

Automatic Rule Clustering for improved, signature based Intrusion Detection

Christopher Kruegel

Reliable Software Group
University of California, Santa Barbara
chris@cs.ucsb.edu

Thomas Toth

Distributed Systems Group
Technical University Vienna
ttoth@infosys.tuwien.ac.at

Abstract

The constant increase of attacks against networks and their resources creates a necessity to protect these valuable assets. In addition to firewalls, many sites have started to deploy intrusion detection systems (IDS) as a complementary mechanism to tighten security.

Most of the installed ID systems follow a signature based detection approach where an attack attempt is pinned down by comparing collected data to predefined signatures that are known to be malicious. The process of comparing the input data to signatures which are usually described as rules in a specification language forms the core of the actual detection. Current systems perform this comparison in a linear fashion where each input item is successively matched against every rule. Although sometimes ad-hoc optimizations are applied, this mechanism is not optimal.

Our paper describes an approach where we apply machine learning clustering techniques to improve the matching process. Given a set of signatures where each specifies a number of properties that the input data has to fulfill in order to match, we generate a decision tree that is utilized to find malicious input items using as few redundant comparisons as possible.

Following this idea, we have implemented a system that replaces the detection engine of Snort [14, 15], an open-source system that is currently the most-widely deployed signature based network IDS. Experiments with our component (which can be obtained under [16]) show that the detection process could be significantly accelerated.

1 Introduction

Intrusion detection systems (IDS) are security tools that are used to detect traces of malicious activities which are targeted against the network and its resources. IDS are traditionally classified as anomaly or signature based. Signature based systems act similar to virus scanners and look for known, suspicious patterns in their input data. Anomaly based systems watch for deviations of actual from expected behavior and classify all ‘abnormal’ activities as malicious.

The advantage of signature based designs is the fact that they can identify attacks with an acceptable accuracy and tend to produce fewer false alarms (i.e. classifying an action as malicious when in fact it is not) than their anomaly based cousins. The systems are more intuitive to build and easier to install and configure, especially in large production networks. Because of this, nearly all commercial systems and most deployed installations utilize signature based detection. Although anomaly based variants offer the advantage of being able to find prior unknown intrusions, the costs of having to deal with an order of magnitude more false alarms is often prohibitive.

Depending on their source of input data, IDSs can be classified as either network or host based. Network based systems collect data from network traffic (e.g. packets by network interfaces in promiscuous mode) while host based systems monitor events at operating system level such as system calls or receive input from applications (e.g. via log files). Host based designs can collect high quality data directly from the affected system and are not influenced by encrypted network traffic. Nevertheless, they often seriously impact performance of the machines they are running on. Network based IDS, on the other hand, can be set up in a non-intrusive manner - often as an appliance box without interfering with the existing infrastructure. In many cases, this makes them the preferred choice.

Although some vendors claim to have incorporated anomaly based mechanisms, the core detection of most intrusion detection systems which are currently installed at production networks is signature based. Commercial IDSs like ISS RealSecure [13], Symantec's Intruder Alert / Net Prowler [17] or Cisco's IDS [2] offer a wide variety of different signatures and regular updates but unfortunately their engines are undocumented. Academic designs like STAT [18, 19] or Bro [10] and open-source tools like Snort [14] also follow a signature based approach. These systems differ significantly in the way that a signature (or rule) can be defined, ranging from single-line descriptions in Snort to complex script languages such as Bro or STATL [4]. Those languages allow one to express complete scenarios that consist of a number of basic alerts in a certain sequence and therefore require that state is kept in the intrusion detection system. The designs mentioned above also offer different input sensors, ranging from network sniffers to operating system analyzers, and output mechanisms which can write alerts into databases or display them on-line. Nevertheless, all these systems require a component that extracts basic alerts from the input data stream by comparing the properties of an input element to the values defined by the rules. This process is done by successively comparing an input element to every rule specification. Some systems such as Snort attempt to improve this process in an ad-hoc manner, but these optimizations are built into the detection engine and are not tailored to the actual rules which are utilized.

This paper describes an approach that improves the matching process by introducing a decision tree which is derived directly from the utilized signatures by means of a clustering algorithm. By using decision trees for the the detection process, it is possible to quickly determine all rules that describe an input element with a minimal number of comparisons. We assume that a signature defines a single, basic alert. When an attack description language (such as STATL) is capable of describing an intrusion scenario signature composed

of a sequence of multiple basic alerts, our mechanism is only applicable for the detection of these basic building blocks.

A system that utilizes decision trees and data mining techniques to extract features from audit data to perform signature based intrusion detection is presented in [7]. In contrast to our approach, however, they derive the decision tree and the signatures from the audit data while we assume existing signature rules as a base for our decision model.

The following two chapters present the idea of rule clustering and the involved data structures in detail while Chapter 4 explains how the comparison between an input element and a single feature value is performed. In Chapter 5, we briefly describe Snort, its detection engine and our modifications to it. Chapter 6 shows the experimental results obtained with the improved system. Finally, we briefly conclude.

2 Rule Clustering

The idea of rule clustering allows a signature based intrusion detection system to minimize the number of comparisons necessary to determine which rules trigger on a certain input data element.

We assume that a signature rule defines required values for a set of features or properties. Each of these feature has a type (e.g. `integer`, `string`) and a value domain. There are a fixed number of features $f_1..f_n$ and each rule may define values drawn from the respective value domain for an arbitrary subset of these properties. Whenever an input data element is analyzed, the actual values for all n features can be extracted and compared to the ones specified by the rules. Whenever a data item fulfills all constraints set by a rule, the signature is considered to *match* it.

A rule defines a constraint for a feature when it requires the feature of the data item to meet a certain specification. Notice, that it is neither required for a rule to specify values for all features, nor that the specification is an equality relationship. It is possible for a signature to require that a feature of type `integer` is less than a constant or inside a certain interval.

The usual technique utilized to compare a data item with a set of rules is to consecutively check every defined feature of a rule against the input element and then advance to the next one, eventually determining every matching signature. Systems such as Snort stop the detection process after the first matching rule has been identified. Obviously, the processing of a single rule can be stopped immediately when the first feature value is found that does not meet the rule's specification. Another possible optimization, for example utilized by Snort, is to consider certain features more important or discriminating than others and check on a combination of those first before considering the rest. Nevertheless, the number of required comparisons is about linear to the number of rules (a more detailed discussion on the number of checks that Snort requires is presented in Section 5), causing the systems to slow down when the number of rules increase. Unfortunately, novel attacks are discovered nearly on a daily basis and the number of needed signatures is increasing

steadily. This problem is exacerbated by the fact that network and processor speeds are also improving, thereby raising the pressure on intrusion detection systems.

We attempt to mitigate this situation by changing the comparison mechanism from a rule-to-rule to a feature-to-feature approach. Instead of dealing with each rule individually, all rules are combined in a large set and partitioned (or clustered) based on their specifications for the different features. By considering a single feature at a time, we partition all rules of a set into subsets. In this clustering process, all rules that specify identical values for the feature are put into the same subset. This clustering is performed recursively on all subsets with the remaining features or until all subsets only contain a single rule.

3 Decision Tree

The subset structure mentioned above can also be represented as a *decision tree*, where the tree's root node represents the set that initially contains all rules while its children are the direct subsets created by partitioning them according to the first feature. Each subset is associated with a node in the tree. When a node contains more than one rule, these rules are subsequently partitioned and the node is labeled with the feature that has been used for this partitioning step. The arrows that are leading from a node to its children are annotated with the value of the feature that is specified by the rules of each respective child node. Every leaf node of the tree contains only a single rule or a number of rules that can not be distinguished by any feature. Rules are indistinguishable when they are identically with respect to all features used for the clustering process.

Consider the following example with four rules and three features. A rule specifies a network packet from a certain source address to a certain destination address and destination port. The source and destination address features have the type IPv4 while the destination port feature is of type `short integer`.

```
(#) Source Address --> Destination Address : Destination Port
```

- (1) 192.168.0.1 --> 192.168.0.2 : 23
- (2) 192.168.0.1 --> 192.168.0.3 : 23
- (3) 192.168.0.1 --> 192.168.0.3 : 25
- (4) 192.168.0.4 --> 192.168.0.5 : 80

A possible decision tree is shown in Figure 1. In order to create this tree, the rules have been partitioned on the basis of the three features, from the left to right, starting with the source address. Notice that nodes with only a single rule need not and cannot be partitioned any further. When the IDS attempts to find the matching rules for an input data item, the detection process commences at the root of the tree. The label of the node determines the next feature that needs to be examined. Depending on the actual value of that feature, the appropriate child node is selected (using the annotations of the arrows

leading to all children). As the rule set has been partitioned by the respective feature, it is only necessary (and possible) to continue detection at a single child node.

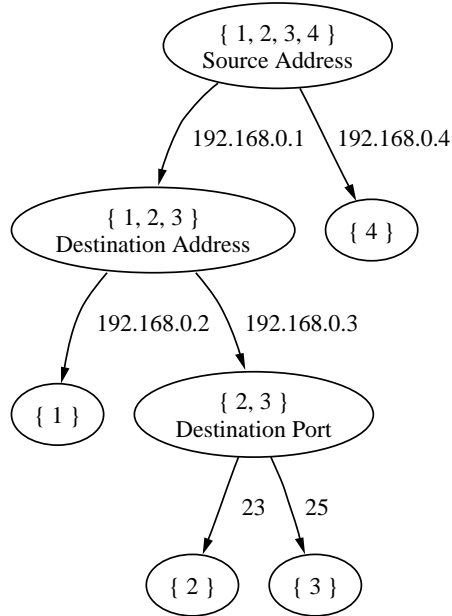


Figure 1: Decision Tree

When the detection process eventually halts at a leaf node, all rules associated with this node are *potential matches*. However, it might still be necessary to check some features. All features that are specified by the potentially matching rules but that have not been used by the clustering process to partition any node on the path from the root to this leaf must be evaluated. Consider Rule 1 in the leftmost leaf node in Figure 1. Both, source address and destination address have been used by the clustering process on the path between this node and the root, but not the destination port. When a packet which has been sent from 192.168.0.1 to 192.168.0.2 is evaluated as input element, the detection process eventually terminates at the leaf node with Rule 1. Although this rule becomes a potential match, it is still possible that the packet was directed to a different port than 23. Therefore, the destination port has to be checked additionally. Our implementation solves this problem by simply expanding the tree for all defined features that have not been used so far. This only requires to be able to further ‘partition’ a node with only one rule, a step that results in a single child node.

At any time, when the detection process can not find a successor node with a specification that matches the actual value of the input element under consideration (i.e. an arrow with a proper annotation), there is no matching rule. This allows to exit immediately.

The decision tree is built in a top-down manner. At each non-leaf node, that is for

every (sub)set of rules, one has to select a feature that is used for extending the tree (i.e. partitioning the corresponding rules). Obviously, features that are not defined by at least one rule are ignored in this process as a split would simply yield a single successor node with the exactly same set of rules. In addition, all features that have been used previously for partitioning at any node on the path from the node currently under consideration to the root are excluded as well. A split on the basis of such a feature would also result in only a single child node with exactly the same rules. This is because of the partitioning at the predecessor node, which guarantees that only rules that specify identical values for that feature are present at each child node. Notice, however, that the choice of the feature among those that are eligible has an important impact on the shape and the depth of the resulting decision tree. As each node on the path from the root to a leaf node accounts for a check that is required for every input element, it is important to minimize the depth of the decision tree. An optimal tree would consist of only two layers - the root node and leaves, each with only a single rule. This would allow the detection process to identify a matching rule by examining only a single feature. As an example, consider the decision tree for the same four rules introduced above as shown in Figure 2. By using the destination port first, the resulting tree has a maximum depth of only two and consists of six nodes instead of seven.

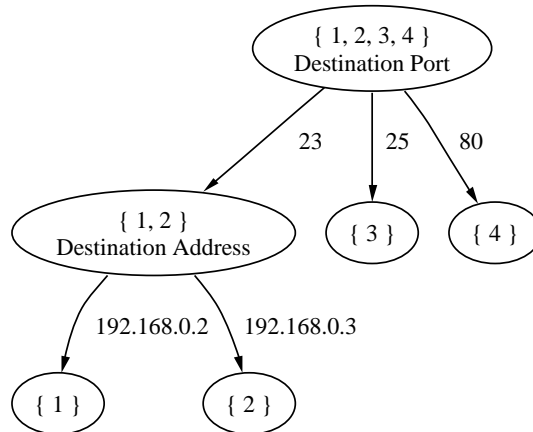


Figure 2: Optimized Decision Tree

In order to create an optimized decision tree, we utilize a variant of ID3 [11, 12], a well-known clustering algorithm applied in machine learning. This algorithm builds a decision tree from a classified set of data items with different features using the notion of information gain. The information gain of an attribute or feature is the expected reduction in entropy (disorder) caused by partitioning the set on this attribute. The entropy of the partitioned data is calculated by weighting the entropy of each partition by its size relative to the original set. The entropy E of a set S of rules is calculated by the following Formula 1.

$$E_S = \sum_{i=1}^{S_{max}} -p_i \log_2(p_i) \quad (1)$$

where p_i is the proportion of examples of category i in S . S_{max} denotes the number of different categories. In our case, each rule itself is considered to be a category on its own, therefore S_{max} is the total number of rules and p_i is equal to $\frac{1}{n}$, with n being the number of rules in S . This simplifies the equation above to

$$E_S = \sum_{i=1}^n -\frac{1}{n} \log_2\left(\frac{1}{n}\right) = -\log_2\left(\frac{1}{n}\right) = \log_2(n) \quad (2)$$

Given the result above, the information gain G for a rule set S and a feature F can be derived as shown in Formula 3.

$$G_{(S,F)} = E_S - \sum_{v=Val(F)} \frac{|S_v|}{|S|} E_{S_v} = \log_2(|S|) - \sum_{v=Val(F)} \frac{|S_v|}{|S|} \log_2(|S_v|) \quad (3)$$

In this equation, $Val(F)$ represents all different values of feature F that are specified by rules in S and v iterates over this set. S_v are the subsets of S that contain all rules with an identical specification for feature F . $|S|$ and $|S_v|$ represent the number of elements in the rule sets S and S_v , respectively.

ID3 performs local optimization by choosing the most discriminating feature, i.e. the one with the highest information gain, for the rule sets at each node. Nevertheless, no optimal results are guaranteed as it might be necessary to choose a non-local optimum at some point to achieve the globally best outcome. Unfortunately, the problem of creating a minimal decision tree that is consistent with a set of data is NP-hard. It is obvious that it is not necessary to choose the same order of features for partitioning rule subsets. It is possible and common that different branches of the same decision tree are split by different features.

So far, we have not considered the situation when a rule completely omits the specification of a certain feature or defines multiple values for it (e.g. instead of a single integer, a whole interval is given). As not defining a feature is equivalent to have the rule specify the feature's whole value domain, we only have to consider the definition of multiple values. Notice that it is sometimes not possible to enumerate the value domain explicitly, for example for floating point numbers. This can be easily solved by specifying appropriate intervals. When a certain rule specifies multiple values for a property, there can be a potential overlap with a value defined by another rule. As the partitioning process can only put two rules into the same subset when both specify the exact same value for the feature used to split, this poses a problem. The solution is to put both rules into one subset and annotate the arrow with the value the two have in common **and** put the rule that defines multiple values into another subset and label the arrow leading to that node with the values that this rules specifies alone.

Obviously, this basic idea can be extended to multiple rules with many overlapping definitions. The value domain of the feature that is used for splitting is partitioned into distinct intersections of all the values which are specified by the rules. Then, for each rule, a copy is put into every intersection that is associated with a value defined by that rule. Consider the example rules that have been introduced above and change the second rule to one that allows an arbitrary destination port as shown below.

(2) 192.168.0.1 --> 192.168.0.3 : **any**

The decision tree that results when the destination port feature is used to partition the root node is shown in Figure 3. Notice, how the value domain $[0, 2^{16}-1]$ of destination port has been divided into the seven intersections represented by the following intervals $[0,22]$, 23, 24, 25, $[26,79]$, 80 and $[81, 2^{16}-1]$. Rules that define the appropriate values are put into the child nodes with the corresponding arrow labels. In addition, a packet sent from 192.168.0.1 to 192.168.0.3 and port 25 satisfies the constraints of both rules, number 2 and 3. This fact is reflected by the leaf node in the center of the diagram that holds two rules but cannot be partitioned any further.

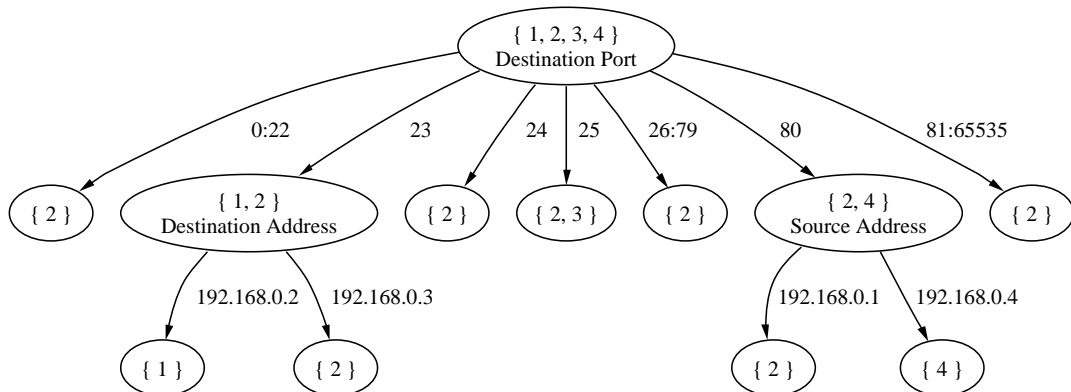


Figure 3: Decision Tree with **any** Rule

As can be seen easily, the total number of rules at the children of a certain node does not necessarily need to be equal to the number of rules in that parent node as one would expect when a set is partitioned. This has effects on the size of the decision tree as well as on the function that chooses the optimal feature for tree construction. When many rules need to be processed and each only defines a few of all possible features, the size of the tree soon becomes unmanageable. Unfortunately, this is, for example, the case with Snort. The rule base currently consists of well over thousand rules and each is only specifying a small subset of the available features. Often, only three or four features out of the 12 that we are considering are actually defined.

In order to deal with this problem, one has to trade execution speed during the detection process for a reduced size of the decision tree. This can be done by dividing the rule set

into several subsets and build separate trees for each set. The detection engine has to process every input element with all trees, combining the results. For our prototype, we implemented two trees for each protocol that Snort supports (i.e. TCP, UDP and ICMP) and were capable of processing all rules in a reasonable amount of time (for more details, refer to Section 6). In addition, we utilize the following simple heuristics while dividing the rule set for each protocol.

As the biggest problem are unspecified properties, we order all features in descending order by the percentage of rules that specify them. Adding a feature at a time, starting with the one that is specified most often, we determine how many rules define all but one of the current first n most-often-specified features. As the number of features is increased, the number of rules that specify all but one of those properties is obviously decreasing. We add features until the number of rules drops below a certain, definable threshold and then use exactly those rules for the first tree. As only rules have been selected that define most of the relevant features, the tree does not need to cope with many unspecified feature entries and remains relatively small. The threshold is chosen in a way that allows the larger fraction of the rule set to go into the first tree. The rest of the rules, which is then only a small fraction of the total, is put into the second tree. When two trees would not suffice, it is obviously possible to increase the number of trees and reduce the rules assigned to each.

Even when using multiple trees, the number of checks that each input element requires while traversing the decision trees is bound by a multiple of the number of features but is still independent of the number of rules. However, our system is not capable of checking input data with a constant overhead. The additional overhead which depends on and grows with the number of rules is now associated with the checks at every node. In contrast to a system that checks all rules in a linear fashion, the comparison of the value extracted from the input element with a rule specification is no longer a simple operation. In our approach, it is necessary to select the appropriate child node that has a label which matches the input data's value. As the number of rules increases and the number of successor nodes grows, this check becomes more expensive. Nevertheless, that comparison can be made more efficient than a linear operation.

4 Feature Comparison

This section discusses mechanisms to efficiently handle the processing of an input element at nodes of the decision tree. As mentioned above, each feature has a type and an associated value domain. When building the decision tree or evaluating input elements, features with different names but otherwise similar types and value domains can be treated identically. It is actually possible to reuse functionality for a certain type even when the value domains are different (e.g. 16 or 32 bit variations of the type `integer`). For our prototype, we have implemented functionality for the types `integer`, `IPv4 address`, `bitfield` and `string`. `Bitfield` is utilized to check for patterns of set bits in a fixed length bit array and is needed to handle the flags in various network protocol headers.

The basic operation that has to be supported in order to be able to traverse the decision tree is to find the correct successor node when getting an actual value from the input item. This is usually a search procedure among all possible child values created by the intersection of the values specified by each rule.

Using binary search, it is easy to implement this search with an overhead of $O(\log n)$ for `integer`, where n is the number of rules. For the `IPv4 address` and the `bitfield`, the different successor values are stored in a tree with a depth that is bound by the length of the addresses or the bitfield in bits, respectively. This yields a $O(1)$ overhead.

The situation is slightly more complicated for the `string` type, especially when a data item can potentially contain a (nearly) arbitrary long string value. When determining the intersections of the property specifications of a rule set for the partitioning, it is possible that the input contains any of all possible combinations of the specified string values. This yields a total of 2^n different intersections or subsets where n is the number of rules under consideration, clearly an undesirable result. We circumvent this problem by demanding that the `string` type may only be used as the last attribute for splitting when creating the decision tree. In this setup, the nodes that partition a rule set according to a string attribute actually become leaf nodes. It is then possible to determine all matching rules (i.e. all rules which define a string value that is actually contained in the input element) during the detection process without having to explicitly enumerating all possible combinations and have their corresponding nodes in memory.

Systems, such as Snort, which compare input elements with a single rule at a time often use the Boyer-Moore [9] or similar optimized pattern matching algorithms to search for string values in their input data. These functions are suitable to find a single keyword in an input text. Often, the same input element has to be scanned repeatedly as more than one rule which all define different keywords might match.

As pointed out in [3], Snort's rule set contains clusters of nearly identical signatures that only differ by slightly different keywords with a common, identical prefix. In order to save redundant comparisons when using the Boyer-Moore algorithm multiple times on the same input string trying to find nearly similar keywords, they introduced a variation of the Aho-Corasick [1] tree to match on several strings with a common prefix in parallel. Unfortunately, this approach is only suitable when keywords share a common prefix. When creating the decision tree, it often occurs that several signatures that only differ by their string property end up in the same node. Nevertheless, they do not necessarily have anything in common.

Instead of invoking the Boyer-Moore algorithm for each string, we utilize a modified version implementing ideas from the Karp-Rabin algorithm [5] to exploit the potential of searching for multiple unrelated strings in a single pass. This variant works by building a global skip table by merging the individual Boyer-Moore skip tables for each keyword. This global skip table is used by the pattern matching process to determine the number of characters that can be advanced in the input stream when searching for those string keywords. Individual skip tables for each keyword are built by counting, for each character of the string alphabet, the distance of the last occurrence of this character in the keyword

to the end of the keyword. When the character does not appear at all, its skip value is set to the length of the keyword. The merge of individual tables is performed by choosing for each character the lowest entry among all individual tables.

The example in Table 1 shows the individual skip tables for the strings `/bin/sh` and `.src` as well as the global skip table after the merge. The global skip table is the only data structure that has to be retained for the detection process.

Character	.	/	b	c	h	i	n	r	s	Others
Skip Table: <code>/bin/sh</code>	7	2	5	7	0	4	3	7	1	7
Skip Table: <code>.src</code>	3	4	4	0	4	4	4	1	2	4
Global Skip Table	3	2	4	0	0	4	3	1	1	4

Table 1: Skip Tables

When the skip value for the current character, which is read from the input stream, yields 0, the detection process cannot advance any further. Now, every keyword has to be matched against the input stream. This expensive process can be accelerated by using hash tables. Using the minimum length m of all keywords, which can be determined easily when creating the global skip table, a hash value is calculated for the last m characters of each keyword. This value is used as index to insert the keyword into the appropriate slot of a hash table. Whenever a character that yields a skip value of 0 is detected in the input stream, a hash value is determined for the previous m characters, including that current one, from the input stream. Now, it is only necessary to perform the exact match between the input stream and those strings which are stored at the corresponding hash table slot, reducing the number of comparisons considerably.

5 Snort

Snort [14] is arguably the most-widely deployed signature based network intrusion detection tool. Besides the fact that Snort is open source, numerous attack signatures are freely available and it is relatively easy to extend with custom pre- and post-processing modules.

A Snort rule is a simple statement that defines properties of a network packet. Each rule additionally states an action in case a packet is detected that fulfills the specification. This can range from logging alert information to a database to the activation of other rules or even active countermeasures.

Snort builds a two-dimensional list structure from the input rules for each supported protocol, one list that consist of **Rule Tree Nodes (RTNs)**, the other one of **Option Tree Nodes (OTNs)**. The RTNs store the values of the the mandatory features of each rule, namely the source and destination addresses and ports, while OTNs hold pointers to properties that

may be defined optionally. A list of OTNs is attached to each RTN - these represent rules with different optional parts but with a similar source/destination address and port combination.

When an input item such as a packet is evaluated, the list of all RTNs is consulted. Whenever a match occurs, the corresponding OTNs are checked next. Notice however that the list of RTNs needs to be traversed completely under all circumstances. A packet that matches a certain RTN might potentially match other RTNs later in the list as well. The only exception occurs when a matching rule is found. In this case, Snort immediately exits and reports the detected alert. This is actually a controversial ‘feature’ that has been in the center of much debate ever since. As only a little fraction of input elements match signatures, this premature exit does not speed up the detection process significantly. In addition, a packet that contains a serious attack might trigger another rule first that only reports suspicious, but not necessarily malicious traffic (such as port scans). This is especially dangerous as the order in the rule configuration file is not reflected by the data structures. It is possible that a rule, which is specified after another one, matches first.

In the common case when no alert is triggered, the input packet needs to be compared to all RTNs of the corresponding protocol. When using the rule set that is shipped with Snort 1.8.7 and enabled per default, its 1239 rules are translated into RTNs and OTNs for the supported protocols as shown below in Table 5. Max., Min. and Avg. show the maximum, the minimum and the average OTNs that are associated with RTNs. As each rule is represented by a corresponding OTN, the entries in the # OTN column sum up to the total number of rules.

Protocol	# RTN	# OTN	Max.	Min.	Avg.
IP	4	32	19	1	8.0
ICMP	4	33	27	1	8.3
UDP	31	82	23	1	2.6
TCP	88	1092	728	1	12.4

Table 2: Statistics - Snort Data Structures

In theory, the two-dimensional structure could allow the length of the lists and therefore the number of required checks to grow proportional to the square root of the total number of rules. However, as can be seen in the table above, the distribution of RTNs and OTNs is very uneven. For UDP, 31 different RTNs are created from only 82 rules which have only two rules associated to it on average, requiring an input packet to be checked against all of them. For TCP, more than half of the rules (i.e. 728 out of 1092) are connected to a single RTN that represents incoming HTTP traffic. Each legitimate packet sent to the web server therefore needs to be compared to at least 728 rules, lots of them requiring expensive string matching operations. As can be seen easily, the ad-hoc selection of source and destination addresses as well as ports provides some clustering of the rules, but it is far from optimal. According to our experience, the destination port and address are

two reasonably discriminating features, while the source port seems to be less important. Often, important features such as ICMP code/type or TCP flags are hidden in the OTN lists while they would provide excellent properties to quickly determine matching rules, allowing the detection process to save many unnecessary string matching operations later. The problem of statically chosen features that are most relevant for quickly excluding rules is solved by our automatic rule clustering algorithm that always selects the most discriminating feature when creating the decision tree. Therefore, we can build trees that are tailored to the actual rule set.

The following Section 6 presents the results that we have obtained with our module which replaces the two dimensional list structures with decision trees. We have implemented patches for Snort 1.8.6 and Snort 1.8.7 that can be downloaded under [16]. The reader is referred to Appendix A for details about the integration of our patch into Snort and interesting findings about the current rule set.

6 Experimental Data

In order to set up a realistic simulation environment that would allow us to achieve repeatable results, we connected two machines directly via a fast Ethernet connection with 100 Mbps. One computer, a Intel Pentium II with 200 MHz running Linux 2.2, acted as the traffic generator that injected network packets read from a file into the wire as fast as possible. The other one, a Pentium III with 550 MHz running Linux 2.4, sniffed and analyzed this traffic with Snort. When performing the measurements, all preprocessor and logging plug-ins of Snort have been disabled to have our results reflect mostly the processing cost of the detection algorithms themselves. Obviously, the overhead of the operating system to process the network traffic as well as the packet capturing and parsing functionality of Snort still influences the numbers, but it does so for both approaches. In addition, our results capture a more realistic performance gain that is actually visible to the user and not only a virtual one by simply comparing the speed-up achieved by the new detection engine.

In a single run, we measured the ratio of packets that Snort was able to analyze in relation to the total number of packets injected into the network. This also gives an indication of the percentage of attacks that Snort would have detected when we assume that attacks are randomly distributed over the log files and that Snort drops random packets when they arrive too fast. For each test data set, we performed five single runs for an increasing number of rules in steps of three. The resulting graphs show that the ratio of analyzed packets drops as more rules are added.

We used two different data sets. One file with a size of 68 MB was produced by MIT Lincoln Labs on the May, 5th 1998 as part of their DARPA intrusion detection evaluation [8]. The other one with 137 MB represents traffic that we captured at the firewall of our institute during a period of 24 hours. The comparison of the results for the MIT/LL traffic is shown in Figure 4, the graphs for the traffic at our firewall are presented in Figure 5.

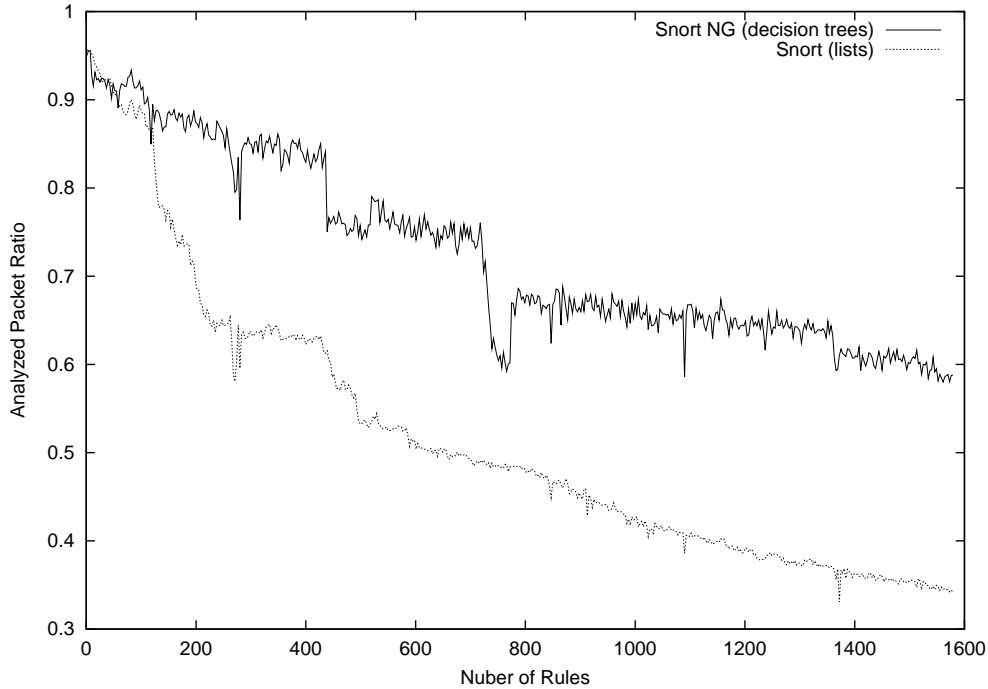


Figure 4: Results for MIT Lincoln Lab Traffic

These figures clearly indicate that a significant number of packets cannot be analyzed anymore when the number of signatures reaches a few hundred. This result is in line with the findings described in [6]. Novel attacks are discovered regularly and signatures have to be developed to detect them. The current rule set of Snort contains at the time of writing a total of 1602 entries and this number is expected to increase over time. New mechanism such as the decision trees presented in this paper are necessary to meet that scalability challenge. Notice that we have achieved a speed-up of 73.7% for the MIT/LL data when all rules are used. In this case, the ratio of analyzed packets with our prototype is 0.589 while the one for the old system is 0.339.

Building our data structures requires some time during start up. Depending on the number of rules and the features which are defined, the tree can contain several tens of thousands of nodes. A few Snort configuration options, such as being able to specify lists of source or destination addresses for certain rules, cause our system to create several signature instances from that rule which are later treated independently during the building of the decision tree. When defining a network topology with different subnets and multiple web servers, the default Snort rules get transformed into 3310 rule instances. It took 5.2 seconds on average to build a decision tree for these rules. Nevertheless, even when slower computers are utilized, it is a cost that is only paid once at start up.

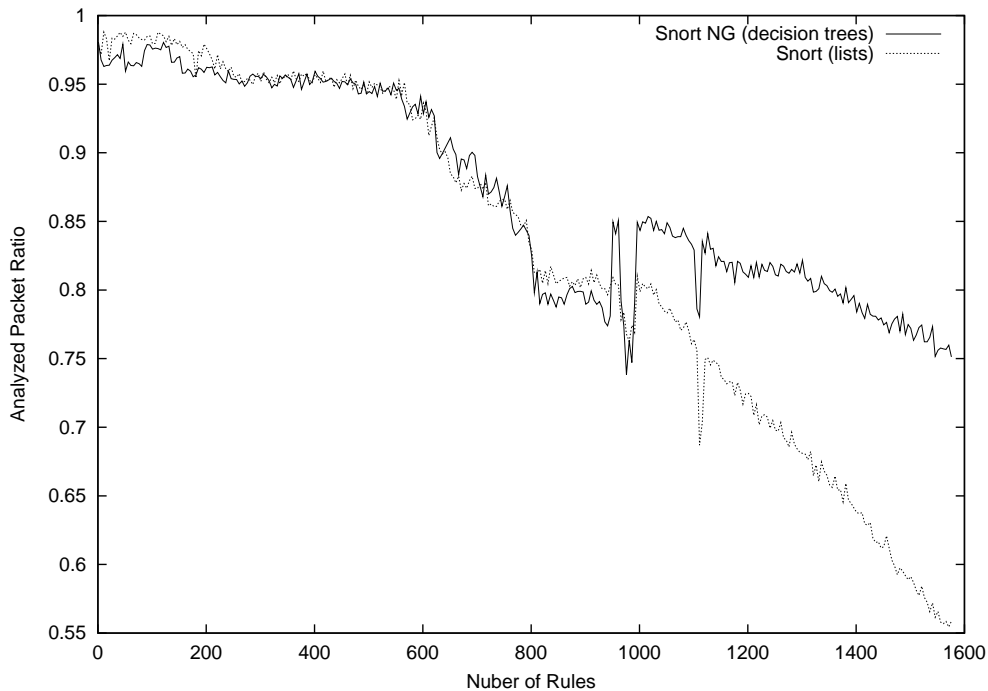


Figure 5: Results for Firewall Traffic

7 Conclusion

Signature based intrusion detection systems face the challenge of an ever increasing number of rules that need to be compared to input elements. Combined with the facts that the amount of data is constantly growing and that users expect results in real-time, current systems have already met their limits in coping with this challenge. Novel approaches to re-structure or cluster the signature rules are necessary in order to relief the detection engines of as many redundant checks as possible.

This paper presents a clustering approach based on decision trees which utilizes machine learning principles to optimize the rules-to-data comparison process. We describe an application of our mechanism for the most popular open-source network intrusion detection Snort and show that a significant improvement of its processing speed was possible.

References

- [1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the Association for Computing Machinery*, 18:333–340, 1975.
- [2] Cisco IDS - formerly NetRanger. <http://www.cisco.com>, 2002.

- [3] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. In *Proceedings of DISCEX 2001*, 2001.
- [4] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *Proceedings of the ACM Workshop on Intrusion Detection Systems*, Athens, Greece, November 2000.
- [5] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. Technical report, Aiken Computation Laboratory, Harvard University, Technical report TR-31-81, 1981.
- [6] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2002.
- [7] Wenke Lee, Sal Stolfo, and Kui Mok. A Data Mining Framework for Building Intrusion Detection Models. In *In Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [8] MIT Lincoln Labs. DARPA Intrusion Detection Evaluation. <http://www.ll.mit.edu/IST/ideval>, 1998.
- [9] J.S. Moore and R.S. Boyer. A Fast String Searching Algorithm. *Communications of the Association for Computing Machinery*, 20:762–772, 1977.
- [10] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *7th USENIX Security Symposium*, San Antonio, TX, USA, January 1998.
- [11] J. R. Quinlan. Discovering rules by induction from large collections of examples. In *Expert Systems in the Micro-Electronic Age*. Edinburgh University Press, 1979.
- [12] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.
- [13] RealSecure. http://www.iss.net/products_services/enterprise_protection, 2002.
- [14] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX Lisa 99*, 1999.
- [15] Snort. Open-source Network Intrusion Detection System. <http://www.snort.org>.
- [16] Snort-NG. Snort - Next Generation: Network Intrusion Detection System. <http://www.infosys.tuwien.ac.at/snort-ng>.
- [17] Symantec - NetProwler and Intruder Alert. <http://www.symantec.com>, 2002.

- [18] Giovanni Vigna, Steve Eckmann, and Richard A. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.
- [19] Giovanni Vigna and Richard A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

Appendix A

Integrating Decision Trees into Snort

When integrating our data structures and the detection process into Snort, we attempted to keep the changes to the original code as little as possible. This ensures that the modifications can be ported to new versions of Snort easily and enables us to test our components independently of the main program. The two major changes occurred in the parser and in the code that calls the original detection process with its two-dimensional lists.

The parser (i.e. the functions `ParseRule()` and `ParseRuleOptions()`) in `rules.c` had to be adapted to extract the relevant signature information from the rules. Snort translates the checks of properties into function pointers which are later called by the detection process and encapsulates their values in private data areas that have a feature dependent layout. Although possible, it seems undesirable to extract values required by our functions from function pointers and their corresponding private data structures, therefore they are directly gathered during parsing. Nevertheless, the original lists structure is still created and utilized by our code (e.g. for dynamic rule activation) whenever possible.

The second part of changes affected the detection function (`Detect()`) in `rules.c`. Instead of calling the original processing routine, it redirects to our decision trees. The modified detection procedure calls response and logging functions in a similar way than the old one. However, it is possible that they are called several times for a single packet as our engine determines all matching signatures for each input element. When this behavior is undesirable, our module can be put into a mode where only the first match per packet is reported (with the command line switch `-j`).

All other changes were only minor modifications of function prototypes to accommodate additional arguments or the addition of variables to data structures such as `OptTreeNode`. Neither the preprocessing nor the response and logging functionality is affected in any way by our patch. It simply replaces the lists with decision trees. Therefore, it is further on possible to use and write new plug-in modules as desired. In addition, it is also possible to add new features (i.e. to introduce new keywords) to the signature language. Although this seems contradicting at first glance as our decision tree requires the knowledge of these features and their corresponding types, it can be done by excluding these properties from the decision tree and simply check them afterwards for all signatures that have triggered for a certain packet. This obviously reduces the effectiveness of our approach but allows one to extend Snort and keep the ability of deploying the modified detection engine.

Discussion of Snort Rules

The rule set of Snort has evolved together with the program itself. Whenever a new threat has been discovered, rules that specify an appropriate signature to detect it have been added. The current version ships with 1602 rules that are stored in 33 files. When testing our implementation, we used Mucus to generate test data for a subset of 848 signatures. Mucus is a tool that reads a rule and creates a network packet with exactly the properties

specified by that signature. When running our prototype on each test packet, we obviously expected to detect the corresponding rule used to create it. Sometimes however, not only the expected signature triggered on a single packet, but several others as well. This has three main reasons.

Rules are identical: A few rule pairs simply specify identical values for the same features.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN SYN FIN";flags:SF;
reference:arachnids,198; classtype:attempted-recon; sid:624; rev:1;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN synscan portscan"; id: 39426;
flags: SF;reference:arachnids,441; classtype:attempted-recon; sid:630; rev:1;)
```

Rules are nearly identical: Several rule pairs specify identical values for all but one feature. For this feature, one rule does not define a value at all, thereby matching all packets that trigger the other one. Notice that for the second rule pair, only the destination ports differ. The content string represented by the ASCII values |57 48 41 54 49 53 49 54| is identical to 'WHATISIT'.

```
alert tcp $HOME_NET 23 -> $EXTERNAL_NET any (msg:"TELNET Bad Login";
content: "Login incorrect"; nocase; flags:A+; classtype:bad-unknown; sid:1251; rev:4;)
alert tcp $HOME_NET 23 -> $EXTERNAL_NET any (msg:"TELNET login incorrect";
content:"Login incorrect"; flags:A+; classtype:bad-unknown; sid:718; rev:5;)

alert tcp $HOME_NET 146 -> $EXTERNAL_NET 1024: (msg:"BACKDOOR Infector.1.x"; content:
"WHATISIT"; flags: A+; sid:117; classtype:misc-activity; rev:3;)
alert tcp $HOME_NET 146 -> $EXTERNAL_NET 1000:1300 (msg:"BACKDOOR Infector 1.6 Server
to Client"; content:"|57 48 41 54 49 53 49 54|"; flags:A+; sid:120; rev:3;)
```

Rules are imprecise: Certain rules specify feature values that can appear with a reasonable high probability in random, usually non-malicious packets as well. This affects many rules which define a very short content string that is searched for inside the packet payload.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP wu-ftp file completion attempt [";
flags:A+; content:"~"; content:["; classtype:misc-attack; sid:1377; rev:7;)
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP wu-ftp file completion attempt {";
flags:A+; content:"~"; content:["; classtype:misc-attack; sid:1378; rev:7;)
```

The problem with multiple matching rules is the fact that Snort only reports the first one. This might result in a packet that triggers a signature which indicates only a minor threat although it would also match one reporting a serious security problem. When using Snort, one has to make sure that signatures are specified as precise as possible and have only a negligible probability of matching benign traffic. We circumvent this limitation by reporting all rules that match a certain packet.