Trees in the List: Accelerating List-based Packet Classification Through Controlled Rule Set Expansion

Sven Hager Stefan Selent Björn Scheuermann

Computer Engineering Group
Humboldt University of Berlin, Germany
{hagersve,scheuermann}@informatik.hu-berlin.de, stefan.selent@hu-berlin.de

ABSTRACT

Network packet classification is performed by a wide variety of network devices, like routers or firewalls. Accordingly, researchers have put great efforts in the development of fast packet classification algorithms. However, despite the fact that such approaches have been around for over a decade, most classification systems used in practice still rely on the slow linear search approach. In this work, we propose a methodology that enables linear search-based systems with jump semantics to take advantage of the superior matching performance of decision tree algorithms, without the need to touch the underlying system implementation. By performing source-to-source transformations on packet classification rule sets, we encode decision trees inside of the modified rule sets in order to guide and tweak the originally linear matching process. We implement this in a proof-of-concept tool which transforms Linux iptables firewall rule sets. Our evaluation demonstrates that throughput performance boosts of one order of magnitude and more are possible—without changing the semantics of the rule set, and without any modifications to the matching engine.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*network management*

General Terms

Algorithms, Performance, Security

Keywords

Packet classification; Rule set modification

1. INTRODUCTION

Network packet classification is a fundamental task that is performed by a multitude of networking devices, including, for instance, routers and firewalls. By matching information about network packets against a set of rules, traffic classes can be distinguished. This forms a key building block of important services such as firewalling, routing, QoS provisioning or traffic shaping.

However, it is well-known that packet classification at line speed in high-speed networks is notoriously difficult, because every single packet has to be matched against a potentially large number of rules [9]. Thus, packet processing devices tend to form performance bottlenecks. In fact, this major problem has attracted many researchers in the last two decades [5, 6, 8, 13, 19, 22, 24, 25].

Yet, while many innovative and efficient algorithms were proposed, they have seldom been practically used—and if, this has mostly happened in high-end devices. The vast majority of devices performing complex packet processing in practice, though, are low-end, low-cost, software-based systems. Nevertheless, they are often used in performance-critical locations. Prominent examples are the widely used Linux netfilter/iptables subsystem [2], pf/pfctl in OpenBSD [3], FreeBSD's ipfw [1], and the OpenFlow reference switch [17].

These practically used packet classification systems are very rich in features and thus very complex. Their matching algorithms, though, are mostly trivial: they are most often based on basic linear search, scanning through the rule set one-by-one for each packet. In contrast, advanced classification algorithms barely support anything but stateless packet classification on certain header fields; they do this fast, but by itself it is not enough for what today's devices usually require. Changing the complex existing implementations at their heart to support more efficient matching is a huge undertaking—and apparently has not led to much over the past decade.[1] Moreover, there exists a huge basis of deployed network devices such as routers, NAT gateways, firewall appliances etc., where existing classification engines are deeply embedded. So, even if an implementation became available at some time in the future, a huge deployed basis of existing devices would not be able to benefit soon.

For these reasons, we take a somewhat different and indeed very pragmatic approach in this paper: we ask the question how we can boost the performance of linear-search classification engines *without the need to modify the implementations of the underlying matching algorithm*. We show that this is possible by building upon the jump ability that is present in many such classification engines.

We developed a proof-of-concept implementation called *HiTables*, which is publicly available at [10]. HiTables performs source-to-source transformations on Linux iptables rule sets without changing their semantics. It starts from the well-known HiCuts packet classification algorithm [8], but applies it in a very unconventional

[1]There have been projects with this aim, the most prominent one likely being nf-HiPAC (www.hipac.org) for Linux netfilter, but they have been abandoned long ago: nf-HiPAC development has ceased in 2005, and it is not compatible with today's kernels.

CoNEXT'14, December 2–5, 2014, Sydney, Australia.
Copyright 2014 ACM 978-1-4503-3279-8/14/12 ...$15.00.
http://dx.doi.org/10.1145/2674005.2675005.

way—not implementing it in the matching engine, but instead resembling it in the rule set structure. In essence, by re-arranging the rules and inserting jump rules with appropriate matching criteria, the linear search of the processing engine is tweaked into a search tree. In doing so, HiTables supports the full breadth of iptables features, because rules that are not amenable to re-structuring are transparently embedded into the transformed rule set.

In contrast to existing approaches in the domain of static rule set transformation [14–16], which are based on reducing the number of rules, we take the opposite approach and enlarge the rule set size during the transformation process. Nevertheless, our measurement results demonstrate that performance gains of more than one order of magnitude in terms of throughput are achievable, because the jump structure encoded into the rule set during HiTables processing leads to rule sets where only a small subset of the rules is actually used for each individual packet.

The remainder of this paper is structured as follows: Section 2 formally introduces the packet classification problem. Next, Section 3 describes related work in the field of packet classification. Section 4 explains the HiCuts algorithm, a major building block of the presented work. Subsequently, Section 5 presents in detail our proposed rule set transformation methodology, which is evaluated in Section 6. Finally, the paper is concluded by Section 7.

2. PACKET CLASSIFICATION

To pave the ground for our discussions throughout the rest of the paper, we first briefly recapitulate the standard terminology and notation of stateless packet classification as used here. Following [9], the problem can be formally defined as follows: Let $S_1, \dots, S_k$ be sets of non-negative integers representing the possible value ranges of packet header fields $1, \dots, k$, and let Act be a set of *actions*. Given a list of N rules (which is often referred to as the *classifier* or *rule set*)

$$C = \langle R_1, \dots, R_N \rangle$$

and a tuple of k non-negative integers (the packet header)

$$H = \langle h_1 \in S_1, \dots, h_k \in S_k \rangle,$$

find the smallest $i \in \{1, \dots, N\}$ such that H *matches* rule R_i. This assumes that rules with a smaller index (higher up in the list) have a higher priority if multiple rules apply to a packet. A rule R_i is defined by value ranges $D_{j,i}$ for each considered header field j and an associated action A_i:

$$R_i = \langle D_{1,i} \subseteq S_1, \dots, D_{k,i} \subseteq S_k, A_i \in Act \rangle.$$

H is said to match R_i if and only if all header fields lie within the respective value ranges, that is

$$\forall j \in \{1, \dots, k\} : h_j \in D_{j,i}.$$

According to previous works [6, 12, 13, 19, 21], it will be assumed for the rest of the paper that the field domains $D_{j,i}$ are (integer) intervals $[a,b]$. This holds for many practical applications, as firewall rules typically perform checks on IP subnets, port ranges, or single values such as protocol numbers, which can all be regarded as intervals. Moreover, any rule with checks on non-consecutive sets of integers can be transformed into a sequence of rules that consist only of interval-based checks. Typical actions include DROP or ACCEPT, meaning that the packet should either be rejected or accepted. Other actions are of course well conceivable (and exist in real packet classification systems).

Note that there exists a connection between the packet classification problem and computational geometry [9]: consider each rule R_i as a k-dimensional box $B(R_i) = D_{1,i} \times \cdots \times D_{k,i}$ within a k-dimensional universe $U = S_1 \times \cdots \times S_k$. Accordingly, a packet header H can be represented by a point $P(H)$ in U. Hence, finding the first matching rule R^* for H in the rule set is equivalent to locating the most highly prioritized box $B(R^*)$ containing $P(H)$.

This observation is important for two reasons: first, it served as inspiration for some of the most relevant classification algorithms. Second, it can be used to show that every algorithm that solves the packet classification problem for arbitrary k in $\mathcal{O}(\log N)$ time takes at least $\mathcal{O}(N^k)$ space, or needs at least $\mathcal{O}(\log^{k-1} N)$ time if it uses linear space [5, 9].

3. RELATED WORK

Packet classification has been the subject of intensive research in the last two decades and motivated a plethora of different approaches. Roughly, these approaches can be categorized into two partially overlapping domains: *rule set transformation* and *matching algorithms*.

3.1 Rule Set Transformation

One technique to improve the performance of packet classification engines is to analyze the characteristics of a specified rule set C_1 in order to compute another rule set C_2 that is equivalent to C_1, but hopefully finds matching rules faster. There exist online and offline variants of such rule set modification schemes. Online rule set modification observes the matching behavior at runtime, and continuously adjusts the data structures such that highly frequented rules can be found more quickly [4, 14].

Offline rule set modification typically aims to reduce the number of rules. This can happen either by detecting and removing redundant rules [15], or by decomposing, reordering, and merging rules [16]. Hence, offline rule set modification can be understood as a preprocessing step.

While the methodology proposed in this paper also performs offline preprocessing, it takes the opposite approach to existing techniques: instead of reducing the size of the rule set, we enlarge it in a controlled way in order to improve the matching performance.

3.2 Matching Algorithms

Classification algorithms which perform better than basic linear search build upon more advanced data structures, such as tries [5], hash maps [21], bit vectors [6, 13], or decision trees [8, 19, 24]. These data structures are precomputed from a given rule set and can be queried much faster than a linear list. However, they also require larger amounts of memory. Two particularly prominent classes of such algorithms are decomposition-based and decision tree-based approaches.

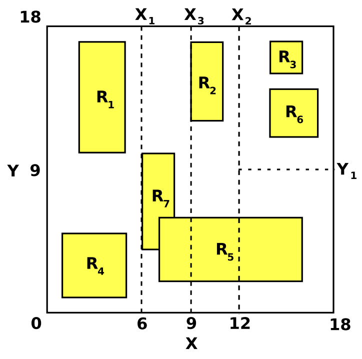
Decomposition-based approaches—like the Bit Vector [6, 13] or Crossproducting schemes [22]—reduce the k-dimensional search problem to k one-dimensional lookups, one for each relevant header field. The lookups can be performed efficiently and are amenable to parallelization. Subsequently, the one-dimensional outcomes are combined in order to obtain an overall match result.

Decision tree algorithms are inspired by the above discussed geometric view of packet classification [8, 19, 24]. During preprocessing, the universe U is decomposed into several areas which correspond to nodes in a tree structure. Subsequently, incoming packet headers are classified by traversing the decision tree. Prominent representative algorithms are HiCuts [8], HyperCuts [19], and EffiCuts [24].

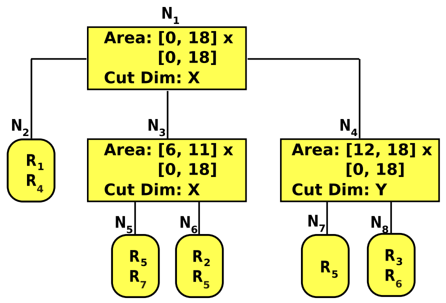
Because the approach taken in this work is based on HiCuts and uses the algorithm in an unorthodox way, Section 4 will repeat the necessary essentials of HiCuts.

Rule	X range	Y range
1	$[2,5]$	$[10,17]$
2	$[9,11]$	$[12,17]$
3	$[14,16]$	$[15,17]$
4	$[1,5]$	$[1,5]$
5	$[7,16]$	$[2,6]$
6	$[14,17]$	$[11,14]$
7	$[6,8]$	$[4,10]$

(a) Example rule set.



(b) Geometric view.



(c) Decision tree.

Figure 1: HiCuts example.

3.3 Practically Used Systems

Despite the existence of advanced matching algorithms like those described in the previous section, most practically used packet filters like iptables [2], pf [3], or ipfw [1] still perform a basic linear search in order to classify incoming network packets. Nevertheless, some of these systems provide a limited amount of optimizations. For instance, OpenBSD's pf offers a basic redundancy removal when loading rules. In addition, pf computes a database of so called *skip-steps*, which are used to skip rules in the list if they cannot match due to a failed check in a previous rule [11]. However, in order to be used efficiently, this technique typically requires a manual reordering of rules by the system administrator. FreeBSD's ipfw system provides the possibility to perform trie-based lookups on single fields in a rule. While this can surely speed up classification performance in specific scenarios, as for instance described in [7], it may be difficult to represent a large and complex rule set efficiently using this technique.

4. THE HICUTS ALGORITHM

The HiCuts algorithm was proposed by Gupta and McKeown in 1999 [8]. It is one of the most well-known packet classification approaches and has inspired many successive works that improved the original approach in matching performance, memory requirements, or preprocessing time [18, 19, 24]. However, the fundamental idea behind HiCuts persisted throughout these works and is sketched in this section.

In order to perform packet classification using HiCuts, a decision tree T is built from the rule set in a preprocessing step. This is done by recursively partitioning the universe U of all possible packet headers, which represents the root of T, hierarchically into nested smaller areas. These subareas correspond to the nodes in T, the tree resembles the nested structure of the areas. For an area A, the respective node in T will be denoted by $N(A)$ in the following.

The hierarchical area subdivision is done through so-called *cuts*. In one subdivision step, $l-1$ cuts ($l \geq 2$) are applied along one dimension (i. e., one header field). They partition an area A into l smaller areas $A_1,\dots,A_l$, resulting in l child nodes $N(A_1),\dots,N(A_l)$. Pointers to the newly created nodes $N(A_1),\dots,N(A_l)$ are stored in $N(A)$. Each of the newly created nodes $N(A_i)$ stores information about the area A_i as well as a pointer to every rule R_j that intersects with A_i. If the number of intersecting rules for a node $N(A_i)$ exceeds a predefined threshold called *binth*, then $N(A_i)$ is partitioned recursively (often with respect to a different header field). Otherwise, $N(A_i)$ represents a leaf in T. Hence, the binth parameter influences both the height of the generated tree as well as the time it takes to build the tree, because smaller binth values typically increase the number of cutting steps during tree construction.

In HiCuts, the partitioning cuts for each node are performed equidistantly along one dimension. The number l of cuts as well as the partitioning dimension are determined individually for each subdivision step by certain heuristics. Tweaks of these parameters and heuristics are a common theme of many HiCuts modifications and enhancements.

In the example shown in Figure 1, the rule set in Figure 1a is transformed into a decision tree depicted by Figure 1c, using a binth value of two. Figure 1b shows the geometric representation of the rules as labeled rectangles. It also shows the cuts that are performed, here indicated by dotted lines. First, the root node N_1 covering the area $[0,18]\times[0,18]$ is cut two times at points X_1 and X_2 along dimension X, which creates the child nodes N_2, N_3, and N_4, covering the areas $[0,5]\times[0,18]$, $[6,11]\times[0,18]$, $[12,18]\times[0,18]$, respectively. The area covered by node N_2 already intersects with only two rules R_1 and R_4, so it becomes a leaf. In contrast, the nodes N_3 and N_4 intersect with three rules each, so they are cut again at points X_3 and Y_1, respectively. The resulting four areas are covered by nodes N_5 to N_8. These nodes intersect with at most two rules each. Hence, the recursion terminates, and the decision tree is complete.

Note that during tree construction, rules can be duplicated when they are divided by cuts. In the example, rule R_5 was duplicated twice, as it was divided by cuts X_2 and X_3. Hence, nodes N_5, N_6, and N_7 each contain a pointer to R_5, as their areas intersect with the area spanned by R_5.

After the tree has been created, incoming packet headers can be matched against the rule set by traversing the decision tree, beginning at its root. Then, for each regarded node, it is first checked whether the node is a leaf or not. If yes, the rules pointed to by the leaf are searched linearly. This is fast, because the number of rules is limited by the binth parameter. If the node is not a leaf, the index of the child where the search continues must be computed. Because the cuts of each subdivision step are equidistant, this is doable in constant time with basic arithmetic operations. For further details on HiCuts, we refer the reader to [8].

HiCuts can accelerate packet matching significantly and is among the best-performing classification algorithms [5]. However, many common packet classification systems still lack an implementation of a sophisticated matching algorithm like HiCuts and rely on basic linear search. This can heavily reduce the maximum available network throughput, as we will demonstrate in Section 6.

5. JUMP-BASED TREE CONSTRUCTION

Even though HiCuts (and similar non-trivial packet matching algorithms) is not implemented in widespread software-based packet processing systems, we will now show how HiCuts-like matching can still be applied "on top" of existing linear search-based implementations, without changing their matching engines. To this end, we perform a source-to-source transformation on the input rule set using our proof-of-concept tool HiTables.

HiTables builds upon a widespread feature which allows to jump to different points in the rule set depending on intermediate match results. For instance, the likely by far most widespread packet classification engine in practice, Linux netfilter/iptables, includes the JUMP target for this purpose, which can be used to jump to a different so-called rule chain. Similarly, OpenFlow allows for GoTo instructions which are used to redirect the matching process to another flow table [17]. For the remainder of this section, our descriptions remain independent from a specific matching engine, but we assume that such JUMP semantics are supported.

Our proposed methodology consists of three basic steps: (1) parse the input rule set and decompose it into sub-lists that are either applicable for transformation or not, (2) build a decision tree for each sub-list that is marked as transformable, and (3) emit a new rule set that consists of those sub-lists that were marked as non-transformable as well as the decision trees which are encoded within the rule set.

5.1 Parsing and Decomposing the Rule Set

In the first step, each rule's semantics, such as its domain, its action, and its matching behavior, must be made accessible by parsing the input rule set. It can then be determined which of the input rules are amenable to transformation according to our approach. Although all practical real-world packet classification engines are able to discriminate packets based on their header fields, many of them support additional features like deep packet inspection or network connection tracking. For example, a rule could specify that it matches only on those TCP packets whose payload contains the sequence of characters "VIRUS", or that it depends on complex stateful filtering criteria. Rules using such features are present in almost every practically used rule set; however, they typically constitute only a small fraction of the rules, while the majority of rules performs stateless matching.

Decision tree algorithms are based on stateless classification as introduced in Section 2, and are not designed to handle complex matching schemes beyond stateless classification. Nevertheless, our proof-of-concept implementation is able to handle rules with arbitrary criteria: they are transparently taken over into the transformed rule set, while optimizations are performed "around" them, leaving the semantics fully intact. Rules which can be handled by a decision tree, namely those which match on the definition in Section 2, are called *transformable rules* in the following. Figure 2 sketches a rule set that consists of transformable rules as well as non-transformable rules.

5.2 Building the Decision Trees

Next, we construct HiCuts tree structures out of the transformable rules within the rule set. This is done separately for each consecutive sequence of transformable rules, interrupted by the non-transformable rules. In this step, the tree construction, in essence, follows the original HiCuts algorithm. The tree construction parameters, such as the binth parameter and the set of heuristics that guide the local cuts, can be specified when calling HiTables.

However, each decision tree introduces a certain amount of control overhead in the resulting rule set. Therefore, an additional

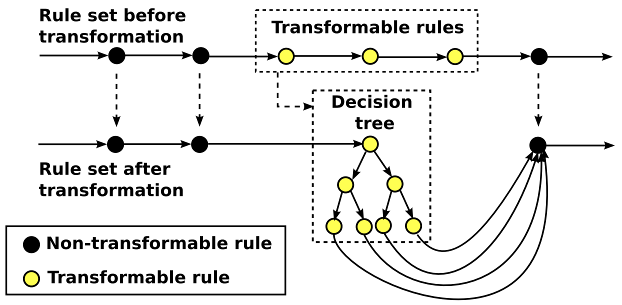


Figure 2: Rule set transformation.

parameter, the *minimum transformation length*, specifies the minimum number of consecutive transformable rules. If a consecutive group of transformable rules is shorter, it is taken over without transformation.

5.3 Emitting the Transformed Rule Set

In the key and final step of the transformation process, a new rule set is generated: the previously built decision trees are encoded in rules that are understandable by the used classification engine. During this step, the central idea of our approach comes into play: by exploiting the JUMP capabilities of the rule set semantics, we can encode the dispatch logic that is executed during the tree traversal in the rule set itself. Traversing the tree is then equivalent to a corresponding series of jumps in the linear list of rules.

Recall that the dispatch decisions at the tree nodes in the original HiCuts approach are performed by determining the respective child node, as discussed in Section 5. As explained above, this can be done in constant time by some basic arithmetic operations. Unfortunately, this necessitates the ability to *perform* such operations. Typical rule set semantics provide only one basic operation: checking whether a packet header matches a rule.

We take this hurdle by generating a sequence of rules that implements a binary search over the domains of the child nodes along the dimension which is used to perform the dispatch. As a simple example, Figure 3 sketches a node N_0 in a decision tree with five child nodes that performs a dispatch in dimension X. It is translated to the list of rules shown in Figure 4. First, a rule R_1 checks if the classification continues at child node N_3. Next, rule R_2 tests if the X packet header is located inside the bounding box of the child nodes N_1 and N_2, and dispatches to the left branch of the binary search. Rule R_3 specifies a wildcard match on the X header, which basically implements an unconditional jump into the right branch: Note that at this point the X header must lie inside of the interval $[5,19]$, otherwise the HiCuts search would not have reached node N_0. Finally, rules R_4 to R_7 implement the left and right branches of the binary search.

In order to embed a decision tree for a sequence of transformable rules in the modified rule set, we traverse the tree and generate a binary search sequence of rules for each inner node. Once we visit a leaf node of the tree, the original rules pointed to by the leaf node are included in the order according to their priority in the input rule set. Therefore, we preserve the semantics of the original rule set in the modified rule set.

If the classification process enters a decision tree and does not find a matching rule in one of the leafs, the search must continue in the remaining part of the rule set in order to keep the semantics intact. Therefore, we add a last rule in each leaf which we call *exit rule*. If no other rule matches, the exit rule redirects the classification process to the subsequent (non-transformable) rule taken

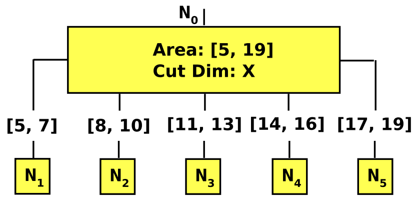


Figure 3: Node dispatch.

```
R_1          <X ∈ [11,13], JUMP N_3>
R_2          <X ∈ [5,10], JUMP LEFT>
R_3          <X ∈ ∗, JUMP RIGHT>
// left branch of binary search
R_4 LEFT:    <X ∈ [5,7], JUMP N_1>
R_5          <X ∈ ∗, JUMP N_2>
// right branch of binary search
R_6 RIGHT: <X ∈ [14,16], JUMP N_4>
R_7          <X ∈ ∗, JUMP N_5>
```

Figure 4: Rules implementing a binary search.

from the original rule set, as sketched in Figure 2. Thereby, the non-transformable rules from the original rule set are embedded. If further sequences of transformable rules follow further down in the original rule set, respective separate decision trees will be emitted.

6. EVALUATION

To demonstrate the feasibility of our approach, we performed measurements with our proof-of-concept implementation HiTables, which was used to transform Linux iptables rule sets. Our evaluation focuses on the achievable gain in classification performance in real and simulated environments with semantically equivalent transformed and non-transformed rule sets. In addition, we investigate the factor by which the size of the rule sets increases during transformation, the computation time required for the transformation itself, and the impact of non-transformable rules on classification performance. In our measurements, we used both self-generated synthetical rule sets as well as publicly available rule sets [20]. We varied the sizes of the self-generated rule sets from 100 to 3500 rules in steps of 200. Each generated rule specifies a random /24 IPv4 source subnet, a fixed destination address, and random source and destination UDP port ranges with a maximum range of 1000. In contrast, the public rule sets (`acl1_`N, `fw1_`N, `ipc1_`N, $N \in \{100, 1K, 5K, 10K\}$, where the suffix N indicates the different rule set size classes) vary widely in the sizes of the specified rule fields and contain a large number of wildcards. Furthermore, while the self-generated rule sets consisted only of transformable rules, the public rule sets also include non-transformable parts.

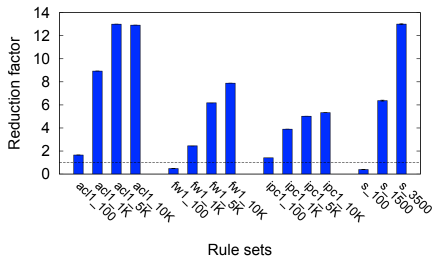
Path lengths. In our first experiment, we compared the number of rules that must be traversed in order to classify a trace of network packet headers in the original and the corresponding transformed rule sets. We did this by counting the number of traversed rules (the *path length*) for each header field specified by the traces in a simulated environment. Hereby, the size of the used traces was ten times as large as the number of rules in the original rule sets. The traces were generated by the ClassBench benchmark tool [23] in a way that they were uniformly distributed over the original rule sets and covered 100% of the rules. Figure 5a shows the speedup in terms of path length reduction for all public rule sets as well as the average speedup for ten self-generated rule sets of size 100, 1500, and 3500 (labeled `s_100`, `s_1500`, and `s_3500` in the figure), respectively. It can be seen that the performance gain grows with the size of the source rule sets, as the decision trees in the modified rule sets can be traversed significantly faster than the linear list in the source rule sets. In addition, Figure 5d illustrates the distribution of the path lengths for the transformed versions of the self-generated rule sets and the public rule sets `ipc1_1K`, `acl1_5K`, and `fw1_10K` in the same setting. While the transformed versions of the self-generated rule sets can classify every packet header from the used traces with less than 40 rules, the path lengths of the transformed public rule sets are more widely scattered, up to a maximum path length of 1939 rules. The reason for this is that the original public rule sets contain many non-transformable parts, which reduces the achievable gain of the transformation process. However, even in these cases, the path lengths of the transformed rule sets are still very small when compared to the sizes of the input rule sets.

Throughput. Next, we conducted a second experiment in order to evaluate the performance of our approach with real network traffic. Therefore, we configured the self-generated original rule sets on a resource-limited machine (Linux 3.14, 700 MHz ARM CPU, 512 MB RAM) which used the fixed destination address from the rule sets. A second, fast computer (Linux 3.13, 2.9 GHz Intel i7 CPU, 16 GB RAM) was used to send minimum-sized UDP packets for ten seconds as fast as possible, following a large ClassBench header trace corresponding to the currently installed rule set. In this setting, each action in each rule was ACCEPT, along with an ACCEPT default policy (which is applied when no rule matches). This is, of course, no sensible filter, but it is well suited for classification performance measurements: with such a configuration, the number of received UDP packets (determined with the netstat tool) corresponds to the number of packets that have been completely processed by the classification engine. We then computed the average processing time T_{orig} for the received packets. Subsequently, we repeated this process for the HiTables-transformed counterparts of the rule sets, using the same traffic traces as before, in order to compute the average processing time T_{mod} for the modified rule sets. We also determined the average packet processing time T_{IO} in the same setting for an empty rule set, which represents the I/O overhead of the packet processing. Finally, we computed the speedup factors with and without I/O overhead, i. e.,

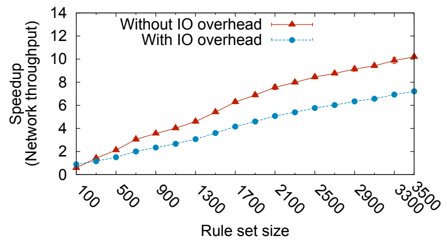
$$\frac{T_{orig}}{T_{mod}} \quad \text{and} \quad \frac{T_{orig} - T_{IO}}{T_{mod} - T_{IO}}.$$

This benchmark was independently repeated ten times, and the average speedup factors as well as the standard deviations are shown in Figure 5b. It can be seen that the source rule sets can be processed faster for very small rule set sizes, which is due to the control overhead implemented by additional rules in the modified rule sets. However, Figure 5b also reveals that the transformation effort quickly pays off as the rule set size increases.

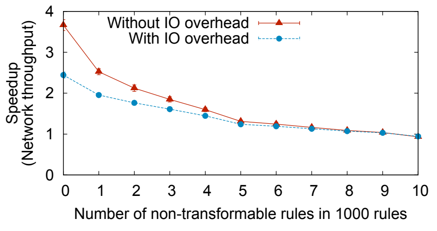
Non-transformable rules. In our third experiment, we examined the impact of non-transformable rules in the source rule sets on classification performance in the modified rule sets. We used synthetic source rule sets like above, each one consisting of 1000 rules. Then, we substituted an increasing number (zero to ten) of rules at equidistant positions with non-transformable ones. Thereby, we forced HiTables to emit $q+1$ decision trees instead of one when translating a source rule set with q non-transformable rules. We again ran this experiment in the real network setting and computed the same factors as above. As expected, Figure 5c shows that the performance of the transformed rule sets decreases. However, it also reveals that even in this sub-optimal setting, considerable per-



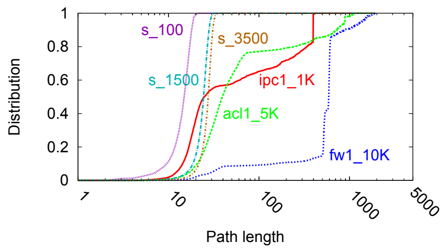
(a) Path length, original vs transformed rule sets.



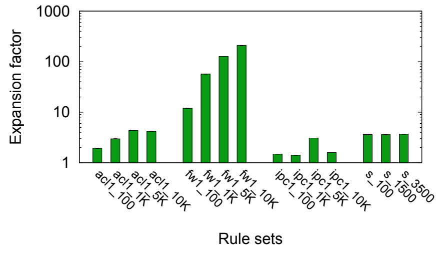
(b) Performance increase, synthetic rule sets.



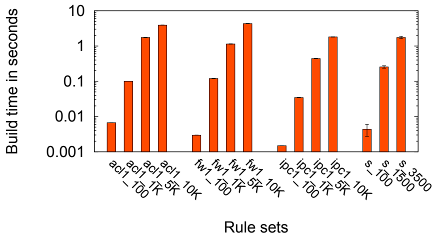
(c) Impact of non-transformable rules on perf.



(d) Cumulative distribution on path lengths.



(e) Rule set size increase.



(f) Transformation times.

Figure 5: Measurement results.

formance gains over the simple linear search are achieved if the number of separate decision trees does not become overly large.

Rule set size increase. Figure 5e illustrates the factors by which the transformed rule sets are increased. Again, we display these values for all used public rule sets as well as for the average values for synthetic rule sets of size 100, 1500, and 3500. Figure 5e reveals that the expansion factor does not directly depend on the size of the original rule sets, but instead is heavily influenced by the geometrical sizes and arrangements of the source rules. While the expansion factor of most rule sets is below ten, the transformed version of the `fw1_`*N* rule sets are very large (up to a factor of 208, in the case of `fw1_10K`). In fact, the `fw1_`*N* rule sets specify up to 38% more wildcarded fields in their rules than the other rule sets of the same size, which greatly favors rule duplications during the tree construction process.

Build time. Finally, Figure 5f displays the amount of time it took to compute the modified rule sets on our fast machine, which was in the order of seconds for all public and self-generated rule sets we used in the previous experiments. Hence, our approach is suitable for environments where the filtering rule sets do not change too frequently, as changes to the source rule set require a recomputation of the modified rule set in order to prevent a degeneration of the embedded decision trees. However, during our evaluation we encountered the problem that the transformation process for some very large rule sets (10K or more rules) or rule sets with many large rules (in a geometrical sense) took very long for a given binth parameter and cut heuristic. In such cases, it was often helpful to increase the binth parameter or to change the cut heuristic that guides the tree construction. For example, increasing the binth parameter from 8 to 20 reduced the transformation time for the `fw1_10K` rule set from 14.3 seconds to 4.3 seconds. Another way to mitigate this problem in the long term would be to apply one of HiCuts' successor algorithms, HyperCuts [19] or EffiCuts [24], which use more advanced techniques to reduce rule duplication during the tree construction process.

7. CONCLUSION

In this work we propose a technique to perform static transformations on rule sets of packet classification engines with JUMP semantics in order to increase their matching performance. In contrast to existing offline approaches for rule set optimization [15,16], which are based on reducing the number of rules, we take the opposite approach and enlarge the number of rules by encoding decision tree semantics within the rule sets themselves. That way, we can leverage the superior matching performance of an advanced classification algorithm without the need to change the implementation of the used classification engine. Our evaluation results reveal possible performance gains by over an order of magnitude when transforming rule sets with our proof-of-concept implementation.

Possible future work includes experimentation with more advanced decision tree algorithms like HyperCuts [19] of EffiCuts [24] in order to achieve even better matching performance at a smaller expansion factor.

Acknowledgments

Our sincere thanks go to the CoNEXT 2014 reviewers and, in particular, to Luigi Rizzo for very valuable feedback and discussions. This work was supported by a grant from the German Federal Ministry of Economic Affairs and Energy.

8. REFERENCES

[1] IPFW packet filter. https://www.freebsd.org/doc/en/books/handbook/firewalls-ipfw.html. Last accessed on September 28, 2014.

[2] The netfilter.org project. www.netfilter.org. Last accessed on June 14, 2014.

[3] OpenBSD packet filter. http://www.openbsd.org/faq/pf/. Last accessed on June 14, 2014.

[4] S. Acharya, M. Abliz, B. Mills, T. Znati, J. Wang, Z. Ge, and A. Greenberg. OPTWALL: A hierarchical traffic-aware firewall. In *NDSS '07: The 14th Annual Network and Distributed System Security Symposium*, Feb. 2007.

[5] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to CAMs? In *INFOCOM '03: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 53–63, Mar. 2003.

[6] F. Baboescu and G. Varghese. Scalable packet classification. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 199–210, Aug. 2001.

[7] M. Carbone and L. Rizzo. An emulation tool for PlanetLab. *Computer Communications, Elsevier*, 34(16):1980–1990, Oct. 2011.

[8] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *HOTI '99: Proceedings of the 7th Symposium on High Performance Interconnects*, pages 34–41, Aug. 1999.

[9] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network: The Magazine of Global Internetworking*, 15(2):24–32, Mar. 2001.

[10] S. Hager. Hitables source code. https://github.com/shager/hitables.

[11] D. Hartmeier. Design and performance of the OpenBSD stateful packet filter (pf). In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 171–180, Berkeley, CA, USA, June 2002.

[12] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (scalable and expressive packet classification). In *SIGCOMM '14: Proceedings of the 2014 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 15–26, Aug. 2014.

[13] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 203–214, Aug. 1998.

[14] H. Lam, D. Wang, and H. Chao. A traffic-aware top-n firewall approximation algorithm. In *INFOCOM WKSHPS '11: 2011 IEEE Conference on Computer Communications Workshops*, pages 1036–1041, Apr. 2011.

[15] A. Liu and M. Gouda. Removing redundancy from packet classifiers. In *SIGCOMM '04: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Sept. 2004.

[16] A. Liu, E. Torng, and C. Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *INFOCOM '08: Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 176–180, Apr. 2008.

[17] A. Nygren et al. OpenFlow switch specification. Technical report, Open Networking Foundation, Oct. 2013.

[18] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *INFOCOM '09: Proceedings of the 28th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 648–656, Apr. 2009.

[19] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 213–224, Aug. 2003.

[20] H. Song. Evaluation of packet classification algorithms. http://www.arl.wustl.edu/~hs1/PClassEval.html. website includes publicly available rulesets, last access: October 3, 2014.

[21] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *SIGCOMM '99: Proceedings of the 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 135–146, Aug. 1999.

[22] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *SIGCOMM '98: Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Aug. 1998.

[23] D. Taylor and J. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3), June 2007.

[24] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. In *SIGCOMM '10: Proceedings of the 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 207–218, Aug. 2010.

[25] T. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM '00: Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1213–1222, Mar. 2000.