Condition Factorization: A Technique for Building Fast and Compact Packet Matching Automata

Alok Tongaonkar and R. Sekar

***Abstract*—Rule-based matching on network packet headers is a central problem in firewalls, and network intrusion, monitoring and access-control systems. To enhance performance, rules are typically compiled into a *matching automaton* that can quickly identify the subset of rules that are applicable to a given network packet. While *deterministic* automata provide the best performance, previous research has shown that such automata can be exponential in the size and/or number of rules. Nondeterministic automata can avoid size explosion, but their matching time can increase 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 on packet header fields so that the result of performing a test can be utilized on behalf of many rules. Our experiments demonstrate major 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 applications that require single-match as well as multi-match.**

***Index Terms*—Packet Classification, Intrusion Detection Systems, Firewalls, Network Monitoring.**

I. Introduction

Firewalls, network monitoring and intrusion detection systems (NIDS) are ubiquitous today. These systems process network packets according to a set of rules:

- *Firewalls* and *access control systems* permit or block network packets based on the conditions specified in firewall or access-control rules. These systems typically look for the *first matching rule* in a linearly ordered rule set.
- *Network intrusion detection systems* such as Snort [26] define suspicious activity in terms of conditions over network packet contents. To ensure that all attacks are detected, NIDS need to identify *all matching rules.*
- *Network monitoring applications* monitor or record a subset of network packets as specified by a set of rules. These systems are concerned with *packet filtering*, i.e., detecting *the presence of a match,* without necessarily identifying the specific rule that matched.

Many of these applications use rules involving packet header fields, while others (e.g., NIDS) rely on a combination of header field matching and *deep packet inspection* that uses string and regular-expression matching on packet payload. The focus of this paper is on efficient packet field matching, a problem that has often been called *packet classification* in the literature. While we do not address string or regular expression matching directly, we show that our approach can be used to narrow down the size of rule sets involved in payload matching, and that this factor can improve the overall performance of systems such as Snort.

Firewalls, network monitoring systems and NIDS face the dual challenges of massive increases in network traffic volumes, and rapid increases in rule set sizes. As a result, these systems need to process many more rules in a much shorter time per network packet. This paper therefore develops techniques for improving the performance of *packet matching:* given a network packet p and a set of *signatures,* which capture a set of conditions on packet fields, identify the subset of signatures that can match p. Our focus in this paper is on software implementations that can run on a general-purpose CPU, a configuration that is common for NIDS such as Snort, as well as many firewalls and network monitoring and logging systems that run on general-purpose hardware.

A naive technique for packet matching is that of sequentially matching each signature against every incoming packet. The performance of such a technique degrades linearly with the number of signatures. To speed up the process, a natural approach is to build a search-tree-like data structure that can be used to quickly narrow down the set of signatures applicable to a packet. The packet can then be matched sequentially against this subset, and this second phase can include more complex operations such as regular expression matching that were not included in the search tree. The popular NIDS Snort builds such a data structure based on a handful of predefined header fields that occur in virtually all rules, such as the protocol (e.g., IP or ICMP) and the source and destination ports.

Limiting to a small predefined attribute set simplifies the search-tree and ensures its compactness. But the drawback is that the number of signatures that remain applicable at a leaf node can be large, thereby slowing down the sequential match phase. To overcome this problem, it is necessary to develop search tree construction algorithms that leverage as many packet fields as possible, instead of limiting to a small number of predefined attributes. However, previous research [31, 32] has shown that the size of such search tree structures can *increase exponentially with the number of signatures*.

Another challenge is that different applications require different flavors of packet matching. Firewalls and NIDS require the identification of the matching rule, while packet filtering does not. Moreover, applications such as NIDS don't rely on rule priorities, while firewalls do. Finally, some applications just need the first match, while others (e.g., intrusion detection) require all matches. We therefore develop a new approach that addresses all these flavors of matching within a uniform framework. Our approach improves matching speed using a novel technique called *condition factorization* that breaks

Alok Tongaonkar (atongaonkar@acm.org) is currently with Symantec. R. Sekar (sekar@cs.stonybrook.edu) is with Stony Brook University, NY, USA.
Research funded in part by NSF grants CNS 0627687, 0831298 and 1319137.
Manuscript received September 10, 2014; revised March 29, 2015.

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, and so on.

Overview of Approach and Contributions

- In Section III, we formalize *prioritized packet matching,* and show how it can capture the flavors of matching required in firewalls, NIDS, and packet filtering applications.
- In Section IV, we develop the concept of *condition factorization,* 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 V we present our automaton construction algorithm. Condition factorization, the core operation behind this algorithm, forms the basis of two key optimizations:
  - It can reason about the relationships between typical tests such as equalities, inequalities and bit-masking operations, and leverage them to avoid *semantically redundant tests.* This is more powerful than the (passive) common subexpression elimination techniques used in previous approaches such as BPF+[5]. Our technique proactively creates opportunities for sharing computation, as shown in Section IX-A.
  - 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* reduces space and time requirements for automaton construction by as much as an exponential factor.
- In Section VI and VII, we present several additional techniques for building space- and time-efficient automata:
  - In Section VI-A, 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 VI-B that ensures polynomial space bounds by trading off some determinism.
  - In Section VI-C, we develop the notion of *benign nondeterminism*, which achieves *substantial space reduction without degrading matching time.*
  - In Section VII, we develop the concept of *utility* that measures a test's contribution in determining a match. By picking tests with high utility values at each state, we can reduce the matching time of the automaton.
- In Sections VIII and IX, we describe our implementation and experimental evaluation. Our algorithms achieve major reduction in space requirements for packet header matching, while also improving matching time. Moreover, the match time remains virtually constant, regardless of the number of rules. In our experiments with Snort, this improvement led to a 30% reduction in overall matching time that includes both header and payload matching.

II. Related Work

Early Works on Packet Classification. A number of previous research efforts on packet header matching were targeted at routers, where all of the rules examine the same set of fields such as the source and destination IP addresses and ports. For each of these fields, the tests can examine if the field is within a range, or if the prefix bits have a certain value. In contrast, our focus is on rule sets where each rule may examine a different set of fields, and moreover, disequality and bit-masking operations are used. Such rules are common in NIDS, e.g., the default rule set that ships with Snort contains rules that examine a total of about 15 fields, with each rule referencing only a small subset of them. Moreover, four of these 15 fields contain tests that use bit-masking operations.

Taylor [38] presents a comprehensive survey of these early works on packet classification. He divides them into three main categories: (i) decision-tree based, (ii) decomposition based, and (iii) tuple search based techniques. Decision tree based techniques are the closest to the technique we present here. One of the seminal works in this area is HiCuts [13], which views the classification problem geometrically. Each filter is viewed as a hyper rectangle in a d-dimensional space, where d is the number of packet fields used in the filters. A test performed at a tree node corresponds to a *cut* in the hyper space. Unfortunately, n filters with d fields can lead to $O(n^d)$ distinct hyper rectangles, with each region corresponding to a distinct leaf in a decision tree. Thus, the worst case space use can be $O(n^d)$. HyperCuts [33] improves the HiCuts algorithm by selecting multiple cuts at each level. While this cuts down the number of interior nodes, it does not fundamentally alter the lower bound on the number of leaves. In particular, Vamanan et al [41] report that on rule sets of sizes 1K to 100K, HyperCuts duplicates rules by a factor of thousands. They therefore propose EffiCuts, which reduces rule duplication among the decision tree leaves by (a) separating the rules into different groups based on their overlap and (b) creating a separate decision tree for each group. They report orders of magnitude reduction in space usage over HyperCuts, but the lookup cost increases since multiple trees need to be traversed for matching a packet. In contrast, we identify two criteria (Sections VI-A and VI-C) by which rule duplication can be avoided without impacting matching time. Another contribution of ours is that we present techniques to ensure polynomial worst-case space use, whereas the above works don't establish any polynomial bound.

Decomposition based techniques decompose multi-field search problem into several instances of a single-field search and aggregate the results. These techniques tend to provide high throughput in hardware due to their amenability to parallel implementation. Cross-producting [37] is perhaps the most representative work in this category. It performs independent

field searches over d fields and then combines the results in a single step. It essentially precomputes the matching filter for every possible combination of results from the d field searches. However, the size of the cross-product table for a set containing n filters can grow to $O(n^d)$, which is acceptable only for small values of d. Gupta et al [12] analyzed several rule sets in use, and identified several important characteristics. Based on these observations, they developed a new technique called Recursive Flow Classification (RFC) that provided high lookup rates in practice. However, no improvement in worst-case space usage was established, in contrast with our polynomial bound.

Tuple search based techniques initially targeted rules consisting only of prefix tests. By dividing the rule sets into groups that examine the same prefix bits for equality, matches within each group can be performed using hashing. However, each packet needs to be matched separately with each distinct tuple. Tuple space techniques can take advantage of parallelism by performing independent probes for separates tuples. However, in the absence of parallelism (e.g., on general-purpose hardware, where the memory accesses needed for these probes will need to be serialized), it is similar to exhaustive search but performs better as the number of distinct tuples is less than the number of filters. (In their experiments, the reduction was by a factor of 4 to 7.)

Hardware-based Techniques. There are many works that exploit hardware based parallelism using either TCAM [44] or GPU [42]. The core of most TCAM based techniques is an exhaustive search, but the search is highly parallelized to achieve very good performance. GPU-based techniques rely on parallelism available on modern GPUs. These techniques are complementary to ours: our focus in this paper is on improving the performance of software-based packet matching techniques that can run on general-purpose hardware.

Term-rewriting. This is another domain where early work on matching complex structures was performed [30]. Terms are used to model composite data in declarative languages, in much the same way `structs` are used in C/C++. Sekar et al [31, 32] presented a technique for adapting the order of examination of fields in order to reduce the space and matching time complexity of term-matching automata. Gustafsson et al [14] 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-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. Consequently, their transitions have an $O(n)$ complexity (where n is the number of filters), while our transitions are $O(1)$ expected time.

Program analysis. Condition factorization descends from constraint-solving framework for analyzing logic programs [24], which, in turn, evolved from previous works [9, 29]. This paper considerably extends those works to handle constraints involving bitmasks, inequalities, disequalities and ranges.

Packet filtering. Techniques such as BPF [20], DPF [10], Pathfinder [2] and SPAF [27] 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. As shown in our evaluation, our techniques result in significant gains in space usage, as compared to a left-to-right traversal.

BPF+ [5] uses global dataflow techniques to identify opportunities for eliminating redundant tests. pFSA [18] improves on BPF+ by handling redundancies in comparisons involving different constants. Our condition factorization technique is more general than those of pFSA, being able to reason about semantic redundancies in the presence of bit-masking operations. 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 [15] 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.

NIDS Systems. Bro [23, 36] is a popular NIDS that first assembles packet sequences into streams before applying signatures. In contrast, many firewalls, packet filtering applications as well as NIDS such as Snort operate primarily by matching network packets, as they apply some tests selectively to some packets (e.g., the first packet in a stream) but not against others. Our work is targeted at such systems.

Today's NIDS have come to rely increasingly on regular expression based rules for matching packet payload. Deterministic finite-state automata (DFA) provide the fastest matching performance for regular expressions, but their size can increase exponentially with the number of regular expressions. As a result, several new techniques have been developed to control space usage of NIDS regular expression matching [17, 43, 3, 34, 4, 22]. Our work is complementary to these approaches, and can be combined with them. In particular, our approach, by making fuller use of conditions on packet fields, can reduce the number of rules involved in regular expression matches, thus warding off space explosion in some cases. Tongaonkar [39] describes a more direct and complete technique to integrate string matching into our automata.

In recent years, there has been a growing interest in using signatures to capture vulnerability conditions rather than specific exploits. This has led to the development of vulnerability signatures [6] (*a.k.a.*, data patches [8]) that are based on accurate modeling of both the network protocol and the application context. Early works on vulnerability signatures relied on matching one signature at a time. Netshield [19], which is based on a systematic design of vulnerability based parsing, achieves high throughput by using techniques similar to Pathfinder for rule matching. It reorders rules to increase sharing of tests across rules.

However, it relies on left-to-right traversal of fields. The techniques presented in Section VII can be used to improve its matching time.

Our work differs from, and complements, the above works in the following ways:

- We present a unified framework in which different flavors of packet header matching can be supported.
- Almost all previous works in packet matching have been based on left-to-right examination of packet fields. Our results show that substantial gains in space and runtime can be obtained by using a customized order of examination. This traversal order is based on our notions of *discriminating tests* that provide *high utility*.
- Our condition factorization, first proposed in [40], is novel in its ability to proactively create opportunities for sharing tests, as opposed to post hoc elimination of tests that happen to be redundant.
- Our approach can classify on a large number of packet fields, and it does not require all filters to use the same set of fields. More importantly, our automaton construction algorithms can naturally handle disequalities and bit-masking operations.

III. Prioritized Packet Matching Automata

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. 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 is:

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

We exclude more complex conditions in filters, e.g.,

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

since they do not seem common 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 (Prioritized Matching): For a set $\mathcal{F}$ of filters, $F \in \mathcal{F}$ is said to ***match a packet*** p, denoted $M_{\mathcal{F}}(F, p)$, if:

- $F(p)$ is true, and
- $\forall F' \in \mathcal{F}$ such that $Pri(F') > Pri(F)$, $F'(p)$ is false.

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 $\mathcal{M}_{\mathcal{F}}(p)$.

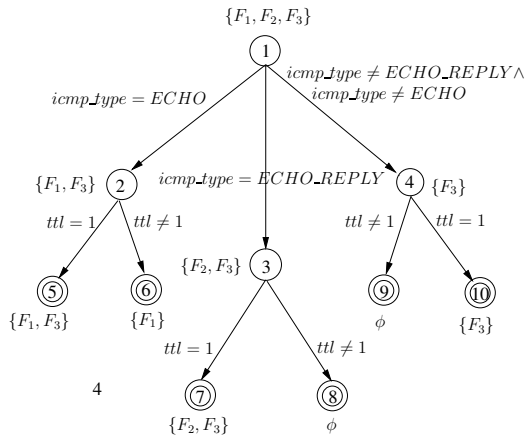

Fig. 1. A deterministic matching automaton.

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.

Figures 1 and 2 illustrate *packet-matching automata* (also known as classification automata) for the above filter set. Figure 1 shows a *deterministic matching automaton* (DMA), in which all of the transitions from any automaton state are mutually exclusive. A *nondeterministic matching automaton* (NMA) 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:

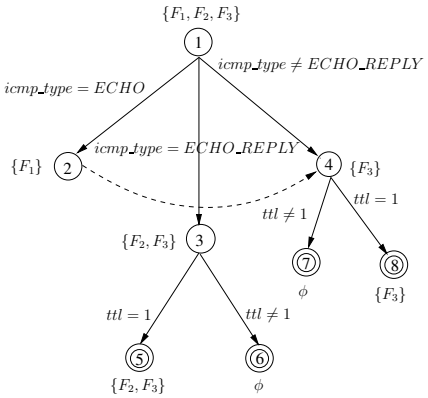

Fig. 2. A nondeterministic matching automaton.

  - 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 root of 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.
  - In a nondeterministic automaton, C is the conjunction of negations of *a subset* of the tests on the other transitions, e.g., the 3rd transition from the root of Figure 2.
- The transitions 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 taking expected $O(1)$ time, which of the transitions out of a state is satisfied by 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 nondeterminism 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.

A. Computational Issues

The two main computational issues in constructing a matching automata are its *size* and *matching time*. Typically, most filters contain a small number of tests, while the number of filters is large. As a result, path lengths in the automata are short as compared to its breadth.

The matching time of an automaton is closely related to path lengths. In particular, the worst-case matching time equals the longest path length in a DMA. The average matching time is dependent on the distribution of packets observed at runtime, but it is common to use the average path length of a DMA as an estimate of average matching cost. In an NMA, note that at each state, two branches may have to be followed at runtime, and this has to be taken into account in computing the worst-case as well as average matching times.

IV. 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 primitive tests that are common across different tests, thus enabling their shared computation.

The basis for condition factorization is the *residue* operation defined below. Suppose that we want to determine if there is a match for a filter C_1, and 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 check if C_1 holds.

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., 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 $\overline{x}$ 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 last row is applicable, yielding the result $(x = 5)$. (Although the two conditions are compatible with each other, the test $x\&\texttt{0x3} \neq 0$ does not contribute to proving $x = 5$.)
- Also from last row, $(x \in [1, 10])/(x \neq 5)$ is $(x \in [1, 10])$.

Note that the *minimal* residue in the last example would be $(x \in [1, 4]) \vee (x \in [6, 10])$, but it is easier to check $(x \in [1, 10])$. Figure 3 returns T_1 in such cases.

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 define residue for conjunctions and disjunctions:

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

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	

Fig. 3. Computation of Residue on Tests.

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

We do not handle the case of the second operand being a disjunction since it is never encountered by the 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 $(y > 7)$.

V. Matching Automata Construction

Our algorithm $Build$ for constructing a matching automata is shown in Figure 4. It 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* $\mathcal{M}_s$ (see Definition 2) 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 are possible. 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.

In the algorithm, 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

```
1.  procedure Build(s, M_s, C_s)
2.    if C_s is empty /* No more filters to match */
3.      then match[s] = M_s /* Annotate final state with match set */
4.    else
5.      (D, T) = select(C_s) /* T_i ∈ T is tested on ith transition */
          /*  d_i ∈ D indicates if this transition is deterministic */
6.      T_o = {⋀_{i|d_i=true} ¬T_i}
          /* Compute test corresponding to the "other"-transition */
7.      for each T_i ∈ (T ∪ {T_o}) do
8.        if (T_i ≠ T_o) then
9.          if d_i then C_i = C_s/T_i else C_i = C_s/T_i − C_s/T_o endif
            /* For a non-deterministic transition, do not duplicate */
            /* filters from the "other" branch */
10.       else C_i = C_s/T_i − ⋃_{j|d_j=false} C_s/T_j endif
11.       compute M_{s_i} and C_{s_i} from C_i and M_s
12.       if no state s_i corresponding to (C_{s_i}, M_{s_i}) is present then
13.         create a new state s_i
14.         Build(s_i, M_{s_i}, C_{s_i})
15.       endif
16.       create a transition from s to s_i on T_i
17.     end
18.   endif
```

Fig. 4. Algorithm for Constructing Matching Automaton

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 $\{F_3 : (ttl = 1)\}$.

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 5–17. 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 whenever T_i is a deterministic transition.

The actual transitions are created in the loop at line 7–17. At line 9, we compute the subset $\mathcal{C}_i$ of filters in $\mathcal{C}_s$ that are compatible with T_i. If this is a deterministic transition, then $\mathcal{C}_i$ consists of those filters in the candidate set (of the current state) 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 in the else clause of line 9. Symmetrically, those filters that are considered on non-deterministic transitions are eliminated from the "other" transition at line 10 of the algorithm. At line 11, $\mathcal{M}_{s_i}$ and $\mathcal{C}_{s_i}$ for the new state s_i are computed. (The procedure for computing match and candidate sets is described below.)

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 12 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 13, and $Build$

recursively invoked on this state. Finally, a transition to this state is created at line 16.

Computing Match and Candidate Sets Line 10 of the above algorithm requires a method for computing match and candidate sets for a newly-constructed automaton state s_i. This method starts with the match set $\mathcal{M}_s$ of the parent state s, and the set $\mathcal{C}_i$ computed in the preceding line, and uses the following steps:

- $\mathcal{M}' = \{M \in \mathcal{C}_i | (M = true)\}$, i.e., $\mathcal{M}'$ consists of the subset of filters in $\mathcal{C}_i$ whose tests have all been checked on the automaton path to s_i.
- $\mathcal{M}'' = \{M \in \mathcal{M}' | \neg\exists C \in \mathcal{C}_i \; Pri(C) > Pri(M)\}$, i.e., we delete those filters from $\mathcal{M}'$ for which a future match with higher priority filters can't be ruled out at this point[1].
- $\mathcal{M}_{s_i}$ is obtained by considering filters with equal priorities in $\mathcal{M}''$, and deleting all but one of them.

Now, $\mathcal{C}_{s_i}$ can be computed using the following equation:

$$\mathcal{C}_{s_i} = \{C \in \mathcal{C}_i \,|\, \neg\exists M \in \mathcal{M}_{s_i} \text{ with } Pri(M) \geq Pri(C)\}$$

A procedural interpretation of this equation will amount to deleting filters in $\mathcal{C}_i$ that are superseded 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[2]. 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; however, we show only the filter labels in these sets, but omit residues.

VI. 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.

Our experimental evaluation considers the number of automaton states as a measure of its size. However, for simplifying mathematical analysis, our discussion in this section, similar to previous works [32], will measure the automaton size in terms of it breadth. Note that the automaton depth is linearly dependent on filter sizes, but the breadth can be exponential. For this reason, breadth is the dominating factor in automaton size, thus justifying our choice.

A. Discriminating Tests

Since *select* determines which packet field is to be examined at each automaton state, it effectively defines an order of examination of packet fields[3]. The simplest approach is to examine the protocol fields in the order of their occurrence in a network packet, as done in most previous works [2, 10]. We call this *left-to-right traversal.* An automaton using this traversal is called an *L-R automaton*. A better strategy, called *adaptive traversal,* was first proposed in the context of term-matching [32], and was then generalized to deal with binary data [14]. 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, $\mathcal{T}$ includes all tests on x that occur 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[4], and hence can form the transitions from s. Moreover, it has been shown [32] 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 results 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 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 $\mathcal{C} = \{x = 5, x = 6, x > 7\}$. This means that 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 automaton that is based entirely on discriminating tests will have at most $O(n)$ breadth.

Note that whether a set of tests is discriminating depends on the filter set as well. For instance, the same set $\mathcal{T}$ discussed above is *not* a discriminating set for the set of filters $\{x = 6, x > 4\}$. This is because the filter $x > 4$ is compatible with more than one of the tests in $\mathcal{T}$.

B. 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

[1] Recall that a match with a filter M cannot be completed until we eliminate the possibility of a match for any filter with a higher priority than M.

[2] Or more accurately, we are keeping track of those tests that aren't already known to be satisfied.

[3] Section VIII-A describes how our algorithm ensures protocol-specified constraints on the order of examination of packet fields.

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

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 [32]. Although some of the previously developed techniques can avoid such explosion, this has been accomplished at the cost of introducing significant backtracking at runtime [20, 10, 2, 5], 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 [25, 14]. With large filter sets that are often found in enterprise firewalls and NIDS, $O(n)$ transition time can become unacceptably large.

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, use of a larger bound, e.g., n^2 instead of $n \log n$, reduces the number of instances where nondeterministic transitions are needed, thus providing better runtime.

Our technique is based on the observation that the breadth of subautomaton rooted at s can be captured using the recurrence:

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

where $B(1) = 1$, and $|\mathcal{C}_s|$ denotes the size of candidate set associated with s. 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 rule sets that cause an exponential blow up — memory will be exhausted in that case and hence the rule set 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 just a worst-case possibility: with the rule sets studied in our experiments, the quadratic bound was not exceeded, and hence nondeterminism was not introduced, except in the special case described below where it does not degrade performance.

C. Benign Nondeterminism

For our final space-reduction technique, we define the concept of benign nondeterminism, which enables us to benefit from the space-savings enabled by nondeterminism *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

- 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. The above definition indicates that the tests appearing in the two automata must be totally disjoint, and hence no runtime reduction is 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 nondeterminism, the size is limited to $k_1 + k_2$. If there are m independent sets, then this technique 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 large increases in space usage if they are not recognized and exploited using our benign nondeterminism 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 multiple subsets are left at this point, they must be independent.

To deal with benign nondeterminism, the interface between $select$ and $Build$ needs to be extended. Instead of returning a test set, $select$ will return a partitioning of the candidate set $\mathcal{C}_s$ of the current automaton state. Let this partitioning consist of sets $\mathcal{C}_1, \ldots, \mathcal{C}_k$, where every pair $\mathcal{C}_i$ and $\mathcal{C}_j$ is

mutually independent. At this point, $Build$ will create a k-way nondeterministic branch, and for $1 \leq i \leq k$, create a child state s_i and invoke $Build(s_i, \mathcal{M}_s, \mathcal{C}_i)$.

VII. Improving Matching Time

To reason about matching time, we need to define a function that assigns computational costs to each test. We discuss two alternatives here. The first, and the simpler, of the two alternatives is to assign the same 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, a preferable alternative is to assign 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. For the discussion below, both alternatives will work well. However, in our evaluation, we have opted for the first alternative due to its simplicity, and because it is a better match for real-world cost of these tests.

The actual matching time is a function of both the automaton and the packet that is matched against it. As a result, obvious cost estimates for an automaton, such as average packet matching time, have to be defined with respect to expected distribution of input packets. In particular, a weighted average path length can be defined as the matching cost of an automaton, provided we have the relative frequencies with which each path in the automaton is taken. Unfortunately, such input distributions are often unavailable, e.g., when a new rule set is installed, or an existing rule set undergoes significant modifications. Moreover, distributions can vary widely across time and/or network installations. For these reasons, we focus on cost measures and optimization techniques that don't require input distributions.

Another difficulty of measures such as average path length is that they can be computed only after the entire automaton is constructed. This makes them unsuitable for use during automaton construction: we need cost estimates before deciding on the transitions from an automaton state, not after all descendant states have already been constructed! We therefore define a notion of *utility* that estimates the local effect of a set of transitions on the overall matching times from an automaton state. Utility is a negative number, with the highest possible value of zero, which indicates the absence of any potentially redundant computation.

Definition 6: The utility $U(T, F)$ of a test T for a filter F is

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

This definition extends naturally to a filter set:

$$U(T, \mathcal{F}) = \sum_{F \in \mathcal{F}} U(T, F)$$

Consider the following examples to illustrate this definition:

- $U(x > 1, x = 1) = 0$, since the test $x > 1$ ensures that $x = 1$ cannot hold.
- $U(x > 1, x > 1) = 0$, since the test $x > 1$ is 100% useful for verifying a match with the filter $x > 1$.
- $U(x > 1, x > 2) = -1$, since the test $x > 1$ does not reduce the cost of verifying $x > 2$.
- $U(x > 1, \{x = 1, x > 1, x > 2\}) = -1$, by adding up the costs of U for each filter.
- $U(x = 1, \{x = 1, x > 1, x > 2\}) = 0$
- $U(x = 2, \{x = 1, x > 1, x > 2\}) = 0$
- $U(x > 2, \{x = 1, x > 1, x > 2\}) = 0$

We now extend the notion of utility to an automaton state s:

Definition 7 (Utility of automaton state): Let $\mathcal{T} = T_1, \ldots, T_k$ be the set of tests performed on transitions out of an automaton state s. If the transitions are all deterministic, then:

$$U_s^d = \frac{1}{k} \sum_{i=1}^{k} U(T_i, \mathcal{C}_s)$$

If s has $n \geq 0$ nondeterministic branches:

$$U_s = U_s^d - \frac{n}{k} * cost(\mathcal{C}_{s_k})$$

At most one of the deterministic transitions can be applicable at any time, and hence U_s^d is simply the average of the utilities of tests on each of the transitions from s. For a nondeterministic state, in addition to one of the deterministic transitions, it may be necessary to backtrack and traverse the "other" transition. The entire time spent within the "other"-transition, given by $cost(\mathcal{C}_{s_k})$, would then be additional work over that given by the term U_s^d. We weight this extra work by the factor n/k that approximates the likelihood of taking one of the n nondeterministic transitions.

Building on the example used to illustrate the utility of a single test, consider a state s with $\mathcal{C}_s = \{x = 1, x > 1, x > 2\}$ and 3 outgoing transitions on the tests $x = 1$, $x = 2$ and $x > 2$. Since these transitions are deterministic, the utility is

$$\frac{U(x = 1, \mathcal{C}_s) + U(x = 2, \mathcal{C}_s) + U(x > 2, \mathcal{C}_s)}{3} = 0$$

If s uses transitions $x = 1$, $x > 1$ and $x > 2$, then U_s is:

$$\frac{U(x = 1, \mathcal{C}_s) + U(x > 1, \mathcal{C}_s) + U(x > 2, \mathcal{C}_s)}{3} - \frac{cost(x > 2)}{3} = -\frac{2}{3}$$

A. Match verification cost: A metric for matching time

We now present a metric for the matching time of an automaton that avoids assumptions about input distributions. Moreover, it provides a way to compare the efficiency of the automata without being closely tied to exact implementations. 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 rely on interpretation. Thus the raw numbers won't accurately capture the benefits of the techniques developed in this paper, 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 nondeterministic matching algorithm that can "guess" the subset of rules that match a given packet p, and then proceeds to verify the soundness of this guess. One can reasonably expect that a deterministic matching algorithm would need to perform more

computation than such a nondeterministic 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 8 (Minimum Match Verification Cost):

- Verifying a match of a filter F is $\Omega(|F|)$.
- Verifying a successful match of all filters in a set $\mathcal{M}$ is $\Omega(k)$, where k is the number of distinct fields (or field-bitmask 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 the lower bound.

VIII. Putting It All Together

We now describe how the techniques described so far can be combined to build an end-to-end system. We begin with our language for specifying filters. We then discuss how to combine our space and matching-time reduction techniques, and generate safe native code. Finally, we describe integration with payload matching, otherwise known as deep-packet inspection.

A. Filter Specification Language

As we detail below, filter specifications consist of a set of packet type declarations, followed by packet processing rules of the form $filter \rightarrow response$.

Packet Structure Description. A simple way to access a packet field is by specifying a numeric offset (from the packet beginning) and width. However, this approach is error-prone, as the same offset can denote different packet fields depending on (a) header lengths for lower-layer protocols, (b) presence of optional fields, and (c) values of preceding fields such as those representing protocol types such as TCP or ICMP. Specification errors can lead to memory safety errors, especially if the filters are translated to native code. It is critical to avoid memory errors in these systems since they are exposed to network traffic from every corner of the Internet. Previous works have proposed safe type systems for network protocols [7, 28] to avoid memory errors in packet field accesses. Our language is based on the latter reference as it provides better decoupling between lower and higher layer protocols. We note that later works such as SPAF [27] and pFSA [18] also take a similar approach of using a specialized type system to ensure the memory safety of the resulting packet filtering code. We begin with an example that illustrates a type declaration for an Ethernet header:

```
#define ETHER_LEN 6
struct ether_hdr {
  byte  e_dst[ETHER_LEN];   /* Ethernet dest. address */
  byte  e_src[ETHER_LEN];   /* Ethernet source address */
  short e_type;             /* Protocol of carried data */
};
```

Layering of protocols is captured through inheritance. For instance, IP header can be defined as a subtype of `ether_hdr`. However, in order for the Ethernet protocol code to forward the higher layer payload (i.e., IP) to the appropriate handler, some fields in the lower layer header need to identify the higher layer protocol. This requirement is captured in our language using inheritance constraints, specified using the keyword `with`:

```
#define ETHER_IP 0x0800
struct ip_hdr : ether_hdr with e_type == ETHER_IP {
  bit          version[4]; /* IP Version */
  bit          ihl[4];     /* Header Length */
  byte         tos;        /* Type Of Service */
  short        tot_len;    /* Total Length */
  ...
  short        check_sum;  /* Header Checksum */
  unsigned int s_addr;     /* Source IP Address */
  unsigned int d_addr;     /* Destination IP Address */
};
```

Our language is expressive enough to capture the fact that a higher layer protocol may be carried over multiple lower layer protocols. If P_1 can be layered over P_2 or P_3, then it will be captured in our language using the declaration:

$$\texttt{struct}\ P_1 : (P_2\ \texttt{with}\ C_2)\ \texttt{OR}\ (P_3\ \texttt{with}\ C_3)\ \ \{\ldots\};$$

where C_2 and C_3 denote respectively the conditions on P_2 and P_3 that signify that the higher layer protocol is P_1.

Rules. Rules take the form $filter \rightarrow response$, where $response$ specifies the action to be taken when a packet that matches the filter $filter$. The syntax and semantics of filters has already been described in the preceding sections of this paper. Actions correspond to function calls into a runtime support library, and are not described further. Note that if multiple filters match at the same time, actions associated with each filter are launched. However, as noted previously, a lower priority rule is considered to match a packet *only after* matches with higher priority rules have been eliminated. Rules may have optional labels. An example rule is:

```
F1: (p.s_addr & 255.255.255.0 == 192. 168.2.0)
     && (p.d_addr == 192. 168.1.103)
     && (p.tcp_dport == 80) -> alert(...);
```

While the type of packet `p` needs to be declared, it is only necessary to specify its base type, which, in our example, is `ether_hdr`. Our compiler can infer higher layer types from the use of `p`, as we describe in subsequent sections.

Priorities are specified using rule labels. By default, rule priorities are incomparable, a setting suitable for NIDS. Other options need to be explicitly specified.

B. Compilation into Native Code

Our compiler translates a rule set into a C function that takes a network packet as input, performs the requisite matching steps, and returns the filters that match. This C-function is then compiled using the C-compiler into native code, and linked with a runtime support library that calls this function for each network packet, and provides the functions used in the action component of the rules.

Packet-field access and precondition insertion. Our compiler translates packet-field names into appropriate offsets. It also accounts for issues such as field width, endianness, etc. For memory/type safety, the compiler takes two steps. Before any field access, it inserts a *precondition* that the packet length is at least as large as the maximum byte offset of the field. The compiler then adds the applicable constraints from packet type definitions as additional preconditions. For instance, consider the example rule described in the preceding section. From the access `p.tcp_dport`, the compiler infers that `p`'s type is `ip_hdr`, and automatically inserts the constraint associated with `ip_hdr`, namely, `p.e_type == ETHER_IP`. This means that the rule won't match non-IP packets, e.g., ARP packets.

Note that $Build$ may examine tests within a filter in any order, which can defeat the purpose of introducing preconditions. To address this problem, the compiler records the preconditions for each test in a filter. Our implementation of $select$ considers only those tests whose preconditions have all been tested (and are determined to hold).

Blindly attaching preconditions to each test can result in duplications. By the nature of condition factorization, these redundancies will be recognized and eliminated, so we do not do any thing special to avoid them.

Automaton Construction. Our implementation uses our $Build$ algorithm to construct a matching automaton for the given filter set. Residues are computed as specified in Table 3, and match and candidate sets computed as described on page 7. Our $select$ implementation considers only those tests whose preconditions have all been satisfied. While selecting from such tests, our implementation prioritizes size reduction over matching time reduction. This is because many of the size reduction techniques also improve matching time:

- *Discriminating tests:* For every filter in the candidate set, a discriminating test will either rule out a match for that filter, or will perform a test that is compatible with that filter. For this reason, the utility of states performing discriminating tests will typically have the (maximum possible) value of zero. (This can be formally proved for the common case of filters containing equality conditions.)
- *Benign nondeterminism:* Since the efforts in matching each independent subset is completely disjoint from those of matching others, none of the tests performed within the subautomata for one of these subsets will be repeated in another. Thus, as in the previous case, we have not performed any additional work that is over and above the minimum that may be required to match these filter sets.

As a result, size reduction techniques will usually have the effect of minimizing utility as well, and hence improving the matching time. So, our implementation of $select$ first identifies a set of choices to minimize automata size, and among these choices, picks the one that provides the lowest utility. Specifically, it proceeds as follows:

- $select$ first attempts to find a discriminating test set (Section VI-A). If several of them exist, our technique selects a set that maximizes utility.
- if no discriminating test sets exist, it examines opportunities for benign nondeterminism (Section VI-C).
- if neither of the above steps succeed, it returns a set of tests that achieves the polynomial size target specified, as described in Section VI-B. Typically, multiple sets satisfy this criteria, and we prefer those sets that minimize nondeterminism. If there are still multiple options, then we pick the one that with the maximizes utility.

An exhaustive search through all possible tests in a candidate set can be quite expensive. To speed up the process, our implementation utilizes a "fast selection" phase. This phase examines the subset of 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.

Code Generation. Once the automaton is constructed, our compiler generates C-code corresponding to the automaton. The code generation is straight-forward and not described in detail here, except to note that the generated code explicitly uses an if-then-else, a binary search, or a hash-based branching to implement transitions efficiently. This choice is dependent on the nature of the test (e.g., inequality tests cannot use hashing), as well as the number of transitions.

The C-code is compiled using a C-compiler into a shared library. If a rule set needs to be updated, then we first compile it into another shared library. At this point, our system can unload the original shared library and load the updated version. We note that this approach is not much more disruptive than that of Snort where the rules need to be re-read and recompiled.

C. Payload Matching

In addition to examining packet header fields, modern NIDS examine packet payloads as well. In this section, we describe how such *deep packet inspection* (DPI) can be integrated with the techniques presented in this paper. For concreteness, our discussion is set in the context of Snort.

Snort rules consist of tests on packet header fields and all of the *strings* that must be found within the payload. Hence, rule matching in Snort involves matching the packet header fields, together with string matching over the packet payload. To reduce false positives, Snort allows rule writers to specify additional constraints on string matching, e.g.,

- match a string s_1 at position k from the payload beginning
- match two strings s_1 and s_2 such that:
  - they are separated by a specific distance d, or,
  - they occur within a certain distance l.

For efficiency, string matching operations across different rules should be shared, so that all matches can be identified in a single scan of payload data. Unfortunately, multi-pattern matching algorithms such as Aho-Corasick [1] cannot be directly used because of the need to support the above constraints. The alternative of matching rules one-by-one is unacceptably slow because Snort operates on thousands of rules. Therefore, Snort uses the following two-stage process to filter out most of the inapplicable rules. In the first stage, it uses a small set of packet fields that appear in almost all rules, e.g., source and destination ports, to divide the rules

into subsets $\mathcal{R}_1, \ldots, \mathcal{R}_n$, each of which agree on these fields. For the second stage, it constructs an Aho-Corasick automaton for each $\mathcal{R}_i$. This automaton includes the longest strings from each rule $R \in \mathcal{R}_i$, and is used to quickly select a subset of $\mathcal{R}_i$ whose longest string matches the payload. Finally, each rule in this subset is matched one-by-one against the payload. We will refer to this final step as *slow search*, and the preceding steps as *fast filtering*.

Since the one-by-one match phase can be time-consuming, the primary goal of the fast filtering phase is to reduce the number of rules that are involved in the slow search. The algorithms presented in this paper are effective in this regard: our technique uses as many packet fields as possible in the first stage to reduce the sizes of the subsets $\mathcal{R}_i$. We then replicate Snort's use of Aho-Corasick automata for each of these sets. Section IX-C shows the improvements that we achieve in the end-to-end performance of Snort using this approach.

In the presence of non-determinism, a modification to this procedure is needed in order to avoid repetition of string-matching tests after backtracking. Specifically, we build the Aho-Corasick at the first non-deterministic node encountered on a root-to-leaf path in the automaton, and perform an intersection of the set of rules returned by the Aho-Corasick automaton with those identified by our algorithm.

We can extend this approach to take advantage of *all the strings* in a filter, as opposed to using just the longest one. The idea is to replace each distinct string match in a filter with a corresponding (boolean-valued) variable, and build a matching automaton for this new set of filters. Next, an Aho-Corasick automaton is constructed for all the strings. Now, given a packet, we initialize all boolean variables to false, and then scan the packet through the Aho-Corasick automaton. Whenever a match for a string is identified by this automaton, the corresponding boolean variable is set to true. We then process the packet, together with these boolean variables, using our packet matching automata. For details, see [39].

IX. Evaluation

We evaluated the effectiveness of our techniques in the context of NIDS (Section IX-B) and firewalls (Section IX-D). Whereas we generate native code from our matching automata, most previous techniques such as Snort [26] and Snort-NG [16] represent the automaton as a data structure that is interpreted at runtime. This factor alone makes our packet matcher many times faster than the corresponding component of most previous systems. Thus, comparison of raw performance does not bring out the benefits many of our techniques. Hence, we focus our evaluation on metrics that are independent of low-level implementation such as the number of states in the automata, path lengths in the automata, etc.

We begin with a comparison of the automata constructed by our technique and the BPF+ packet filtering technique. We then compare performance with Snort [26] and Snort-NG [16], two NIDS systems that operate on network packets. Finally, we present the performance of our algorithms for firewall rules.

A. Packet Filtering

Our goal in this section is to visually illustrate the differences in the automata built by our technique and the one

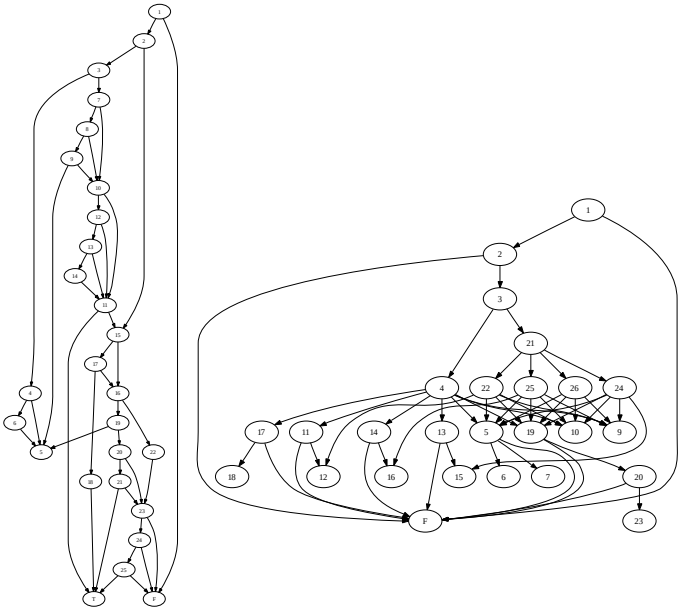

Fig. 5. Backdoor detection: BPF+ (left) and Condition factorization (right)

constructed by the BPF+ [5] technique. Although both works avoid redundant tests, our approach for achieving this goal differs from theirs. In particular, condition factorization proactively creates opportunities for sharing computation, whereas BPF+ takes a more passive approach, eliminating later tests whose satisfaction can be proved using data-flow analysis.

A visual illustration is necessarily limited by space constraints, so we consider just a single example here, namely, the signatures used for detecting backdoors [45] for services such as ssh and ftp. We instrumented a version of `pcap` library that uses BPF+ to compile the filter expressions to generate the corresponding automata. Figure 5 shows that even for a small number of simple filters. The height of automaton generated by BPF+ can be significantly larger than that of our technique. The longest path in the BPF+ automaton is *thrice as long* as that of condition factorization. Note that path lengths correspond closely to matching times.

It is important to note that shorter path lengths of condition factorization result from our techniques for $select$, whereas the long path lengths of BPF+ result from its use of left-to-right traversal order. For this reason, similar differences in path lengths are to be expected for many other works that also examine packet fields in the order of their occurrence.

B. Packet-field matching in NIDS

For our experiments we use Snort [26] which is a popular open-source NIDS. Snort signatures consist of two main components: tests involving packet fields, and content-matching operations on the payload. According to [11], packet-field matching and content-matching are the most expensive parts of Snort, accounting for 21% and 31% of the execution time. This section presents a comparative evaluation of packet-field matching, while the next section evaluates end-to-end performance that includes content-matching time as well.

Although earlier versions of Snort relied largely on sequential matching (i.e., matching a packet against one signature at a time), subsequent versions (specifically, version 2.0 and

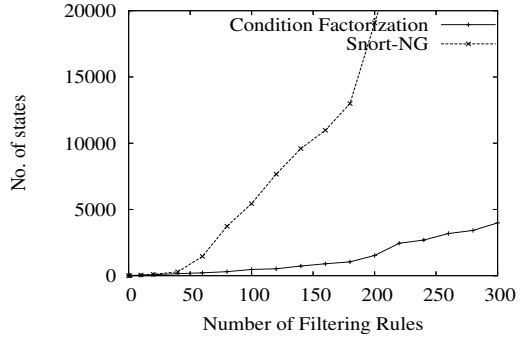

Fig. 6. Automaton size for NIDS rule set

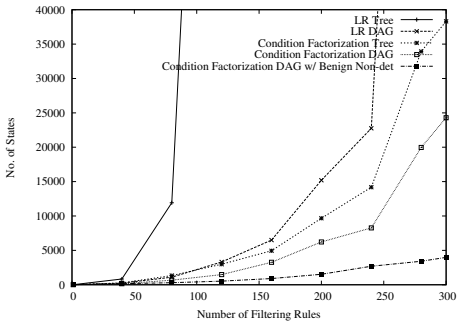

Fig. 7. Optimization effect on automaton size

later) match the signatures in parallel. An ad-hoc approach for parallelizing is utilized, where a small set of hand-picked packet fields, such as destination ports, are tested first, but there is no systematic technique for matching other packet fields in parallel. In contrast, Kruegel and Toth developed the `Snort-NG` [16] system, which demonstrated the performance gains achievable by parallelizing 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-matching algorithm for Snort-type rules. Hence we compare our performance results with them. To simplify this comparison, we used all the *alert* rules from the rules that come with `Snort-NG-1.8.7` [35], which total 1635 rules. (This rule set is available at http://www.seclab.cs.sunysb.edu/seclab/cfac/snortng/exp.html.) Since our focus is on matching packet fields, we combined the rules that differ only in payload contents, leading to 305 unique rules. Fifteen distinct fields were tested in these rules. Of these:

- 5 fields involve only equality tests
- 4 fields involve bitmasking with equality and disequality
- 3 fields involve only equality and inequality, and
- 3 fields involve equality, inequality, and disequality.

Automata size comparison with Snort-NG. Figure 6 shows the effect of increasing the number of rules on the number of automaton states. We note that Snort-NG decision trees contain some states that perform tests and others that are used for purposes such as identifying the type of field being tested. For our experiments we only counted 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. This happens in spite of the fact that Snort-NG divides the rules into several subsets, and builds an independent decision tree for each group. This is done because Snort NG experiences an unacceptable space explosion if it attempts to build a single decision tree for the entire rule set. This means that packets have to be processed sequentially by each of these decision trees, leading to repeated computation. In contrast, condition factorization is able to construct *a single matching automaton that avoids repetition of any tests.* Despite this, it achieves more than an order of magnitude size reduction.

Effect of Optimizations on Automaton Size. Figure 7 shows the effects of various optimizations on the automaton size.

- *Order of testing fields.* As compared to left-to-right (LR) order for examining packet fields, our techniques for realizing $select$ produce tree automata that are much smaller: for 120 rules, the LR automaton has 150,000 states, whereas our automaton has less than 3000 states.
- *DAG Vs tree automata.* For our technique, DAG automata were smaller than tree automata by about a quarter to one-third. Much larger space reductions were achieved for LR automata. Despite these reductions, space usage for LR automata remains much higher than with other techniques.
- *Benign nondeterminism.* By exploiting benign non-determinism, we achieved 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.

Some combinations of our techniques worked much better than others. For instance, benign nondeterminism leads to large improvements in size when combined with discriminating tests. It was much less effective when used with LR order. In contrast, LR automata provide far more opportunities for DAG optimization, as compared to alternative techniques.

Matching time comparison with Snort and Snort-NG. For measuring runtime performance, we used two sets of data. The first one 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 trace contains about 21 million packets collected over a few days. Figure 8 plots the matching time against the number of rules for Snort, `Snort-NG` and our technique.

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 [21], consisting of 17 million packets. Figure 9 shows the matching times for this data. 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 rule set, 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 8 and 9, it can be seen the matching time remains essentially constant with our technique, even as the

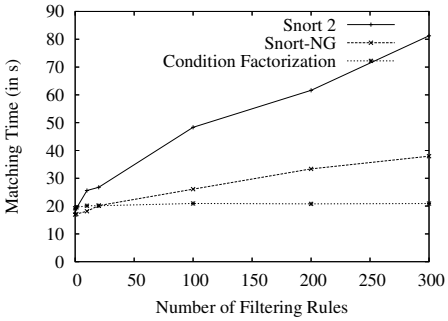

Fig. 8. Matching Time (Lab Traffic)

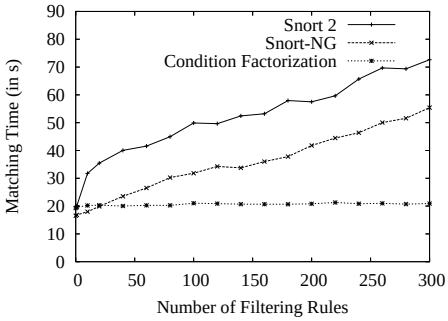

Fig. 9. Matching Time (Lincoln Labs Data)

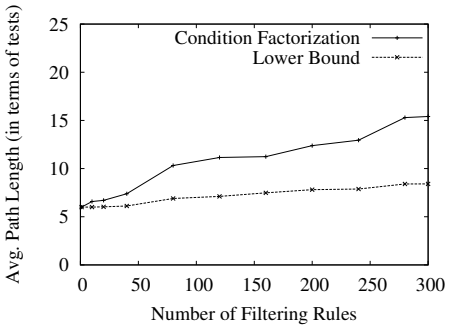

Fig. 10. Path Length for Snort Rules

number of rules are increased from 0 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 is roughly the same, as it corresponds to the time spent by Snort to read network packets and perform other book-keeping operations related to matching.

The flatness of the graph for condition factorization can be attributed to two factors.

- Our automata are compiled into native code, whereas Snort-NG and Snort use an interpreted approach. Native code provides much faster performance.
- Our techniques yield automata that are mostly deterministic, and don't repeat tests. As a result, the number of tests performed won't increase significantly as the number of rules is increased. In contrast, Snort and Snort-NG end up repeating tests, and these repetitions increase as the rule set size increases.

In an effort to separate these two factors, we undertook additional experiments on our system. In particular, Figure 10 plots the average path lengths of our automata as the number of rules is increased. As the number of rules increases from 10 to 300 — a factor of 30 — the average path length increases by less than a factor of 3. Since path lengths capture the number of tests performed before announcing a match, they correlate well with matching times, and hence provide a measure of automaton "goodness" that is decoupled from lower level implementation choices.

We also compared the rate of increase in path lengths of our automata with the *match verification cost (MVC)* metric defined in Section VII-A. Recall that MVC($\mathcal{F}$) defines a lower bound on the time taken by any packet matching technique to announce a match for all the filters in $\mathcal{F}$. We plotted the average of MVC($\mathcal{F}_l$) for each leaf l in our automata in Figure 10, as the number of rules is increased. Given that MVC is typically an unachievable lower bound, the fact that our technique comes within a factor of two is remarkable.

C. End-to-end Performance of NIDS

In this experiment, we compare the aggregate performance benefits of our techniques. In particular, we compare the total time taken by Snort (when using its original packet and string matching components) with a Snort-version that we modified to use condition factorization. The times measured include the total processing time, including the times for reading packets, matching them against rules, and raising alerts. Our modified Snort version uses the payload matching techniques described in Section VIII-C. The performance measurements use the laboratory packet trace described previously.

Figure 11 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) at about a 30% performance advantage for condition factorization at few hundred signatures. (Recall that our default signature set had about 1600 rules, but the number was reduced to about 300 if content-matching conditions were deleted.)

According to Fisk et al [11], only about 50% of Snort's execution time is attributable to packet-field and content matching, the two tasks whose performance is improved by our techniques. As a result, even a 3-fold reduction in matching times can be expected to yield just one-third reduction in end-to-end runtime. This suggests that the 30% reduction we achieve in end-to-end performance must be underpinned by a much larger improvement in matching times. To get a better handle on this, we measured the number of times the slow-search phase was invoked in original and modified Snort. It was invoked about 120M times originally, but was reduced by a factor of three to 40M by condition factorization — an observation that seems consistent with our expectation on the degree of improvement needed in the matching phase.

D. Firewall Rule Matching

The firewall rule set 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 to the outside world. The rules are in the form of iptable rules for a Linux machine. There were a total of 140 filtering rules.

Figure 12 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 13 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

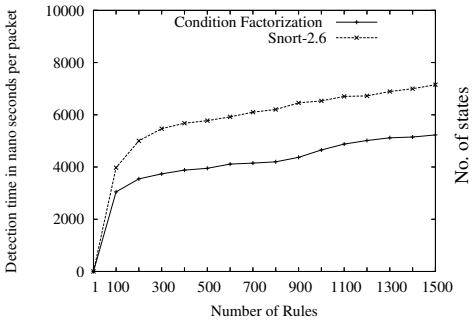

Fig. 11. End-to-end Matching Time

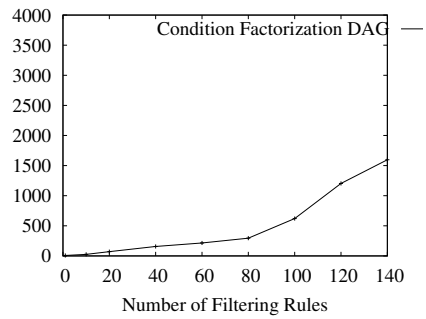

Fig. 12. Automaton Size for Firewall Rules

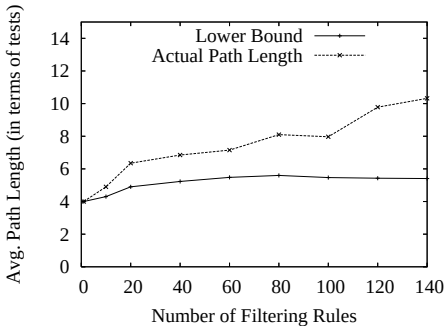

Fig. 13. Matching Time for Firewall Rules

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 rules used with NIDS.

X. 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 repetitions of tests. Our experiments show that the technique is very effective in reducing automata size as well as matching time.

References

[1] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Communications of the ACM*, 1975.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A Pattern-Based Packet Classifier. In *ACM SOSP*, 1994.

[3] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *CoNEXT*, 2007.

[4] M. Becchi and P. Crowley. A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation. *ACM Trans. Archit. Code Optim.*, 2013.

[5] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In *SIGCOMM*, 1999.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-based Signatures. In *IEEE Security And Privacy Symposium*, 2006.

[7] S. Chandra and P. McCann. Packet types. In *Workshop on Compiler Support for Systems Software*, 1999.

[8] W. Cui, M. Peinado, and H. Wang. Shieldgen: Automatic data patch generation for unknown vulnerabilities. In *SIGCOMM*, 2007.

[9] S. Dawson, C. Ramakrishnan, I. Ramakrishnan, and R. Sekar. Extracting determinacy in logic programs. *ICLP*, 1993.

[10] D. Engler and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing Using Dynamic Code Generation. In *SIGCOMM*, 1996.

[11] M. Fisk and G. Varghese. Fast Content-Based Packet Handling for Intrusion Detection. *Tech. Rep. CS2001-0670, UC, San Diego*, 2001.

[12] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *SIGCOMM*, 1999.

[13] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*, 1999.

[14] P. Gustafsson and K. Sagonas. Efficient manipulation of binary data using pattern matching. *Journal of Functional Programming*, 2006.

[15] H. Hamed, A. El-Atawy, and E. Al-Shaer. On Dynamic Optimization of Packet Matching in High-Speed Firewalls. *IEEE JSAC*, 2006.

[16] C. Kruegel and T. Toth. Using Decision Trees to Improve Signature-Based Intrusion Detection. In *RAID*, 2003.

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions for Deep Packet Inspection. In *SIGCOMM*, 2006.

[18] M. Leogrande, F. Risso, and L. Ciminiera. Modeling Complex Packet Filters With Finite State Automata. *IEEE Trans. on Networking*, 2015.

[19] Z. Li, G. Xia, Y. T. Hongyu Gao, Y. Chen, B. Liu, J. Jiang, and Y. Lv. NetShield: Matching with a Large Vulnerability Signature Ruleset for High Performance Network Defense. In *SIGCOMM*, 2010.

[20] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, 1993.

[21] MIT Lincoln Labs. DARPA Intrusion Detection Evaluation, 1999.

[22] J. Patel, A. X. Liu, and E. Torng. Bypassing space explosion in high-speed regular expression matching. In *IEEE Trans. on Networking*, 2014.

[23] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *USENIX Security*, 1998.

[24] C. Ramakrishnan, I. Ramakrishnan, and R. Sekar. A symbolic constraint solving framework for analysis of logic programs. In *ACM PEPM*, 1995.

[25] R. Ramesh, I. Ramakrishnan, and D. Warren. Automata-Driven Indexing of Prolog Clauses. In *ACM POPL*, 1990.

[26] M. Roesch. Snort — lightweight intrusion detection for networks. In *Large Installation Systems Administration*, 1999.

[27] P. Rolando, R. Sisto, and F. Risso. SPAF: Stateless FSA-Based Packet Filters. *IEEE Trans. on Networking*, 2011.

[28] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A High-Performance Network Intrusion Detection System. In *ACM CCS*, 1999.

[29] R. Sekar and I. Ramakrishnan. Fast strictness analysis based on demand propagation. *ACM TOPLAS*, 1995.

[30] R. Sekar, I. Ramakrishnan, and A. Voronkov. *Term indexing, Handbook of automated reasoning*. Elsevier Science Publishers BV, Amsterdam, The Netherlands, 2001.

[31] R. Sekar, R. Ramesh, and I. Ramakrishnan. Adaptive Pattern Matching. In *ICALP*, 1992.

[32] R. Sekar, R. Ramesh, and I. Ramakrishnan. Adaptive Pattern Matching. *SIAM Journal on Computing*, 1995.

[33] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *SIGCOMM*, 2003.

[34] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *IEEE Security and Privacy*, 2008.

[35] Snort-NG-1.8.7. Available at http://www.iseclab.org/snort-ng/.

[36] R. Sommer and V. Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *ACM CCS*, 2003.

[37] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *SIGCOMM*, 1998.

[38] D. Taylor. Survey and Taxonomy of Packet Classification Techniques. In *Journal ACM Computing Surveys*, 2005.

[39] A. Tongaonkar. *Efficient Techniques for Fast Packet Classification*. PhD thesis, Stony Brook University, August 2009.

[40] A. Tongaonkar, R. Sekar, and S. Vasudevan. Fast packet-classification using condition factorization. In *Applied Cryptography and Network Security (ACNS)*, 2009.

[41] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *SIGCOMM*, 2010.

[42] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-Layer Packet Classification with Graphics Processing Units. In *CoNEXT*, 2014.

[43] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *ACM/IEEE ANCS*, 2006.

[44] F. Yu and R. Katz. Efficient Multi-Match Packet Classification with TCAM. In *Hot Interconnects*, 2004.

[45] Y. Zhang and V. Paxson. Detecting Backdoors. In *USENIX Security Symposium*, 2000.