

Fast Packet Classification using Condition Factorization

Alok Tongaonkar, R. Sekar, Sreenaath Vasudevan

Stony Brook University

Abstract. Rule-based packet classification plays a central role in network intrusion detection systems such as Snort. To enhance performance, these rules are typically compiled into a *matching automaton* that can quickly identify the subset of rules that are applicable to a given network packet. The principal metrics in the design of such an automaton are its size and the time taken to match packets at runtime. Previous techniques for this problem either suffered from high space overheads (i.e., automata could be exponential in the number of rules), or matching time that increased quickly with the number of rules. In contrast, we present a new technique that constructs polynomial size automata. Moreover, we show that the matching time of our automata is insensitive to the number of rules. Our experimental results demonstrate substantial improvements in space requirements, as well the runtime of Snort.

1 Introduction

Given a network packet p and a set of *signatures* (which capture a set of conditions on the content of network packets), the problem of *packet classification* is that of identifying the subset of signatures that match p . It is the central computation performed in network intrusion detection systems (NIDS) such as Snort [14].

A naive technique for packet classification is that of sequentially matching each signature against an incoming packet. The performance of such a technique degrades linearly with the number of signatures. Since the number of signatures used in NIDS applications is typically large (e.g., Snort rule sets consist of several thousand rules), this naive technique will not scale to even moderate speed networks.

A natural way to speed up classification is to build a search-tree-like data structure that can be used to narrow down the set of signatures that are applicable to a packet, and then match the packet sequentially against each of the signatures in this subset. A common technique is to base the search tree on a small set of packet attributes that are present in almost all rules, e.g., Snort (versions 2.x) uses a search tree that first branches on the protocol (e.g., IP or ICMP), and then on source and destination ports (for TCP- and UDP-related rules).

By limiting to a small number of predefined attributes, we can simplify the search-tree construction algorithm, and also ensure that the tree is small in size. But the drawback is that the number of signatures that remain applicable at a leaf node can be substantial. As a result, the sequential matching phase can still take significant time. To further reduce this time, it would be desirable to build search trees that can make use of all (or most) of the attributes that occur in signatures, instead of limiting to a small number of predefined attributes. However, building such search trees becomes complex because some of the attributes may not be present in all signatures. Consider a search tree node that examines such an attribute. If node has two children, signatures that do

not examine this attribute would need to be duplicated across these children. Repeated duplication leads to search trees whose size, in the worst-case, is *exponential in the number of signatures* [16].

In contrast with previous techniques, we develop a new, systematic approach that ensures a polynomial bound on the size of the search tree. In addition to space-reductions, our approach improves classification speed using a novel technique called *condition factorization* that breaks down tests involving packet fields in such a manner as to expose commonalities across different types of tests such as equality tests, inequality tests, tests involving bit-masking operations, etc. Our experimental results indicate an overall performance gain of 30% for Snort. Moreover, as compared to previous techniques for constructing packet classification search trees such as Snort-NG [9], our techniques lead to search trees that are tens to hundreds of times smaller. Below, we present an overview of our approach and summarize its key contributions.

1.1 Overview of Approach and Contributions

- In Section 2, we formalize the problem of packet classification as applicable to intrusion detection systems.
- In Section 3, we develop the concept of *condition factorization* that provides the foundation for the optimizations developed in this paper. Condition factorization is based on the notion of a *residual* of a condition with respect to another. Intuitively, if we think of logical conjunction as analogous to the product operation on integers, then residuals are analogous to the division operation. Just as division provides the basis for finding common factors among integers, residuals provide the basis for “factorizing” complex conditions originating from different rules so as to “share” the testing of their common parts.
- In Section 4 we present our automaton¹ construction algorithm. Condition factorization is the core operation behind this algorithm, and it contributes directly to two key optimizations:
 - It can reason about the relationships between the typical operations that arise in rules (e.g., equalities, inequalities, disequalities, and bit-masking operations) and leverage them to avoid *semantically redundant tests* even if they aren’t syntactically identical. It is more general than the techniques developed in BPF+[3] for eliminating semantic redundancies — our technique proactively creates opportunities for sharing computation, whereas BPF+ is limited to checking whether previously performed tests obviate the need for a new test.
 - By working with residuals of rules, our automaton construction algorithm can recognize equivalence between automata states even before constructing the descendant states. Such *direct construction* is important, since a tree automaton is usually much larger (in theory, exponentially larger) than a DAG automaton. As a result, techniques that minimize tree automaton into a DAG automaton are bound to significantly increase space and time needed for automata construction.
- In Section 5, we present several additional techniques for building space- and time-efficient automata:

¹ Henceforth, we use the term “automaton” instead of the term “search-tree.”

- In Section 5.1, we develop the notion of a *discriminating test*. If such tests are selected at every state of the automaton, its size would be polynomial in the size of input rules. Unfortunately, discriminating tests may not always exist, which can lead to an explosion in automaton size. We therefore present a new technique in Section 5.2 that guarantees polynomial space bounds (where the degree of the polynomial can be user-specified) by trading off some determinism. We point out that this theoretical possibility of nondeterminism wasn't observed in our experiments. Thus, our technique was able to guarantee quadratic worst-case space requirement, without incurring, in practice, the performance penalties associated with nondeterminism.
 - In Section 5.3, we develop the notion of *benign nondeterminism*, which enables the introduction of nondeterministic branches in the automaton *without any increase in matching times*. Our experiments indicate dramatic reductions in automata size as a result of this technique.
- In Section 6, we describe our implementation, followed by an experimental evaluation in Section 7. Our technique achieves over a *10-fold reduction* in space requirements as compared to previous packet classification techniques for NIDS, while improving matching times. Moreover, the experimentally observed matching time remains virtually constant, regardless of the number of rules. In contrast, previous techniques experience a significant slowdown as the number of rules are increased. Our experiments also show that each of the techniques presented in previous sections contributes to significant reduction in space requirements.
 - Related work is described in Section 8, followed by concluding remarks in Section 9.

We point out that string-matching and regular-expression matching techniques are orthogonal to the techniques developed in this paper. In particular, a common strategy used in NIDS is to build a search-tree based on packet fields. At each leaf of this search-tree, a string-matching (or finite-state) automaton corresponding to the signatures S associated with this leaf is built. In the case of Snort, an Aho-Corasick automaton [1] is used, which identifies a subset of S' of S whose longest string matches the current network packet. Our techniques reduce the size of S by building a search-tree based on most packet fields, and hence the size of S' is also correspondingly reduced, which translates into faster times for the final (sequential) matching phase.

2 Preliminaries

In the rest of this paper, we use the term *filter* to refer to signatures. We associate a label to identify a filter.

Definition 1 (Tests, Filters and Priorities) *A test involves a variable x and one or two constants (denoted by c) and has one of the following forms.*

- Equality tests of the form $x = c$
- Equality tests with bitmasks of the form $x \& c_1 = c$
- Disequality tests of the form $x \neq c$
- Disequality tests with bitmasks of the form $x \& c_1 \neq c$

- Inequality tests of the form $x \leq c$ or $x \geq c$

A **filter** F is a conjunction of tests.

An example of a filter, as defined above, is

$$(dport = 22) \wedge (sport \leq 1024) \wedge (flags \& 0xb = 0x3)$$

We exclude more complex conditions that don't satisfy the definition of a filter, e.g.,

$$(sport + dport < 1024) \wedge (sport < ttl),$$

since they do not seem to arise in practice.

A filter F can be “applied” to a network packet p , denoted $F(p)$, by substituting variables, which denote the names of packet fields, with the corresponding values from p . We define the notion of matching based on whether the filter evaluates to *true* after this substitution.

Definition 2 (Matching) For a set \mathcal{F} of filters, we say that $F \in \mathcal{F}$ **matches a packet** p if $F(p)$ is true. The **match set** of p , denoted $\mathcal{M}_{\mathcal{F}}(p)$ consists of all filters that match p .

To illustrate matching, consider the following filter set \mathcal{F} :

- $F_1 : (icmp_type = ECHO)$
- $F_2 : (icmp_type = ECHO_REPLY) \wedge (ttl = 1)$
- $F_3 : (ttl = 1)$

Also consider an *icmp echo* packet p_1 and an *icmp echo reply* packet p_2 , both having a *ttl* of 1. For these filters and packets, F_1 matches p_1 , F_2 matches p_2 , and F_3 matches both. As a result, $\mathcal{M}_{\mathcal{F}}(p_1) = \{F_1, F_3\}$ and $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_2, F_3\}$

Examples of *packet-matching automata* (also known as matching or classification automata) for the above filter set are shown in Figures 1 and 2. Figure 1 shows a *deterministic automaton*, in which all of the transitions from any automaton state are mutually exclusive. A *non-deterministic automaton* is shown in Figure 2, where the transitions may not be mutually exclusive. We make the following observations about the structure of matching automata:

- All but one of the transitions from each state are labeled with a *test* as defined above; the remaining (optional) transition, called an “other” transition, is labeled with a more complex condition C as follows:
 - In a non-deterministic automaton, C is the conjunction of negations of a *subset* of the tests on the rest of the transitions, e.g., the third transition from the start state in Figure 2.
 - In a deterministic automaton, C is the conjunction of negations of *all* the tests on the rest of the transitions, e.g., the third transition from the start state in Figure 1. In this case, the “other” transition is mutually exclusive with the rest of the transitions, and hence is also called an “else” transition.
- The transitions from each automaton state are *simultaneously distinguishable*, i.e.,
 - apart from the “other”-transition, the tests on the rest of the transitions are mutually exclusive

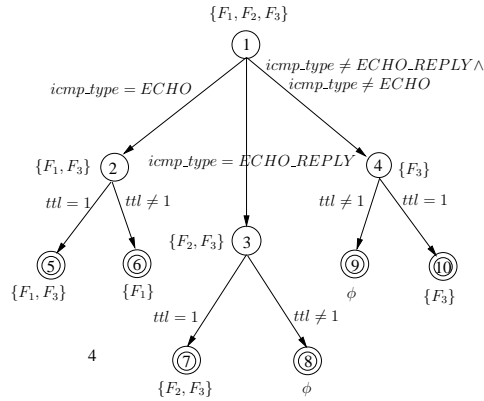


Fig. 1. A deterministic matching automaton.

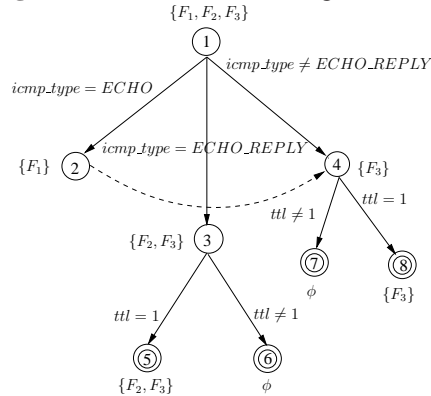


Fig. 2. A non-deterministic matching automaton.

- it is possible to determine, using a single operation with $O(1)$ expected time complexity, which of the transitions out of a state is applicable to a given packet.
- Each final state S correctly identifies the match set corresponding to any packet satisfying all the tests along a path from the start state to S .

Note that non-determinism has a runtime cost, as it needs to be simulated using backtracking. For instance, consider a packet that satisfies the $icmp.type = ECHO$ condition on the first transition from the start state of Figure 2. This packet is also compatible with the condition $icmp.type \neq ECHO_REPLY$ on the third transition from the start state. Thus, after exploring down the first transition, it is necessary to explore down the third transition as well. This need for backtracking is depicted in Figure 2 using a dotted transition.

3 Condition Factorization

In this section, we introduce the novel concept of condition factorization. It refers to the process of decomposing filters into combination of more primitive tests — a process that is intuitively similar to factorization of integers. This decomposition exposes

those primitive tests that are common across different tests, and thus enables shared computation of these common primitive tests.

The basis for condition factorization is the residue operation defined below. It is analogous to integer division. Suppose that we want to determine if there is a match for a filter C_1 . Also assume that we have so far tested a condition C_2 . A residue captures the additional tests that need to be performed at this point to verify C_1 .

Definition 3 (Residue) For conditions C_1 and C_2 , the *residue* C_1/C_2 is another condition C_3 such that:

- (1) $C_2 \wedge C_3 \Rightarrow C_1$, and
- (2) $C_1 \wedge C_2 \Rightarrow C_3$.

For a filter set, $\mathcal{F}/C = \{F/C \mid F \in \mathcal{F} \wedge F/C \neq \text{false}\}$.

Ideally, C_3 would be the weakest condition such that (1) holds. In practice, however, we may not want the minimal condition since it may be expensive to compute, or be inefficient to use, e.g., may contain many disjunctions. For this reason, we do not require C_3 be the weakest such condition. But C_3 shouldn't be too strong, or else we may miss matches for C_1 . This motivates the condition (2) above.

The rules in Figure 3 specify how to compute residues on tests. In the figure, the notation \bar{x} denotes bit-wise complement of x , while $\&$ denotes bit-wise “and” operation. In addition, inequalities are expressed using interval constraints, e.g., $x \leq 7$ is represented as $x \in [-\infty, 7]$, if x is an integer-valued variable. Note that a single interval constraint can represent a pair of inequalities involving a single variable, e.g., $(x \leq 7) \wedge (x > 3)$ can be represented as $x \in [4, 7]$.

For any pair of tests T_1 and T_2 , the first row in the table that matches the structure of T_1 and T_2 yields the value of T_1/T_2 . We illustrate residue computation using several examples:

- $(x \neq a)/(x = a)$ is *false*, as given by the second row in the table (which defines $T/\neg T$).
- $(x = 5)/(x \& 0x3 \neq 1)$ is *false*, as given by the 5th row.
- for $(x = 5)/(x \& 0x3 \neq 0)$, 5th row is no longer applicable since the condition $c \& c_1 = c_2$ does not hold. (Here, $c = 5$, $c_1 = 0x3$, and $c_2 = 0$.) Hence the applicable row is the last row, which yields $(x = 5)/(x \& 0x3 \neq 0) = (x = 5)$. The result is understandable: although the two conditions are compatible with each other, the test $x \& 0x3 \neq 0$ does not contribute to proving $x = 5$.
- $(x \in [1, 10])/(x \neq 5)$ is also given by the last row to be $(x \in [1, 10])$.

Note that the *minimal* residue in the last example would be $(x \in [1, 4]) \vee (x \in [6, 10])$. In this sense, Figure 3 makes approximations in computing residues. Intuitively, we make this approximation since there does not seem to be any way to evaluate $(x \in [1, 4]) \vee (x \in [6, 10])$ more efficiently than $(x \in [1, 10])$.

In general, approximations such as those used above have the potential to lead our matching algorithm to perform multiple tests that have some semantic overlap. However, the first line in Figure 3 ensures that two syntactically identical tests would never be performed.

T_1	T_2	T_1/T_2	Conditions
T	T	$true$	
T	$\neg T$	$false$	
T	$x = c$	$T[x \leftarrow c]$	
$x = c$	$x \& c_1 = c_2$	$x \& \bar{c}_1 = c \& \bar{c}_1$	$c \& c_1 = c_2$
		$false$	$c \& c_1 \neq c_2$
$x = c$	$x \& c_1 \neq c_2$	$false$	$c \& c_1 = c_2$
$x = c$	$x \in [c_1, c_2]$	$false$	$c \notin [c_1, c_2]$
$x \neq c$	$x \& c_1 = c_2$	$x \& \bar{c}_1 \neq c \& \bar{c}_1$	$c \& c_1 = c_2$
		$true$	$c \& c_1 \neq c_2$
$x \neq c$	$x \& c_1 \neq c_2$	$true$	$c \& c_1 = c_2$
$x \neq c$	$x \in [c_1, c_2]$	$true$	$(c < c_1)$ $\vee (c > c_2)$
$x \in [c_1, c_2]$	$x \in [c_3, c_4]$	$true$	$c_1 \leq c_3$ $\leq c_4 \leq c_2$
		$x \in [-\infty, c_2]$	$c_1 \leq c_3$ $\leq c_2 \leq c_4$
		$x \in [c_1, \infty]$	$c_3 \leq c_1$ $\leq c_4 \leq c_2$
		$x \in [c_1, c_2]$	$c_3 \leq c_1$ $\leq c_2 \leq c_4$
		$false$	$(c_2 < c_3)$ $\vee (c_4 < c_1)$
$x \in [c_1, c_2]$	$x \& c_3 = c_4$	$false$	$c_4 > c_2$
$x \& c_1 = c_2$	$x \& c_3 = c_4$	$x \& (c_1 \& \bar{c}_3)$ $= (c_2 \& \bar{c}_3)$	$c_2 \& c_3$ $= c_1 \& c_4$
		$false$	otherwise
$x \& c_1 = c_2$	$x \in [c_3, c_4]$	$false$	$c_2 > c_4$
$x \& c_1 \neq c_2$	$x \& c_3 = c_4$	$x \& (c_1 \& \bar{c}_3)$ $\neq (c_2 \& \bar{c}_3)$	$c_2 \& c_3$ $= c_1 \& c_4$
		$true$	otherwise
$x \& c_1 \neq c_2$	$x \in [c_3, c_4]$	$true$	$c_2 > c_4$
T	T'	T	

Fig. 3. Computation of Residue on Tests.

To illustrate residues on filter sets, consider

$$\mathcal{F} = \{F_1 : (x = 5), F_2 : (x = 7), F_3 : (x < 10)\}.$$

Then

- $\mathcal{F}/(x = 5) = \{F_1 : true, F_3 : true\}$
- $\mathcal{F}/(x < 7) = \{F_1 : (x = 5), F_3 : true\}$

Finally, we specify how to compute residues on more complex conditions that are formed using conjunction and disjunction operations on tests:

- $(C_1 \oplus C_2)/C_3 = (C_1/C_3) \oplus (C_2/C_3)$, for $\oplus \in \{\wedge, \vee\}$

$$- C_1/(C_2 \wedge C_3) = (C_1/C_2)/C_3$$

We have ignored the case where the second operand to the residue operator contains a disjunction, since this case does not arise in our automata construction algorithm. Using this definition, we can see that:

- $((x > 2) \vee (y > 7))/(x = 5)$ is *true*, and
- $((x > 2) \wedge (y > 7))/(x = 5)$ is $(y > 7)$.

4 Matching Automata Construction

Our algorithm *Build* for constructing a matching automata is shown in Figure 4. *Build* is a recursive procedure that takes an automaton state s as its first parameter, and builds the subautomaton that is rooted at s . It takes two other parameters: (i) the *match set* \mathcal{M}_s that consists of all filters for which a match can be announced at s , and (ii) the *candidate set* \mathcal{C}_s that consists of filters that haven't completed a match, but future matches can't be ruled out either, i.e., matches for these filters will be reported at some of the descendants of s . To illustrate the concepts of match and candidate sets, we have annotated the final states in Figures 1 and 2 with match sets, and non-final states with the union of match and candidate sets.

We maintain only the residuals of the original filters in \mathcal{C}_s and \mathcal{M}_s , after factoring out the tests performed on the path from the root of the automaton to the state s . For example, in Figure 1, at state 2, we have completed a match for F_1 , and hence its match set is $\{F_1 : true\}$. Note that the condition component of F_1 has become *true* since we computed the residue of the original condition (i.e., $(icmp_type = ECHO)$) with respect to the condition $(icmp_type = ECHO)$ on the path from the automaton root to state 2. In addition, note that we can rule out a match for F_2 at this state, but a match for F_3 is still possible. Thus, the candidate set for this state is $\{F_3 : (ttl = 1)\}$.²

A final state is characterized by the fact that there are no more filters left in \mathcal{C}_s . This condition is tested at line 2, and s is marked final, and is annotated to indicate \mathcal{M}_s as its match set. If the condition at line 2 isn't satisfied, then the construction of automaton is continued in lines 5–16. First, a procedure *select* (to be defined later) is used at line 5 to identify a set of tests T_1, \dots, T_k that would be performed on the transitions from s . This procedure also indicates whether T_i is going to be a deterministic transition or not: in the former case d_i is set to *true*, while in the latter case, $d_i = false$. Based on which T_i are deterministic, the condition T_o associated with the “other”-transition is computed on line 6: $\neg T_i$ is included in T_o iff T_i is to be a deterministic transition.

The actual transitions are created in the loop at line 7–16. At line 8, we compute the subset \mathcal{C}_i of filters in \mathcal{C}_s that are compatible with T_i . However, if this is going to be a nondeterministic transition, then a match would be tried down the transition labeled T_i and then subsequently down the “other”-transition. For this reason, we can eliminate from \mathcal{C}_i any filter that will be considered on the “other”-transition. This elimination is performed on line 9. At line 10, \mathcal{M}_{s_i} and \mathcal{C}_{s_i} for the new state s_i are computed.

² \mathcal{M}_s and \mathcal{C}_s can be formally defined as follows. Let \mathcal{P}_s denote the conjunction of tests on the path from the start state of the automaton to the state s . Then $\mathcal{M}_s = \{F \in \mathcal{F}/\mathcal{P}_s | (F = true)\}$. Similarly, $\mathcal{C}_s = \{F \in \mathcal{F}/\mathcal{P}_s | (F \neq true)\}$

```

1. procedure Build( $s, \mathcal{M}_s, \mathcal{C}_s$ )
2.   if  $\mathcal{C}_s$  is empty /* No more filters to match */
3.     then  $match[s] = \mathcal{M}_s$  /* Annotate final state with match set */
4.   else
5.      $(D, \mathcal{T}) = select(\mathcal{C}_s)$  /*  $T_i \in \mathcal{T}$  is tested on  $i$ th transition */
6.     /*  $d_i \in D$  indicates if this transition is deterministic */
7.      $T_o = \{\bigwedge_{d_i \in D | d_i = true} \neg T_i\}$ 
8.     /* Compute test corresponding to the “other”-transition */
9.     for each  $T_i \in (\mathcal{T} \cup \{T_o\})$  do
10.       $\mathcal{C}_i = \mathcal{C}_s / T_i$ 
11.      if  $((T_i \neq T_o) \wedge \neg d_i)$  then  $\mathcal{C}_i = \mathcal{C}_i - \mathcal{C} / T_o$  endif
12.      /* For a non-deterministic transition, do not duplicate */
13.      /* filters from the “other” branch */
14.      compute  $\mathcal{M}_{s_i}$  and  $\mathcal{C}_{s_i}$  from  $\mathcal{C}_i$  and  $\mathcal{M}_s$ 
15.      if a state  $s_i$  corresponding to  $(\mathcal{C}_{s_i}, \mathcal{M}_{s_i})$  isn't present
16.        create a new state  $s_i$ 
17.        Build( $s_i, \mathcal{M}_{s_i}, \mathcal{C}_{s_i}$ )
18.      endif
19.      create a transition from  $s$  to  $s_i$  on  $T_i$ 
20.    end
21.  end

```

Fig. 4. Algorithm for Constructing Matching Automaton

Since the behavior of *Build* is determined entirely by the parameters \mathcal{C}_s and \mathcal{M}_s , two invocations of *Build* with the same values of these parameters will yield identical subautomata. Hence a check is made at line 11 to examine if an automaton state already exists corresponding to \mathcal{C}_{s_i} and \mathcal{M}_{s_i} , and if not, a new state is created at line 12, and *Build* recursively invoked on this state. Finally, a transition to this state is created at line 15.

5 Improving Automata Size

The algorithm presented in the last section incorporated two main optimizations to reduce automaton size and matching time, both derived from our definition of condition factorization: detecting and sharing equivalent states, and avoiding repetition of (semantically) redundant tests. In this section, we present techniques for realizing the *select* function that yields significant additional reduction in automata size.

Although our experimental evaluation considers the number of automaton states as a measure of its size, for simplifying mathematical analysis, our discussion in this section will use the automaton breadth as the size metric. Since the automaton is acyclic, and since tests are never repeated, it can be shown that the total number of automaton states can, in the worst case, be at most S times its breadth, where S is the number of distinct tests across all the filters³.

³ In practice, the factor is closer to average size of filters, which can be significantly smaller than S .

5.1 Discriminating Tests

Definition of *select* amounts to determining the test that should be performed at a particular state of the automaton. Since the test identifies the packet field to be examined, *select* can be viewed as defining an order of examination of packet fields. Not all orders of examination may be acceptable, since some packet fields (e.g., the protocol field) may need to be examined before others (e.g., the port field). We use a type system similar to packet types [4] that captures such ordering constraints among tests. Our implementation of *select* ensures that these constraints are respected.

The simplest approach for defining *select* is to test the fields in the order of their occurrence in a network packet, as done in some of the previous works [2,5]. We call such a traversal order as *left-to-right traversal* and refer to an automaton using this traversal order as *L-R automaton*. A better strategy, called *adaptive traversal*, was first proposed in the context of term-matching [16], and was then generalized to deal with binary data in [7]. In the terminology of this paper, an adaptive traversal would select a set of tests \mathcal{T} at an automaton state s as follows. It identifies a packet field x that occurs in every filter in \mathcal{C}_s . (If no such field can be found, it falls back to another choice, e.g., choosing the left-most field that hasn't yet been examined.) Now, \mathcal{T} consists of all tests on x that occur in any of the filters in \mathcal{C}_s .

Since adaptive traversal was developed in a context where the tests were all restricted to be simple equalities with constants, it is easy to see that the set \mathcal{T} described above consists of tests that can be simultaneously distinguished⁴, and hence can form the transitions from s . Moreover, it has been shown [16] that, as compared to other choices, this choice of transitions will simultaneously reduce the automaton size as well as matching time. Unfortunately, none of these hold in the more general setting of packet matching, where disequalities and inequalities also need to be handled. For instance, consider a candidate set that consists of two filters ($x \neq 25$) and ($x < 1024$). These tests are not simultaneously distinguishable. Moreover, neither of these tests contributes towards verifying a match with the other. More generally, it can be shown that, in the presence of disequality and inequality tests, the choices that decrease automaton size do not necessarily decrease matching time (and vice-versa). We therefore focus first on a criterion for reducing automaton size.

Definition 4 (Discriminating Set) A set \mathcal{T} of conditions is said to be a **discriminating set** for a filter set \mathcal{F} iff for every $F \in \mathcal{F}$ there exists at most one $T \in \mathcal{T}$ such that F belongs to the candidate set of \mathcal{F}/T .

The set $\mathcal{T} = \{x = 5, x = 6, (x \neq 5) \wedge (x \neq 6)\}$ is discriminating for the filter set $\mathcal{C} = \{x = 5, x = 6, x > 7\}$, but not for $\{x = 6, x > 4\}$. This means if we create 3 outgoing transitions corresponding to the three tests in \mathcal{T} from an automata state s with the candidate set \mathcal{C} , none of the filters in \mathcal{C} will be duplicated among the children of s . As a result, in an automaton that uses only discriminating tests, the candidate sets (as well as the match sets) associated with the leaves will be disjoint. Since there are at most n disjoint subsets of a set of size n , it immediately follows that any automata that is based entirely on discriminating tests will have at most $O(n)$ breadth.

⁴ Recall that simultaneous distinguishability refers to the ability to identify the matching transition in $O(1)$ expected time.

5.2 Ensuring Polynomial-Size Automata

Since discriminating tests may not always exist, it may be necessary to choose non-discriminating tests. This choice introduces overlaps among the candidate sets of sibling states in the automaton. These overlaps, in turn, mean that at any level in the automaton, there may be as many as 2^n distinct candidate sets. Thus, the breadth of the automaton can become exponential in the number of filters. Exponential *lower bounds* have previously been established even in the simple case where all tests are restricted to be equalities [16]. Although some of the previously developed techniques can avoid such explosion, this has been accomplished at the cost of introducing significant backtracking at runtime [11,5,2,3], especially for the kinds of filters that occur in the context of intrusion detection. Other techniques avoid exponential size by introducing $O(n)$ operations for each transition at runtime, as they require runtime maintenance of match sets [13,7]. With large filter sets that are often found in enterprise NIDS, $O(n)$ time complexity for transitions becomes unacceptable.

We present a new technique that can provide a polynomial size bound, while limiting nondeterminism in practice. Indeed, any desired polynomial bound $P(n)$ can be achieved by our technique. However, by using a larger bound, e.g., n^2 instead of $n \log n$, one can obtain deterministic automata in almost all cases.

Our technique is based on the observation that the breadth of subautomaton rooted at s can be captured, in terms of the sizes of candidates sets associated with s and its children, using the recurrence

$$B(|\mathcal{C}_s|) = \sum_{i=1}^k B(|\mathcal{C}_{s_i}|),$$

where $B(1) = 1$. Let $P(n)$ be the desired polynomial on n that bounds the automaton size. Based on the above recurrence, we can show, by induction on the height of s that the bound will be satisfied as long as the following condition holds at every state s of the automaton.

$$P(|\mathcal{C}_s|) \geq \sum_{i=1}^k P(|\mathcal{C}_{s_i}|) \tag{1}$$

By selecting tests that satisfy this constraint, our implementation of *select* ensures that the automaton size will be $O(P(n))$. If no such test can be found, our technique picks a test that comes the closest to satisfying this constraint, and then makes some of the outgoing transitions nondeterministic so as to ensure that sizes of candidate sets associated with the descendant automaton states satisfy the above constraint. Recall from line 9 of *Build* that making a test \mathcal{T}_i nondeterministic enables us to avoid overlaps between \mathcal{C}_i and \mathcal{C}_o . So, our algorithm makes one or more transitions out of an automaton state nondeterministic until Inequality 1 is satisfied. In our implementation, we have set $P(n)$ to be n^2 , which guarantees a quadratic worst-case automaton size.

To understand the importance of the above technique, note that a purely deterministic technique ensures good performance at runtime, but risks catastrophic failure on large rulesets that cause an exponential blow up — memory will be exhausted in that case and hence the ruleset can't be supported. In contrast, our approach converts this catastrophic risk into the less serious risk of performance degradation. Unlike previous

techniques for space reduction that led to increases in runtime in practice, performance degradation remains a theoretical possibility with our technique, rather than something observed in our experiments. (This is because of the fact that with the rulesets we have studied in our experiments, the quadratic bound was never exceeded, and hence nondeterminism was not introduced.)

5.3 Benign Nondeterminism

For our final space-reduction technique, we define the concept of benign non-determinism, which enables us to benefit from the space-savings enabled by non-determinism *without incurring any performance penalties*. It is based on the following notion of *independence* among filter sets.

Definition 5 (Independent Filters) *Two filters F_1 and F_2 are said to be independent of each other if $F_2/T = F_2, \forall T \in F_1$, and $F_1/T = F_1, \forall T \in F_2$. \mathcal{F}_1 and \mathcal{F}_2 are said to be independent if $\forall F_1 \in \mathcal{F}_1, \forall F_2 \in \mathcal{F}_2, F_1$ and F_2 are independent.*

Suppose that there is a filter set \mathcal{F} that can be partitioned into two independent subsets \mathcal{F}_1 and \mathcal{F}_2 . We can then build separate automata for \mathcal{F}_1 and \mathcal{F}_2 . Packets can now be matched using the first automaton and then the second one. From the above definition, it is clear that the tests appearing in the two automata are completely disjoint, and hence no decrease in runtime can be achieved by constructing a single automaton for \mathcal{F} .

Our experiments show that the above technique leads to dramatic reductions in space usage. The intuition for this is as follows. If F_1 and F_2 are independent, then a packet may match F_1 , F_2 , both, or neither. A deterministic automaton must have a distinct leaf corresponding to each of these possibilities. Extending this reasoning to independent filter sets, if an automaton for the set \mathcal{F}_1 has k_1 states, and the automaton for \mathcal{F}_2 has k_2 states, then a deterministic automaton for $\mathcal{F}_1 \cup \mathcal{F}_2$ will have $k_1 * k_2$ states. In contrast, using benign non-determinism, the size is limited to $k_1 + k_2$. If there are m independent filter sets, then the use of benign nondeterminism can reduce the automaton size from a product of m numbers to their sum.

The second reason for significant reductions in practice, is as follows. After examining some of the fields that are common across many rules, as we get closer to the automaton leaf, independent sets arise frequently. For instance, we may be left with one set that examines only the destination port, another set that examines only the source port, yet another set that examines only the destination network, and so on. Thus, independent rule sets tend to arise frequently, and lead to massive increases in space usage if they are not recognized and exploited using our benign non-determinism technique.

There is a simple algorithm for checking if \mathcal{F} contains two independent subsets. First, partition \mathcal{F} into singleton subsets corresponding to each rule. Now, these subsets are taken two at a time, and merged if they are *not* independent. This process is repeated until no more merges are possible. If there are multiple subsets left at this point, then these subsets are independent.

To deal with benign non-determinism, the interface between *select* and *Build* needs to be extended so that the former can return a set of independent filter sets $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$, instead of a test set. At this point, *Build* will create a k -way non-deterministic branch. On the i th branch, it will invoke $Build(s_i, \mathcal{F}_i, \mathcal{F}_i \cap \mathcal{M}_s)$.

6 Implementation: Putting It All Together

Our implementation first compiles the given filter set into an automaton using the *Build* algorithm. Residues were computed as specified in Table 3. Our *select* implementation proceeds as follows:

- *select* first attempts to find a discriminating test set (Section 5.1).
- if no discriminating test sets exist, it examines opportunities for benign non-determinism (Section 5.3).
- if neither of the above steps succeed, it returns a set of tests that achieves the polynomial size target specified, as described in Section 5.2.

In order to speed up *select*, our implementation starts by examining fields that occur in all filters in a candidate set, giving preference to those fields that contain primarily equality tests. Such fields have a high likelihood of yielding discriminating tests at which point *select* returns this set. As mentioned earlier, any constraints regarding the order of examination of fields are enforced by *select*.

Once the automaton is constructed, our compiler generates C-code corresponding to the automaton, which is then compiled into native code using a C-compiler. The code generation is straight-forward and not described in detail here, except to note that the code explicitly uses an if-then-else, a binary search, or a hash-based branching to implement transitions.

A runtime system is responsible for reading network packets and calling the generated code to perform matching. For experiments on network intrusion detection, our runtime system was essentially Snort, with modifications that were needed to integrate with our automata code.

7 Evaluation

The goal of our experimental evaluation is demonstrate performance improvements that can be gained in typical network intrusion detection systems as a result of using the packet classification techniques presented in the previous sections. To this end, we undertook two main experiments, both performed on a Linux system with 1.70Ghz Pentium 4 processor and 520MB memory, running CentOS-4.2 (Linux kernel 2.6.9).

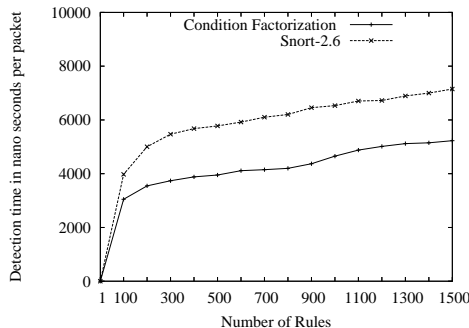


Fig. 5. Total Matching Time

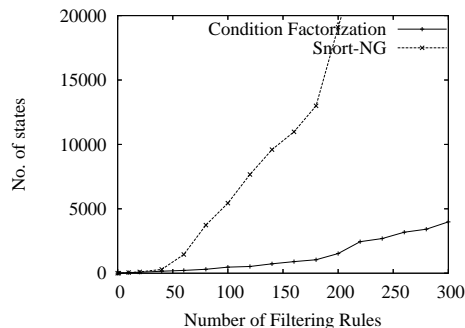


Fig. 6. Automaton Size for Snort Rules

7.1 End-to-End Performance Improvement of NIDS

In the first experiment, we replaced the simple packet classification used in Snort 2.6, the popular open-source NIDS, with our technique. Snort divides signatures into groups based on protocol, source port and destination port. For each such group, it extracts the longest string contained within the content-matching part of the signature, and builds an Aho-Corasick automaton for these signatures. At runtime, a simple packet classification technique is used to identify the rule group against which a packet needs to be matched. Then the content of the packet is matched using the Aho-Corasick automaton associated with this group. Since this automaton only considers the longest string from each signature, some of the signatures returned by this automaton may not really match the packet. (However, the automaton will always return a superset, not a subset of matching signatures.) Moreover, the signatures may contain complex conditions, e.g., a constraint on the distance between two strings within a signature. To handle these aspects, Snort performs a one-on-one match between a packet and each of the signatures returned by the automaton.

In our experiment, we replaced the first stage with the matching automaton constructed by our technique. At each leaf of this automaton, we replicate the technique used by Snort, i.e., we build an Aho-Corasick automaton to recognize the longest string contained in each of the signatures in the candidate set of the leaf⁵. Finally, a one-on-one match is performed between the signatures returned by this automaton and the network packet. Our implementation reuses almost all of Snort code, including the code for Aho-Corasick automaton, and the final one-on-one match. It only replaced the initial packet classification component. As a result, the performance improvements obtained by our technique are entirely due to the use of our sophisticated packet classifier.

We measured the total time taken by original Snort, and the version of Snort we modified to use our matching automaton. These times were computed for a 21-million packet trace collected at a University laboratory consisting of about 30 hosts. Since the firewall is fully open to the Internet (i.e., the traffic is not pre-screened by another layer of firewalls in the University or elsewhere), the traffic is a reasonable representative of what one might expect a NIDS to be exposed to. We used the default signature set that is shipped with Snort.

In this experiment, we observed that the one-on-one matching phase was invoked about 120M times in the original Snort, whereas it was invoked only 40M times with our packet classifier in place. This reduction in the number of one-on-one matches translates to about 30% reduction in the overall time taken by Snort.

Figure 5 shows the overall time taken by Snort with and without our modification, as we vary the number of rules. While the performance is nearly identical for small rule sets, it quickly increases to (and stabilizes at) about 30% at a few hundred signatures.

⁵ In the presence of non-determinism, we needed to modify the above technique so as to avoid repetition of string-matching tests after backtracking. Specifically, we built the Aho-Corasick at the first non-deterministic node encountered on a root-to-leaf path in the automaton, and performed an intersection of the set of signatures returned by the Aho-Corasick with the signature sets of each of the matching leaves.

7.2 Improvement in Space Usage

In this experiment, we evaluated gains in space usage obtained using our packet classification technique. We first compared our technique against that of Snort-NG [9], which was the only other implementation of a sophisticated packet classifier that we are aware of that is applicable to NIDS like Snort. Snort-NG uses a different strategy from ours for eliminating redundant tests: they convert all tests into a canonical form so that semantically identical tests would also be syntactically identical. However, tests in canonical form can in general be more expensive than the original test, e.g., in order to support tests on IP addresses that may sometimes involve bit-masking operations and at other times involve equality, they convert both tests into smaller tests that examine one bit of address at a time. Secondly, Snort-NG uses an entropy-based algorithm (instead of the criteria developed in Section 5 of this paper) to decide which packet field to test at each node. These factors lead to significant differences in the sizes of the automaton constructed, as illustrated in Figure 6.

We focused this evaluation on packet classification, and ignored the content-matching components of signatures. (Recall that content-matching *was* considered in the experiments in the previous section.) Since many signatures are identical except for the content-matching part, the default signature set that came with Snort-NG was reduced from a size of 1635 to 305.

Figure 6 shows the effect of increasing the number of rules on the number of automaton states. We can see from the graph that as the number of rules increases, the number of states in Snort-NG increases much faster than our technique. For 300 rules, Snort-NG automaton contains over 45K states whereas the automaton constructed by our technique has only about 4K states, representing an order of magnitude improvement in space utilization.

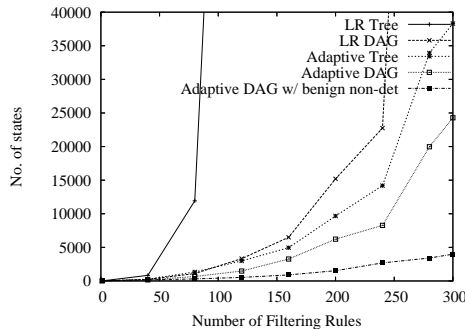


Fig. 7. Effect of Optimizations on Automaton Size for Snort Rules

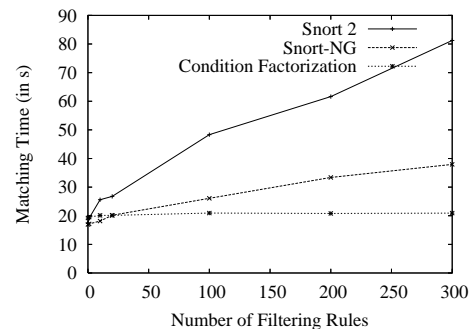


Fig. 8. Matching Time for Packet Classification

Effect of Optimization Techniques Figure 7 illustrates the effects of different optimizations on the automaton size. We studied different combinations of techniques: with and without sharing of equivalent states in the automata, and with different traversal orders.

- *Order of testing fields.* As compared to left-to-right (L-R) order for examining packet fields, our technique (which uses the *select* function as described in Sec-

- tion 6 produces tree automata that are much smaller: for 120 rules, the L-R automaton had 150,000 states, whereas the tree automaton had less than 3000 states.
- *DAG Vs tree automata.* Our results show that DAG automata were smaller than tree automata by about 25% for our technique. Larger space reductions were achieved with DAG optimization for L-R automata, but still, L-R automata remain significantly larger than the one constructed by our technique.
 - *Benign nondeterminism.* By exploiting benign non-determinism, we were able to achieve dramatic reductions in space usage. This is because Snort contains many rules which test some common fields. Our technique prefers these common fields for testing, since they are the ones that are likely to be discriminating. Once these common fields are tested, the residual rule sets contain many independent subsets.

We point out that a combination of our techniques was necessary to achieve the size reductions we have reported. In particular, benign nondeterminism leads to large improvements in size when combined with discriminating tests. It is much less effective when used with L-R technique, since the factors contributing to the occurrence of independent filter sets do not arise frequently when the L-R technique is used.

7.3 Packet Classification Performance

In this section, we describe experiments to study the runtime performance of packet classification. Unlike Section 7.1, which considered packet classification as well as content-matching time, this section focuses exclusively on the packet classification time in order to measure the raw performance benefits provided by our technique. For these experiments, we used the same 21M packet trace mentioned earlier.

Figure 8 shows the matching time taken by Snort, Snort-NG and our technique for classifying these packets as the number of rules change. In the Figure 8, it can be seen the matching time remains essentially constant with our technique, even as the number of rules are increased from about 10 to 300. In contrast, the matching times for Snort and Snort-NG increase significantly with the number of rules. The base matching time for all the techniques (with no rules enabled) is basically the same, as it corresponds to the time spent by Snort to read the packets from a file and do all related processing except matching.

8 Related Work

[19],[10],[20], [17] are techniques targeted at routers where they can restrict the problem so as to work on a small, predefined set of attributes such as IP address and port. Our focus is on NIDS, where a much larger number of attributes may be tested, and moreover, the tests can be complex.

Pattern matching automata have been extensively studied in the context of term rewriting and theorem proving [15]. Sekar et al [16] presented a technique for adapting the order of examination of fields in order to reduce space and matching time complexity of term-matching automata. Gustafsson and Sagonas [7] extended this technique to handle binary data such as network packets. Our technique generalizes their technique

further by adding support for inequalities and disequalities. Moreover, our bit-mask operations are more general than their bit-field operations. More importantly, their automata has an exponential worst-case space complexity. Although they describe a technique for constructing linear-size *guarded sequential automata*, these automata require runtime operations to manipulate match and candidate sets as a result, their transitions have an $O(N)$ complexity (where N is the number of patterns), while our transitions are $O(1)$ expected time.

Techniques such as BPF [11], DPF [5] and Pathfinder [2] can also be viewed as building matching automata where the packet fields are examined in the order they occur, i.e., they rely on a left-to-right traversal instead of relying on the techniques described in Section 5 for selecting the tests that are performed. As shown in our evaluation, our techniques result in significant gains in space usage, as compared to a left-to-right traversal.

BPF+ [3] uses global dataflow techniques to identify opportunities for eliminating redundant tests. Our condition factorization technique is more general than those of BPF+, being able to reason about semantic redundancies in the presence of bit-masking operations, and comparisons involving different constants. More importantly, condition factorization takes a step beyond the passive approach of recognizing redundant tests and eliminating them: it proactively decomposes complex tests into more primitive ones so that their common components are exposed and shared.

DPF uses dynamic code generation, which allows dynamic reordering of tests. Dynamic reordering improves performance by detecting match failures earlier. Al-Shaer et al [8] and Gupta et al [6] significantly improve on the dynamic reordering technique used in DPF by using efficient algorithms to maintain statistics regarding the traffic. Their techniques are analogous to profile-based optimizations in compilers, whereas ours is analogous to static-analysis based optimizations. Thus, the two techniques can complement each other.

Vern Paxson [12] developed Bro which is another popular NIDS. Sommer and Paxson [18] enhanced Bro signature matching to use regular expressions. An important difference between Bro and Snort is that Bro is primarily stream-oriented: it assembles packet sequences into streams before applying signatures. Commercial NIDS such as those from CISCO and IBM employ a combination of packet-oriented and stream-oriented matching techniques. The packet classification techniques developed in this paper fit naturally in the context of packet-oriented NIDS (Snort or commercial systems), and can speed them up. Integrating them into a stream-oriented NIDS can be a bit more involved, as these systems may apply certain tests on packet fields against some packets (e.g., the first packet in a stream) but not against others.

9 Conclusions

In this paper we presented new techniques for packet-matching. Our approach is based on the concept of condition factorization, and proactively creates opportunities for sharing common tests across different signatures. Unlike previous techniques, our techniques provide a worst-case polynomial bound on the size of matching automata, while ensuring excellent runtime performance in practice. Our experimental results demonstrate a 30% gain in end-to-end performance of the popular Snort NIDS due to the use

of our techniques. They also demonstrate an order of magnitude reduction in space usage as compared to previous systematic packet classification techniques developed in the context of NIDS, as well as matching times that remain virtually constant as the number of rules is increased.

References

1. A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, vol 18, no. 6, pages 333–343, 1975.
2. M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.
3. A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.
4. S. Chandra and P. McCann. Packet types. In *Second Workshop on Compiler Support for Systems Software (WCSSS), May 1999.*, 1999.
5. D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, 1996.
6. P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.
7. P. Gustafsson and K. Sagonas. Efficient manipulation of binary data using pattern matching. *J. Funct. Program.*, 16(1):35–74, 2006.
8. E. A.-S. Hazem Hamed, Adel El-Atawy. On dynamic optimization of packet matching in high-speed firewalls. In *IEEE Journal on Selected Areas in Communications*, Vol 24, No. 10, Oct 2006.
9. C. Kruegel and T. Toth. Using decision trees to improve signature-based intrusion detection. In *6th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.
10. T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, pages 203–214, 1998.
11. S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.
12. V. Paxson. Bro: A system for detecting network intruders in real-time. In *USENIX Security*, 1998.
13. R. Ramesh, I. Ramakrishnan, and D. Warren. Automata-driven indexing of prolog clauses. In *Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, San Francisco, 1990. Revised version appears in *Journal of Logic Programming*, May 1995.
14. M. Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference, USENIX*, 1999.
15. R. Sekar, I. Ramakrishnan, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1964. Elsevier Science, 2001.
16. R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Automata, Languages and Programming*, pages 247–260, 1992.
17. S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM*, 2003.
18. R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, 2003.
19. V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM '98*, pages 191–202, sep 1998.
20. T. Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM*, 2000.