activities that do not involve a payload (e.g., a port scan or a DDoS) cannot be automatically classified. As most of the harmful attacks inject some data in target systems, we do not see this as a serious limitation. However, Panacea cannot work with an ABS that detects attacks by monitoring network flows. Here we consider only attacks that target networks, however it is possible to extend the approach to include host-based IDSs too.

1 Architecture

Panacea consists of two interacting components: the Alert Information Extractor (AIE) and the Attack Classification Engine (ACE). The AIE receives alerts from the IDS(s), processes the payload, and extracts significant information, outputting alert meta-information. This meta-information is then passed to the ACE that automatically determines the attack class. The classification process goes through two main stages. First, the ACE is trained with several types of alert meta-information to build a classification model. The ACE is fed alert meta-information and the corresponding attack class. The attack class information can be provided in several ways, either manually by an operator or automatically by extracting additional information from the original alert (only when the alert has been raised by an SBS). Secondly, when the training is completed, the ACE is ready to classify new incoming alerts automatically. We now describe each component and the working modes of Panacea in detail. Figure 1 depicts Panacea and its internal components.

1.1 Alert Information Extractor

The first component we examine is the AIE. The extraction of relevant information from alert payloads is a crucial step, as it is the basis for attack class inference. Requirements for this phase are that the extraction function should capture enough features from the original information (i.e., the payload) to distinguish alerts belonging to different classes, and it should be efficient w.r.t. the required memory space. We now describe the analysis techniques we have chosen.

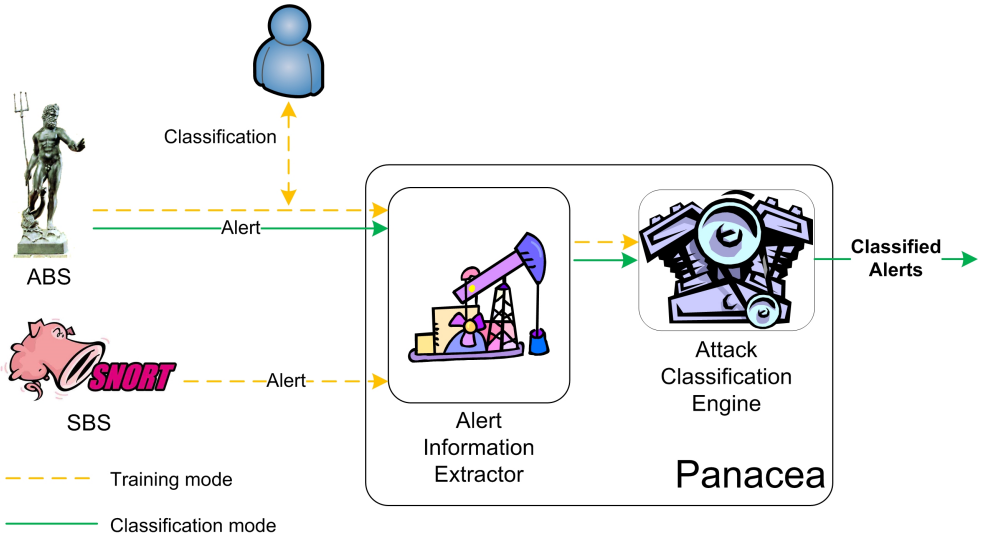


Fig. 1. An overview of the Panacea architecture and the internal components

Extracting and storing relevant information N-gram analysis [8] allows to capture features of data in an efficient way, and it has been used before in the context of computer security to detect attacks (Forrester and Hofmeyr [9], Wang and Stolfo [10]). N-gram analysis is a suitable technique to capture data features also for the problem of attack classification.

As Wang et al. note [11], by using higher order n-grams (i.e., n-grams where $n > 1$) it is possible to capture more data features and to achieve a more precise analysis. One has to consider that the whole feature space size of a higher-order n-gram is 256^n (where n is the n-gram order). The comparison of byte frequency values becomes quickly infeasible, also for values of n such as 3 or 4, because the space needed to store average and standard deviation values for each n-gram grows exponentially (e.g., 640GB would be needed to store 5-grams statistics). Although a frequency-based n-gram analysis may seem to model data distribution accurately, Wang et al. experimentally show that a binary-based n-gram analysis is more precise in the context of network data analysis. In practice, the fact that a certain n-gram has occurred is stored, rather than computing average byte frequency and standard deviation statistics. The reason why the binary approach performs better is that high-order n-grams are more sparse than low-order n-grams, thus it is more difficult to gather accurate byte-frequency statistics as the order increases. This approach has an additional advantage, other than being more precise. Because less information is required, it requires less space in memory, and we can consider higher-order n-grams (such as 5).

Bitmap The ideal data structure to store binary-based n-gram information is a bitmap. A bitmap is a type of memory organization used to store information

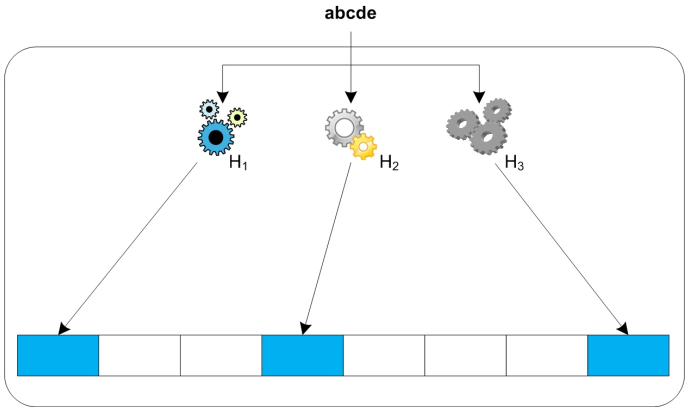
as spatially mapped arrays of bits. In our case, each map entry (a bit) maps a certain n-gram: thus the bitmap size depends on the n-gram order. For 3-grams the bitmap size is 2MB, and for 5-grams the size goes up to 128GB. Here we follow Wang et al. and we use Bloom filters to overcome the space dimension problem.

Bloom filter A Bloom filter [12]) is a method to represent a set of S elements (n-grams in our embodiment) in a smaller space. Formally, a Bloom filter is a pair $\langle b, H \rangle$ where b is a bit map of l bits, initially all set to 0, and H is a set of k independent hash functions $h_1 \dots h_k$. H determines the storage of b in such a way that, given an element s in S: $\forall h_k$, $b_i = 1 \iff h_k(s) \ mod \ l = i$. In other words, for each n-gram s in S, and for each hash function h_k, we compute $h_k(s) \ mod \ l$, and we use the resulting value as index to set to 1 the bit in b corresponding to it. When checking for the presence of a certain element s, the element is considered to be stored in the Bloom filter if and only if: $\forall h_k$, $b_{h_k(s) \ mod \ l} = 1$. A BF with a size of 10KB is sufficiently large to store the alert meta-information resulting from 5-grams analysis. Figure 2 shows an example of insertion in a BF.

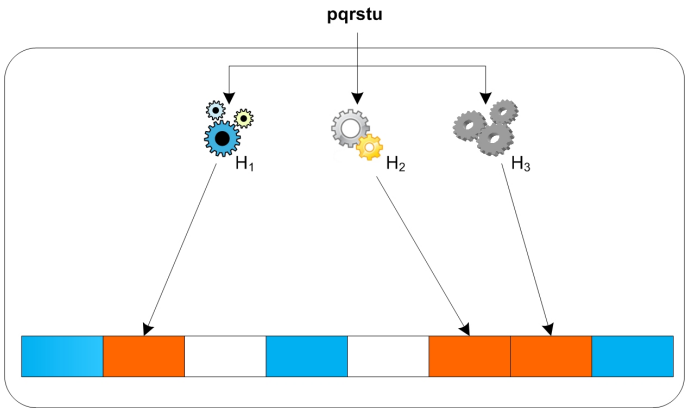
A Bloom filter employs k different hash functions at the same time to decrease the probability of a false positive (the opposite situation, a false negative, cannot occur). False positives occur when all of the bit positions calculated for a given element have been set to 1 when inserting previous elements (as depicted in Figure 3), due to the collisions generated by hash functions. The false positive rate for a given Bloom filter is $(1 - e^{\frac{kn}{l}})^k$, where n is the number of elements already stored.

Operational modes The AIE is also responsible for forwarding the attack class information to the classification engine, when the latter is in training mode. The attack class can be provided either automatically or manually. In case an SBS is deployed next to the ABS and it is monitoring the same data, it is possible to feed the ACE during training both the payload and the attack class of any alert generated by the SBS. We define this the *automatic* mode, since no human operator is required to carry out the attack classification. A human operator can classify the alerts raised by the ABS (in consistent manner with the SBS classification), hence integrating those with the alerts raised by the ABS during the ACE training. We call this the *semi-automatic* mode. The last possible operative mode is the *manual* mode. In this case, *any* alert is manually classified by an operator.

Each mode presents advantages and disadvantages. In automatic mode, the workload is low, but on the other hand the classification accuracy is likely to be low as well. In fact, the SBS and the ABS are likely to detect different attacks, only for those alerts that are raised by both engines. Thus, the classification engine could be trained to classify only a subset of the alerts correctly. The manual mode requires human intervention but it is likely to produce better results, since each alert is consistently classified (the classification that comes out-of-the-box with an SBS could not be suitable). We assume that the alerts



(a) Inserting n-gram “abcde”



(b) Inserting n-gram “pqrst”

Fig. 2. Examples of inserting two different 5-grams. H_1, H_2 and H_3 represent different hash functions.

raised by the SBS and ABS have already been verified and any false positive alert has already been purged (e.g., using ALAC [13] or our ATLANTIDES).

1.2 Attack Classification Engine

The ACE includes the algorithm used to classify attacks. Since we are aware of the attack class information, we consider only *supervised* machine learning algorithms. These algorithms generally achieve better results than unsupervised algorithms (where the algorithm, e.g. K-medoids, deduces classes by measuring inter-data similarity). The classification algorithm must meet several requirements, namely:

– support for multiple classes, as alerts fall in several classes;

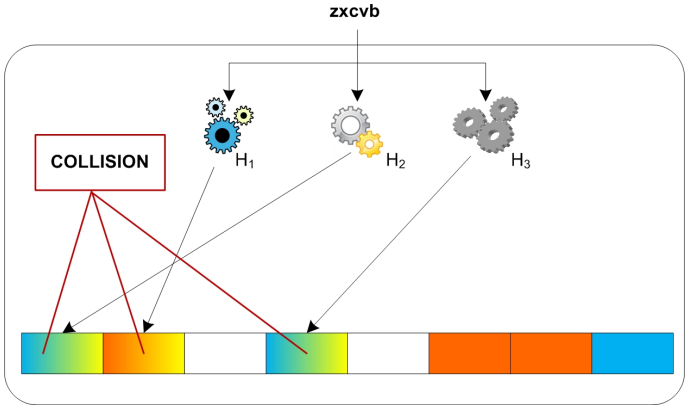


Fig. 3. Example of a false positive. The element "zxcvb" has not been inserted in the Bloom filter. Due to the collisions generated by the hash functions, the test for its presence returns "true".

- classification of high-dimensional data, since each bit of the BF data structure the ACE receives in input is seen as a dimension of analysis;
- fast training phase (the reason for this will be clarified later);
- (optional) estimate classification confidence when in classification phase.

We consider the last requirement optional, as it does not directly influence the quality of the classification, though it is useful to improve the system usability. Confidence measures the likelihood of having a correct classification for a given input. Users can instruct the system to forward any alert whose confidence value is lower than a given threshold for manual classification, hence reducing the probability of misclassification (at the price of an increased workload).

We chose two algorithms for our experiments: (1) Support Vector Machine (SVM) and (2) the RIPPER rule learner. These algorithms implement supervised techniques, their training and classification phases are fast and handle high-dimensional data. Both algorithms perform non-incremental learning. A non-incremental algorithm iterates on samples several times to build the best classification model by minimizing the classification error. The whole training set is then needed at once, and additional samples cannot be incorporated in the classification model unless the training phase is run from scratch. On the other hand, an incremental algorithm can modify the model after the main training phase as new samples become available. An incremental algorithm usually performs worse than a non-incremental algorithm, because the model is not re-built. Thus, a non-incremental algorithm is the best choice to perform an accurate classification. However, because it is highly unlikely that we can collect all alerts for training at once the choice of non-incremental algorithms could be seen as a limitation of our system.

In practice, thanks to the limited BF memory size, we can store a huge number of samples and, by applying a "batch training", we can simulate incremental

learning in non-incremental algorithms. As new training samples become available, we add them to the batch training set and build the classifier using the entire set only when a certain number of samples is reached. Then, the classifier is re-built with the set of "batches" available at that time. Because both SVM and RIPPER are fast in training, there are no computational issues.

We chose SVM and RIPPER, not only because they meet the requirements, but for two additional reasons. First, they yield high-quality classifications. Meyer et al. [14] test the SVM against several other classification algorithms (available from the R project [15]) on real and synthetic data sets. An SVM outperforms competitors in 50% of tests and ranks in the top 3 in 90% of them. RIPPER has been used before in the context of intrusion detection (e.g., on data relative to system calls and network connections [16, 17]) with good results. Secondly, because they approach the classification problem differently (geometric for SVM, and rule-based for RIPPER), the algorithms are supposed to behave heterogeneously in some circumstances. Hence, we can evaluate which algorithm is more suitable in different contexts. We will now provide some detail on the algorithms.

Support Vector Machines (Vapnik and Lerner [18]) is a set of supervised learning methods used for classification. In the original formulation, an SVM is a binary classifier. It uses a non-linear mapping to transform the original training data into a higher dimension. Then, it searches for the linear optimal separating hyperplane, i.e., a plane that separates the samples of one class from another. An SVM uses "support vectors" and "margins" to find the optimal hyperplane, i.e., the plane with the maximum margin. Lets us consider Figure 4. By selecting an appropriate non-linear mapping to a sufficiently high dimension, data samples from two classes can always be separated by a hyperplane. In fact, there are a number of separating hyperplanes. However, we expect the hyperplane with the larger margin to be more accurate at classifying future data samples, because it gives the largest separation "distance" between classes. The margin is calculated by constructing two parallel hyperplanes, one on each side of the hyperplane, and then by "pushing them up against" the data sets. Any data samples that fall on these side hyperplanes are called support vectors.

The original SVM algorithm has been modified to classify non-linear data and to use multiple classes. Boser et al. [19] introduce non-linear data classification by using kernel functions (i.e., non-linear functions). The resulting algorithm is similar, but every dot product is replaced with a non-linear kernel function. Then, the algorithm proceeds to find a maximal separating hyperplane in the transformed space.

To support multiple classes, the problem is reduced to multiple binary subproblems. Given m classes, m classifiers are trained, one for each class. Any test sample is assigned to the class corresponding to the largest positive distance.

RIPPER (Cohen [20]) is a fast and effective rule induction algorithm. RIPPER uses a set of IF-THEN rules. An IF-THEN rule is an expression in the form

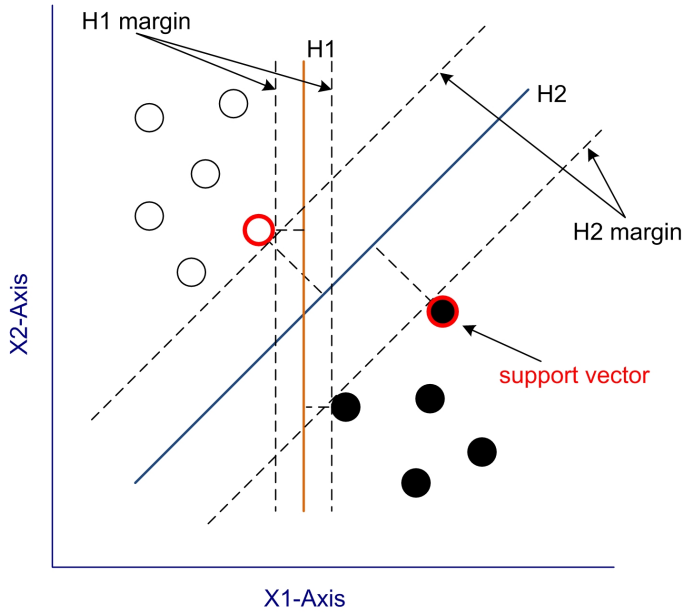


Fig. 4. Hyperplanes in a 2-dimensional space. H1 separates samples sets with a small margin, H2 does that with the maximum margin. The example refers to linearly separable data. The support vectors are shown with a thicker red border.

IF <*condition*> **THEN** <*conclusion*>. The IF-part of a rule is called the rule antecedent. The THEN-part is the rule consequent. The *condition* consists of one or more attribute tests, that are logically ANDed. A test t_i is in the form $t_i = v$ for categorical attributes (where v is a category label) or either $t_i \geq \theta$ or $t_i \leq \theta$ for numerical attributes (where θ is a numerical value). The conclusion contains a class prediction. If, for a given input, the condition (i.e., all of the attribute tests) holds true, then the rule antecedent is satisfied and the corresponding class in the conclusion is returned (the rule is said to "cover" the input). Since RIPPER employs ordered rules, when a match occurs, the algorithm does not evaluate other rules. Some examples of rules are:

$$\textbf{IF } bf[i] = 1 \ AND \ \ldots \ AND \ bf[k] = 1 \ \textbf{THEN } class = cross\text{-}site\ scripting$$
$$\textbf{IF } bf[l] = 1 \ AND \ \ldots \ AND \ bf[m] = 1 \ \ldots \ AND \ bf[n] = 1 \ \textbf{THEN } class = sql\ injection$$

RIPPER builds the rule set for a certain class SC_i as follows. The training data set is split into two sets, a pruning and a growing sets. The classifier is built using these two sets by repeatedly inserting rules starting from an empty rule set (the growing set). The algorithm heuristically adds one condition at a time until the rule has no error rate on the growing set.

RIPPER implements also an optimisation phase, in order to simplify the rule set. For each rule r_i in the rule set, two alternative rules are built; the

replacement of r_i and the revision of r_i. The replacement of r_i is created by growing an empty rule $r_i^{'}$, and then pruning it in order to reduce the error rate of the rule set including $r_i^{'}$ on the pruning data set. The revision of r_i is constructed similarly, but the revision rule is built heuristically by adding one condition at a time to the original r_i rather than to an empty rule. Then the three rules are examined on the pruning data to select the rule with the least error rate.

When multiple classes $C_1 \ldots C_n$ are used, RIPPER sorts classes on a sample frequency basis and induces rules sequentially from the least prevalent class SC_1 to the second most prevalent class SC_{n-1}. The most prevalent class SC_n becomes the *default* class, and no rule is induced for it (thus, in case of a binary classification, RIPPER induces rules for the minority class only).

1.3 Implementation

We have implemented a prototype of Panacea to run our experiments. The prototype is written in Java, since we link to the libraries provided by the Weka platform [21]. Weka is a well-known collection of machine learning algorithms, and it contains an implementation of both SVM and RIPPER. Weka provides also a comprehensive framework to run benchmarks on several data sets under the same testing conditions. The attacks samples generated by network IDSs, in the form of alerts, are stored in a database that the system fetches to extract the alert payload information.

2 Benchmarks

Public data sets for benchmarking IDSs are scarce. It is even more difficult to find a suitable data set to test Panacea, since no research has systematically addressed the problem of (semi)automatically classifying attacks detected by an ABS before. Hence, we have collected three different data sets (referred to as DS_A, DS_B and DS_C, see below for a description of the data sets) to evaluate the accuracy of Panacea. These data sets are used to evaluate the accuracy of Panacea in different scenarios: (1) when working in automatic mode (DS_A), (2) when using an ad hoc taxonomy and the manual mode (DS_B) and (3) when classifying unknown attacks (e.g., generated by two ABSs), having trained the system with alerts from known attacks (DS_B and DS_C).

In the literature there are several taxonomies and classifications of security events. For instance, Howard [22], Hansman and Hunt [23], and the well-known taxonomy used in the DARPA 1998 [24] and 1999 [25] data sets. Only the latter classification has been used in practice (in spite of its course granularity, as it contains only four classes which are unsuitable to classify modern attacks). A modern IDS employs its own attack classification, which is usually non-standard and – according to Andersson et al. [26] – difficult to translate into a standardized classification, e.g., the XML-based Intrusion Detection Message Exchange Format [27]. In our experiments, we use the Snort classification for benchmarks

with DS_A (see [28] for a detailed taxonomy) and the Web Application Security Consortium Threat Classification [29] for benchmarks with DS_B and DS_C.

To evaluate the accuracy of the classification model, we use two approaches. For test (1) and (2), we employ cross-validation. In cross-validation, samples are partitioned into sub-sets. The analysis is first performed on a single sub-set, while the other sub-set(s) are retained to validate the initial analysis. In k-fold cross-validation, the samples are partitioned into k sub-sets. A single sub-set is retained as the validation data for testing the model, and the remaining $k-1$ sub-sets are used as training data to build the model. The process is repeated k times (the "folds"), using each of the k sets exactly once to validate the model. Usually the k fold results are combined (e.g., averaged) to generate a single estimation. The advantage of this method is that all of the samples are used for both training and validation, and each sample is used for validation exactly once. We use 10 folds in our experiments, which is a standard value, used in the Weka testing environment too.

For test 3), we use one of DS_B and DS_C for training and the other for testing. The accuracy is evaluated by counting the number of correctly classified attacks.

Attack Class	**Description**	**# of samples**
attempted-recon*	Attempted information leak	1379
web-application-attack*	Web application attack	1032
web-application-activity*	Access to a potentially vulnerable web application	599
unknown	Unknown traffic	66
attempted-user*	Attempted user privilege gain	45
misc-attack	Miscellaneous attack	44
attempted-admin	Attempted administrator privilege gain	32
attempted-dos	Attempted Denial of Service	14
bad-unknown	Potentially bad traffic	13

Table 1. DS_A (alerts raised by Snort): attack classes and samples. It is not surprising that web-related attacks account for more than 50%, since most Snort signatures address web vulnerabilities. * marks classes that contain web-related attacks.

DS_A contains alerts raised by Snort (see Table 1 for alert figures). To collect the largest number of alerts possible, we have used several tools to automatically inject attack payloads (Nessus and a proprietary vulnerability assessment tool). Attacks have been directed against a system running some virtual machines with both Linux- and Windows-based installations, which expose several services (e.g., web server, DBMS, web proxy, SMTP and SSH). We collected more than 3200 alerts in total, classified in 14 different (Snort) attack classes. However, some classes have few alerts, thus we select only classes with at least 10 alerts. This data set (and DS_B as well) is synthetic. We do not see this as a limitation since the alerts cover multiple classes and trigger a large number of different

signatures. We test how the system behaves in automatic mode, the whole set being generated by Snort.

DS_B contains a set of more than 1400 Snort alerts related to web attacks (Table 2 provides alert details). To generate this data set, we have used Nessus [30], Nikto [31] (a web vulnerability scanner), and we have manually injected attack payloads collected from the well-known site Milw0rm, that hosts a large collection of web exploits [32]. The attack classification has been performed manually (manual mode), since Snort does not provide a fine-grained classification of web-related attacks (alerts are allocated to different classes with other alerts, see Table 1). Attacks have been classified according to the Web Application Security Consortium Threat Classification [29].

Attack Class	**# samples**
Path Traversal	931
Cross-site Scripting	399
SQL Injection	73
Buffer Overflow	8

Table 2. DS_B: attack classes and samples. Attacks have been classified according to the Web Application Security Consortium Threat Classification.

DS_C is a collection of alerts generated over a period of 2 weeks by two ABSs, i.e., our POSEIDON [33] and Sphinx [34]. We recorded network traffic directed to a main web server of the university network, and did not inject any attack. Afterwards, we processed this data with POSEIDON and Sphinx to generate alerts. The inspection of alerts and the classification of attacks has been performed manually (using the same taxonomy we apply for DS_B). The data set consists of a set of 100 alerts, and Table 3 reports attack details.

Attack Class	**# samples**
Path Traversal	53
Cross-site Scripting	27
SQL Injection	16
Buffer Overflow	4

Table 3. DS_C: attack classes and samples. Attacks have been classified according to the Web Application Security Consortium Threat Classification.

Tests with DS_A We use DS_A to validate the general effectiveness of our approach. There are three factors which influence the classification accuracy, namely: (1) the number of alerts processed during training, (2) the length of n-grams used, and (3) the classification algorithm selected. This preliminary test

aims to identify which parameter combination(s) results in the most accurate classification.

Testing methodology We proceed with a 3-step approach. First, we want to identify an adequate number of samples required for training: in fact, a too low number of samples could generate an inaccurate classification. On the other hand, while it is generally a good idea to have as many training samples as possible, after some point the benefit from adding additional information could become negligible. Secondly, we want to identify the best n-gram length. Short n-grams are likely to be shared among many attack payloads, and the attack diversification would be poor (i.e., a number of different attacks contains the same n-grams). On the other hand, long n-grams are unlikely to be common among attack payloads, hence it would be difficult to predict a class for a new attack that does not share a sufficient number of long n-grams. Finally, we analyse how the classification algorithms work by analysing the overall classification accuracy (i.e., considering all of the attack classes) and the per-class accuracy. The two algorithms approach the classification problem in two totally different ways, and each of them could be performing better under different circumstances.

To avoid bias by choosing a specific attack, we randomly select alerts in the sub-sets. In fact, by selecting alerts for training in the same order they have been generated (as opposed to random), we could end up with few (or no) samples in certain classes, hence influencing the accuracy rate (i.e., a too good, or bad, value). To enforce randomness, we also run several trials (five) with different sub-sets and calculate the average accuracy rate. Table 4 reports benchmark results (the percentage of correctly classified attacks) for SVM and RIPPER.

	SVM n-gram length				**RIPPER n-gram length**			
# samples	1	2	3	4	1	2	3	4
1000	62.6%	76.8%	**77.3%**	76.7%	66.1%	75.9%	**76.2%**	75.7%
2000	65.9%	78.6%	**78.9%**	77.7%	69.4%	76.7%	**76.9%**	76.4%
3000	66.3%	79.4%	**79.6%**	78.6%	72.7%	77.2%	**77.5%**	76.9%

Table 4. Test results on DS_A with SVM and RIPPER. We report the average percentage of correctly classified attacks of five trials. As the number of samples in the testing sub-set increases, the overall effectiveness increases as well. Longer n-grams generally produce better results, up to length 3. SVM performs better than RIPPER by a narrow margin.

Discussion Tests with DS_A indicate that the approach is effective in classifying attacks. As the number of training samples increases, accuracy increases as well for both algorithms. Also the n-gram length directly influences the classification. The number of correctly classified attacks increases as n-grams get longer, up to 3-grams. N-grams of length 4 produce a slightly worse classification, and the same

happens for 1-grams (which achieve the worst percentages). SVM and RIPPER present similar accuracy rates on 3-grams, with the former being slightly better. However, if we perform an analysis based on per-class accuracy (see Table 5), we observe that, although both classification algorithms score high on accuracy level for the three most populated classes, RIPPER is far more precise than SVM (in once case, the "web-application-activity" class, by nearly 15%).

When we look at the overall accuracy rate, averaged among the 9 classes, for DS_A, SVM performs better because of the classes with few alerts. If we zoom into the classes with a significant number of samples, we observe an opposite behaviour. This means that, with a high number of samples, RIPPER performs better than SVM.

In Table 5, a sub-set with fewer samples seems to achieve better results (although percentages differ by a narrow margin), when considering the same algorithm. This happens for SVM once then using 1000 training samples ("attempted-recon" class) and twice when using 2000 training samples ("web-application-attack" and "web-application-activity" classes). When using 2000 training samples, RIPPER performs best in the "web-application-activity" class. The reason for this is that alerts in the sub-sets are randomly chosen, thus a class could have a different number of samples among trials.

	SVM # of samples			**RIPPER # of samples**		
Attack Class	1000	2000	3000	1000	2000	3000
attempted-recon	**90.9%**	90.5%	90.7%	90.4%	93.9%	**94.0%**
web-application-attack	79.8%	**89.0%**	88.8%	97.4%	98.8%	**99.1%**
web-application-activity	80.8%	**81.2%**	80.9%	93.7%	**96.1%**	95.8%

Table 5. Per-class detailed results on DS_A, using 3-grams. We report the average percentage of correctly classified attacks of five trials. RIPPER performs better than SVM in classifying all attacks, .

Tests with DS_B DS_B is used to validate the manual mode and the use of an ad hoc classification. To perform the benchmarks, we use the same n-gram length that achieves the best results in the previous test. Table 6 details our findings for SVM and RIPPER.

Discussion The test results on DS_B show that Panacea is effective also when using a user-defined classification, regardless of the classification algorithm is chosen. Regarding accuracy rates, RIPPER shows a higher accuracy for most classes, although SVM scores the best classification rate (by a narrow margin).

Only the "buffer overflow" class has a low classification rate. Both algorithms have wrongly classified most of buffer overflow attacks in the "path traversal" class. This is because (1) the number of samples is lower than for the other classes, which are at least 10 times more numerous, and 2) a number of the

Attack Class	SVM	RIPPER
Path Traversal	98.6%	**99.1%**
Cross-site Scripting	97.5%	**98.4%**
SQL Injection	**97.6%**	96.2%
Buffer Overflow	37.5%	37.5%
Percentage of total attacks correctly classified	**98.0%**	97.7%

Table 6. Test details (percentage of correctly classified attacks) on DS_B with SVM and RIPPER. RIPPER achieves better accuracy rates for the two most numerous classes, although by a narrow margin. We observe the same trend for the rates reported in Table 5.

path traversal attacks present some byte encoding that resembles byte values typically used by some buffer overflow attack vectors. In the case of RIPPER, the "path traversal" class has the highest number of samples, hence no rule is induced for it and any non-matching samples is classified in this class.

Tests with DS_C An ABS is supposed to detect previously-unknown attacks, for which no signature is available yet. Hence, we need to test how Panacea behaves when the training is accomplished using mostly alerts generated by an SBS but afterwards Panacea processes alerts generated by an ABS. For this final test we simulate the following scenario. A user has manually classified alerts generated by an SBS during the training phase (DS_B) and she uses the resulting model to classify unknown attacks, detected by an ABS (POSEIDON). Since we collected few buffer overflow attacks, we use the Sploit framework [35] to mutate some of the original attack payloads and increase the number of samples for this class, introducing attack diversity at the same time. Thus, we obtain additional training samples with a different payload. Table 7 shows the percentage of correctly classified attacks by SVM and RIPPER. For the buffer overflow attacks, we report accuracy values for the original training set (i.e. representing real traffic) and the "enlarged" training set (in brackets).

Attack Class	SVM	RIPPER
Path Traversal	**98.1%**	94.4%
Cross-site Scripting	**92.6%**	88.9%
SQL Injection	**100.0%**	87.5%
Buffer Overflow	**50.0%** (75.0%)	25.0% (50.0%)
Percentage of total attacks correctly classified	**92.0%** (93.0%)	89.0% (90.0%)

Table 7. Test details (percentage of correctly classified attacks) on DS_C with SVM and RIPPER. SVM perform better than RIPPER in classifying any attack class. For the "buffer overflow" class and the percentage of total attacks correctly classified we report (in brackets) the accuracy rates when Panacea is trained with additional samples generated using the Sploit framework.

Discussion Tests on DS_C show that the SVM performs better than RIPPER when classifying attack instances that have not been observed before. The accuracy rate for the "buffer overflow" class is the lowest, and most of the misclassified attacks have been classified in the "path traversal" class (see the discussion of benchmarks for DS_B). However, with a higher number of training samples (generated by using Sploit), the accuracy rate increases w.r.t. previous tests. This suggest that, with a sufficient number of training samples, Panacea achieves high accuracy rates.

2.1 Summary of benchmark results

From the benchmarks results, we can draw some conclusions after having observed the following trends:

- the classification accuracy is always higher than 75%
- SVM performs better than RIPPER when considering the classification accuracy for all classes, when not all of them have more than 50-60 samples (DS_A, DS_B and DS_C)
- RIPPER performs better than SVM when the class has a good deal of training samples, i.e., at least 60-70 in our experiments (DS_A and DS_B)
- SVM performs better than RIPPER when the class presents high diversity and attacks to classify have not been observed during training (DS_C)

We can conclude that SVM works better when few alerts are available for training and when attack diversity is high, i.e., the training alert samples differ from the alerts received when in classification phase. On the other hand, RIPPER shows to be more accurate when trained with a high number of alerts.

Evaluating confidence However good Panacea is, the system is not error-free. The consequences of a misclassification can have a direct impact on the overall security. Think of a buffer overflow attack, for which usually countermeasures must take place immediately (because of the possible consequences), that is misclassified as a path traversal attack, for which the activation of countermeasures can be delayed (e.g., after other actions taken by the attacker). This event occurs often in our benchmarks when the system selects the wrong class. Both SVM and RIPPER can generate a classification confidence value for each attack. This value can be used to evaluate the accuracy of the classification. The lower the classification value is (in a range from 0.0 to 1.0), the more likely the classification is wrong (see Table 8 for average confidence values for DS_C).

The confidence value can be taken into consideration to detect possible misclassification. Users can set a minimum confidence value (e.g., 0.5). Any alert with a lower confidence value is forwarded to a human operator for manual classification. With this additional check, we are able to increase the percentage of total attacks correctly classified up to 95% for SVM and 94% for RIPPER (when using the standard training set, without additional training samples generated with Sploit). The additional workload involves also the manual classification of

	SVM	RIPPER
Average confidence value for correctly classified attacks	0.75	0.62
Average confidence value for misclassified attacks	0.37	0.43
Percentage of total attacks correctly classified without confidence evaluation	92.0%	89.0%
Percentage of total attacks correctly classified with confidence evaluation	95.0%	94.0%
# of alerts forwarded for manual classification	10/100	13/100
# of forwarded attacks that were actually wrongly classified	3/10	5/13
# of forwarded attacks that were actually correctly classified	7/10	8/13

Table 8. Effects of confidence evaluation for DS_C, when Panacea is trained with the standard DS_B. When considering the classification confidence to forward alerts for manual classification, the human operator classification increases by 3% and 5% the overall accuracy rate by inspecting 10 and 13 alerts, out of 100, when Panacea uses SVM and RIPPER respectively.

alerts which have been correctly classified by the system but whose confidence value is lower than the set threshold. However, less than 10 alerts (out of 100) have been forwarded for manual classification when this action was not needed. Table 8 reports the details regarding the evaluation of the confidence value.

2.2 Usability in Panacea

Panacea aims not only to provide automatic attack classification for an ABS, but to improve usability as well. In automatic mode, Panacea performs an accurate classification (more than 75% of correctly classified attacks). In semi-automatic and manual modes, users take actively part in the classification process: however, users are requested to provide a limited input (i.e., a class label). Panacea classifies attacks systematically and automates (1) the extraction of relevant information used to distinguish an attack class from another and (2) the update of the classification model. These tasks are usually left to the user experience and knowledge, thus they are not said to be neither error-free nor comprehensive. Table 9 reports actions that users have to take with and without the support of Panacea.

	User actions	
	Without Panacea	**With Panacea**
DS_A	Classify any alert	No action to take
DS_B	Classify any alert	Classify alerts used during training
DS_C	Classify any alert	No action to take (alerts have been previously classified)

Table 9. Actions that users have to take with or without Panacea w.r.t. alert classification for each data set we use during benchmarks.

3 Related work

Although the lack of attack classification is a well-known issue in the field of anomaly-based intrusion detection, little research has been done on this topic.

Robertson et al. [7] suggest to use some heuristics to infer the class of (web-based) attacks. This approach has several drawbacks. Users have to generate heuristics (e.g., regular expressions) to identify attack classes. They have to enumerate all of the possible attack variants, and update the heuristics each time a new attack variation is detected. This is a time consuming task. Panacea can operate in an automatic way, by extracting attack information from any SBS, or employ an *ad-hoc* classification, with the user providing only the attack class.

Wang and Stolfo [10] use a "Z-String" to distribute among other ABSs attack payloads to enhance detection. A Z-String contains the information resulting from the n-gram analysis of the attack payload. Once a certain payload has been flagged as malicious, the corresponding Z-String can be distributed to other IDSs to detect the attack also, and stop it at an early stage (think of a worm). If some traffic matches a certain Z-String, that data is likely to be a real attack. Although a Z-String is not used for attack classification, by attaching a class label it would be possible to classify each attack. However, this approach is not systematic, as each attack that does not exactly match any Z-String would have to be manually classified. A Z-String is based on a frequency-based n-gram analysis, thus an exact match could be difficult to achieve. On the other hand, Panacea applies a systematic classification using the more precise binary-based n-gram analysis. Panacea can also use as a source of information the alerts generated by an SBS, and not only by an ABS.

4 Conclusion

In this chapter we present Panacea, a system that automatically and systematically classifies attacks generated by a payload-based ABS (and consequently the generated alerts). Panacea extracts information from alerts during a training phase, then predicts the attack class for new alerts. The alerts used to train the classification engine can be generated by an SBS as well as an ABS. In the former case, no manual intervention is requested (the system operates in automatic mode), as Panacea automatically extracts the attack class from the alert. In the latter case, the user is required to provide the attack class for each alert used to train the classification engine.

Panacea improves the usability and makes it possible to integrate anomaly-based with signature-based IDSs, for instance by using security information management tools. Benchmarks show that the approach is effective in classifying attacks, even those that have not been detected before (and not used for training). Although Panacea works in an automatic way, users can employ ad-hoc classifications, and even manually tune the engine for more precise classifications.

Future work Panacea can use different algorithms to classify alerts. The benchmarks with SVM and RIPPER, which approach the classification problem in two different ways, show that each algorithm has its strong points, depending on the circumstances. A possible extension is to use a cascade of SVM and RIPPER. We would then use SVM for early classification (when the number of samples is low, and when RIPPER does not perform well), then, when the number of alerts increases, we can train RIPPER, thanks to the batch training mode, and use it for classification as well (RIPPER performs better than SVM when the number of training samples is high). By applying a voting schema to the classification results provided by both algorithms for a given alert (for instance by considering the confidence value and the number of training samples in a certain class), we could be able to increase the overall accuracy.

References

1. Ning, P., Cui, Y., Reeves, D.: Constructing attack scenarios trough correlation of intrusion alerts. In: CCS '02: Proc. 9th ACM Conference on Computer and Communication Security, ACM Press (2002) 245–254
2. Cuppens, F., Ortalo, R.: LAMBDA: A Language to Model a Database for Detection of Attacks. In: RAID '00: Proc. 3rd International Symposium on Recent Advances in Intrusion Detection, Springer (2000) 197–216
3. Debar, H., Wespi, A.: Aggregation and Correlation of Intrusion-Detection Alerts. In: RAID '00: Proc. 4th International Symposium on Recent Advances in Intrusion Detection, Springer (2001) 85–103
4. Ning, P., Xu, D.: Learning attack strategies from intrusion alerts. In: CCS '03: Proc. 10th ACM conference on Computer and Communications Security, ACM Press (2003) 200–209
5. Valeur, F., Vigna, G., Kruegel, C., Kremmerer, R.: A comprehensive approach to intrusion detection alert correlation. IEEE Trans. Dependable Secur. Comput. **1**(3) (2004) 146–169
6. AlienVault: Open Source Security Information Management (OSSIM) `http://www.ossim.net`.
7. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.: Using generalization and characterization techniques in the anomaly-based detection of web attacks. In: NDSS '06: Proc. 13th ISOC Symposium on Network and Distributed Systems Security. (2006)
8. Damashek, M.: Gauging similarity with n-grams: Language-independent categorization of text. Science **267**(5199) (1995) 843–848
9. Forrest, S., Hofmeyr, S.: A Sense of Self for Unix Processes. In: S&P '96: Proc. 17th IEEE Symposium on Security and Privacy, IEEE Computer Society Press (2002) 120–128
10. Wang, K., Stolfo, S.: Anomalous Payload-Based Network Intrusion Detection. In: RAID '04: Proc. 7th Symposium on Recent Advances in Intrusion Detection. Volume 3224 of LNCS., Springer (2004) 203–222
11. Wang, K., Parekh, J., Stolfo, S.: Anagram: a Content Anomaly Detector Resistant to Mimicry Attack. In: RAID '06: Proc. 9th International Symposium on Recent Advances in Intrusion Detection. Volume 4219 of LNCS., Springer (2006) 226–248
12. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7) (1970) 422–426

13. Pietraszek, T.: Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In: RAID '04: Proc. 7th Symposium on Recent Advances in Intrusion Detection. Volume 3224 of LNCS., Springer (2004) 102–124
14. Meyer, D., Leisch, F., Hornik, K.: The support vector machine under test. Neurocomputing **55**(1-2) (2003) 169–186
15. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. `http://www.R-project.org`.
16. Lee, W.: A data mining framework for constructing features and models for intrusion detection systems. PhD thesis, Columbia University, New York, NY, USA (1999)
17. Lee, W., Fan, W., Miller, M., Stolfo, S., Zadok, E.: Toward cost-sensitive modeling for intrusion detection and response. Journal of Computer Security **10**(1-2) (2002) 5–22
18. Vapnik, V., Lerner, A.: Pattern recognition using generalized portrait method. Automation and Remote Control **24** (1963)
19. Boser, B., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Proc. 5th Annual ACM Workshop on Computational Learning Theory, ACM Press (1992) 144–152
20. Cohen, W.: Fast effective rule induction. In: Proc. 12th International Conference on Machine Learning, Morgan Kaufmann (1995) 115–123
21. : The University of Waikato. Weka 3: Data Mining Software in Java `http://www.cs.waikato.ac.nz/ml/weka/`.
22. Howard, J.: An analysis of security incidents on the Internet 1989-1995. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1998)
23. Hansman, S., Hunt, R.: A taxonomy of network and computer attacks. Computers & Security **24**(1) (2004) 31–43
24. Lippmann, R., Cunningham, R., Fried, D., Garfinkel, S., Gorton, A., Graf, I., Kendall, K., McClung, D., Weber, D., Webster, S., D.W̃yschogrod, M.Z.: The 1998 DARPA/AFRL off-line intrusion detection evaluation. In: RAID '98: Proc. 1st International Workshop on the Recent Advances in Intrusion Detection. (1998)
25. Lippmann, R., Haines, J., Fried, D., Korba, J., Das, K.: The 1999 DARPA off-line intrusion detection evaluation. Computer Networks: The International Journal of Computer and Telecommunications Networking **34**(4) (2000) 579–595
26. Andersson, D., Fong, M., Valdes, A.: Heterogeneous Sensor Correlation: A Case Study of Live Traffic Analysis (2002)
27. Debar, H., Curry, D., Feinstein, B.: The intrusion detection message exchange format (IDMEF). RFC 4765 (2007)
28. Team, S.: Snort user manual `http://www.snort.org/docs/snort_htmanuals/htmanual_2832/node220.html`.
29. Web Application Security Consortium: Web Security Threat Classification `http://www.webappsec.org/projects/threat/`.
30. Security, T.N.: Nessus Vulnerabilty Scanner `http://www.nessus.org/`.
31. CIRT.net: Nikto web scanner `http://www.cirt.net/nikto2`.
32. Milw0rm: `http://milw0rm.com`.
33. Bolzoni, D., Zambon, E., Etalle, S., Hartel, P.: POSEIDON: a 2-tier Anomaly-based Network Intrusion Detection System. In: IWIA '06: Proc. 4th IEEE International Workshop on Information Assurance, IEEE Computer Society Press (2006) 144–156
34. Bolzoni, D., Etalle, S.: Boosting Web Intrusion Detection Systems by Inferring Positive Signatures. In: Confederated International Conferences On the Move to

Meaningful Internet Systems (OTM '08). Volume 5332 of LNCS., Springer (2008) 938–955
35. Vigna, G., Robertson, W., Balzarotti, D.: Testing network-based intrusion detection signatures using mutant exploits. In: CCS '04: Proc. 11th ACM Conference on Computer and Communications Security, ACM Press (2004) 21–30