# Using decision trees to improve signature-based intrusion detection

Christopher Kruegel[1] and Thomas Toth[2]

[1] Reliable Software Group
University of California, Santa Barbara
chris@cs.ucsb.edu
[2] Distributed Systems Group
Technical University Vienna
ttoth@infosys.tuwien.ac.at

**Abstract.** Most deployed intrusion detection systems (IDSs) follow a signature-based approach where attacks are identified by matching each input event against predefined signatures that model malicious activity. This matching process accounts for the most resource intensive task of an IDS. Many systems perform the matching by comparing each input event to all rules sequentially. This is far from being optimal. Although sometimes *ad-hoc* optimizations are utilized, no general solution to this problem has been proposed so far.
This paper describes an approach where machine learning clustering techniques are applied to improve the matching process. Given a set of signatures (each dictating a number of constraints the input data must fulfill to trigger it) an algorithm generates a decision tree that is used to find malicious events using as few redundant comparisons as possible.
This general idea has been applied to a network-based IDS. In particular, a system has been implemented that replaces the detection engine of Snort [14, 15]. Experimental evaluation shows that the speed of the detection process has been significantly improved, even compared to Snort's recently released, fully revised detection engine.

**Keywords:** Signature-based Intrusion Detection, Machine Learning, Network Security

## 1 Introduction

Intrusion detection systems (IDSs) are security tools that are used to detect evidence of malicious activity which is targeted against the network and its resources. IDSs are traditionally classified as anomaly-based or signature-based.

This work has been supported by the FWF (Fonds zur Förderung der wissenschaftlichen Forschung), under contract number P13731-MAT. The views expressed in this article are those of the authors and do not necessarily reflect the opinions or positions of the FWF.

Signature-based systems are similar to virus scanners and look for known, suspicious patterns in their input data. Anomaly-based systems watch for deviations of actual behavior from established profiles 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 they 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 easier to implement and simpler to configure, especially in large production networks. As a consequence, nearly all commercial systems and most deployed installations use signature-based detection. Although anomaly-based variants offer the advantage of being able to find prior unknown intrusions, the costs of dealing with a large number of 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 from network interfaces in promiscuous mode) while host-based systems collect events at the operating system level, such as system calls, or at the application level. 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 without interfering with the existing infrastructure. In many cases, these characteristics make network-based IDS the preferred choice.

Although some vendors claim to have incorporated anomaly-based detection techniques into their system, the core detection of most intrusion detection systems is signature-based. Commercial IDSs like ISS RealSecure [13], Symantec's Intruder Alert/Net Prowler [19] or Cisco's IDS [2] offer a wide variety of different signatures and regular updates. Unfortunately, their engines are mostly undocumented. Academic designs like STAT [20] or Bro [10] and open-source tools like Snort [14] also follow a signature-based approach. They differ significantly in the way a signature (or rule) can be defined, ranging from single-line descriptions in Snort to complex script languages such as Bro or STATL [4]. The latter allows one expressing complete scenarios that consist of a number of related basic alerts in a certain sequence and therefore require that state is kept. Nevertheless, all systems require a component that produces basic alerts as a result of comparing the properties of an input element to the values defined by their rules. These basic alerts can then be combined as building blocks to describe the more complex scenarios.

Most systems perform the detection of basic alerts by comparing each input event to all rules sequentially. Some of the aforementioned programs utilize ad-hoc optimizations, but they require domain specific knowledge and are not optimal for different rule sets. Therefore, a general solution to this problem is needed. Our paper describes an approach that improves the matching process by introducing a decision tree which is derived directly from and tailored to the installed intrusion detection signatures by means of a clustering algorithm. By using decision trees for the detection process, it is possible to quickly determine all firing rules (i.e., rules that match an input element) with a minimal number of comparisons.

The paper is organized as follows. Section 2 discusses related work and describes current rule matching techniques. Section 3 and Section 4 present the idea of applying rule clustering and the creation of decision trees in detail. Section 5 explains how the comparison between an input element and a single feature value is performed. Section 6 shows the experimental results obtained with the improved system. Finally, in Section 7, we briefly conclude.

## 2 Related Work

The simplest technique for determining whether an input element matches a rule is to sequentially compare it to the constraints specified by each element of the rule set. Such an approach is utilized by STAT [20] or by SWATCH [18], the simple log file watchdog.

Consider a STAT (state transition analysis) scenario that consists of three states, one start state and two terminal states. In addition, consider that a transition connects the start state to each of the two terminal states (yielding a total of two transitions). Every transition represents a rule such that it has associated constraints that determine whether the transition should be taken or not, given a certain input element. In our simple scenario with two transitions leading from the start node to each terminal node, none, one or both transitions could be taken, depending on the input element. To decide which transitions are made, every input element is compared sequentially to all corresponding constraints. In addition, as STAT sensors keep track of multiple scenarios in parallel, an input element has to be compared to all constraints of all currently active scenarios. No parallelism is exploited and even when multiple transitions have constraints that are identical or that are mutual exclusive, no optimization is performed and multiple comparisons are carried out. The same is true for the much simpler SWATCH system. All installed regular expressions (i.e., SWATCH rules) are applied to every log file entry to determine suspicious instances.

Some systems attempt to improve this process using *ad-hoc* techniques, but these optimizations are hard-wired into the detection engine and are not flexibly tailored to the set of rules which is actually used. A straightforward optimization approach is to divide the rule set into groups according to some criteria. The idea is that rules that specify a number of identical constraints can be put together into the same group. During detection, the common constraints of a rule group need only be checked once. When the input element matches these constraints, each rule in the group has to be processed sequentially. When the constraints are not satisfied by the input element, the whole group can be skipped.

This technique is utilized by the original version of Snort [14], arguably the most deployed signature-based network intrusion detection tool. Snort builds a two-dimensional list structure from the input rules. One list consists of `Rule Tree Nodes` (`RTNs`), the other one of `Option Tree Nodes` (`OTNs`). The `RTNs` represent rule groups and store the values of the group's common rule constraints (the source and destination IP addresses and ports in this case). A list of `OTNs` is attached to each `RTN` – these lists represent the individual rules of each group and hold the additional constraints that are not checked by the group constraints of the corresponding `RTN`.

In theory, Snort's two-dimensional list 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, the distribution of `RTNs` and `OTNs` is very uneven. The 1092 `TCP` and 82 `UDP` rules that are shipped with `Snort-1.8.7` and enabled by default are divided into groups as shown below in Table 1. The `Maximum`, `Minimum` and `Average` columns show the maximum, the minimum and the average number of rules that are associated with each rule group.

| Protocol | # Groups | # Rules | Maximum | Minimum | Average |
|---|---|---|---|---|---|
| UDP | 31 | 82 | 23 | 1 | 2.6 |
| TCP | 88 | 1092 | 728 | 1 | 12.4 |

**Table 1.** Statistics - Snort Data Structures

For `UDP`, 31 different groups are created from only 82 rules and each group has only three rules associated with it on average. This requires every input packet to be checked at least against the common constraints of all 31 groups. For `TCP`, more than half of the rules (i.e., 728 out of 1092) are in the single group that holds signatures for incoming `HTTP` traffic. Therefore, each legitimate packet sent to a web server 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 discriminating features, while the source port seems to be less important. However, valuable features such as `ICMP` code/type or `TCP` flags are not used and are checked sequentially within each group.

The division of rules into groups with common constraints is also used for packet filters and firewalls. Similar to Snort, the OpenBSD packet filter [6] combines rules with identical address and port parameters into skip-lists, moving on when the test for common constraints fails.

With the introduction of `Snort-2.0` [17] and its improved detection engine, the two-dimensional list structure and the strict sequential search within groups have been abandoned. The idea is to introduce more parallelism when checking rules, especially when searching the content of network packets for matching character strings. A rule optimizer attempts to partition the set of rules into smaller subsets which can then be searched in parallel.

The goal of the revised detection engine is similar to our decision trees in the sense that both systems attempt to partition the set of rules in smaller subsets where only a single subset has to be analyzed for each input element. The differences to our approach are the mechanism to select rule subsets and the extent of parallelism that is introduced. In `Snort-2.0`, rules are partitioned only based on at most two statically chosen constraints (source and destination port for `TCP` and `UDP`, type for `ICMP` packets). Within each group, a parallel search is only performed for content strings, while all other feature constraints are still evaluated sequentially. Our decision trees, on the other hand, dynamically pick the most discriminating features for a rule set and allow to perform parallel evaluation of every feature. This yields superior performance (as shown in Section 6), despite the fact that the detection engine of `Snort-2.0` is heavily tailored to the Snort rule set (which has many similar rules that only specify different content strings – and the content string is the only feature that can be evaluated in parallel).

Another system that uses 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 an existing set of signature rules as the basis for our decision model.

## 3 Rule Clustering

The idea of rule clustering allows a signature-based intrusion detection system to minimize the number of comparisons that are necessary to determine rules that are triggered by a certain input data element.

We assume that a signature rule specifies required values for a set of features (or properties) of the input data. Each of these features 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 corresponding 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 example, that a signature requires a feature of type `integer` to be less than a constant or inside a certain interval.

The basic 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 move to the next one, eventually determining every matching signature.

As described above, a popular *ad-hoc* optimization is implemented by considering certain features more important or discriminating than others and by checking on a combination of those first before considering the rest. This technique, which is, for example, used by the original Snort and the OpenBSD packet filter, bases on domain specific knowledge and still requires a number of comparisons that is about linear to the rule set size. 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. Although Snort has been recently released with an improved detection engine that addresses some of these issues, its parallelization efforts are limited to searching strings in packet payloads and the discriminating features are chosen based on domain knowledge. This limits the general applicability of the solution and forfeits potential gains by processing all features in parallel.

Similar to the revised detection engine of Snort, we attempt to mitigate the performance problem 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 this feature are put into the same subset. The clustering process is then performed recursively on all subsets until each subset contains only a single rule or there are no more features left to split the remaining rules into further subsets. In contrast to the Snort engine, our solution is applicable to different kinds of signature-based systems and not limited to input from the network. It requires no domain specific feature selection and is capable of performing parallel checks for all features.

## 4 Decision Tree

The subset structure obtained by the partitioning of the rule set can also be represented as a *decision tree*. Given this representation, the set that initially contains all rules can be considered as the tree's root node while its children are the direct subsets created by partitioning the rule set 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. An arrow that leads from a node to its child is annotated with the value of the feature that is specified by all the rules in this 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 the 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 address` 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 left to right, starting with the source address. 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 to continue detection at a single child node.

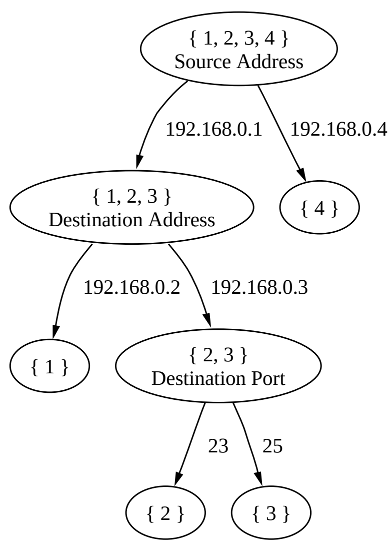


**Fig. 1.** Decision Tree

When the detection process eventually terminates at a leaf node, all rules associated with this node are *potential matches*. However, it might still be necessary to check additional features. To be precise, all features that are specified by the potentially matching rules but that have not been previously used by the clustering process to partition any node on the path from the root to this leaf must be evaluated at this point. 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 the ability 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 cannot 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 the matching process to exit immediately.

### 4.1 Decision Tree Construction

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.

The choice of the feature used to split a subset 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 levels - 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 of the impact of feature selection, consider the decision tree of Figure 2 which has been built from the same four rules introduced above. By using the destination port as the first selection feature, the resulting tree has a maximum depth of only two and consists of six nodes instead of seven.

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 (i.e., disorder) caused by partitioning the set using this attribute. The entropy of the partitioned data is calculated by weighting the entropy of each partition by its size relative to the

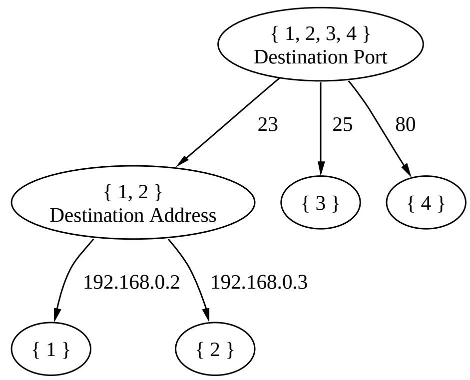


**Fig. 2.** Optimized Decision Tree

original set. The entropy $E_S$ of a set $S$ of rules is calculated by the following Formula 1.

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

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

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

The notion of entropy could be easily extended to incorporate domain specific know-ledge. Instead of assigning the same weight to each rule (that is, $\frac{1}{n}$ for each one of the $n$ rules), it is possible to give higher weights to rules that are more likely to trigger. This results in a tree that is optimized toward a certain, expected input.

Given the result about entropy in Formula 2 above, the information gain $G$ for a rule set $S$ and a feature $F$ can be derived as shown in Formula 3.

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

In this equation, $\text{Val}(F)$ represents the set of different values of feature $F$ that are specified by rules in $S$. Variable $v$ 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, creating a minimal decision tree that is consistent with a set of data is NP-hard.

### 4.2 Non-trivial Feature Definitions

So far, we have not considered the situation of a rule that 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 specifying the feature's whole value domain, we only consider the definition of multiple values. Notice that it is sometimes not possible to enumerate the value domain of a feature (such as floating point numbers) explicitly. This can be easily solved by specifying intervals instead of single values.

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 set and annotate the arrow with the value that the two have in common **and** additionally put the rule which defines multiple values into another set, labeling the arrow leading to that node with the value(s) that only that rules specifies.

Obviously, this basic idea can be extended to multiple rules with many overlapping definitions. The value domain of the feature 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 previously introduced 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. 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 successor nodes with the corresponding arrow labels. Notice that 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.

The total number of rules in all node's successors does not necessarily need to be equal to the number of rules in the ancestor node (as one might expect when a set is partitioned). This has effects on the size of the decision tree as well as on

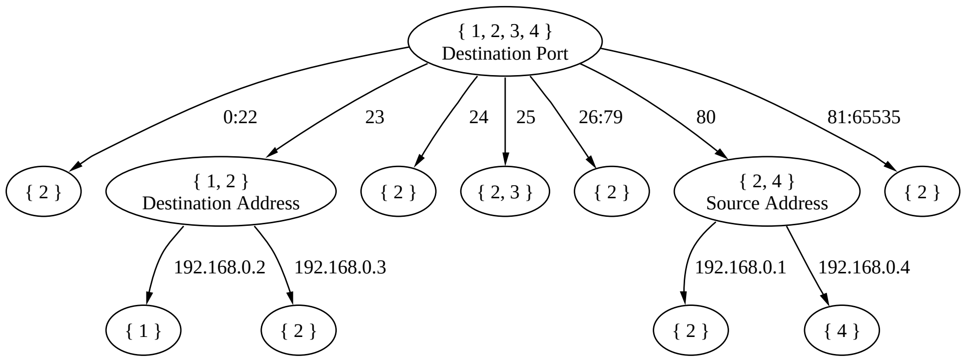


**Fig. 3.** Decision Tree with **any** Rule

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 can become large. To keep the size manageable, one can trade execution speed during the detection process for a reduced size of the decision tree. This is achieved by dividing the rule set into several subsets and building separate trees for each set. During detection, every input element has then to be processed by all trees sequentially. For our detection engine implementation, we have used this technique to manage the large number of Snort rules (see Section 6 for details).

The number of checks that each input element requires while traversing the decision trees is bound by the number of features, which is independent of the number of rules. However, our system is not capable of checking input data with a constant overhead independent of the rule set size. The additional overhead, which depends on 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 by choosing the arrow which matches the input data value. As the number of rules increases and the number of successor nodes grows, this check becomes more expensive. Nevertheless, the comparison can be made more efficient than an operation with a cost linear (i.e., $O(n)$) in the number of elements $n$.

## 5 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 bits in a fixed length bit array and is needed to handle the flag fields of 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 successor 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 `bitfield` types, the different successor values are stored in a tree with a depth that is bound by the length of the addresses or the bitfields, 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 attempting to determine the intersections of the string property specifications of a rule set during the partition process, it is necessary to assume that the input can contain 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. This is clearly undesirable. We tackle this problem by requiring 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 enumerate all possible combinations and keep 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. But often, the same input string has to be scanned repeatedly because multiple rules all define different keywords.

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. As a result, the matching process generates a number of redundant comparisons that emerge where the Boyer-Moore algorithm is applied multiple times on the same input string trying to find nearly similar keywords. The authors propose to use a variation of the Aho-Corasick [1] tree to match several strings with a common prefix in parallel and reduce overhead. Unfortunately, the approach is only suitable when keywords share a common prefix. When creating the decision tree following our approach, it often occurs that several signatures that specify different strings end up in the same node. They do not necessarily have anything in common. Instead of invoking the Boyer-Moore algorithm for each string individually, we use an efficient, parallel string matching implementation introduced by Fisk and Varghese [5]. This algorithm has the advantage that it does not require common prefixes and delivers good performance for medium sized string sets (containing a few up to a few dozens elements).

In the Fisk-Varghese approach, hash tables are utilized to reduce the number of strings that need to be evaluated on an expensive character-by-character basis when a partial match between the rule strings and the input string is detected. However, when a few hundred strings are compared in parallel, some hash table buckets can contain so many elements that the efficiency is negatively effected. This is solved by selectively replacing hash tables by tries when a hash table bucket contains a number of elements above a certain, definable threshold (the default value is 8).

A trie is a hierarchical, tree like data structure that operates like a dictionary. The elements stored in the trie are the individual characters of 'words' (which are, in our case, the string features of the individual rules). Each character of a word is stored at a different level in the trie. The first character of a word is stored at the root node (first level) of the trie, together with a pointer to a second-level trie node that stores the continuation of all the words starting with this first character. This mechanism is recursively applied to all trie levels. The number of characters of a word is equal to the levels needed to store it in a trie. A pointer to a leave node that might hold additional information marks the end of a word.

When a partial match is found by the detection process, the trie is utilized to perform the expensive character-by-character search for all string candidates in parallel. It is no longer necessary to sequentially match all words of a hash bucket against the input string (as with the Fisk-Varghese approach).

Although tries would be beneficial in all cases, we limit their use to the replacement of large hash tables only because of the significant increase in memory usage.

## 6 Experimental Data

This section presents the experimental data that we have obtained by utilizing decision trees to replace the detection engine of Snort. We have implemented patches named `Snort NG` (next-generation) for `Snort-1.8.6` and `Snort-1.8.7` that can be downloaded from [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. Our performance results are directly compared to the results obtained with the latest version of Snort and its improved detection engine, that is `Snort-2.0rc1`.

For our first experiment, we set up `Snort-2.0` and our patched `Snort NG` with decision trees on a `Pentium IV` with 1.8 GHz running a `RedHat Linux 2.4.18` kernel. Both programs read `tcpdump` log files from disk and attempted to process the data as fast as possible. When performing the measurements, most preprocessors have been disabled (except for `HTTP`-decoding and `TCP` stream reassembling) and only fast-logging was turned on to have our results reflect mostly the processing cost of the detection algorithms themselves. Obviously, the overhead of the operating system to read from the file and the parsing functionality of Snort still influences the numbers, but it does so for both approaches.

We measured the total time that both programs needed to complete the analysis of our test data sets. For each of these data sets, we performed ten runs and averaged the results. For the experiment, the maximum number of 1581 `Snort-2.0` rules that were available at the time of testing have been utilized. As `Snort NG` bases on `Snort-1.8.7` that uses a rule language incompatible to `Snort-2.0`, all rules have been translated into a suitable format. Both programs were executed consecutively and did not influence each other while running.

We used the 'outside' tcpdump files of the ten days of test data produced by MIT Lincoln Labs for their 1999 DARPA intrusion detection evaluation [8]. These files have different sizes that range from 216 MB to 838 MB. The comparison of the results for the ten days of the MIT/LL traffic is shown in Figure 4. For each test set, both systems reported the same alerts. Although the actual performance gain varies considerably depending on the type of analyzed traffic (as `Snort-2.0` is tuned to process `HTTP` traffic), the decision trees performed better for every test case and yielded an average speed up of 40.3%. The maximum speed up seen during the tests was 103%, and the minimum was 5%.

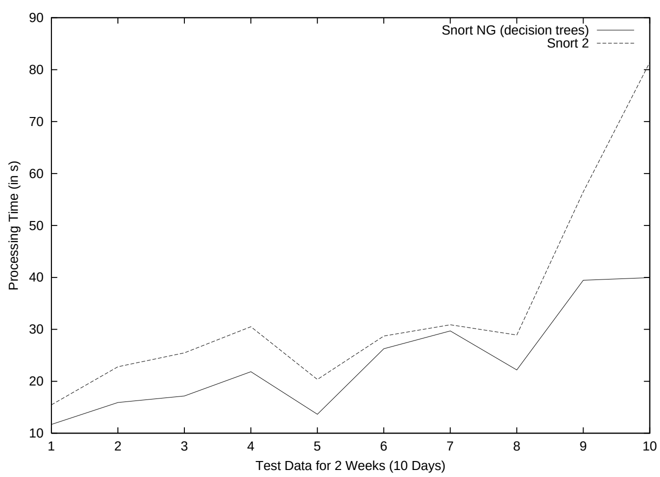


**Fig. 4.** Time Measurements for 1999 MIT Lincoln Lab Traffic

The second experiment used the same setup as the first one. This time, however, the number of rules were increased (starting from 150 up to the maximum of 1581) while the input file was left the same (we used the first day of the 1999 MIT Lincoln Labs data set). The rules were added in the order implied by the default rule set of `Snort-2.0`. All default rule files were sorted in alphabetical order and their rules were then concatenated. From this resulting list, rules were added in order. Similar to the previous test, both programs reported the same alerts for all test runs. Figure 5 depicts the time it took both programs to complete the analysis of the test file given the increasing number of input rules. The graph shows that the decision tree approach performs better, especially for large rule sets.

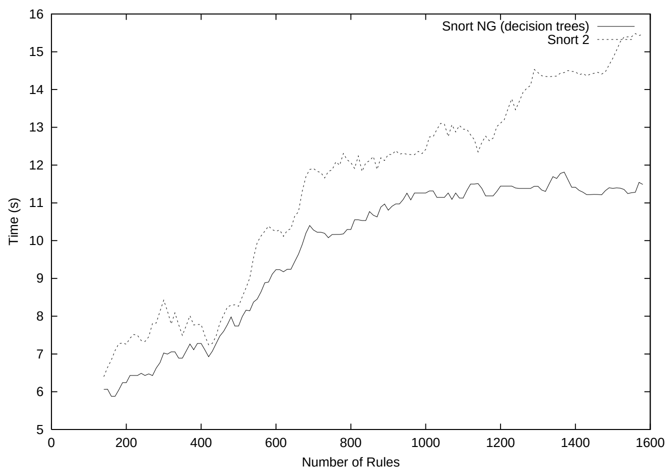


**Fig. 5.** Time Measurements for increasing Number of Rules

Building the decision tree requires some time during start up and increases the memory usage of the program. 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 internal 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 (as needed for the MIT/LL data), the complete rule set used for our evaluation is transformed into 2398 rule instances that need to be processed internally. As a single tree would be too large for this amount of rules, the detection engine splits the rule set for each supported protocol into two subsets and builds two separate trees.

Figure 6 shows the total memory consumption of the patched version of Snort for increasing amounts of rules. It indicates that even when the complete set of rules is loaded, the memory demands are reasonable given todays main memory sizes. The time to build the tree (including the case for the maximum number of rules) has never exceeded 12 seconds.

Notice the interesting irregularity that Figure 6 shows for the modified version of Snort around rule number 700. The reason is a change in the shape of the decision tree. Given the tree for the previous rules and adding a single additional one, the ID 3 algorithm creates a tree which has the same height but is much broader. It contains noticeable more nodes (mostly due to copied rule instances with unspecified feature values) and therefore consumes more memory. However, additional rules fit well into the resulting tree structure and the detection time does not increase significantly after that as more rules are added (as can be seen in Figure 5).

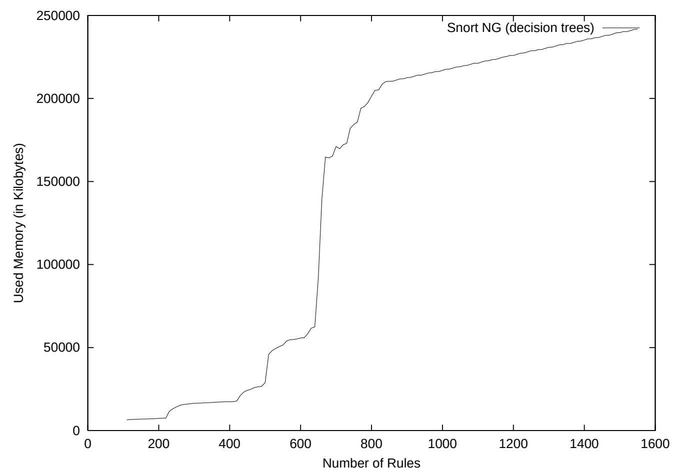


**Fig. 6.** Memory Consumption

## 7 Conclusion

Signature-based intrusion detection systems face the challenge of an constantly 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 relieve 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-input comparison process. We describe an application of our mechanism to Snort, the most popular open-source network intrusion detection system, and show that a significant improvement of its processing speed was possible. Decision trees, however, are a general solution that can be of benefit to other intrusion detection systems (host- and network-based), packet filters and firewalls as well.

## Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, or the U.S. Government.

## 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/warp/public/cc/pd/sqsw/sqidsz/index.shtml`, 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. M. Fisk and G. G. Varghese. An analysis of fast string matching applied to content-based forwarding and intrusion detection. Technical Report UCSD TR CS2001-0670, UC San Diego, 2001.
6. Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *USENIX Annual Technical Conference – FREENIX Track*, 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`, 1999.
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`.
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. Sourcefire. Snort 2.0. `http://www.sourcefire.com/technology/whitepapers.htm`.

18. Swatch: Simple Watchdog. http://swatch.sourceforge.net.
19. Symantec - NetProwler and Intruder Alert. http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=%50, 2002.
20. 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.

# 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`). In this mode, our system imitates the original reporting behavior of Snort.

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 afterward 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 1581 rules that are stored in 47 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;
     classtype:attempted-recon; sid:624; rev:1;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"SCAN synscan portscan";
     id: 39426; flags: SF; 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+; sid:1251; )
alert tcp $HOME_NET 23 -> $EXTERNAL_NET any (msg:"TELNET login incorrect";
     content:"Login incorrect"; flags:A+; sid:718; rev:5;)

alert tcp $HOME_NET 146 -> $EXTERNAL_NET 1024: (msg:"BACKDOOR Infector";
     content: "WHATISIT"; flags: A+; sid:117; )
alert tcp $HOME_NET 146 -> $EXTERNAL_NET 1000:1300 (msg:"BACKDOOR Infector
     to Client"; content:"|57 48 41 54 49 53 49 54|"; flags:A+; sid:120;)
```

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

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 (when desired).