Fast Packet Classification using Condition Factorization

Alok Tongaonkar and R. Sekar
Department of Computer Science, Stony Brook University.

ABSTRACT

Rule-based packet classification plays a central role in network intrusion detection systems, firewalls, network monitoring and access-control systems. 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. The key idea in our approach is that of decomposing and reordering the tests contained in the rules so that the result of performing a test can be utilized on behalf of many rules. Our experiments demonstrate dramatic reductions in space requirements over previous techniques, as well as significant improvements in matching speed. Our technique can uniformly handle prioritized and unprioritized rules, and support single-match as well as multi-match classification.

1. Introduction

A number of problems in network monitoring and security involve processing rules of the form "$cond \longrightarrow actn$," where $actn$ specifies the action to be taken on a packet that matches the condition $cond$. Given a set of rules $\{R_1, ..., R_n\}$, a *packet filter* determines if a given network packet matches the condition component of any of these rules. A *packet classifier* identifies the subset of rules that match the packet. They play a central role in:

- *Network intrusion detection* systems (NIDS) that match network packets with intrusion detection rules,
- *Firewalls* and *access control systems* that permit or deny incoming (or outgoing) network packets based on the conditions specified in firewall or access-control rules, and
- *Network monitoring systems* that selectively monitor, record, or analyze network packets that match the conditions specified in packet-filtering rules

Packet classifiers have to operate in real-time on high-speed networks without dropping packets or misclassifying them. The performance of naive techniques, which match one rule at a time, degrades quickly as the number of rules is increased. Such techniques repeat computations involved in matching: in particular, a test that occurs in multiple rules is tested once on behalf of each rule. This repetition can be avoided by building a finite-state automaton: the states of the automaton can be used to "remember" the tests already performed before reaching a state, and avoid repetitions. Unfortunately, previous research has established an exponential lower bound on the size of deterministic automata [18], even in the simple case where we are restricted to equality checks with constants. Previous packet classification techniques either suffered from this exponential worst-case complexity [9, 11], or relied on nondeterministic (aka "backtracking") automata [14, 5, 3, 2, 9] whose matching times, in practice, can increase quickly with the number of rules. Clearly, neither alternative is satisfying:

- Exponential blow-ups can't be tolerated since the number of rules can be large, e.g., several hundred to many thousands in the case of NIDS and firewalls.
- Even a modest rate of increase in matching time with number of rules may not be acceptable in high-speed networks. For instance, consider a technique whose matching time increases at the rate of $\sqrt{N}$, where N is the number of rules. It slows down by a factor of 50 when the number of rules is increased to 2500.

We present a solution to the packet classification problem that addresses both these problems: it guarantees a polynomial size matching automata in the worst-case, while providing matching times that, in practice, remain virtually constant. Our experimental results, performed in the context of firewalls and Snort NIDS, demonstrate substantial reduction in space requirements of automata, as well as significant improvements in matching time.

1.1 Overview of Approach and Contributions

- In Section 2, we define the notion of prioritized packet matching, where matches for lower priority rules can be announced only after ruling out the possibility of matching higher priority rules. Using partially ordered priorities, we can support different flavors of matching needed in the above applications, including packet filtering (applicable to network monitoring), single-match classification (applicable to firewalls) and multi-match classification (applicable to NIDS) . This contrasts with previous research efforts that addressed only one of these applications, e.g., packet-filtering (BPF [14], BPF+ [3], DPF [5] and PathFinder [2]), firewalls ([9]) and NIDS (Snort-NG [11]).
- In Section 3, we present a novel concept of *condition factorization* that provides the foundation for the many 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 automaton 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, and is more space-efficient than the Snort-NG [11] techniques.
  - 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:
  - 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 develop the notion of *utility,* which provides the basis for improving matching times.
- In terms of matching time, one of the contributions of this paper is that we develop a metric for matching time that can be understood independent of the specifics of underlying hardware or software implementations. In particular, assume that there exists an *oracle* that can guess the exact set of rules that match a given packet. We define a lower bound on the number of tests needed to verify the correctness of this guess. We then compare the number of tests performed by our technique to this number.

  Intuitively, match verification is significantly simpler than match identification, since a solution to the latter problem cannot rely on an oracle. Nevertheless, we show that in practice, our technique identifies matches using no more than twice the lower bound for match verification. Our results also show that as the number of rules is increased by over a factor of 100, the number of operations needed for matching a packet increase by only a factor of two, even in the worst-case.
- In Section 7, we describe our implementation, followed by an experimental evaluation in Section 8. 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 9, followed by concluding remarks in Section 10.

Although the techniques presented in this paper do not address content-matching components of NIDS signatures, it can still contribute significantly to improving NIDS performance. The most direct gains come from reductions in packet classification times, which, according to [6], constitutes a significant (although not dominant) fraction of the runtime of Snort [17], the popular open-source NIDS. More importantly, improvements in packet classification can contribute significantly to reduce content-matching times. In particular, researchers have been developing techniques to speed up content matching operations in NIDS by sharing the content-matching operations across multiple rules. Techniques such as deterministic finite state automata used for this purpose can require space that increases exponentially as the number of rules is increased. The technique presented in this paper can be used to first subdivide the rules based on packet fields. These subsets can be significantly smaller than the original rule sets — our experiments with the Snort default ruleset shows that the average size of subsets formed by our technique is one-third the size of the subsets formed by Snort. Since the size of DFA can increase exponentially with the size of subsets, we expect that our technique can contribute significantly to reducing the space needed for content-matching.

2. Preliminaries

In the rest of this paper, we are concerned only with the condition components of classification rules, which are referred to as *filters* henceforth. We associate a label with each filter to identify the corresponding rule.

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. A set $\mathcal{F}$ of filters may be partially ordered by a priority relation. The priority of F is denoted as $Pri(F)$.*

An example of a filter, as defined above, is

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

We do not consider more complex conditions that do not satisfy the definition of a filter, e.g.,

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

since they do not seem to arise in practice in our application domains (firewalls and NIDS).

A filter C can be "applied" to a network packet p, denoted $C(p)$, 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 (PRIORITIZED MATCHING). *For a set $\mathcal{F}$ of filters, we say that $F \in \mathcal{F}$ **matches a packet** p, denoted $M_{\mathcal{F}}(F, p)$, if $F(p)$ is true, and $F'(p)$ is $false$ $\forall F' \in \mathcal{F}$ that have a strictly higher priority than F. The **match set** of p, denoted $\mathcal{M}_{\mathcal{F}}(p)$ consists of all filters that match p, with the exception that among equal priority filters, at most one is retained in $\mathcal{M}_{\mathcal{F}}(p)$.*

Thus, a filter cannot match a packet unless matches with higher priority filters are ruled out. 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.

- If these filters have incomparable priorities, then 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 $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_2, F_3\}$
- If $Pri(F_1) > Pri(F_2) > Pri(F_3)$, then $\mathcal{M}_{\mathcal{F}}(p_1) = \{F_1\}$, and $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_2\}$.
- If $Pri(F_3) > Pri(F_2) > Pri(F_1)$, then $\mathcal{M}_{\mathcal{F}}(p_1) = \mathcal{M}_{\mathcal{F}}(p_2) = \{F_3\}$.

- If $Pri(F_1) = Pri(F_3) > Pri(F_2)$, then $\mathcal{M}_{\mathcal{F}}(p_1)$ can either be $\{F_1\}$ or $\{F_3\}$, while $\mathcal{M}_{\mathcal{F}}(p_2) = \{F_3\}$.

These examples illustrate how various flavors of matching can be captured using priorities.

- *Packet-filtering* can be done by setting equal priorities for all filters. By virtue of the definition of match sets, this priority setting causes a match to be announced as soon as a match for any filter is identified.
- *Ordered matching,* as used in firewalls and access control lists can be done by assigning priorities that decrease monotonically with the rule number.
- *Multi-matching,* as used in NIDS, can be solved by using incomparable priorities among filters.

Examples of *packet-matching automata* (also known as matching or classification automata) for the above filter set with incomparable priorities 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
  - 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. To motivate the need for defining residues, 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, $\mathcal{F}/C = \{F/C | F \in \mathcal{F} \wedge F/C \neq false\}$.

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., contains many disjunctions. (Our examples below illustrate this point). 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 $\overline{x}$ 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 $T/\neg T$).
- $(x = 5)/(x\&\texttt{0x3} \neq 1)$ is $false$, as given by the 5th row.
- for $(x = 5)/(x\&\texttt{0x3} \neq 0)$, 5th row is no longer applicable since the condition $c\&c_1 = c_2$ does not hold. (Here, $c = 5$, $c_1 = \texttt{0x3}$, and $c_2 = 0$.) Hence the applicable row is the last row, which yields $(x = 5)/(x\&\texttt{0x3} \neq 0) = (x = 5)$. The result is understandable: although the two conditions are compatible with each other, the test $x\&\texttt{0x3} \neq 0$ does not contribute to proving $x = 5$.

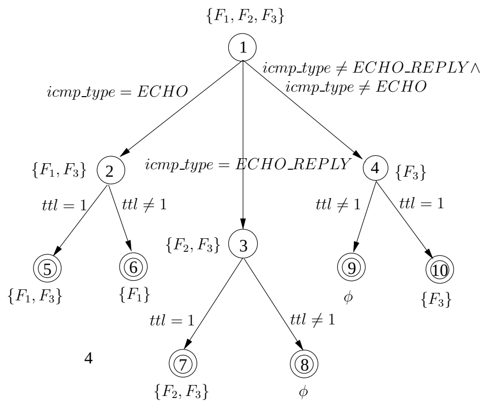

Figure 1: A deterministic matching automaton.

- $(x \in [1, 10])/(x \neq 5)$ 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	$x = c$	$T[x \leftarrow c]$	
$x = c$	$x \,\&\, c_1 = c_2$	$x \,\&\, \overline{c_1} = c \,\&\, \overline{c_1}$	$c \,\&\, c_1 = c_2$
		$false$	$c \,\&\, c_1 \neq c_2$
$x = c$	$x \,\&\, c_1 \neq c_2$	$false$	$c \,\&\, c_1 = c_2$
$x = c$	$x \in [c_1, c_2]$	$false$	$c \notin [c_1, c_2]$
$x \neq c$	$x \,\&\, c_1 = c_2$	$x \,\&\, \overline{c_1} \neq c \,\&\, \overline{c_1}$	$c \,\&\, c_1 = c_2$
		$true$	$c \,\&\, c_1 \neq c_2$
$x \neq c$	$x \,\&\, c_1 \neq c_2$	$true$	$c \,\&\, c_1 = c_2$
$x \neq c$	$x \in [c_1, c_2]$	$true$	$(c < c_1)$ $\vee (c > c_2)$
$x \in [c_1, c_2]$	$x \in [c_3, c_4]$	$true$	$c_1 \leq c_3$ $\leq c_4 \leq c_2$
		$x \in [-\infty, c_2]$	$c_1 \leq c_3$ $\leq c_2 \leq c_4$
		$x \in [c_1, \infty]$	$c_3 \leq c_1$ $\leq c_4 \leq c_2$
		$x \in [c_1, c_2]$	$c_3 \leq c_1$ $\leq c_2 \leq c_4$
		$false$	$(c_2 < c_3)$ $\vee (c_4 < c_1)$
$x \in [c_1, c_2]$	$x \,\&\, c_3 = c_4$	$false$	$c_4 > c_2$
$x \,\&\, c_1 = c_2$	$x \,\&\, c_3 = c_4$	$x \,\&\, (c_1 \,\&\, \overline{c_3})$ $= (c_2 \,\&\, \overline{c_3})$	$c_2 \,\&\, c_3$ $= c_1 \,\&\, c_4$
		$false$	otherwise
$x \,\&\, c_1 = c_2$	$x \in [c_3, c_4]$	$false$	$c_2 > c_4$
$x \,\&\, c_1 \neq c_2$	$x \,\&\, c_3 = c_4$	$x \,\&\, (c_1 \,\&\, \overline{c_3})$ $\neq (c_2 \,\&\, \overline{c_3})$	$c_2 \,\&\, c_3$ $= c_1 \,\&\, c_4$
		$true$	otherwise
$x \,\&\, c_1 \neq c_2$	$x \in [c_3, c_4]$	$true$	$c_2 > c_4$
T	T'	T	

Figure 3: Computation of Residue on Tests.

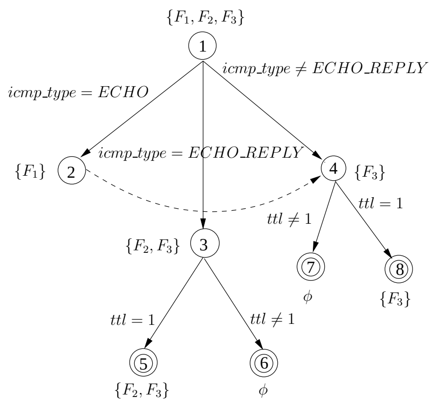

Figure 2: A non-deterministic matching automaton.

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: $\mathcal{C}_s$, the *candidate set* of the state s, and $\mathcal{M}_s$, the *match set* of s. $\mathcal{M}_s$ consists of all filters for which a match can be announced at s, while $\mathcal{C}_s$ consists of those filters that haven't completed a match, but future matches can't be ruled out either. In other words, the elements $\mathcal{C}_s$ will appear in the match sets of some descendants of s.

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.,

```
1.  procedure $Build(s, \mathcal{C}_s, \mathcal{M}_s)$
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 */
          /* $d_i \in D$ indicates if this transition is deterministic */
6.      $T_o = \{\bigwedge_{d_i \in D | d_i = true} \neg T_i\}$
          /* Compute test corresponding to the "other"-transition */
7.      for each $T_i \in (\mathcal{T} \cup \{T_o\})$ do
8.        $\mathcal{C}_i = \mathcal{C}_s / T_i$
9.        if $((T_i \neq T_o) \wedge \neg d_i)$ then $\mathcal{C}_i = \mathcal{C}_i - \mathcal{C}/T_o$ endif
            /* For a non-deterministic transition, do not duplicate */
            /* filters from the "other" branch */
10.       compute $\mathcal{M}_{s_i}$ and $\mathcal{C}_{s_i}$ from $\mathcal{C}_i$ and $\mathcal{M}_s$
11.       if a state $s_i$ corresponding to $(\mathcal{C}_{s_i}, \mathcal{M}_{s_i})$ isn't present
12.         create a new state $s_i$
13.         $Build(s_i, \mathcal{C}_{s_i}, \mathcal{M}_{s_i})$
14.       endif
15.       create a transition from $s$ to $s_i$ on $T_i$
16.     end
17.   endif
```

Figure 4: Algorithm for Constructing Matching Automaton

$(icmp\_type = ECHO))$ 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 $\{F_3\colon (ttl = 1)\}$.

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 $T_1, ..., T_k$ 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 non-deterministic 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 are computed. (The procedure for computing match and candidate sets is described in Section 4.1.)

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.

4.1 Computing Match and Candidate Sets

Given a state s of a matching automaton for a prioritized filter set $\mathcal{F}$, we denote the conjunction of tests on the path from the start state to s by $\mathcal{P}_s$. We can compute the match set $\mathcal{M}_s$ corresponding to an automata state s using the following steps:

- $\mathcal{M}_1 = \{M \in \mathcal{F}/\mathcal{P}_s | (M = true)\}$, i.e., $\mathcal{M}_1$ consists of those filters that are implied by the conditions examined on the automaton path reaching s.
- $\mathcal{M}_2 = \{M \in \mathcal{M}_1 | \neg\exists M' \in \mathcal{F}/\mathcal{P}_s \; Pri(M') > Pri(M)\}$, i.e., $\mathcal{M}_2$ is obtained by deleting those filters from $\mathcal{M}_1$ for which a future match with higher priority filters can't be ruled out.
- $\mathcal{M}_s$ is obtained by considering filters with equal priorities in $\mathcal{M}_2$, and deleting all but one of them.

Now, $\mathcal{C}_s$ can be computed using the following two equations:

$$\mathcal{C}_s = \mathbf{C}(\mathcal{F}/\mathcal{P}_s, \mathcal{M}_s)$$
$$\mathbf{C}(\mathcal{F}, \mathcal{M}) = \{C \in \mathcal{F} \,|\, \neg\exists M' \in \mathcal{M} \text{ with } Pri(M') \geq Pri(C)\}$$

These equations can be interpreted procedurally as follows. First, identify the list of all filters that are compatible with the automaton path reaching s. Next, eliminate filters that are superceded by higher (or equal) priority filters for which a match has already been completed.

Note once again that we are computing residues of original filters, thereby conveniently keeping track of those tests in each filter that haven't yet been performed. (Or more accurately, we are keeping track of those tests that aren't already known to be satisfied.) In Figures 1 and 2, we have annotated final states with match sets, and non-final states with the union of match and candidate sets.

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 size 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 filters[1].

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, $select$

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

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 [18], and was then generalized to deal with binary data in [9]. 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 leftmost 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 distinguished[2], and hence can form the transitions from s. Moreover, it has been shown [18] 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 automataon that is based entirely on discriminating tests will have at most $O(n)$ breadth.

[2]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 [18]. Although some of the previously developed techniques can avoid such explosion, this has been accomplished at the cost of introducing significant backtracking at runtime [14, 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 [16, 9]. With large filter sets that are often found in enterprise firewalls and 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.

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

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.

The above technique can be extended further: rather than looking at one level of the automaton at a time, we could examine all of the ancestors of a state s, and ensure that collectively, they stay within the budget permitted by $P(n)$. This would permit a greater degree of overlap at s if the degree of overlap among (the children of) the ancestors of s was smaller than the budget.

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*

- *$Pri(F_1)$ and $Pri(F_2)$ are either equal or incomparable,*
- *for every test T in F_1, $F_2/T = F_2$, and*
- *for every test T in F_2, $F_1/T = F_1$.*

$\mathcal{F}_1$ 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[3]. 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, especially in the case of NIDS rules, 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 subsets such that any two rules F_1, F_2 such that $Pri(F_1) > Pri(F_2)$ are in a single subset. 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 $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$, instead of a test set. At this point, $Build$ will create a k-way non-deterministic branch. On the ith branch, it will invoke $Build(s_i, \mathcal{F}_i, \mathcal{F}_i \cap \mathcal{M}_s)$.

6. Improving Matching Time

To reason about matching time, we need to define a function that assigns computational costs to each test. A simple cost model is one that assigns unit cost to all tests. Note that such a measure would treat tests on 1-bit fields the same as on 32- or 64-bit fields. While this may seem reasonable, it does not capture the intuition that checking a test $y\&\texttt{0xff} = 3$ contributes partially towards checking $y = \texttt{0x703}$. For this reason, we prefer to use a measure that assigns a cost of r to tests involving r-bit quantities. In this case, $cost(y\&\texttt{0xff} = 3)$ will be 8, while $cost(y = \texttt{0x703})$ will be 16, assuming y is a 16-bit field. However, to simplify our presentation, we will use the uniform cost model below, and ignore priorities. Our technique for reducing matching time is based on the following notion:

DEFINITION 6 (UTILITY). *The utility $U_s(T, F)$ of a test T at an automaton state s for a filter $F \in \mathcal{C}_s$ is*

- *0, if a match for F is ruled out when T is satisfied*
- *$cost(F) - cost(F/T) - cost(T)$, otherwise.*

The utility U_s of the set $\mathcal{T}$ of tests on the transitions from s

[3]For the packet-filtering case, characterized by equal priorities among all filters, we need to match with the second automaton only if the first automaton reports no matches.

is the weighted average,

$$\frac{\sum_{F\in\mathcal{C}_s}\sum_{T\in\mathcal{T}} U_s(T,F)}{|\mathcal{T}| * |\mathcal{C}_s|}.$$

We assume that filters do not contain redundant tests. In this case, the utility value can never be greater than zero. A negative value, which indicates that potentially unnecessary computation was carried out, is characterized by the fact that a test T costs more to perform than the cost it takes away from future tests that need to be performed for verifying a match for F. The lowest possible value of $U(T,F)$ is $-cost(T)$.

Our technique for improving matching time is based on choosing tests that have high utilities. Our implementation of $select$ places more importance on size reduction than matching time. As such, it chooses test sets that maximize utility among those that minimize size.

6.1 Measuring Match Time

We now proceed to develop an implementation-independent metric for quantifying the overall matching cost of an automaton. Such a metric is preferable to raw runtimes that are heavily influenced by low-level implementation decisions. For instance, since our matching automaton is compiled into native code, it is many times faster than some of the techniques that we compare against. Thus the raw numbers don't necessarily reflect the benefits obtained using the algorithms developed in this approach, which are applicable to compiled as well as interpretation-based implementations.

Our metric is based on lower bounds on *match verification cost.* In particular, suppose that there exists a *non-deterministic matching algorithm* that can "guess" the subset of rules that match a given packet p, and then proceeds to verify the correctness of this guess. One can reasonably expect that a *deterministic* matching algorithm would need to perform more computation than such a non-deterministic algorithm. For this reason, a deterministic algorithm that comes fairly close to the lower bound for nondeterministic algorithms, say, within a factor of two, could be considered a very good algorithm. We therefore use the ratio of actual matching cost to the lower bound for match verification cost as a metric for evaluating an automaton. In our experiments, we computed this metric statically: in particular, we computed the average of this ratio across all paths in the automaton.

OBSERVATION 7 (MINIMUM MATCH VERIFICATION COST).

- *The lower bound for verifying a successful match of a filter F is $O(|F|)$.*
- *The lower bound for verifying a successful match of all filters in a set $\mathcal{M}$ is $O(k)$, where k is the number of distinct fields (or distinct number of field and bit-mask combinations) tested across all the filters in $\mathcal{M}$.*

It is clear that a match cannot be announced without testing all conditions in F and hence the bound in the first case. In the second case too, it is clear that all the fields present in all the filters in $\mathcal{M}$ have to be examined before announcing a match for all of them, and hence $O(k)$ time is needed.

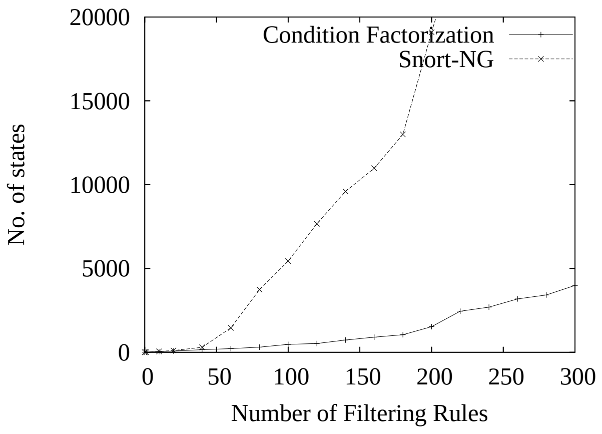

Figure 5: **Automaton Size for Snort Rules**

7. 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, and match and candidate sets computed as specified in Section 4.1. Our $select$ implementation proceeds as follows:

- $select$ first attempts to find a discriminating test set (Section 5.1). If several of them exist, our technique selects a set that maximizes utility.
- 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 with zero (i.e., maximum possible) utility, 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.

8. Evaluation

We evaluated the effectiveness of our techniques in the context of NIDS (Section 8.1), firewalls (Section 8.2) and packet-filtering (Section 8.3). Our experiments were performed on a system with 1.70Ghz Pentium 4 processor and 520MB memory, running CentOS-4.2 (Linux kernel 2.6.9).

8.1 Experiments using IDS

For our experiments we use Snort [17] which is a popular open source NIDS. Snort comes with default signatures that are comprehensive and up-to-date. Snort signatures consist of two main components: tests involving packet fields, and content-matching operations on the payload. According to [6], packet classification and content-matching are the most expensive parts of Snort, accounting for 21% and 31% of the execution time. Within NIDS research community, researchers have been investigating techniques for parallelizing packet-field matches as well as content matches. The primary goal of this paper is to address packet-field matching, and hence our evaluation focusses only on this component of NIDS.

Although earlier versions of Snort relied largely on sequential matching (i.e., matching a packet against one signature at a time), newer versions of Snort (specifically, version 2.0 and later) attempt to match the signatures in parallel.

In contrast, Kruegel and Toth developed the Snort-NG [11] system, which demonstrated the performance gains achievable by parallelizing the signature matching. They use an entropy-based algorithm to decide which packet field to test at each node. Their technique is the only one that we are aware of that uses a sophisticated packet-classification algorithm for Snort-type rules. Hence we compare our performance results with them. To simplify this comparison, we used the default ruleset that comes with Snort-NG, which consists of 1635 rules. Since our focus is only on matching packet fields, we combined the rules that differ only in terms of payload contents. This resulted in a ruleset with 305 unique rules.

Bro [15, 20] is another popular NIDS for which a comprehensive set of signatures is freely available. Unlike Snort, which uses signatures that are based on individual packets, Bro performs matching on data streams obtained after packet reassembly. As noted in their paper [20], this factor complicates a direct comparison with a Snort-based technique such as ours, which assumes packet boundaries, and operates on one packet at a time. As a result, we do not provide a comparison of runtimes with their approach.

8.1.1 Automaton Size

We provided `Snort NG` and our technique with the same set of 305 rules. The `Snort NG` algorithm suffers from a space explosion if all of the rules are put into a single decision tree. To cope with this, they arbitrarily divide the rules into several subsets, and build multiple decision trees, which can degrade runtime performance. Our technique builds a single deterministic automaton.

Figure 5 shows the effect of increasing the number of rules on the number of automaton states. We note that `Snort NG` decision tree has a complicated structure, where some of the states do not perform any tests, but are used to identify the type of field being tested. For our experiments we count only the states which actually perform some test. 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 45,000 states whereas the automaton constructed by our technique has only about 4000 states. *This translates to a size reduction by a factor of about 10.*

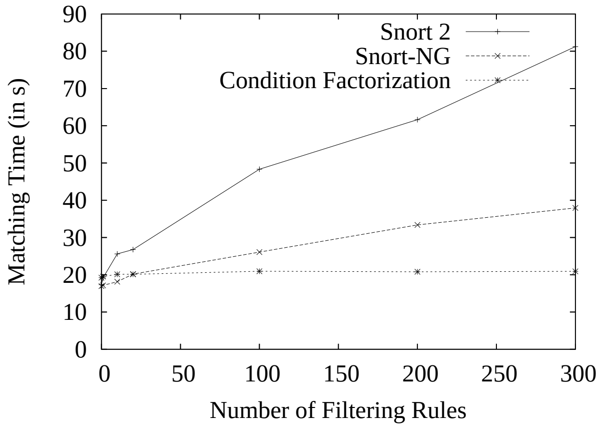

Figure 6: **Matching Time for Lab Traffic**

8.1.2 Matching Time

For measuring runtime performance, we used two sets of data. The first one was consists of all packets captured at the external firewall of a medium-size University laboratory that hosts 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. Our packet trace consisted of about 21 million packets collected over a few days. Figure 6 shows the matching time taken by Snort, Snort-NG and our technique for classifying these packets as the number of rules change.

We also used a second packet trace for performance measurement. This data corresponds to 10 days of packets from the MIT Lincoln Labs IDS evaluation data set [12], consisting of 17 million packets. While there has been some criticism of this data for the purpose of evaluating IDS, they primarily concern artifacts in the data that may make it easier to detect attacks. Since our focus is not on evaluating the quality of the ruleset, these concerns are not that significant in our context. Moreover, we note that the results obtained with both data sets are similar.

In the Figures 6 and 7, 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

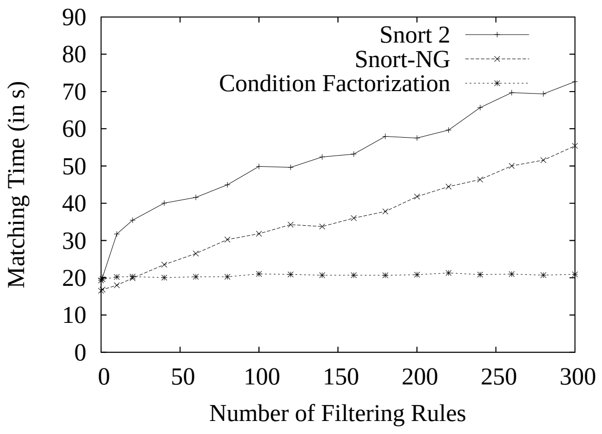

Figure 7: **Matching Time for Lincoln Lab Traffic**

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.

One of the reasons for a drastic difference in the rate of increase in matching time is due to the fact that we compile our automata into native code, whereas Snort-NG and Snort use an interpreted approach. We note here that the packet classification code that we generate is compiled into a shared library. So to update the rules, all that one needs to do is to compile the rules offline and then reload the shared library. We note that this approach is no more disruptive than that of Snort where the rules need to be re-read and recompiled.

In order to better understand the effectiveness of our technique in reducing the matching time, we compared the cost of our automata with the lower bounds for match verification cost in Figure 8. Our results show that our matching cost is within a factor of two from this bound. More remarkably, the number of tests performed increases by only a factor of 3 when the number of rules is increased from 1 to 300.

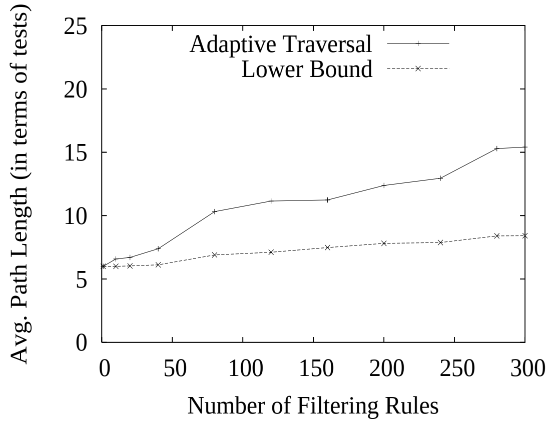

Figure 8: **Path Length for Snort Rules**

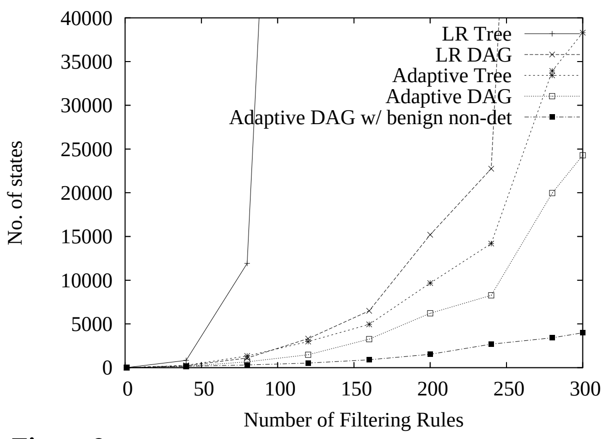

Figure 9: **Effect of Optimizations on Automaton Size for Snort Rules**

8.1.3 Impact of Our Technique on Content Matching

Although our technique does not address content-matching directly, it can have significant benefits for content-matching (both in terms of automata sizes and runtime). In particular, there has been a lot of interest in speeding up string matching operations by sharing common matching operations across different rules. Opportunities for sharing are maximized by building a deterministic finite-state automata (DFA) for the string matching operations. Unfortunately, use of DFAs can cause massive explosion in the size of the automaton, so techniques are needed to control this size explosion. A technique such as ours reduces the opportunities for explosion by using space-efficient packet-field based classification. Now, at each final state of this classification automata, we are left with subsets of the original rule set that are significantly smaller than the original rule set, thereby correspondingly decreasing the likelihood of an explosion due to string matching. In our experiments with the 1635 default Snort rules containing string matching operations, we observed that the average size of sets for string matching using our technique was *3 times smaller* than the groups formed by Snort 2.6.

Moreover, Fisk and Verghese [6] have shown that performance of set-wise algorithms for string matching depend on the size of the sets. They conclude that the optimal solution is to have sets of pattern strings for every rule that matches that packet, but no additional strings. Since we use this 2-stage approach for signature matching we expect to have this actual runtime improvements.

8.1.4 Effect of Optimizations on Automaton Size

Figure 9 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 L-R order for examining packet fields, our technique (which uses the $select$ function as described in Section 7 produces tree

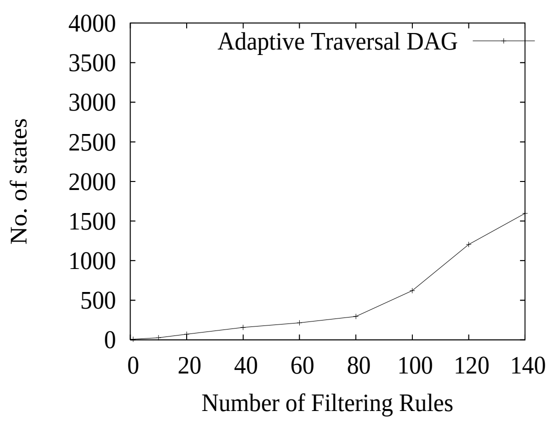

Figure 10: **Automaton Size for Firewall Rules**

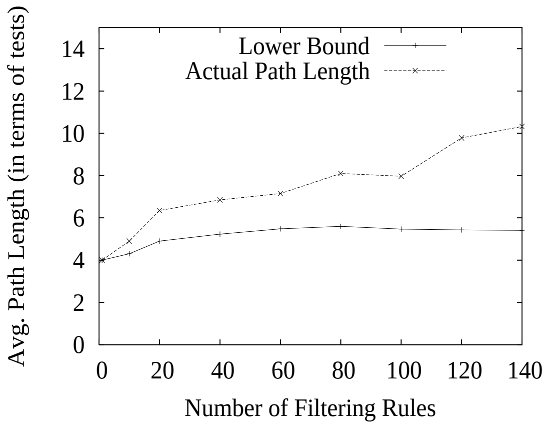

Figure 11: **Matching Time for Firewall Rules**

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 apply in the case of L-R technique.

8.2 Experiments with Firewall Rules

The firewall ruleset we considered is typical for a small to medium scale organization such as a department in a University. It divides a network into several subnets: the main network (all servers, workstations, etc), DMZ network, a wireless network, and a testbed network. The firewall is used for the traffic between these subnets and also between the outside world. The firewall rules are in the form of iptable rules for a Linux machine. There are different chains of rules for each of the subnets. Excluding the rules for defining and branching to user-defined chains, there were 140 actual filtering rules.

Figure 10 shows the automaton size as a function of the number of rules. The automaton size increases at a somewhat faster rate than in the case of NIDS because firewall rules are totally ordered in terms of priorities. As a result, they can never have independent subsets of filters, and hence the benign nondeterminism technique cannot be applied.

Figure 11 compares the cost of our automata with the lower bounds for match verification. Although the results in this case seem similar to that obtained for NIDS rules, we point out that they are actually better than what they appear to be. In particular, to verify a match for a filter F in the presence of priorities, it is not sufficient to just verify if the tests in F hold, but we also need to verify that at least one of the tests in each of the higher priority rules don't match. As a result, the match verification lower bound is strictly higher than the number for unprioritized rulesets that arise in the context of NIDS.

8.3 Experiments using Detecting Backdoors Signatures

To illustrate the difference in the automaton generated by our technique and BPF+ [3], we used the signatures used for detecting backdoors [23]. We observed that as new filters are added, the maximum path length of their automaton increased to about 3 times that of our automaton. To ob-

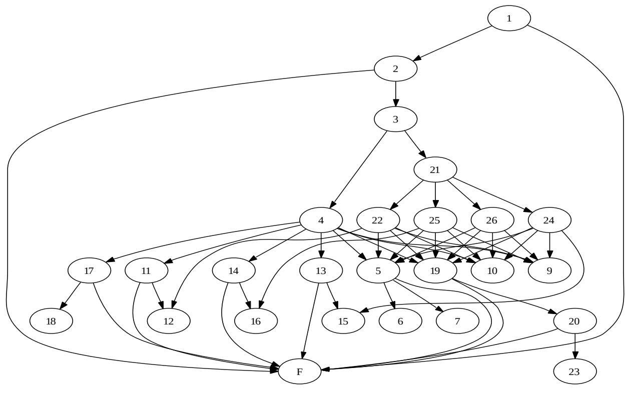

Figure 12: Automata constructed by our technique

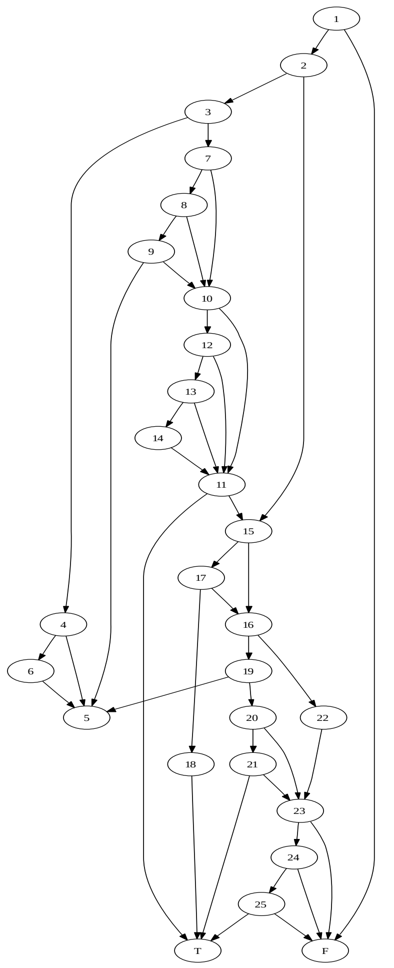

Figure 13: Automata constructed by BPF+

tain this result, we instrumented a version of `pcap` library on Solaris that uses BPF+ to compile filter expressions to generate the corresponding graphs. The filters used to detect backdoors for applications like ssh and ftp were *OR*'d to get a combined filter which could detect any of the backdoors. Figure 13 shows that even for a small number of simple filters the height of automaton is significantly larger than of the automaton generated by our technique (Figure 12). This suggests that the matching time for BPF+ can be significantly more than with our technique.

9. Related Work

A number of related works have already been discussed in the paper, so we focus our discussion here to those works that haven't been covered before.

Srinivasan et al [21] and Lakshman et al [13] cast packet classification as a multidimensional searching problem. Woo [22] and Gupta et al [8] presented decision tree based techniques for packet classification. Singh et al [19] improve on the performance of these techniques by performing tests on multiple fields at a node. This performance improvement can be at the cost of storage. All these techniques are targeted at routers, where they can restrict the problem so as to work on a fixed number of fields. Our focus in this paper is on techniques that do not place such restrictions.

Pattern matching automata have been extensively studied in the context of term rewriting, functional and equational programming, theorem proving and rule-based systems. Augustsson described pattern matching techniques for functional programming that are based on left-to-right traversal [1]. BPF [14], DPF [5], Pathfinder [2], and BPF+ [3] also use left-to-right traversal, but work on network packet that are more complex than terms used in functional programming. To improve matching time, DPF and Pathfinder combine common prefixes to share some tests. BPF+ uses global dataflow techniques to identify more 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.

DPF uses dynamic code generation, which allows dynamic reordering of tests. Dynamic reordering improves performance by detecting match failures earlier. Al-Shaer et al [10] and Gupta et al [7] 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 [15] developed Bro which is another popular NIDS. Bro has a powerful policy language that allows the use of sophisticated signatures. Sommer and Paxson [20] enhanced Bro signature matching to use regular expressions. They build DFAs from the regular expressions. They overcome the problem of exponential space requirements of DFAs by building the DFAs incrementally at runtime. Moreover, they mention using the constraints on packet fields to reduce the size of the sets of signatures that are compiled into a DFA. In that respect, our technique can help them in avoiding the exponential blowup, as we generate small sets for content-matching.

Kruegel et al [11] present an approach for avoiding redundancy, where, by restricting the form of allowable tests, every test was converted 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. Moreover, the canonical form places more restrications on the form of tests.

Sekar et al [18] presented a technique for adapting the traversal order to reduce space and matching time complexity of term-matching automata. Gustafsson and Sagonas [9] extended this technique to handle binary data like network packets. Our technique generalizes their technique further by adding support for inequalities, disequalties, and bit-masks that are more general than their notion of bit-fields. More importantly, their automata has an exponential worst-cast 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. In particular, their transitions, strictly speaking, become $O(N)$ operations, which contrasts with our approach that takes $O(1)$ expected time per transition.

10. Conclusions

In this paper we presented a new technique for fast packet-matching. Unlike previous techniques, our technique is flexible enough to support filtering as well as classification applications. It can support prioritized rules such as those used in firewalls, as well as unprioritized rules requiring all matches to be reported, such as those used in intrusion detection systems. We developed novel techniques and algorithms that guarantee polynomial size automata, while, in practice, avoiding any repetition of tests. Our experimental results show that the technique is very effective in reducing automata size as well as matching time. In the context of network intrusion detection, our technique constructed automata that were over ten times smaller than some of the previous techniques. Matching times were significantly better than previous techniques, and remained virtually constant, even as the number of rules was increased by two orders of magnitude.

11. References

[1] L. Augustsson. Compiling pattern matching. *Functional Programming and Computer Architecture*, 1985.

[2] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.

[3] Andrew Begel, Steven McCanne, and Susan 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] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, 1996.

[6] M. Fisk and G. Varghese. Fast contentbased packet handling for intrusion detection, 2001.

[7] P. Gupta and N. McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[8] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*, 1999.

[9] Per Gustafsson and Konstantinos Sagonas. Efficient manipulation of binary data using pattern matching. *J. Funct. Program.*, 16(1):35–74, 2006.

[10] Ehab Al-Shaer 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.

[11] Christopher Kruegel and Thomas Toth. Using decision trees to improve signature-based intrusion detection. In *6th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.

[12] MIT Lincoln Labs. Darpa intrusion detection evaluation, 1999.

[13] T. V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, pages 203–214, 1998.

[14] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[15] V. Paxson. Bro: A system for detecting network intruders in real-time. In *USENIX Security*, 1998.

[16] R. Ramesh, I.V. Ramakrishnan, and D.S. 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.

[17] Martin Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference, USENIX*, 1999.

[18] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Automata, Languages and Programming*, pages 247–260, 1992.

[19] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM*, 2003.

[20] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, 2003.

[21] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM '98*, pages 191–202, sep 1998.

[22] Thomas Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM*, 2000.

[23] Yin Zhang and Vern Paxson. Detecting backdoors. In *Proc. 9th USENIX Security Symposium*, pages 157–170, aug 2000.