Nebula – Generating Syntactical Network Intrusion Signatures

Tillmann Werner, Christoph Fuchs, Elmar Gerhards-Padilla, Peter Martini
University of Bonn, Roemerstrasse 164, 53117 Bonn, Germany
{werner,cf,padilla,martini}@cs.uni-bonn.de

Abstract

Signature-based intrusion detection is a state-of-the-art technology for identifying malicious activity in networks. However, attack trends change very fast nowadays, making it impossible to keep up with manual signature engineering. This paper describes a novel concept for automatic signature generation based on efficient autonomous attack classification. Signatures are constructed for each class from syntactical commonalities and go beyond a single, contiguous substring. Each part of a signature is combined with positional information, which drastically improves signature accuracy and matching performance. We argue that a general description of zero-day attacks is immanently restricted to syntactical features and outline how valid signatures for novel real-world attacks were successfully generated.

1. Introduction

Intrusion detection systems play an important role in protecting IT infrastructures against attacks: They enable networks to identify or even block malicious traffic. Although recent results show a promising progress in the field of anomaly-based intrusion detection [3], the major part of production setups relies on *misuse detection* by matching network traffic against known attack patterns [12], generally referred to as *signatures*. A signature is a formal description of distinct attack features. In this paper, the term *intrusion detection system* (IDS) refers to a misuse detection system that uses a set of signatures to identify attacks in a network.

Misuse detection has the immanent conceptual property of being reactive: An attack can only be detected after it occurred or, at best, while it is occurring. This implies that unknown attacks are likely to be missed if they do not match a general signature. Constructing a signature usually requires detailed knowledge about the attacks. But as new attacks are executed automatically and area-wide, sometimes within hours after a new vulnerability has been published, it is not possible to keep up with manually engineered signatures. As a solution we suggest a methodology for generating signatures from attack traces automatically.

The rest of this paper is organized as follows: The next section introduces a general methodology for automated signature generation. Section 3 proposes a specific realization of a generation algorithm, implemented in the *nebula* system, which was designed to perform well even for large input sets. We evaluate the quality of signatures generated by nebula in section 4. After comparing work related to our approach with the proposed system in section 5, some real-world applications and scenarios are discussed in section 6. We conclude our work with a final discussion of nebula's strenghts and advantages in section 7.

2. Methodology

An intrusion signature describes the distinctive features of a class of attacks. While a signature should be *generic* to catch different variants of an attack, it should also be *specific* at the same time to prevent false positives [16]. These two goals are contrary, and a signature generation system has to perform an adequate classification to find a good compromise. The classification step (also referred to as *clustering*) is one of the two critical parts of a signature generation algorithm. The second important task is to extract relevant features from each class.

There are no guidelines that must be fulfilled by a class' features. For instance, an attack family might contain all attempts to exploit a particular vulnerablility, in which case its features are related to the security hole. Or alternatively, a class might cover attacks with more general characteristics, e.g., traces that contain specific shellcode sequences. However, it is important to understand that a general signature generation system is restricted to syntactical features: Although it would be theoretically possible to include semantic knowledge in a signature, this kind of information is usually not automatically obtainable for previously unknown attacks.

The following formalization is a basis for a general definition of syntactic features. Each input is interpreted as a byte stream (string) $\in \Sigma^+, \ \Sigma = \{0,1\}^8$. Based on this, a generic pattern can be defined:

Definition 1. A *pattern* $\mathcal{P} := \{f_1, \ldots, f_n\}$ is a set of features $f_i \in \Sigma^+$ which assigns any string x either to the set $A := \{x \mid \forall f \in \mathcal{P} : f \text{ is substring of } x\}$ or to the set $\overline{A} = \Sigma^* \backslash A$. $\mathcal{P}$ is called a *pattern for* A.

More intuitively, $\mathcal{P}$ is a set of substrings characteristic for a certain attack type. However, such a pattern is too general as it does not contain any positional feature requirements. As this kind of information is available in the input and can easily be extracted during the generation process, it should be used to build a more specific description:

Definition 2. A *signature* $\sigma := \{s_1, \ldots, s_n\}$ is a set of triples $s_i := (f_i, k_i, l_i)$ which assigns any string x either to the set A or to the set $\overline{A}$. The set $\{f_1, \ldots, f_n\}$ is a pattern for A; $k_i, l_i \in \mathbb{N}$ denote the minimium and maximum possible offset of the feature f_i in a byte stream $\in A$.

With this signature definition it is possible to generate attack descriptions based solely on syntactical information.

2.1. Input Classification

Given a set of attack traces, the first step is to find a classification that groups similar inputs together. As there are many different attack types which generally do not share syntactical commonalities, it is not reasonable to construct a single signature that covers the whole set. Such a signature would not only break the claim for being specific, it doesn't even exist in cases where the feature sets for different attacks are disjoint. Instead, one signature should be constructed for each cluster where the granularity of the generated signatures depends on the degree of commonality, which is used as classification criterion.

How can the degree of commonality of two attacks be expressed? A similarity measuring scheme must be limited to syntactical information, otherwise it would break the generality of the overall methodology. Information theory offers several ways to check and express the similarity of two strings. For instance, the amount of information stored in a string is expressed by its *Entropy* or *Kolmogorov Complexity*. Other techniques rate the similarity of two strings by comparing *N-Gram frequencies* or by computing their *Edit Distance*. Another idea is to calculate and compare hashes that conserve the similarity of the original inputs [13]. All these approaches have been successfully used for semantic-independent input classification (e.g., [1, 2, 3]). The clustering can be performed independently from the specific simliarity measure s as long as the following requirements are met:

- $\forall a, b \in \Sigma^+ : s(a, b) \in [0, 1] \subset \mathbb{R}$
- a, b are considered equal $\Leftrightarrow s(a, b) = 1$
- a, b are considered maximal dissimilar $\Leftrightarrow s(a, b) = 0$

In practice it is often beneficial to use dissimilarities instead of similarities. A measure for the dissimilarity of two input strings can be defined as $d(a, b) := 1 - s(a, b)$. d can be interpreted as a distance function and often satisfies the metric conditions (positive definiteness, symmetry, and triangle inequality). A clustering can be calculated from the distance of the inputs as follows: We assume that all inputs are given and that a complete distance graph is calculated. An edge between each two points in the graph is labeled with the distance of the corresponding inputs. By removing all edges with a distance greater than a certain threshold value, the graph splits into several partitions (the remaining connected components).

Whenever a new input is processed, the distance to all previously clustered elements is calculated to determine its nearest neighbor (we will, however, see that this step can often be aborted early if certain conditions are met). If the minimum distance is above the threshold, the new input is stored as an *outlier*, otherwise it gets assigned to the neighbor's cluster. Thus the classification step's complexity is linear in the number of already clustered inputs in the worst-case. However, several optimizations are possible in practice that usually result in a much better average performance. Such implementation details will be discussed in section 3.1.

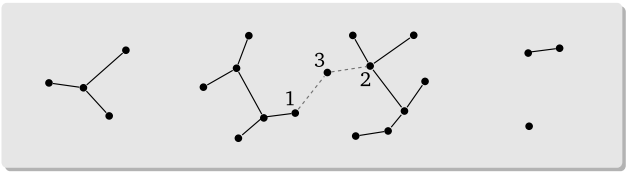

Figure 1. A distance graph's components

During the classification of a new input, one of the following three different cases can occur: First, the distance to all other elements may be above the threshold value. Then the new string is categorized as an outlier. In the second case, the classification criterion is met for an existing outlier, which then initializes a new cluster together with the new input. In the remaining case there is (at least) one element in an existing cluster with a distance below the threshold, and the new input is assigned to that cluster. If there are more clusters containing elements with an adequate distance value, they are merged into one cluster. This situation is depicted in figure 1 where the classification of node 3 merges two clusters into one as it has a sufficiently small distance to the nodes 1 and 2.

The graph-based approach has two important advantages. One is that two inputs stay combined once they are grouped together in a cluster. This is a fundamental requirement for an efficient online algorithm where input is processed piece-wise in a serial fashion and recalculations must

be avoided. Second, the method offers a tradeoff between generality and accuracy: Few signatures must be more general to cover the input set but might perform faster. On the other hand, more signatures represent a more specific description of the different attack types and are thus not as prone to false positives. The size and number of clusters is controllable via the distance threshold that can be varied in order to tune the classification result.

2.2. Information Extraction

The elements within a cluster are considered as members of an attack class and carry the information that is needed to construct a signature. Our algorithm composes a signature from *common substrings* of a cluster's elements.

Definition 3. Let $x = x_1 \dots x_n \in \Sigma^+$ be a string of length $n \in \mathbb{N}$. A *substring* of x is a string $x' = x_i \dots x_{i+j}$ where $1 \leq i, j \leq n \in \mathbb{N}$. A *common substring* is a substring common to all strings in a set.

Several common substrings can be combined to a *common subsequence*, which is compliant with the above understanding of a signature if position information is added:

Definition 4. A *subsequence* of a string $x = x_1 \dots x_n \in \Sigma^+$ of length $n \in \mathbb{N}$ is defined by an index sequence $1 \leq i_1 < \dots < i_k, k \leq n$. The corresponding subsequence is $x_{i_1} \dots x_{i_k}$. A *common subsequence* is a subsequence common to all strings in a set.

A common subsequence is *pattern*, but it also makes a *signature* that is compliant with definition 2 as each substring has a fixed position in the sequence. A common subsequence that is taken as a signature is denoted by σ; a common substring that is part of such a subsequence is called *σ-segment*. We can now reformulate the task as follows: A signature generator has to extract σ-segments from an input cluster and assemble a signature σ from these. The longer the signature is, the better is its coverage of the corresponding attack class. An optimal result would be a *longest common subsequence* (LCS). However, the LCS problem is NP-hard and thus not feasible in practice [4]. Our method implements an algorithm for efficient extraction of all common substrings. The actual signature composition can then be approximated, again with a tradeoff between accuracy and algorithmic complexity. A simple LCS approximation, which has proven to generate sequences of convenient length, will be introduced in the next section.

The number of all common substrings for a set of inputs can be huge. A naive approach would be to calculate sets of all substrings for each input and then intersect them to get the final list. This would require actually comparing substrings, which is expensive. We developed an algorithm that computes all common substrings in time and space linear in the size of the input, namely the length of the concatenation of all input strings, and does not need to perform any string comparisons at all. The underlying data structure is a *generalized suffix tree* (GST) that represents all substrings (as a substring is a prefix of a suffix) and takes $\mathcal{O}(n)$ time and space [15]. Once the GST is computed, our algorithm explores the tree to find those substrings that are common to all inputs. A side effect of using a GST is that it also provides the positions on which a substring occurs in the input strings. This information is needed for the succeeding sequence assembly. It should be noted that the whole information extraction process is possible without string comparisons. Section 3.2 describes how to compose an appropriate signature from the list of extracted substrings.

3. Implementation – Nebula

In this section we will detail a specific implementation of the described methodology that is realized in the nebula framework. It allows for setting the focus on different aspects, like accuracy or performance. Our method emphasizes performance in terms of the time needed for a signature to be generated.

3.1. Attack Clustering

The nebula signature generation framework presumes that submitted input is already considered malicious. It does not contain any functionality to rate incoming data – this task has to be performed outside the system prior to submissions, e.g., by obtaining input from *honeypots* like in [6, 16], or by invoking separate traffic classification components *flow classifiers* [8, 11, 5, 14]. The performance of the clustering as described in section 2.1 heavily depends on the distance metric used. We established the following selection criteria:

- **Accuracy**: A metric should be accurate, that is, the similarity value of two attacks is close to their normalized edit distance. This is a meaningful relation as we aim for a nearly optimal corresponding string alignment (or a long common subsequence, respectively).
- **Processing time**: The metric should be fast to compute, preferably in time linear in the input size.
- **Space complexity**: An aspect concerning the space complexity is that the calculation of a similarity value should not cause much extra cost due to memory management overhead. Again, at most linear additional space should be sufficient.

We implemented a procedure based on the *spamsum* similarity hashing scheme. A comprehensive description of the whole scheme is given in [13]. Spamsum hashes are well-suited for persistent storage (*tries* as special data structures need little space and answer hash queries efficiently and can be calculated and compared in linear time and space. For the sake of completeness it must be mentioned that a spamsum-based distance function does not fulfil the metric conditions: Hash collisions for different inputs violate the definiteness requirement, i.e. there exist $x \neq y$, but $d(x, y) = 0$. However, this is not really a problem in practice: It suffices if the measure is a *semi metric*, i.e. $d(x, y) = 0 \Rightarrow \forall z : d(x, z) = d(y, z)$, which is the case for spamsum. Cryptographic hashes can additionally be considered to distinguish between different inputs even if a spamsum collision occurs.

The clustering process calculates a nearest neighbor for each input and therefore performs comparisons with all other inputs in the worst case. Hence, the growth of the effort is proportional to the number of already processed attacks, leading to a quadratic overall complexity in the worst case. The expected number of comparisons is much lower if we take into account that the remaining elements of a cluster can be skipped as soon as a new attack gets assigned to it. To further restrict the processing time per attack, our implementation makes use of fixed-sized queues to store outliers, clusters and cluster elements. This has the effect that only a limited amount of historic knowledge is considered with the drawback that attacks with a low frequency might already have been "forgotten" and have no chance to form a cluster. Another advantage of queues is that a new input can be compared with more recent attacks by first starting with the elements at the queues' heads. Assuming that similar attacks occur cumulated on the time scale, there is a good chance that the major part of a cluster can be skipped because of an early hit.

3.2. Signature Composition

Having a list of all common substrings, i.e. the candidates for σ-segments, different signature assembly strategies are possible. Just concatenating all extracted substrings is not possible as they are not disjoint as would be required for a subsequence. Instead we have to choose an appropriate set of non-overlapping substrings so that they form a preferably long subsequence common to the input strings. This task can be formulated as an optimization problem where a solution is feasible when the cost function, i.e. the length of the corresponding sequence, is maximal. However, as finding a LCS is NP-hard, a practical method must implement heuristics for an informed search and can not guarantee that the determined solution is optimal.

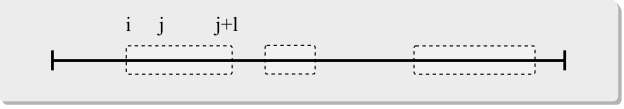

Figure 2. A sequence of σ-segment intervals

We chose a greedy algorithm that is simple to implement and has good performance qualities: The substring list is preprocessed and sorted by length. An iterative greedy algorithm then takes the longest entry from the list, places it in the sequence constructed so far and proceeds with the next list entry. For each segment candidate, all occurences in the input strings are considered to determine a minimum offset i and a maximum offset j. These define an interval $[i, j + l]$ where the substring is allowed to occur in order to finally trigger the signature (see figure 2). Each candidate falls in one of the following categories:

1. If this interval overlaps with one that is already set, the candidate is discarded. Such a situation is visualized in figure 3 where two (light-shaded) substrings already form a sequence and a third (dark-shaded) segment shall be added. As this third segment lies between the present ones for attack x and after them in attack y, there is a syntactical conflict – its position in the sequence is indeterminable.

2. If a substring's interval is disjoint with all ranges of present segments, the segment is added to the sequence. This whole step can be performed in $\mathcal{O}(\log n)$ time where n is the number of segments in the preliminary sequence if the intervals are organized in a sorted array and only $\mathcal{O}(n \log n)$ time and space is necessary to build a signature of n σ-segments.

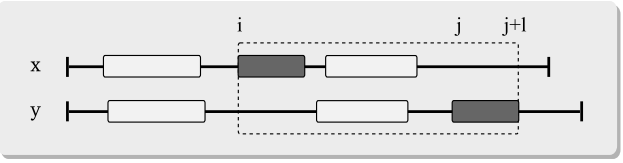

Figure 3. A syntactical conflict

The iterative construction has the advantage that filters can be applied during substring examination. For example, it might be reasonable to skip very short substrings, which often carry no real descriptive features of a certain attack family, but are rather noise that results from an insufficient number of inputs and would be eliminated by further samples. Another restriction would be to only include substrings with an entropy above some threshold as these are more likely to contain characteristic information. An example for a low-entropy string is the NOP sled part in a

```
 1  alert tcp any any -> $HOME_NET 8800 (msg: "nebula rule 2000001 rev. 1";
 2   content: "GET / HTTP/1.0|0d 0a|User-Agent\: DFXPDFXPAAA|eb 03|Y|eb 05 e8 f8 ff ff ff|II"; offset: 0; depth: 51;
 3   content: "A0A"; distance: 23; within: 91;
 4   content: "XP8"; distance: 1; within: 21;
 5   content: "Oy"; distance: 18; within: 525;
 6   content: "|0d 0a|Authorization\: Basic UVVGQlFVRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlF
 7             VRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlFVRkJRVUZCUVVGQlFVRkJ
 8             RVUZCUVVGQlFVRkJRVUZCUVVGQm5TOEFFSkNRa0pBejBtYUJ5djhQUWxKcUFsak5MandGV25UdnVFUkdXRkNMK3E5MTZxOTE1Ly9uUWtKQ1FrSkNRa0p
 9             DUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWt
10             KQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1F
11             rSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkN
12             Ra0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0p
13             DUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWt
14             KQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1F
15             rSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkN
16             Ra0pDUWtKQ1FrSkNRa0pDUWtKQ1FrSkNRa0pDUWtKQ1FrST0=|0d 0a 0d 0a|"; distance: 58; within: 1845;
17   sid: 2000001; rev: 1;)
```

Figure 4. An auto-generated snort signature

buffer overflow exploit trace, which is generally not very descriptive for the particular attack and therefore should be excluded. Our implementation makes use of both filter types and allows for easy extensions. The signature quality improves automatically by publishing new revisions for growing clusters, i.e. when more information about an attack family is available.

3.3. Signature Translation

The de facto standard in network intrusion detection is the open-source IDS framework *snort* [12]. It defines a rule language that allows the specification of attack characteristics as used in our concept via the `content` keyword. Absolute position information can be specified via `offset` and `depth`, and relative offsets with `distance` and `within`, respectively. As both binary content and positional information are stored in the GST, all data needed to translate a σ-segment into a snort rule part is available from the data structures used. Figure 4 shows an example (with line breaks for better readability). The signature matches a buffer overflow attack against a web server on port 8800/tcp and was auto-generated from real attack traces. The first σ-segment describes the start of a `HTTP GET` request and a `User Agent` parameter that contains a typical *GetPC* sequence of assembler instructions. The segments in line 3 to 5 are noise and were included in the signature as it was generated without a minimum substring length restriction. The last segment contains a large string delivered as argument to the `Authorization` parameter and is properly suited as a signature feature because of its length. The translation layer is independent from the rest of the method and thus easily extensible to support other formats as well.

4. Evaluation

A well-known data set for IDS evaluation is the IDEVAL corpus. It contains traces of benign as well as malicious network activity that belong to different categories. Earlier work analyzed its aptitude as training set for intrusion detection systems [10] and checked whether the results for synthetic data can also be applied to real data [9]. It was shown that the IDEVAL corpus is not comparable to real network traffic. Hence we decided to create our own test set. We tested our system against a corpus of 6631 unique attacks, observed with two *honeytrap* honeypots in different production networks. The traces were collected at any first of August, September, and October 2007. All traces were manually analyzed and labeled according to the type of attack. 55 different labels have been identified. However, some labels represented sub-classes of a common group and were marked correspondingly (e.g., the labels for two groups a and b of a supergroup c were denoted by $c.a$ and $c.b$). 34 labels represent clusters with more than one attack, the other 21 labels denote clusters of only one element.

4.1. Cluster Quality

As the performance of signatures depends on both the quality of the classification and on the segment assembly routine, the clustering was evaluated separately. A classification of the labeled corpus was performed for varying similarity thresholds in 5 percent steps, starting with 5 percent. Additionally, the marginal cases for 1 and 99 percent were computed. Two criteria were used to rate the results: The *accuracy* of a corpus partition P is defined as

$$a(P) := \frac{|\{clusters\}|}{\sum_{clusters} |label : label \in cluster|} \in [0,1]$$

and is the reciprocal value of the average number of different labels per cluster. If a partition is *accurate*, i.e. has an accuracy value close to 1, the clusters are well separated and nearly congruent to the partition defined by the labels. Further, the *sensitivity* expresses the amount of outliers that actually belong to a cluster and were not correctly assigned. It calculates as

$$s(P) := 1 - \frac{|\{outlierlabels\} \cap \{clusterlabels\}|}{|\{outlierlabels\} \cup \{clusterlabels\}|} \in [0,1].$$

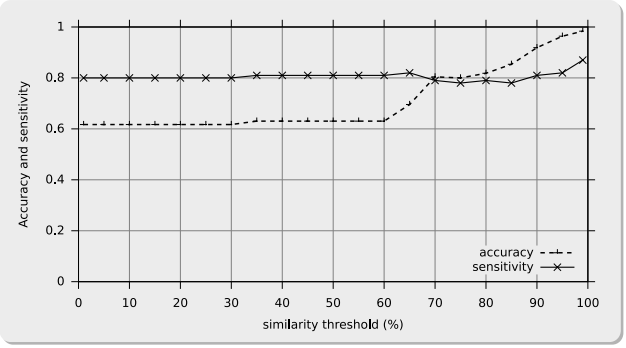

Figure 5. Accuracy and sensitivity

The closer to 1 this value is, the less outliers are classified incorrectly. Summing up, a partition is better, the higher its accuracy and sensitivity is. Figure 5 shows that the sensitivity is relatively static for the evaluated corpus while the accuracy performs a significant refinement around 70 percent and nearly reaches the maximum of 1 for a 99 percent threshold. This suggests to choose the similarity threshold as high as possible but at least above 65 percent. However, the value should be restricted also: Figure 6 shows the number of clusters and outliers for an increasing threshold. Starting from 90 percent, both numbers grow exponentially because attack families begin to split into subclusters. That means that in practice queue limits would be hit sooner, rendering the system more insensitive for low-frequency attacks while the effort per input would be huge. Combining the results from both evaluations, we suggest a similarity threshold of 70 percent.

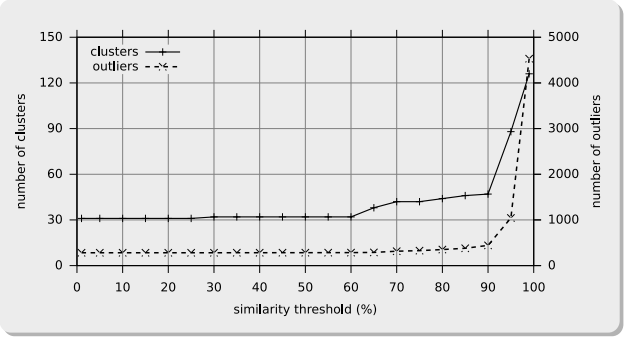

Figure 6. Cluster and outlier quantities

4.2. Signature Quality

To evaluate the signature quality, the tests we performed were threefold. First it was cross-checked that a signature covers all attacks it was generated from. If this was not the case, the algorithm would be either erroneous or incorrectly implemented. Our system worked as expected. In a second test, signatures were generated as if the test inputs were submitted by live sensors. The signature base was then matched against the whole corpus to measure the false positive rate. Table 1 shows the list of clusters (labels), their size and the number of hits for the corresponding signature. As there are 34 clusters in the corpus, 34 different signatures were generated, each from the attacks of one cluster.

Table 1. Signature hits vs. cluster size

Size	Hits	Label	Size	Hits	Label
16	16	2	**48**	**50**	**14.1.2**
2	2	3	7	7	14.2
146	146	4.2	6	6	14.3
3	3	5.3	2	2	14.4.1
4	**114**	**6.1.2**	10	10	14.5
110	110	6.2	17	17	14.6
5	5	7	**12**	**13**	**16**
115	115	8	**38**	**40**	**17.1**
12	12	10	**2**	**0**	**17.2**
32	32	11	46	46	20
16	16	12	6	6	21.2
7	7	13.1	5	5	22.3.1
3	3	13.2	102	102	22.3.2
27	27	13.3	**32**	**34**	**27.1**
13	13	13.4	2	2	27.2
9	9	13.5	3	3	28
37	37	14.1.1	9	9	29.1

The light-shaded lines of table 1 contain cases where the signature matched all cluster elements plus other unrelated attacks. The most sigificant difference was measured for label 6.2.1, but there is an explanation for it: The clusters 6.1.2 and 6.2 share a common superclass – they are slightly different variations of an attack. The signature for 6.1.2 also matches all 110 attacks of the other class (but not vice versa), resulting in a match count of 114. A fully automated classification would construct a common cluster for both labels and consequently produce a single signature that matches the whole superclass. The same reason holds for the other cases, with the exception of label 16 where the signature produced a true false positive. Expressing the false positive sensitivity without considering the previous explanations results in a rate of 11.48 percent. Taking into account that our framework does not distinguish between subclasses of a superclass when it performs the clustering step autonomously, there is only one real false positive, resulting in a rate of 0.1 percent. The dark-shaded line shows a case where the constructed signature did not match its cluster elements, resulting in two false negatives. That was due to technical properties, the signature could not be processed with snort. In fact, false negatives would be an indication for an error in the system itself, so a rate of 0 confirms the correctness of the method.

In a final experiment, the evaluation corpus was split into two, each half containing the attacks from one sensor. Then, a set of signatures was generated for each part and run against the other. Our system was able to successfully detect about 80 percent of all unknown attacks, which is quite good, given that splitting the corpus led to relatively small training sets. Our experience is that the signature quality and the detection rate improve significantly for an increasing number of inputs.

5. Related Work

Several systems have been proposed to automatically generate signatures for self-spreading worms. An example is *Autograph* [5], which monitors a network for traffic patterns typical for spreading worms. Input from sources that are identified as *scanners* is then fed into the signature generation engine. The generator uses a method called *Context-based Payload Partitioning* (COPP) to extract substrings from the traffic and manages a list to identify the most frequent ones which are then taken as a signature basis. This procedure is comparable to nebula's substring extraction as only syntactical information is evaluated. However, the dependancy on the scanner detection component restricts the kind of signatures that can be generated as less frequent attacks are likely to remain unrecognized.

A similar method was proposed by Singh et al. [14]. Their system *Earlybird*, also developed in 2004, uses Rabin fingerprints to count string prevalences. Once the counter for a certain byte sequence hits a threshold, a different counter is initialized to store the number of host pairs for corresponding sessions. A high number is assumed to indicate a spreading worm, and the corresponding sequence is transformed into a signature. Earlybird also includes a detection component that performs rudimentary session tracking: A signature is always matched against the latest 40 bytes. A window of such a small size is obviously not sufficient for longer signatures and reveals another restriction.

Polygraph, a system developed by Newsome et al. in 2005, monitors network traffic for worm-specific behavior and maintains a *suspicious* and a *innocuous* flow pool [11]. Three different signature types are supported: *conjunction signatures* which consist of an unordered set of substrings (called *tokens*), *token-subsequence signatures* in which the list of tokens is ordered, and *Bayes signatures*. An important contribution is the conclusion that signatures which consist solely of a single substring are insufficient [11]. While Polygraph's high-level concepts are similar to our ideas, the method has a worse complexity and cannot implemented in an online algorithm: The running time for token extraction is $\mathcal{O}(sn^2)$ for s inputs of length n and the classification step is quadratic in s and must be performed from scratch for each new input.

A system developed by Li et al, *Hamsa*, addresses the performance issues with Polygraph by using a suffix array for token bookkeeping [8]. It is similar to our approach in the way that identification of dangerous traffic is seen as an outside task. Hamas, like Polygraph, maintains two datasets, one with known benign and one with suspicious traffic. A signature is an unordered set of tokens that are chosen based on the assumption that worm flows will constitute a significant fraction of the suspicous pool. Substrings that occur often are thus interpreted as invariant parts in traffic produced by a spreading worm. As exactly one signature is generated from the list of tokens, Hamas cannot distinguish between many independent simultaneous attacks and thus produces overly generalized signatures. Further, the complexity of the signature generation process is not only dependent on the input size but also on the number of signature tokens. Nebula's generation algorithm outperforms Hamsa as it is strictly linear in the input size only.

All four systems were designed with a focus on catching worms. They contain restrictions that limit their use to this area. However, the basic ideas provide an excellent basis for further research and expose the relevant aspects of signature generation systems. Examples for systems that also work on honeypot data are *Honeycomb*, which was developed by Kreibich and Crowcroft in 2003 as one of the earlier works in the field [6], and *Nemean* by Yegneswaran et al. from 2004 [16]. However, signatures produced by Honeycomb consist of a single longest common substring which is less accurate than multi-token signatures. Nemean requires protocol knowledge, which is generally not available.

6. Real-World Applications

In a classical setup, a signature generator computes signatures from network traffic in a fully automated fashion and installs them for immediate use. But activating filter rules in a production network without any kind of user influence can be dangerous as it might also affect legitimate functioning, although it is arguable that this can be the only way to contain a new threat at an early stage. However, our experience shows that the application area of nebula is not restricted to this field. The system can also be used in an ad-hoc fashion to extract structural information from input data. An example is our analysis of exploits produced by the Conficker worm [7]. We provided a nebula system with 175 different exploit traces produced by the worm's A variant. It took 7.8 seconds to generate a signature for these on a 1.2 GHz Pentium IV. The signature contained an eye-catching segment which turned out to be static shellcode. Using nebula as a structure extraction tool revealed that this part does not change, and saved a lot of time finding a well-suited pattern. Based on this knowledge, manually fine-tuned signatures could be published within minutes. With the help of

nebula, similar structural analysis has been performed with success on completely different data like malware samples or even source code. This is possible because there are no constraints regarding the format of input data.

Because of its availability as an open-source tool, nebula found its way into the package repository of Fedora Linux. Larger projects like the *Network of Affined Honeypots* (NoAH), which is part of a European research program, are evaluating the tool for their early warning framework. The code has already been downloaded 884 times from our project site at `http://nebula.carnivore.it`.

7. Conclusion

In this paper we proposed a general methodology for generating intrusion signatures from syntactical information in attack traces. The first step is to identify and cluster similar inputs so that the resulting clusters represent groups of similar attacks. After that, common features are extracted from the elements of a cluster. These features are finally assembled to a signature. Positional and ordering information is included for the individual parts of a signature, which provides more accuracy than signatures produced by the systems proposed in [5, 11, 14, 8]. Our implementation was optimized to publish new signatures as fast as possible. The input classifier has a quadratic space and time worst case complexity, the signature composition algorithm has a running time in $\mathcal{O}(n)$ for n input bytes.

Nebula is able to compute signatures for previously unknown attacks by requiring no other knowledge than syntactical structure. It is, to our knowledge, the only system that does not make any demands on the format of the data submitted by contributing sensors. We argue that including other than syntactical information renders a signature generator incapable of processing arbitrary unknown input.

One design goal was to create a flexible framework with as few restrictions as possible. The underlying methodology sketches the borders for a specific implementation with independent components. This modular design allows for easy adaptation of single parts to a concrete scenario without affecting the others. The code was made public to the community under an open-source license. Signatures have been generated automatically for novel threats, e.g., the exploit used by the Conficker worm. Real-world experiences show that nebula's value as an ad-hoc tool for semi-automated structure analysis is one of its major strengths.

Acknowledgements

The authors would like to thank the anonymous reviewers of this paper for discussions and comments. We are also grateful to the people who provided valuable suggestions to the implementation of the nebula framework.

References

[1] Abou-Assaleh, T., Cercone, N., Kešelji, V., Sweidan, R. Detection of New Malicious Code Using N-grams Signatures. In *Proc. of the 2nd Annual Conference on Privacy, Security and Trust*, pages 193–196, University of New Brunswick, Canada, 2004.

[2] Bayley, M., Andersen, J., Morleymao, Z., Jahanian, F. Automated Classification and Analysis of Internet Malware. In *Proc. of the 3rd International Workshop on Recent Advances in Intrusion Detection*, pages 178–197, 2007.

[3] Earl Eiland, E., Liebrock, L. M. An Application of Information Theory to Intrusion Detection. In *Proc. of the 4th IEEE International Workshop on Information Assurance*, pages 119–134, 2006.

[4] Gusfield, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[5] Kim, H.-A., Karp, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proc. of the 13th USENIX Security Symposium*, pages 271–286, 2004.

[6] Kreibich, C., Crowcroft, J. Honeycomb – Creating Intrusion Detection Signatures Using Honeypots. In *Proc. of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[7] Leder, F. Werner, T. Know Your Enemy: Containing Conficker. Technical report, The Honeynet Project, 2009.

[8] Li, Z., Sanghi, M., Chen, Y., Kao, M., Chavez, B. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *Proc. of the 2006 IEEE Symposium on Security and Privacy*, pages 32–47, 2006.

[9] Mahoney, M. V., Chan, P. K. An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. Technical report, Computer Science Department, Florida Institute of Technology, 2003.

[10] J. McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. In *ACM Transactions of Information and System Security*, volume 3, pages 262–294, 2000.

[11] Newsome, J., Karp, B., Song, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proc. of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[12] Roesch, M. Snort – Lightweight Intrusion Detection for Networks. In *Proc. of the 13th Conference on Systems Administration*, pages 229–238, Seattle, WA, 1999.

[13] Roussev, V., Richard III, G. G., Marziale, L. Multi-resolution similarity hashing. *Digital Investigation*, 4:105–113, 2007.

[14] Singh, S., Estan, C., Varghese, G., Savage, S. Automated Worm Fingerprinting. In *Proc. of the 6th Conference on Symposium on Operating Systems Design & Implementation*, pages 45–60, 2004.

[15] Ukkonnen, E. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.

[16] Yegneswaran, V., Giffin, J. T., Barford, P., Jha, S. An Architecture for Generating Semantics-Aware Signatures. In *Proc. of the 14th USENIX Security Symposium*, pages 97–112, 2005.