WIND: Workload-aware INtrusion Detection

Sushant Sinha, Farnam Jahanian, and Jignesh M. Patel
Electrical Engineering and Computer Science,
University of Michigan,
Ann Arbor, MI-48109
{sushant, farnam, jignesh}@umich.edu

Abstract. Intrusion detection and prevention systems have become essential to the protection of critical networks across the Internet. Widely deployed IDS and IPS systems are based around a database of known malicious signatures. This database is growing quickly while at the same time the signatures are getting more complex. These trends place additional performance requirements on the rule-matching engine inside IDSs and IPSs, which check each signature against an incoming packet. Existing approaches to signature evaluation apply statically-defined optimizations that do not take into account the network in which the IDS or IPS is deployed or the characteristics of the signature database. We argue that for higher performance, IDS and IPS systems should adapt according to the workload, which includes the set of input signatures and the network traffic characteristics. To demonstrate this idea, we have developed an adaptive algorithm that systematically profiles attack signatures and network traffic to generate a high performance and memory-efficient packet inspection strategy. We have implemented our idea by building two distinct components over Snort: a profiler that analyzes the input rules and the observed network traffic to produce a packet inspection strategy, and an evaluation engine that pre-processes rules according to the strategy and evaluates incoming packets to determine the set of applicable signatures. We have conducted an extensive evaluation of our workload-aware Snort implementation on a collection of publicly available datasets and on live traffic from a border router at a large university network. Our evaluation shows that the workload-aware implementation outperforms Snort in the number of packets processed per second by a factor of up to 1.6x for all Snort rules and 2.7x for web-based rules with reduction in memory requirements. Similar comparison with Bro shows that the workload-aware implementation outperforms Bro by more than six times in most cases.

Keywords: Intrusion detection and prevention, deep packet inspection, workload aware, adaptive algorithm

1 Introduction

New critical software vulnerabilities are a common occurrence today. Symantec documented 1,896 new software vulnerabilities from July 1, 2005 to December 31, 2005, over 40% more than in 2004 [1]. Of these, 97% were considered moderately or highly severe, and 79% were considered easy to exploit. To address this rapid increase in vulnerabilities, organizations around the world are turning to

Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) to detect and prevent attacks against networked devices.

The core component of popular IDSs, like Snort [2], is a deep packet inspection engine that checks incoming packets against a database of known signatures (also called rules). The performance of this signature-matching system is critical to the scalability of IDS and IPS systems, including packet per second rate. The dominant factor in determining the performance of this signature matching engine, whether implemented in software or hardware, is the number and complexity of the signatures that must be tested against incoming packets. However, both the number and complexity of rules appears to be increasing. For example, the recent Windows Meta-File (WMF) exploit [3] required inspecting and decoding more than 300 bytes into the HTTP payload which could quickly overwhelm the CPU of the IDS or IPS, causing massive packet drops [4].

As a result, there has been significant effort in developing methods for efficient deep packet inspection. Current IDSs like Snort and Bro attempt to evaluate as few rules as possible in a highly parallel way. For example, Snort pre-processes rules to separate them by TCP ports, and then parallelizes the evaluation based on port. However, these groupings can be inefficient because all of the rules in a given group do not apply to incoming packets. Moreover, separating rules by multiple protocol fields in a naive way does not solve the problem because of the additional memory overhead associated with managing groups.

In this paper, we argue that IDS and IPS should dynamically adapt the parallelization and separation of rules based on the observed traffic on the network and the input rules database. That is, all IDS and IPS workloads are not the same, and systems should adapt to the environment in which they are placed to effectively trade-off memory requirements for run-time rule evaluation. To demonstrate this idea, we have developed an adaptive algorithm that systematically profiles the traffic and the input rules to determine a high performance and memory efficient packet inspection strategy that matches the workload. To effectively use memory for high performance, the rules are separated into groups by values of protocol fields and then these rule groups are chosen to be maintained in memory following a simple idea of "the rule groups that have a large number of rules and match the network traffic only a few times should be separated from others." This idea follows our observation that if rules with value v for a protocol field are grouped separately from others, then for any packet that does not have value v for the protocol field, we can quickly reject all those rules, and if only a few packets have that value, then those rules will be rejected most of the time. Therefore, our workload-aware scheme aims to determine a small number of effective groups for a given workload.

Our algorithm determines which rule groups are maintained in the memory by choosing protocol fields and values recursively. It first determines the protocol field that is most effective in rejecting the rules, and then separates those groups with values of the chosen protocol field that reject at least a threshold number of rules. After forming groups for each of these values, the algorithm recursively splits the groups by other protocol fields, producing smaller groups. In this way,

we generate a hierarchy of protocol fields and values for which groups are maintained. By lowering the threshold, memory can be traded-off for performance. Using this systematic approach for computing a protocol evaluation structure, we automatically adapt an IDS for a given workload.

In this paper we develop a prototype Snort implementation based on our workload-aware framework, which we call Wind. The implementation has two main components. The first component profiles the workload (i.e., the input rules and the observed network traffic) to generate the hierarchical evaluation tree. The second component takes the evaluation tree, pre-processes the rules, and matches incoming packet to the rules organized in the tree.

We evaluate our prototype workload-aware Snort implementation on the widely recognized DARPA intrusion detection datasets, and on live traffic from a border router at a large live academic network. We find that our workload-aware algorithm improves the performance of Snort up to 1.6 times on all Snort rules and up to 2.7 times for web-based rules. Surprisingly, we also find that the algorithm reduces memory consumption by $10 - 20\%$. We also compare the workload-aware algorithm with Bro, and find it outperforms Bro by more than six times on most workloads.

To summarize, the main contributions of this paper are:

- We propose a method for improving the performance of IDS and IPS systems by adapting to the input rules and the observed network traffic.
- To demonstrate our idea, we constructed a workload-aware Snort prototype called Wind that consists of two components: a component that profiles both the input rules and the observed network traffic to produce an evaluation strategy, and a second component that pre-process the rules according to the evaluation strategy, and then matches incoming packets.
- We evaluate our prototype on publicly-available datasets and on live traffic from a border router. Our evaluation shows that Wind outperforms Snort up to 1.6 times and Bro by six times with less memory requirements.

The rest of the paper is organized as follows: Section 2 presents background and related work. Section 3 presents the design of Wind, and Section 4 presents empirical results comparing Wind with existing IDSs. Section 5 discusses techniques for dynamically adapting Wind to changing workloads. We finally conclude with directions for future work in Section 6.

2 Background and Related Work

The interaction between high-volume traffic, number of rules, and the complexity of rules has created problems for Intrusion Detection Systems that examine individual flows. Dreger *et. al.* [5] present practical problems when Intrusion Detection Systems are deployed in high-speed networks. They show that current systems, like Bro [6] and Snort [2], quickly overload CPU and exhaust the memory when deployed in high-volume networks. This causes IDS to drop excessive number of packets, some of which may be attack incidents. Therefore, they propose some optimizations to reduce memory consumption and CPU usage which are orthogonal to the Wind approach.

Lee *et. al.* [7] find that it is difficult to apply all possible rules on an incoming packet. Therefore, they evaluated the cost-benefit for the application of various rules and determined the best set of rules that can be applied without dropping packets. However, they trade-off accuracy for achieving high bandwidth. Kruegel and Valeur [8] propose to slice traffic across a number of intrusion detection (ID) sensors. The design of their traffic slicer ensures that an ID sensor configured to apply certain rules on a packet does not miss any attack packet.

Sekar *et. al.* [9] developed a high-performance IDS with language support that helps users easily write intrusion specifications. To specify attack signatures within a payload, they used regular expressions. This specification is different from Snort in which attack signatures contain exact substrings, in addition to regular expressions, to be matched with a payload. Using regular expressions is a more generic approach than using substrings to specify an attack signature. However, regular expressions are more expensive to evaluate than exact substring matches. (The complexity of checking a regular expression of size m over a payload of size n is $O(mn)$ [10] and it is more expensive than checking for exact substring within a payload, which has a time complexity of $O(n)$ [10]). Aho-Corasick [10] matches a *set* of substrings over a payload in $O(n)$. Alternative schemes like Wu-Manber [11] speed up matching by processing the common case quickly. The multi-pattern optimizations to speed up an Intrusion Detection System are complementary to our approach, as we speed up an IDS by reducing the expected number of patterns to be checked with a packet.

Versions of Snort prior to 2.0 evaluated rules one by one on a packet. This required multiple passes of a packet and the complexity of intrusion detection grew with the number of rules. To eliminate redundant checking of protocol fields, rules that have the same values for a protocol field can be pre-processed and aggregated together. Then, a check on the protocol field value would equivalently check a number of rules. By clustering rules in this way and arranging the protocol fields by their entropy in a decision tree, Kruegel and Toth [12], and Egorov and Savchuk [13] independently demonstrated that Snort (version 1.8.7) performance can be improved up to three times. However, these papers only examined the input rules to determine the rule evaluation order. In contrast, we analyze the traffic, as well as the rules, to determine the rule evaluation order. Secondly, they use entropy as an ordering metric, whereas we use a more intuitive metric for selecting as few rules as possible. Lastly, a naive arrangement of protocol fields would drastically increase memory usage, and these papers have not considered the memory costs associated with their approaches. Wind improves performance and at the same time reduces the memory usage of an intrusion detection and prevention system.

Snort 2.0 [14] uses a method in which rules are partitioned by TCP ports, and a packet's destination and source port determines the sets of applicable rules. Then, the content specified by these applicable rules are checked in one pass of the payload, using either the Aho-Corasick or the Wu-Manber algorithm, for multiple substring search. If a substring specified in some attack rule matches with the packet, then that rule is evaluated alone. We found that the parallel

evaluation significantly sped up Snort. Snort now takes 2-3 microseconds per packet, when compared to earlier findings of 20-25 [13] microseconds per packet for Snort versions prior to 2.0 [1]. This optimization significantly improved Snort performance. Nevertheless, we further speed up a multi-rule Snort on many workloads. This is achieved by partitioning the rules in an optimized evaluation structure.

Recently, specialized hardware [15, 16] for intrusion detection in high-volume networks has been developed. However, hardware-based solutions are complex to modify (e.g., to change the detection algorithm). Nevertheless, the techniques presented here will further enhance performance of these systems.

Our work is also related, and inspired, by database multi-query optimization methods that have long been of interest to the database community (see [17–20] for a partial list of related work). However, rather than finding common subexpressions amongst multiple SQL queries against a static database instance, the problem that we tackle requires designing a hierarchical data structure to group network rules based on common subexpressions, and using this data structure in a data streaming environment.

3 Designing a Workload-Aware IDS

In this section, we first show that checking a protocol field can reject a large number of rules, and the number of rejected rules varies significantly with the protocol field. Then, we take this observation a step further and construct an evaluation strategy that decomposes the set of rules recursively by protocol fields and constructs a hierarchical evaluation tree. However, a naive strategy that separates rules by all values of a protocol field will use too much memory. To address this issue, we present a mathematical model that addresses the trade-off between memory occupied by a group of rules and the improvement in run-time packet processing. Finally, we present a novel algorithm and a concrete implementation to capture statistical properties of the traffic and the rule set to determine a high-performance and memory-efficient packet inspection strategy.

3.1 Separating rules by protocol fields

An IDS has to match a large number of rules with each incoming packet. Snort 2.1.3 [2] is distributed with a set of 2,059 attack rules. A rule may contain specific values for protocol fields and a string matching predicate over the rest of the packet. For example, a Snort rule that detects the Nimda exploit is shown below:

```
alert tcp EXTERNAL_NET any -> HOME_NET 139 (msg:‘‘NETBIOS nimda
          .nws’’; content:‘‘|00|.|00|N|00|W|00|S";)
```

This rule matches a packet if the value of transport protocol field is TCP, the value in the source address field matches the external network, the destination address field contains an address in the home network, the value of destination TCP port field is 139, and if the payload contains the string `‘‘|00|.|00|N|00|W|00|S"`.

[1] difference in computing systems and rules not taken into account for rough discussion

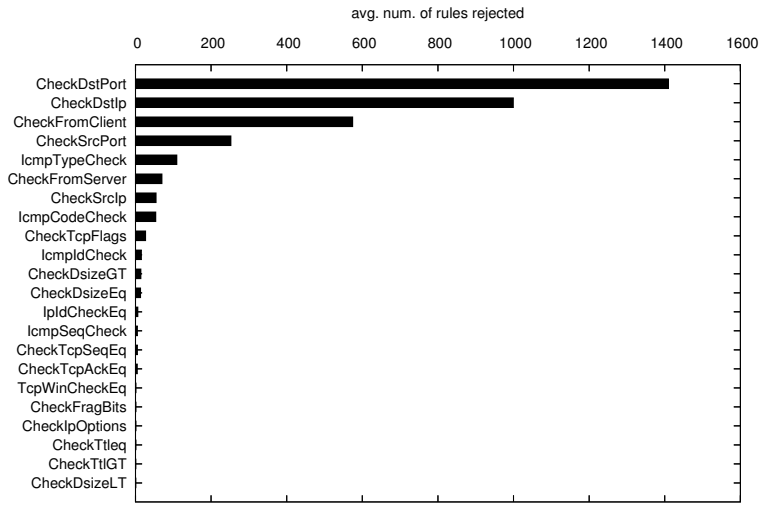

Fig. 1. Average number of rules (out of 2,059) rejected by checking different protocol fields for the DARPA dataset (99-test-w4-thu).

A simple approach for evaluating multiple rules on an incoming packet is to check each rule, one-by-one. However, this solution involves multiple passes over each packet and is too costly to be deployed in a high-speed network. Therefore, the evaluation of the rules should be parallelized as much as possible and evaluated in only a few passes over the packet. To evaluate a protocol field in the packet only once, we need to pre-process rules and separate them by the values of the protocol field. Then, by checking the value of just one protocol field, the applicable rules can be selected. The advantage of separating rules by the protocol field values is that a large number of rules can be rejected in a single check. In Snort, the rules are pre-processed and grouped by destination port and source port. The TCP ports of an incoming packet are checked to determine the set of rules that must be considered further, and all other rules are immediately rejected. The expected number of rules that will be rejected by checking a protocol field of an incoming packet depends on two factors: the traffic characteristics and the rule characteristics. Consider an input rule set with a large number of rules that check if the destination port is 80. Assuming that the rules are grouped together by the destination port, for a packet not destined to port 80, a *large* number of port-80 rules will be rejected immediately. If only a few packets are destined to port 80, then a *large* number of rules will be rejected *most* of the time.

Figure 1 shows the number of rules that can be rejected immediately for an incoming packet when rules are grouped by different protocol fields. For this figure, we used the 2,059 rules that came with Snort 2.1.3 distribution and the traffic is from the Thursday on the fourth week of 99 DARPA dataset (99-test-w4-thu). Figure 2 shows a similar graph, using the same set of rules on a border router in a large academic network. The graphs show that checking the destination port rejects the maximum number of rules, which is followed

by destination IP address and then by the check that determines whether the packet is from a client. The source IP address is fourth in the list for the border router traffic and seventh in the DARPA dataset. After this, most other protocol fields reject a small number of rules. Therefore, the graphs show that the rule set and the traffic mix cause varying number of rules to be rejected by different protocol fields.

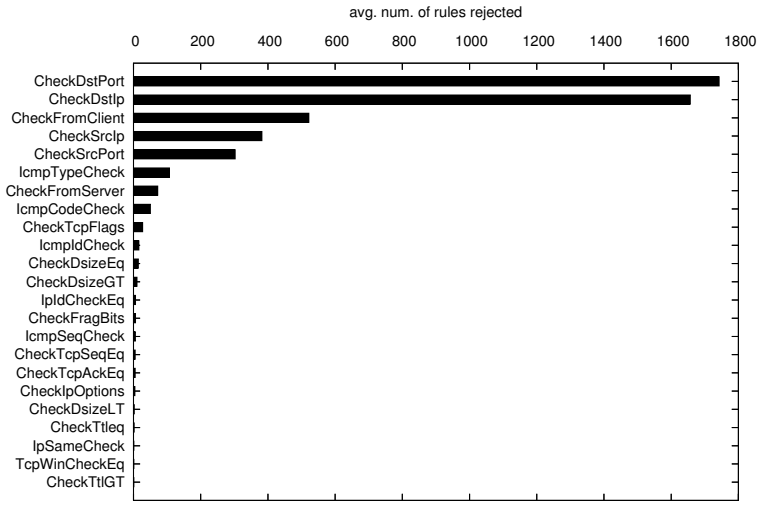

Fig. 2. Average number of rules (out of 2,059) rejected by checking different protocol fields for data from the border router of a large academic network.

Now, checking whether a payload contains a particular string is a costly operation, but checking the value of a protocol field is cheap. So, it is preferable to check protocol fields to reduce the number of applicable rules. To use multiple protocol fields for reducing the applicable rules, the rules have to be pre-processed in a hierarchical structure in which each internal node checks a protocol field and then divides the rules by the values of the protocol field. Finally, the leaf node is associated with a set of rules and a corresponding data structure for evaluating multiple patterns specified in the rules. We are agnostic to the multi-pattern search algorithm and the only objective of this hierarchical evaluation is to reduce the number of applicable rules, so that a packet is matched with as few rules as possible.

Figure 3 shows an example of an evaluation tree in which protocol fields are hierarchically evaluated to determine the set of applicable rules. It first checks for destination port. If the destination port matches a value for which the set of rules is maintained then those groups of rules are further analyzed, or else the generic set of rules is picked. If the destination port is 21, the connection table is checked to determine if the packet came from the client who initiated the connection, and the corresponding rules are picked. If the destination port is 80, then the destination IP address of the packet is checked. Then, depending

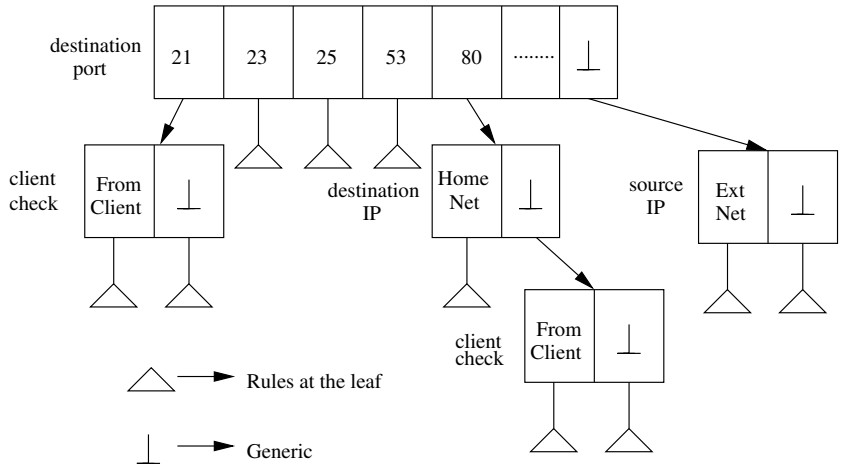

Fig. 3. An example evaluation tree that checks protocol fields to determine the set of rules for matching an incoming packet.

on whether the packet is destined to the Home Network or not, the correct set of rules are picked to further evaluate on the packet. However, maintaining a naive hierarchical index structure, in which every specific value of a protocol field is separated, consumes a significant amount of memory for the following two reasons:

1. **Groups require memory:** Multiple patterns from a set of rules have to be searched in a payload in only one pass of the payload. Therefore, additional data structures are maintained for fast multi-pattern matching. This structure can be a hash table as in the case of the Wu-Manber [11] algorithm, or a state table as in the case of the Aho-Corasick [10] algorithm. These structures consume a significant amount of memory.
2. **Rules are duplicated across groups:** If groups are formed by composing two protocol fields hierarchically, then the number of distinct groups may increase significantly. For example, assume that the rules are first divided by destination port, and then each group so formed is further divided by source port. A rule that is specific in source port but matches *any* destination port has to be included in all groups with a particular destination port. If the groups that are separated by destination port are further divided by source port, then separate source port groups would be created within all the destination port groups. For a set of rules with n source port groups and m destination port groups, the worst number of groups formed, when rules are hierarchically arranged by the two protocol fields, is $n \times m$.

To investigate the memory consumed when rules are grouped hierarchically by different protocol fields, we instrumented Snort to construct this structure for a given list of protocol fields. We then measured the memory consumed for different combinations of protocol fields. Figure 4 shows the memory consumed when different protocol fields are hierarchically arranged, and a separate bin is

maintained for every specific value in a protocol field (trace data was 99-w4-thu from DARPA dataset and the 2059 rules of Snort-2.1.3 distribution). This shows that the memory consumed by the combination of destination port and client check is 50% more than just the destination port. The memory required for the combination of destination port and destination IP address is two times, and for the combinations of destination port, destination IP address and client check, the memory consumed is three times than only using the base destination port. From the graph, the increase in memory is evident when the rules are hierarchically grouped by destination port and source port. Therefore, constructing such a hierarchy immediately raises two important questions:

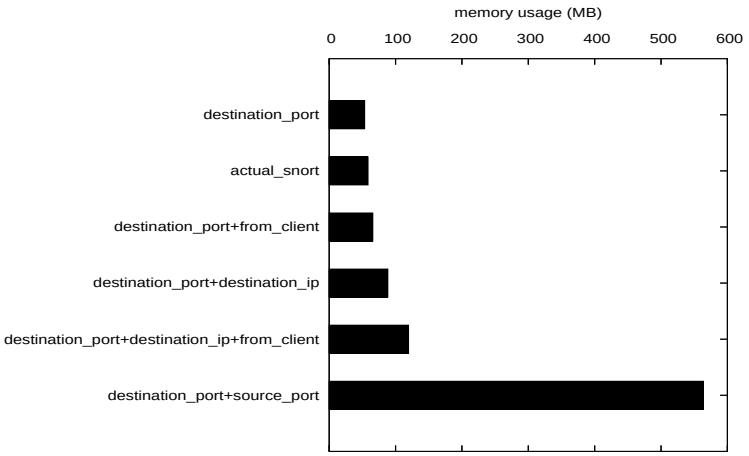

Fig. 4. Memory usage when rules are hierarchically arranged by protocol fields in the specified order using the DARPA dataset (99-w4-thu).

1. What is the order in which the protocol fields are evaluated in the hierarchy?
2. What are the values of a protocol field for which groups are maintained?

In what follows, we first present a mathematical description of this problem, analyzing the cost and benefit of different orders of protocol fields and the field values for which the rules are maintained. We then argue that these questions can be answered by capturing properties of the workload, namely the traffic-mix characteristics and the input rule set characteristics.

3.2 Formal description

In this section, we formulate the problem of determining the order of evaluation and the values of protocol fields for which the groups are maintained. As argued earlier, the cost in maintaining a separate group is mostly the memory consumed by the group. Intuitively, the benefit obtained by maintaining a group of rules can be measured by how *many* rules this group separates from the rule set

and how *frequently* this group is rejected for an incoming packet. We begin by formalizing the problem.

Consider n protocol fields $F_1, F_2, \ldots, F_n$. Let $v_1^i, v_2^i, \ldots, v_{m_i}^i$ be m_i specific values of the protocol field F_i present in various rules in the rule set. Let $\mathcal{P} = (F_{r_1} = v_{j_1}^{r_1}) \wedge (F_{r_2} = v_{j_2}^{r_2}) \wedge \ldots \wedge (F_{r_i} = v_{j_i}^{r_i})$ be the predicate for a group that is picked only when the packet matches specific values for i protocol fields, and $f(\mathcal{P})$ denote the probability that the protocol fields for an incoming packet matches the predicate $\mathcal{P}$, i.e., $v_{j_1}^{r_1}$ for protocol field F_{r_1}, $v_{j_2}^{r_2}$ for protocol field F_{r_2}, $\ldots$, $v_{j_i}^{r_i}$ for protocol field F_{r_i}. $f(\mathcal{P})$ actually captures statistics on the network traffic. The probability that an incoming packet does not have the values for protocol fields as the predicate $\mathcal{P}$ is $1 - f(\mathcal{P})$. Let the benefit of rejecting rule R be measured by improvement of b_R in run time. Then, every time a packet does not have the values for protocol fields as $\mathcal{P}$, benefit of $\sum_{R=rule\ with\ value\ \mathcal{P}} b_R$ is obtained by maintaining a separate group of rules with values $\mathcal{P}$. Therefore, the overall benefit of creating a group with specific values for protocol fields present in the predicate $\mathcal{P}$ includes traffic characteristics in $f(\mathcal{P})$ and rule properties in the rule set as:

$$(1 - f(\mathcal{P})) \times \sum_{R=rule\ with\ value\ \mathcal{P}} b_R \tag{1}$$

Assume that $c(\mathcal{P})$ is the memory cost of creating a group for a set of rules that satisfies the predicate $\mathcal{P}$. Then, the problem of an effective hierarchical structure is to determine the set of groups such that they maximize the benefit measured by improvement in run time for a given total cost, measured by the total amount of memory that is available. Formally, the objective is to determine m and m distinct predicates $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$ that maximizes

$$\sum_{i=1}^{m} [(1 - f(\mathcal{P}_i)) \times \sum_{R=rule\ with\ value\ \mathcal{P}_i} b_R] \tag{2}$$

with the cost constraint

$$\sum_{i=1}^{m} c(\mathcal{P}_i) \leq maximum\ memory \tag{3}$$

3.3 Our approach

In this section, we design an algorithm that captures the properties of input rules and traffic characteristics to produce an effective set of rule groups, separated by values of protocol fields. These groups are then arranged in a hierarchical evaluation structure, which determines the order in which protocol fields are evaluated on an incoming packet. We begin with some assumptions that simplify the above mathematical model for a realistic treatment and then present our algorithm.

Assumptions It is not easy to precisely determine the cost of creating a data structure for matching multiple patterns and the absolute benefit achieved by rejecting a rule. For some exact substring-match algorithms (like Aho-Corasick), the memory space occupied by the data structure may not grow linearly with the number of patterns. For hash-based algorithms, the memory consumed is independent of the number of patterns. This makes estimating cost for a multi-pattern-matching algorithm difficult. At the same time, for most algorithms that perform multi-pattern matching, it is hard to estimate the benefit of excluding a single pattern. Therefore, we will make two simplifying assumptions that allows us to easily compute the cost and benefit:

1. The cost of creating a multi-pattern data structure for any group of patterns is constant. This assumption is valid for hash-based matching algorithms, like Wu-Manber, that allocate fixed hash space. However, this assumption is incorrect for the Aho-Corasick algorithm in which the required space may increase with the increase in the number of patterns.
2. The benefit of rejecting any rule is a one-unit improvement in run time (i.e., $b_R = 1$) except for rules that have content of maximum length one. The rules that have content length one significantly degrade multi-pattern matching and should be separated if possible. Therefore, rules with a content length of one are assigned a large benefit (mathematically infinity). It is possible that other patterns may adversely impact the performance in multi-pattern search, but we choose to ignore such interactions for simplicity.

It is important to note that our assumptions help us to easily estimate the cost and benefit of creating a group, and more accurate estimates will only improve our scheme.

The algorithm Instead of specifying a fixed memory cost and then maximizing the benefit, we specify the trade-off between the cost and the benefit. We say that any specific value of a protocol field that rejects at least a minimum THRESHOLD number of rules should be assigned a separate group and hence, memory space. This specification allows us to more easily tune real-time performance.

The mathematical model allows us to compare two groups with specific values for a number of protocol fields. The problem is then to determine a set of groups in which each group rejects at least a THRESHOLD number of rules and the set maximizes the overall benefit. However, this may require generating all possible sets, which is computationally infeasible. Therefore, we do not attempt to produce an optimal set of groups, but instead to discover possible groups heuristically. The main intuition behind our algorithm is to place all rules in a bin and iteratively split that bin by the protocol field that produces the maximum benefit, and at each split separate values of the chosen protocol field that reject at least a THRESHOLD number of rules on average.

We now explain our algorithm in detail. First, all rules are placed in a bin. Then, a few packets are read from the network and protocol fields in each rule are evaluated. Then, the benefit obtained by a value in a protocol field is computed using the benefit Equation 1. For value v_j^i of the protocol field F_i, $f(\mathcal{P})$ reduces

to $f(F_i = v_j^i)$ and $\sum_{R=rule\ with\ value\ \mathcal{P}} b_R$ reduces to $S_{F_i=v_j^i}$ where $S_{\mathcal{P}}$ indicates the number of rules with protocol field values specified by the predicate $\mathcal{P}$. This simplification is possible because b_R is one. The overall benefit of a protocol field is the sum of benefit of all values, and the protocol field is chosen that produces maximum benefit. Then groups are formed for each specific value in the protocol field that rejects at least THRESHOLD number of rules, or has a rule with content length one. Then, we partition the bin into those specific values and recursively compute other protocol fields for each of these bins. We stop splitting a bin if none of the protocol fields can reject at least THRESHOLD number of rules.

When we partition a bin into specific values, we replicate a rule that may match multiple of these specific values in all those bins. For example, if the rules are divided by destination port, then a rule that matches `'any'` destination port is included in all of those bins. This ensures that when a set of rules with a specific value for a protocol field are picked, other applicable rules are also matched with the packet. This is essential for correctness. Generally a rule with value v_j for a protocol field is included in a rule set with specific value v_i if $v_j \cap v_i \neq 0$. If there is an order in which the values are checked during run-time, then a rule v_j is included in v_i only if it appears before it, and if it satisfies the previous property.

Packets rejected by a protocol field may correlate with packets rejected by another protocol field, and so computing protocol fields independently may give misleading information. For example, a source port and a source IP address may reject exactly the same packets, in which case we do not gain anything by checking both of them. Our recursive splitting of a bin removes this problem of correlated values. This is because for a bin, we evaluate the benefit of remaining protocol fields only on those packets that match the values specified in the bin. For example, to split a bin containing port-80 rules, we only evaluate the remaining protocol fields on packets that have port 80. This ensures that the remaining protocol fields reject only the rules that were not rejected by port 80.

By choosing the protocol field that produces maximum benefit for each bin, we get an order in which the protocol field is checked for a packet. By choosing values that produce benefit above a threshold, we get the values that determines which groups should be maintained.

Implementation: We implemented two distinct components to develop a workload-aware Intrusion Detection System. The first component profiles the workload (i.e., the input rules and the live traffic) to generate the evaluation tree. The second component takes the evaluation tree, pre-processes the rules, and matches any incoming packet on the tree. These components are general enough to be applied to any IDS. We implemented our algorithm that generates an evaluation tree for a given workload over Snort 2.1.3. We chose Snort as it already provides an interface to read the rules into proper data structures. It also provides an interface to read the incoming traffic and check for different protocol fields.

As a second component, we modified Snort 2.1.3 to take the bin profiles and construct a hierarchical evaluation plan. Snort 2.0 [14] introduced an interface for

parallel evaluation of rules on a packet. Our hierarchical evaluation tree provides the set of applicable rules for a packet according to its values for different protocol fields. We pre-computed the data structure required for parallel matching for each of these groups. For every packet, we used our evaluation tree to determine the set of applicable rules and allowed Snort to perform the evaluation. We implemented three protocol fields by which the hierarchical structure can be constructed, namely: destination port, source port, destination IP address, and whether the packet is from the client. Since rules contain a large number of distinct protocol fields and we want to immediately detect the applicable rules, we implemented a check for destination port using an array of 65,536 pointers. Source port and destination IP address was checked by looking for possible match in a linked list. We did this because only a few destination IP addresses/source ports have to be checked, and because maintaining a pointer for each specific value consumes significant memory. For client checks the rules were divided into two parts: those that required to check if the packet is coming from client, and the rest were others. Every time a bin was split, we ensured that a rule was included in all new bins whose specific value can match the value in the rule. This ensured the correctness of our approach. We also validated our system by matching the number of alerts that our system raises, when compared to the number of alerts raised by unmodified Snort on a large number of datasets.

4 Evaluation

In this section, we evaluate Wind on a number of publicly-available datasets and on traffic from a border router at a large academic network. On these datasets, we compared real-time performance of Wind with existing IDSs using two important metrics: the number of packets processed per second and the amount of memory consumed. To measure the number of packets processed per second, we compiled our system and the unmodified Snort with gprof [21] options and then evaluated the dataset with each one of them. Then we generated the call graph, using gprof, and examined the overall time taken in the `Detect` function, which is the starting point of rule application in Snort. Finally, using the time spent in `Detect` and the number of times it was called, we computed the number of packets processed per second. To compute the memory used, we measured the maximum virtual memory consumed during the process execution by polling each second the process status and capturing the virtual memory size of the process. We now describe the datasets and the computing systems that we used for our experiments.

4.1 Datasets and computing systems

We evaluated the performance of our system on a number of publicly-available datasets and on traffic from a large academic network. For publicly available datasets, we used traces that DARPA and MIT Lincoln Laboratory have used for testing and evaluating IDSs. We used two-week testing traces from 1998 [22], and two-week testing traces from 1999 [23]. This gave us 20 different datasets

with home network 172.16.0.0/12. For evaluating the system on real-world, live traffic, we chose a gateway router to a large academic network with address 141.212.0.0/16. This router copies traffic from all ports to a span port, which can be connected to a separate machine for analyzing the traffic.

For DARPA dataset experiments, we used a dual 3.06 GHz Intel Xeon machine with 2 GB of main memory. The machine was running FreeBSD 6.1 with SMP enabled. We connected the span port of the gateway router to a machine with dual 3.0 GHz Intel Xeon processors and 2GB of main memory. The machine was running FreeBSD 5.4 with SMP enabled. The results that follow are the averages over 5 runs and with the THRESHOLD value set to 5.

4.2 Processing time and memory usage

We compared Wind with Snort 2.1.3 for all rules included with the distribution. There were 2,059 different rules, and both Wind and Snort were run using default configuration. Figure 5 shows the amount by which we improved the number of packets processed per second by Snort. For most datasets, we find that our system processes up to 1.6 times as many packets as Snort. We also compared the memory used by our system with that of Snort. Figure 6 shows the memory saved by our system when compared to Snort. We find that our system uses about 10-20% less memory when compared to the unmodified Snort. In other words, we perform up to 1.6 times better in processing time and save 10-20% of the memory.

Wind and Snort were run on the border router for analyzing a million packets at a few discrete times in the week. Figure 7 shows the amount by which Wind improved the number of packets processed per second by Snort. It shows that the improvement factor on this dataset varied from 1.35 to 1.65. During the runs, Wind consumed 10-15% less memory than Snort.

4.3 Application-specific rules

Until now, all our experiments were conducted by enabling all rules that came with the Snort distribution. However, in many networks, only application-specific rules can be used. For example, in many enterprise networks, the only open access through the firewall is web traffic. Since web traffic forms the dominant application allowed in many networks, we compared our system with Snort for web-based rules [2]. Figure 8 shows the magnitude by which our system improves Snort, in the terms of number of packets processed per second, for traffic at the border router. We found that for web-based rules, our system improves performance by more than two times when compared to Snort. Figure 9 shows a similar graph for the DARPA datasets. We observed that Wind outperforms Snort by a factor of up to 2.7 times. In this case, we saved 2-7% of the memory when compared to Snort.

[2] web-cgi, web-coldfusion, web-iis, web-frontpage, web-misc, web-client, web-php, and web-attack rules with Snort 2.1.3

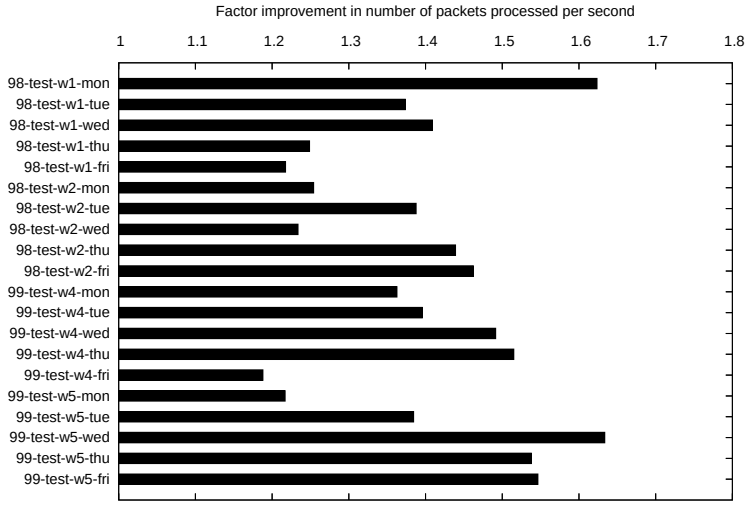

Fig. 5. Factor improvement, in terms of number of packets processed per second, when compared to Snort for the 1998 and 1999 DARPA testing datasets.

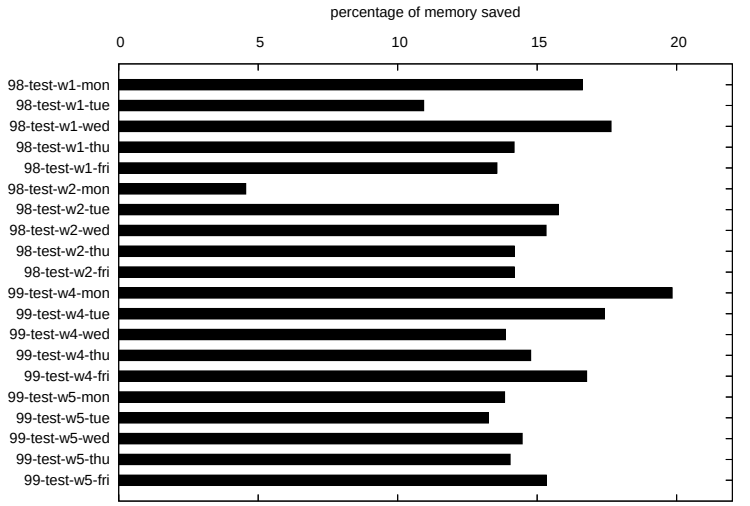

Fig. 6. Percentage of memory saved for each of the 1998 and 1999 DARPA datasets, when compared to Snort.

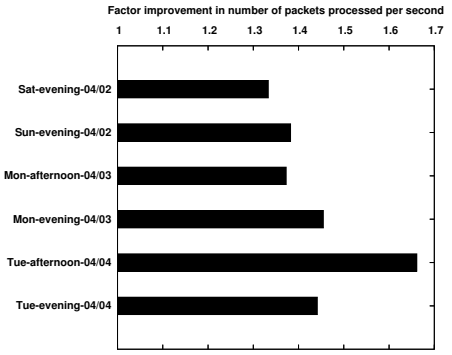

Fig. 7. Factor improvement in number of packets processed per second, when compared to Snort, on data from a border router in an academic network.

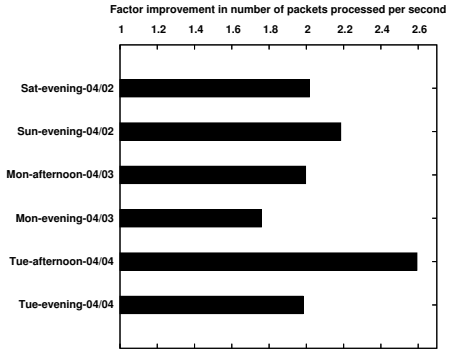

Fig. 8. Factor improvement in number of packets processed per second, when compared to Snort, for web-based rules. These experiments were on traffic from a border router at an academic network.

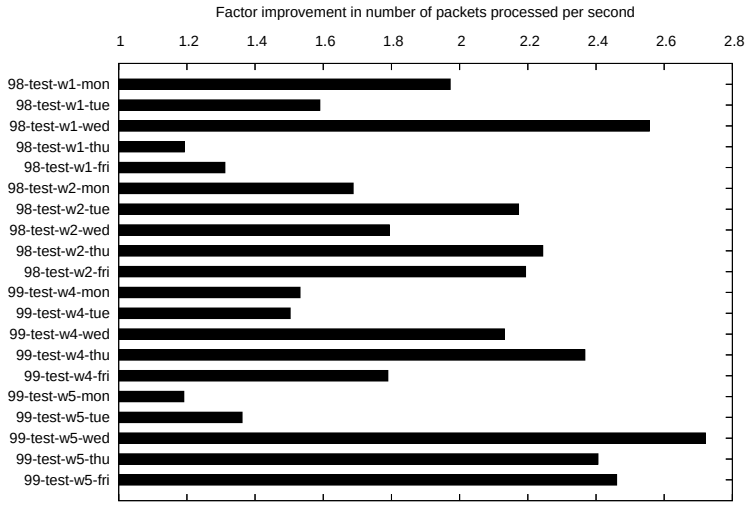

Fig. 9. Factor improvement in number of packets processed per second by Wind when compared to Snort for web-based rules. The datasets include the 1998 and 1999 DARPA intrusion detection datasets.

4.4 Variation with threshold

In order to investigate how the threshold affects the performance of our system, we evaluated the DARPA dataset, 98-test-w1-mon, for different values of the threshold. Figure 10 shows the performance variation of our system with the increasing threshold. As expected, the performance of the system decreases with increasing cost assigned by threshold. However, we find that the changes are more pronounced only for lower threshold values. We find that the memory saved by our system increases with increasing threshold values, significantly only for lower threshold values. Therefore, we find that increasing the threshold reduces performance, but saves more memory, and this difference is more pronounced for lower threshold values.

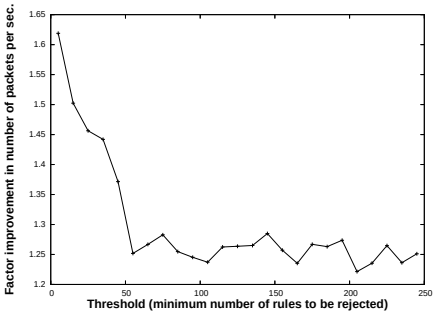

Fig. 10. The change in number of packets processed with the threshold for minimum number of rules to be rejected, when compared to Snort (dataset: 98-test-w1-mon).

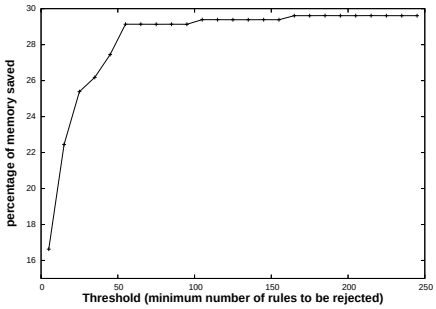

Fig. 11. Variation in memory saving with the threshold for minimum number of rules to be rejected, when compared to Snort (dataset: 98-test-w1-mon).

4.5 Comparison with Bro

We also compared Wind with another IDS Bro [6]. We first converted Snort signatures using a tool already provided by Bro [24]. However, only 1,935 signatures were converted and regular expressions in the rules were ignored. We then compared Bro 0.9 with Wind and Snort for various DARPA workloads. As shown in Fig. 12, Snort is faster than Bro by 2 to 8 times, and Wind is 3 to 11 times faster than Bro. This result is partly because Bro uses regular expression for signature specification rather than Snort, which uses exact substrings for signature matching. Bro uses a finite automata to match regular expressions [24], whereas Snort uses the Wu-Manber algorithm for matching sets of exact substrings.

5 Dynamically Adapting to Changing Workload

The Wind system that we have described so far analyzes observed network traffic and input rules to speed up the checking of network packets in an IDS in a

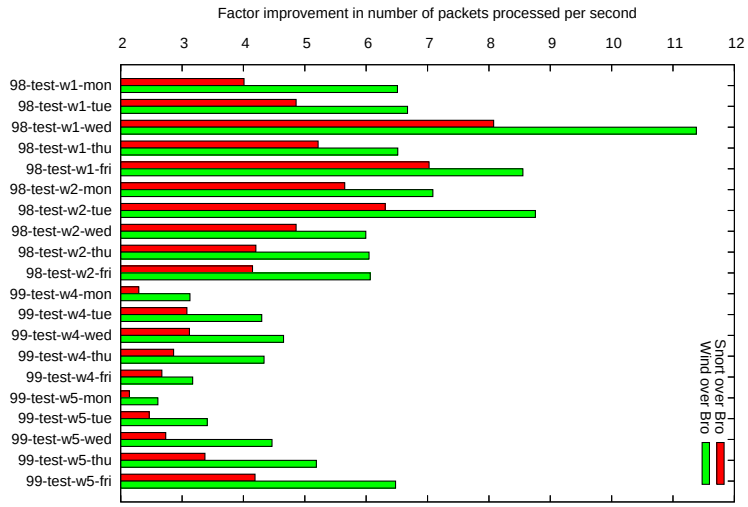

Fig. 12. Factor improvement when comparing Bro with Snort and Wind for the number of packets processed per second (dataset: 99-test-w1-wed).

memory-efficient way. However, traffic characteristics can change over time, or the rule set can change as new vulnerabilities are announced. Therefore, we need to adapt our evaluation structure dynamically without restarting the system.

To adapt to changing traffic characteristics, we plan to collect traffic statistics in the intrusion detection system itself, and reorganize the evaluation structure when necessary. It would be too intrusive and costly to update statistics for each packet. Therefore, one could update statistics for a small sample of incoming packets. Then, we can use these statistics to determine the utility of specific groups in the structure, and determine the benefit that rules in the generic group would provide if they are separated from other rules in the generic group. We can then remove specific groups whose utility decreases over time and make new groups for rules in the generic group that provide increased benefit. However, to ensure the correct application of rules, these changes may require updating a portion of the evaluation tree atomically, thereby disrupting the incoming traffic. Therefore, one could develop algorithms that use the updated statistics to dynamically detect a significant change in traffic and trigger reconfiguration of the structure when the benefits far outweigh the disruption.

Vulnerabilities are announced on a daily basis. Sometime a number of vulnerabilities for a single application are announced in a batch, demanding a set of rules to be updated with the intrusion detection and prevention system. One naive solution is to add the set of rules to the existing evaluation structure, and then let the reconfiguration module decide over time if there is a need to create additional groups. However, this strategy may affect the performance significantly if a large set of rule is added to the generic group. This performance degradation would continue till new groups are created. Therefore, one could add rules whose values match with already existing groups directly to those specific groups. If a large number of rules still remain to be added to the generic

group, then we can use our algorithm described in this paper to determine the groups that should be separated. Then, additional groups can be created within the existing structure and the new rules added into those groups.

6 Conclusions and Directions for Future Work

In this paper, we have argued that an intrusion detection and prevention system should adapt to the observed network traffic and the input rules, to provide optimized performance. We have developed an adaptive algorithm that captures rules and traffic characteristics to produce a memory-efficient evaluation structure that matches the workload. We have implemented two distinct components over Snort to construct a workload-aware intrusion detection system. The first component systematically profiles the input rules and the observed traffic to generate a memory-efficient packet evaluation structure. The second component takes this structure, pre-processes the rules, and matches any incoming packet. Finally, we have conducted an extensive evaluation of our system on a collection of publicly-available datasets and on live traffic from a border router at a large academic network. We found that workload-aware intrusion detection outperforms Snort by up to 1.6 times for all Snort rules and up to 2.7 times for web-based rules, and consumes 10-20% of less memory. A Snort implementation of Wind outperforms existing intrusion detection system Bro by six times on most of the workloads.

In future, we believe application decoding will be more common in intrusion detection and prevention systems [24]. As part of future work, we plan on evaluating our workload-aware framework on such systems. We also plan on evaluating Wind with more context-aware signatures, and porting it to other available IDSs and IPSs. Finally, we also plan on developing a dynamically-adaptive IDS, and deploying it in real networks.

Acknowledgments

This work was supported in part by the Department of Homeland Security (DHS) under contract number NBCHC040146, and by corporate gifts from Intel Corporation. We thank Evan Cooke and Michael Bailey for providing valuable feedback on the draft and anonymous reviewers for critical and useful comments.

References

1. Symantec: Symantec Internet threat report: Trends for July '05 - December '05. `http://www.symantec.com/enterprise/threatreport/index.jsp` (March, 2006)
2. Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of Usenix Lisa Conference. (November, 2001)
3. Microsoft: Vulnerability in graphics rendering engine could allow remote code execution. `http://www.microsoft.com/technet/security/bulletin/ms06-001.mspx` (January, 2006)
4. Knobbe, F.: WMF exploit. `http://www.securityfocus.com/archive/119/420727/30/60/threaded` (December, 2005)
5. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational experiences with high-volume network intrusion detection. In: CCS '04: Proceedings of the 11th ACM conference on Computer and communications security. (2004) 2–11

6. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks **31(23-24)** (1999) 2435–2463
7. Lee, W., Cabrera, J.B.D., Thomas, A., Balwalli, N., Saluja, S., Zhang, Y.: Performance adaptation in real-time intrusion detection systems. In: Proceedings of Recent Advances in Intrusion Detection (RAID). (2002) 252–273
8. Kruegel, C., Valeur, F., Vigna, G., Kemmerer, R.: Stateful intrusion detection for high-speed networks. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2002) 285–
9. Sekar, R., Guang, Y., Verma, S., Shanbhag, T.: A high-performance network intrusion detection system. In: ACM Conference on Computer and Communications Security. (1999) 8–17
10. Gusfield, D.: Algorithms on strings, trees, and sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
11. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical report, Department of Computer Science, University of Arizona (1993)
12. Kruegel, C., Toth, T.: Automatic rule clustering for improved signature-based intrusion detection. Technical report, Distributed systems group: Technical Univ. Vienna, Austria (2002)
13. Egorov, S., Savchuk, G.: SNORTRAN: An optimizing compiler for snort rules. Technical report, Fidelis Security Systems (2002)
14. Norton, M., Roelker, D.: SNORT 2.0: Hi-performance multi-rule inspection engine. Technical report, Sourcefire Inc. (2002)
15. Schuehler, D., Lockwood, J.: A modular system for FPGA-based TCP flow processing in high-speed networks. In: 14th International Conference on Field Programmable Logic and Applications (FPL), Antwerp, Belgium (2004) 301–310
16. Cho, Y.H., Mangione, W.H.: Programmable hardware for deep packet filtering on a large signature set. http://citeseer.ist.psu.edu/699471.html (2004)
17. Finkelstein, S.: Common expression analysis in database applications. In: Proceedings of the 1982 ACM SIGMOD international conference on Management of data, New York, NY, USA (1982) 235–245
18. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1) (1988) 23–52
19. Sellis, T., Ghosh, S.: On the multiple-query optimization problem. IEEE Transactions on Knowledge and Data Engineering **2**(2) (1990) 262–266
20. Park, J., Segev, A.: Using common subexpressions to optimize multiple queries. In: Proceedings of the Fourth International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (1988) 311–319
21. Graham, S., Kessler, P., McKusick, M.: gprof: A call graph execution profiler. In: Proceedings of the SIGPLAN '82 Symposium on Compiler Construction. (June, 1982) 120–126
22. Lippmann, R.P., Fried, D.J., Graf, I., Haines, J.W., Kendall, .K.R., McClung, D., Weber, D., Webster, S.E., Wyschogrod, D.., Cunningham, R.K., Zissman, M.A.: Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In: Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX). (2000) 12–26
23. Lippmann, R.P., Haines, J.: Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In: Proceedings of Recent Advances in Intrusion Detection (RAID), Springer Verlag (2000) 162–182
24. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS-03), New York (2003) 262–271