On Rule Width and the Unreasonable Effectiveness of Policy Verification

H. B. Acharya
Email: acharya@cs.utexas.edu

***Abstract*—Policies, such as routing tables and firewalls, are fundamental components of networking infrastructure. Unfortunately, existing policy verification and optimization algorithms require $O(n^d)$ time, where n is the number of rules (thousands), and d the number of fields (usually < 10). However, these algorithms perform very well in practice. In this paper, we provide the explanation for this result: n and d are not the only parameters of interest! Through experimental study of our Parallel Next-step Lookup system PaNeL, as well as the FDD and Probe algorithms for policy verification, we clearly demonstrate the importance of our proposed new metric - the "width index". Some established algorithms (such as FDD, used for structured firewall design) indeed become intractable for policies with poor width index values. We therefore suggest that the "unreasonable effectiveness" of such algorithms for practical policies is possible because such policies have a reasonable width index.**

I. Introduction

Policies, such as routing and filtering policies (implemented in routing tables and firewalls) are essential to the operation of packet-switched networks, such as the Internet. Routers and middleboxes examine incoming packets and decide, based on the relevant policy, what course of action to pursue for each packet. Another important example occurs in system security, where in addition to packets, messages and system calls can be checked by Access Control Lists such as firewalls and Intrusion Detection Systems.

As may be expected, there has been considerable research into the problem of how to perform fast, correct *resolution* of packets, i.e. deciding which rule of the policy to apply to a given packet. In our own previous work, we developed a fast packet processing engine, PaNeL [1], which runs on a standard and cheap parallel computer (the eXtensible Multi Threading architecture by Vishkin [2]), rather than a specific-purpose machine such as a TCAM [3]. PaNeL showed clearly superior performance to serial resolution for policies of a few thousand rules, and by adopting the idea of processing a policy in "batches" of rules, we were able to obtain a speedup of around 20 times even for large policies of 100000 rules. However, in our studies comparing PaNeL with serial resolution, we found that factors besides the length of the policy (i.e. the number of rules) played a very important role in our results. On a more thorough investigation into these factors, we were able to identify the issue : serial resolution is fast for policies whose first rules match large numbers of packets, while parallel resolution is much more uniform in performance.

In this paper, we begin by proposing a new metric of a policy called the "width index"; this metric seeks to quantify how "fat" the rules of the policy are, i.e. how many packets can be expected to match them. As we show, across policies of different lengths, the width index is the factor that determines whether PaNeL or serial resolution show better performance. We then extend the scope of our work, by studying the impact of width index on FDD and Probe, two representative algorithms for policy verification. (We selected FDD as an example of an algorithm that preprocesses policies and builds a fast data structure, because it is used in structured firewall design. Probe was selected as an example algorithm that does no such expensive precomputation.) Our results show that the width index has a very strong effect on the performance of these algorithms - so much so, that we propose that the reason for the good performance of algorithms like FDD in practical usage (despite their poor theoretical running time of $O(n^d)$) is that practical policies have "friendly" values of width index.

We begin by presenting our definitions and concepts, and a very brief description of XMT, in the next section. Next, we discuss PaNeL, FDD and Probe algorithms, and how they may be affected by variation of the width index. We then present our experimental results, and show that they agree quite well with what we intuitively expect. Finally, we mention how our work fits into the context of related research in the area, and offer some concluding remarks.

II. Terms and Concepts

In this section, we define the terms and concepts used in the paper, such as policies and properties, and formally introduce our concept of the "width index".

A. Packets, Rules, and Matching

In our work, we model a *packet* as a d-tuple of non-negative integers. The reason for this model is that, in order to decide what to do with a packet (such as whether to forward it, which port to forward it on, etc.), network routers and firewalls examine its various attributes - mostly values in the packet header, such as source address, destination address, source port, destination port, protocol, and so on. (In 'deep packet inspection', attributes from the packet payload are also examined.) The d fields of our packet model represent the d attributes examined.

A *rule* represents a single rule in a routing table or firewall, and consists of two parts: a *predicate* and a *decision*.

The rule predicate is of the form

$$x_1 \in [x_{1,1}, x_{1,2}] \wedge x_2 \in [x_{2,1}, x_{2,2}] \ldots x_d \in [x_{d,1}, x_{d,2}]$$

where each interval $[x_{k,1}, x_{k,2}]$ is an interval of non negative integers, drawn from the domain of field k. (For example, suppose the third field in packets and rules represents source IP address. In IPv4, the domain of this field is $[0, 2^{32} - 1]$. Then, in any rule, $0 \leq x_{3,1} \leq x_{3,2} \leq 2^{32} - 1$.)

978-1-4799-3780-6/14/$31.00 ©2014 IEEE

The decision is an action, such as (in a firewall) *accept* or *discard*.

A packet that satisfies the predicate of a rule is said to *match* the rule. For example, the packet $(1, 26, 7)$ clearly matches the rule

$$x_1 \in [0, 108] \wedge x_2 \in [21, 65535] \wedge x_3 \in [7, 616] \rightarrow accept$$

A rule that cannot be matched by any packet is called a *match-none* rule, while a rule matched by every possible packet (in the domain of packets under consideration) is called an *match-all* rule.

B. Policies and Packet Resolution

A *policy* consists of multiple rules (as defined in the previous subsection), and a specification of which action to execute, in the event that multiple rules match a given packet. There are two principal methods in use to decide the precedence of matching rules.

1) *First Match.* The rules are arranged in sequence in the policy, and the action of the first rule in the sequence that is matched by a packet is the action executed.
   This is the method usually used in firewalls.
2) *Best Match.* One specific field is chosen and, out of the rules matched by the packet, the one with the smallest interval for this particular field has precedence.
   This is the method used in routing tables. The field chosen is the destination IP, and this technique is called 'longest prefix matching'. (In routing tables, IP address intervals are usually denoted by prefixes, e.g. $[1687603200, 1687603455]$, which in standard dotted-quad notation is $100.150.200.0 - 100.150.200.255$, is written $100.150.200.0/24$. Thus the best match is by the rule which had the longest prefix and was matched by the packet.)

The practical order of deciding precedence in a router is quite complicated. In short,

1) First, find the best match.
2) In case of conflict, choose rules in the order:
   a) Static routes
   b) Dynamic routes, in order (usually EIGRP, OSPF, ISIS, RIP)
   c) Default route
3) If no rule matches, discard the packet.

We note that this entire procedure can be effectively reduced to first match, by placing the rules in order with the primary sort key being the specified prefix length, and the secondary sort key being the type of rule. Hence, in our work, we simply assume first match semantics for all policies.

The 'winning' rule, the decision of which is implemented for a packet, is said to *resolve* the packet.

C. Width Index, PaNeL and XMT, and Verification

In policy research, the complexity of a policy is considered to be affected by two principal factors. The first is n, the number of rules in the policy, which can be up to several thousand; the other is d, the number of fields in a rule, which is usually around $5 - 10$. For example, the complexity of the simple standard algorithm for resolution - check, for each of d fields of the packet, that the value falls in the interval specified by the rule; repeat until a rule is matched - is clearly $O(nd)$.

However, we contend that this is not sufficient information to predict the complexity of a policy. Most rules in practical policies are concerned with very specific uses, and have multiple fields set to either a single specific value, or to "All" - i.e. the entire domain of the field. (For example, firewalls built using Structured Firewall Design have many fields set to All.) As we demonstrate in this paper, policies with high proportions of fields set to single values, or to All, can show significantly different behavior than other policies with the same values of n and d. Accordingly, we introduce a new concept, the *width index*.

The width index of a policy consists of two values, $(allprob, oneprob)$. $allprob$ is the probability that, on randomly choosing a field and a rule in the policy, the interval specified by the rule for the field is "All". Similarly, $oneprob$ is the probability that the interval specified by a randomly chosen rule for a randomly chosen field, is a single value, such as $[6, 6]$.

We owe the idea of the width index to our study of Parallel Next-Step Lookup, i.e. PaNeL, a system to perform fast resolution using a standard parallel machine, XMT.

XMT is an approximate implementation of a Parallel Random Access Machine (PRAM) - theoretically, a shared-memory machine in which an unbounded number of processors have unit-time access to unbounded memory. Its main limitation is that if multiple threads write to a variable at once, it is arbitrary which value gets written (formally, it provides arbitrary concurrent-read, concurrent-write semantics). Despite this limitation, and the fact that a practical implementation can only support a finite number of threads, we demonstrated in our previous work [1] that PaNeL outperforms serial resolution by more than an order of magnitude.

During this study, we made a very simple observation: fast policies are usually those whose first rules match many packets. Expanding on this observation, we gradually developed the idea of the width index, and how it might affect the performance characteristics of resolution.

We then expanded the scope of our study to check how width index affects the performance of other policy algorithms, besides packet resolution. In particular, we looked at policy verification, the process of determining whether a policy satisfies a *property*. (We adopt the convention that a property is simply a rule. The policy satisfies the property iff, for every packet p that matches the property P, the decision implemented by the policy for packet p is the decision of property P.)

There are two main types of policy verification algorithms. The first preprocess the property into a decision diagram, and then use this structure to rapidly answer queries about various properties. The second type use information that is available

only after receiving the property, and use it to query the policy in an optimized manner.

Over the next few sections, we provide brief accounts of PaNeL and of policy verification algorithms of both types, together with a discussion of how the width index may be expected to affect their performance characteristics.

III. PaNeL and Fast Resolution

In this section, we give a brief description of the PaNeL system for resolution on a parallel machine, and discuss how varying the width index can affect its performance compared to simple (linear) resolution.

Resolution involves finding the decision of the highest-priority rule in the given policy that is matched by the given packet. A parallel machine can speed up the search: it checks if the given packet matches the rule, for all the rules in the policy, in parallel. In case there are multiple matches, however, deciding priority is a problem - as all the rules are now checked in parallel, it is no longer possible to check the rules in order (most significant to least significant) and stop as soon as a rule is matched by the packet.

PaNeL [1] solves this problem by reducing it to finding the first 1 in an array A of $0/1$ elements. Element $A[i]$ of the array holds the Boolean value of whether the packet matches rule R_i; the rules are sorted in order of decreasing precedence, so the first 1 corresponds to the resolving rule. To resolve a packet, it is sufficient to return the decision of rule R_i, such that $A[i] == 1$ and there exists no $j < i$ s.t. $A[j] == 1$.

To find the first 1, PaNeL computes the *prefix sum* of array A. The prefix sum of an element $A[i]$ in an array A is the sum $A[1] + A[2] + ..A[i]$, and the prefix sum of the array A is the array A^+ such that for all i, $A^+[i]$ is the prefix sum of $A[i]$. The first 1 is the only element in A which is 1 and whose prefix sum is also 1. (All the other elements either have the value 0, or if they contain 1, this 1 adds to the prefix sum so its value changes.) Therefore, it is sufficient to return the decision of rule R_i, such that $A[i] == A^+[i] == 1$.

In theory, this algorithm should have high performance. It is possible to check if a packet matches a rule in $O(d)$ time; the prefix sum for an array of n elements can be computed in $O(\log n)$ time [4]; and after this computation, the first matching rule can be identified in constant time (by checking if $A[i] == A^+[i] == 1$, for all i, in parallel). The total time to resolve a packet, $O(d) + O(\log n) + O(1)$, i.e. $O(d + \log n)$, is far superior to the $O(nd)$ time expected for simple serial resolution (checking all the rules in sequence for the first match).

However, in practice, we found that the algorithm was fast for policies of a few thousand rules, but not larger, as it always had to process all the rules. In contrast, the serial solution stops at the first match - and if this match comes early in the sequence, the time taken is small.

Based on this insight, PaNeL processes a given policy in "batches" of rules. The function for parallel resolution is called with the given packet and a policy consisting of the first $step$ rules of the given policy Y. If no rule in the first batch resolves the given packet, ParallelResolve is called again, this time with a policy consisting of the next $step$ rules, and so on, until some rule resolves the packet, or there are no more rules in Y. This is the actual PaNeL system, which is competitively fast for large policies [1].

Fig. 1. The PaNeL algorithm for Packet Resolution

```
procedure PRESOLVE(Policy [R_1, R_2..R_n], Packet p)
    A[n], A^+[n] : integer arrays
    ans ← −1 : default ans
    for i ← 1, n pardo
        A[i] ← Match(R_i, p)
    end for
    A^+ ← PrefixSum(A)
    for i ← 1, n pardo
        if A[i] == 1 ∧ A^+[i] == 1 then
            ans ← R_i.D
        end if
    end for
    return ans
end procedure
procedure PANEL(Policy Y = [R_1, R_2..R_n], Packet p)
    step ← 1000, index, start, stop
    ans ← −1 : default ans
    for index ← 1, n/step do
        start ← (index − 1) ∗ step + 1
        stop ← min(start + step − 1, n)
        ans ← PResolve([R_start, ...R_stop], p)
        if ans ≠ −1 then
            break
        end if
    end for
    return ans
end procedure
```

We now come to the question of how width index can affect the performance of PaNeL relative to simple resolution. The width index of a policy consists of two parameters - $(allprob, oneprob)$ - the probability that the specified interval (for a field in a rule) covers its domain, and the probability that the interval is a single value. Clearly, increasing the value of $allprob$ increases the number of packets that match each rule, and therefore reduces the expected number of rules that have to be checked (in order of decreasing priority) until one of them resolves the packet. Increasing $oneprob$ has the opposite effect. Hence for high values of $allprob$ and low values of $oneprob$, most packets are resolved after checking only a few rules: serial resolution is fast. Conversely, for low values of $allprob$ and high values of $oneprob$, serial resolution is slow. PaNeL should show similar behavior, but with a much flatter curve, as it aggregates the rules into batches; the curve should be flatter for larger batch sizes.

After introducing and discussing the FDD and Probe algorithms for verification in the next two sections, we show in our results section (Section VI) that the experimental evidence agrees quite well with this analysis.

IV. Firewall Decision Diagrams

In this section, we describe the verification of policies using Decision Diagrams, as proposed by Gouda [5]. (As the

algorithm was originally employed for firewalls, we will use their traditional name of Firewall Decision Diagrams.)

A firewall decision diagram (over fields $f_1..f_d$) is an acyclic, rooted digraph. In an FDD, every node with no outgoing edges is marked with a decision (in case of firewalls, *accept* or *discard*), and is a *terminal node*. Similarly, every node with at least one outgoing edge, i.e. a *nonterminal node*, is marked with the name of a field. The outgoing edges from a nonterminal node are marked with values for its field, and the values marked on the edges (from a single node) do not overlap.

An FDD also has two properties.

1) It is weakly connected: there is at least one directed path from the root to every other node.
2) No directed path in the FDD has more than one node labeled with the same field.

Thus, an FDD is a representation for a simple deterministic finite automaton. Given any packet $p = (p.f_1, ..p.f_d)$, there is exactly one path from the root corresponding to the field values of p, and it terminates in a terminal node marked with a decision; this is the decision of D for p.

Clearly, given a policy with n rules of d fields, an FDD can perform packet resolution in $O(d)$ time. However, for policy verification, it is essential to follow every path from the root corresponding to packets that match the property; as the property can conceivably specify all possible packets, in the worst case it is necessary to follow all paths from the root to the terminal nodes of the FDD, i.e. it is lower bounded by the size of the FDD.

We now consider the worst case size of an FDD. Clearly, the worst case is when the FDD is a tree (say, for example, we may have a tree with root f_1; its neighbors are f_2, and so on down to f_d, whose neighbors are the leaf i.e. terminal nodes) and the branching factor of each node is as large as possible.

As there are n rules in the policy, there can be a maximum of $2n-1$ outgoing edges for a node. (Each edge must be labeled with at least one interval. The n rules have $2n$ end points - each interval has a start and an end. These $2n$ end points thus divide the domain into a maximum of $2n-1$ intervals. Hence we can have at most $2n-1$ edges from a node.)

Thus, the best known upper bound on the size of an FDD for a policy of n rules and d fields is $(2n-1)^d$, i.e. $O(n^d)$.

In our study, we consider how varying the width index affects the size of FDDs. For our explanation, it is necessary to examine the algorithm for FDD construction, presented in Figure 2 (summarized from Gouda [6]).

The FDD is built by adding rules to a root node, initially a completely empty node with no outgoing edges. The algorithm is recursive; when a rule is passed to a node labeled with a field, it may add new paths outgoing from the node, then passes rules with values for the remaining fields down to lower nodes in the FDD. There are two operations that increase the size of the FDD.

Fig. 2. Building FDD from policy

procedure ADDNEWRULE(Rule R, node x)
  if x is a terminal node **then**
    Label x with the decision of R.
  else
    i is the field of node x.
    $R.i$ is the interval for field i in R.
    $x.i_1, x.i_2..$ are the values on the edges from node x.
    Build new path from x, forming decision diagram of R. (Path has new nodes and edge labels as per the fields in R, other than field of x. Path ends in decision of R.)
    Label outgoing edge connecting x to new path with $R.i - \{x.i_1, x.i_2..\}$. If label is empty, delete this new path.
    for All $x.i_k$ **do**
      if $x.i_k \cap R.i$ is empty **then**
        continue
      else
        y is the target of edge labeled $x.i_k$.
        Copy the entire subgraph rooted at y.
        Let y' be the new copy of y.
        Add edge $x \rightarrow y'$, labeled $x.i_k - R.i$.
        Relabel edge $x \rightarrow y$ with $x.i_k \cap R.i$.
        $AddNewRule(R, y)$
      end if
    end for
  end if
end procedure

procedure BUILDFDD(Policy $\{R_1, R_2..R_n\}$)
  Create empty node x with no outgoing edges.
  Label x with desired field for root.
  for $index \leftarrow n, 1$ step -1 **do**
    $AddNewRule(R_{index}, x)$
  end for
  return FDD rooted at x.
end procedure

1) Adding a new path, when the rule specifies new values for the field at the node, for which no outgoing edges exist.
2) When the interval specified by the rule *partly* overlaps with the label of an outgoing edge, we 'split' the edge into two edges, labeled with $x.i_k \cap R.i$ and $x.i_k - R.i$, and make two copies of the subgraph below. (The edge with $x.i_k \cap R.i$ deals with the part that overlaps, so we pass the remainder of the rule recursively to the node along that path.)

Now we consider the effect of the width index $(allprob, oneprob)$ of a policy on the size of its corresponding FDD.

A high value of $oneprob$, clearly, will lead to many field values being a single integer, and thus, very few overlaps; it may be expected that the resulting FDD will have low branching factor at the lower nodes, and the size of the FDD does not become too large. The size of the FDD should decrease with increasing $oneprob$.

The behavior of $allprob$ is more complicated. Increasing the value of $allprob$ leads to "wide" rules, which overlap with each other; for moderate values, this leads to a complicated

FDD. However, when $allprob$ grows sufficiently large, most of the cases are of total (rather than partial) overlap, which does not lead to an increase in the size of the FDD. In fact, in the extreme case of $allprob = 100$, the FDD reduces to a linked list (from the root to the decision of the first rule, which resolves all packets). Thus, we expect that FDD size should increase and then decrease with increasing $allprob$.

V. Probe and the Projection-Division Algorithm

Probe, by Acharya and Gouda [7], is a policy verification algorithm. It attempts to find a *witness* packet, i.e., a packet p such that the given policy Y and the property P do not return the same decision for packet p. Iff there is no such packet, Y satisfies P.

For simplicity, we will label the decision of P to be "true" and all other decisions to be "false". Probe appends a match-all rule to Y with the decision "false", so P decides "true" and Y decides "false" for witness packets (rather than having cases where Y returns no decision at all for a witness packet).

Next, as the search space is constrained to packets that match P, we perform *projection* of Y over P. The predicate of every rule in Y is replaced by its intersection with the predicate of P. (Rules for which this intersection is empty, are simply removed.) For example,

$$R_1 = x_1 \in [0, 108] \wedge x_2 \in [21, 255] \wedge x_3 \in [7, 61] \rightarrow R_1.D$$
$$P = x_1 \in [0, 100] \wedge x_2 \in [0, 100] \wedge x_3 \in [0, 100] \rightarrow P.D$$
$$R_1/P = x_1 \in [0, 100] \wedge x_2 \in [21, 100] \wedge x_3 \in [7, 61] \rightarrow R_1.D$$

Further, as soon as we find a rule (say R_P) that *covers* P - i.e., is matched by every packet that matches P - we stop: no subsequent rules will resolve a witness packet, so they are of no importance.

The result of projecting Y over P is the policy Y/P. Given $Y = [R_1, ..R_n]$, we have $Y/P = [R_1/P, ..R_P/P]$.

Next, Probe examines the values of field f_1 in the rules in Y/P. Suppose in rule R_i, the corresponding interval is $[a_i, b_i]$. If R_i has the decision "true", Probe adds a to a set S_1; if the decision is "false" (and $b + 1$ does not fall outside the f_1 interval of P) Probe adds $b + 1$ to the set.

The same procedure is followed for all the fields, forming sets S_2 .. S_d.

Finally, Probe uses Y/P to resolve all the packets p in the Cartesian product $S_1 \times S_2 \times ...S_d$. Iff there is no witness packet in this set, Y satisfies P. [The formal proof of the correctness of this algorithm is published in [7].]

In the worst case, Probe examines m^d packets, where m, the length of Y/P, is upper bounded by n. Thus the best known bound on the complexity of Probe is also $O(n^d)$.

"Division" (also known as slicing) is not integral to Probe, but is an important optimization. It is based on the observation that a witness packet is always resolved by some rule with decision "false" in Y.

In division, we choose any rule R_j in Y with decision "false". We then create the projection Y/R_j. Finally, we remove from Y/R_j all the other rules (i.e. other than R_j) with decision "false". The resulting policy is the "slice" Y_j.

If R_j resolves packet p in Y, it will also resolve p in Y/R_j, and in Y_j.

Thus, after division, we need only apply Probe to all the slices of Y, rather than Y itself; iff there is a witness packet, it will be resolved to "false" by some slice. As the slices are usually very much smaller than Y, this greatly speeds up the algorithm; however, it does not improve the theoretical bound of $O(n^d)$ time.

Now we consider the probable impact of the width index ($allprob, oneprob$) on the Probe algorithm.

As discussed for FDDs in Section IV, a high value of $oneprob$ should cause most rules to not overlap with each other or with the property, leading to very small projected slices Y_j/P and thus fast performance.

A large value of $allprob$ is likely to result in short slices and projections, as a covering rule is quickly found. It should also ensure that the number of distinct values in the sets $S_1, S_2..S_d$ is relatively small (if the rules are very wide, the end values of intervals in the projection will usually be the end values of the intervals specified in the property). Thus, our intuition is that high values of $oneprob$ as well as $allprob$ should correspond to good performance.

In the next section, we present our experimental measurements of the actual response of PaNeL, FDD, and Probe to changes in width index.

VI. Experimental Results

In this section, we discuss our implementations of PaNeL, FDD and Probe, and note how their performance is affected by changes in width index.

A. Experiment 1: PaNeL

As the developers of PaNeL, we used the reference implementation, which is coded in XMT-C and runs on the XMT parallel machine. (XMT-C is almost identical to C; the only non standard command used in the code for PaNeL is 'spawn'.) For timing, we use the built-in simulator macro, `xmt_readtimer32(time1)`, which when called loads the (32-bit) cycle count into integer `time1`.

In our experiments, we used rules with 5 fields, each of which has the domain $[0, 65535]$. The length of the policy was varied from 1000 to 10000 rules, and the width index ($allprob, oneprob$) was varied as $(0, 0), (0, 20)..(0, 100), (20, 0), ..(80, 0), (80, 20)$. (Clearly, the case $(100, 0)$ is trivial: all packets are resolved by the first rule.) For each length of policy, we tested 100 random packet resolutions against 10 distinct policies and reported the average time.

The general trend seen in our results is that, as expected, packet resolution is much faster with high values of $allprob$ - PaNeL only pulls ahead for $allprob$ values below 40. Also, resolution is slower for higher values of $oneprob$, as more rules

Fig. 3. Resolution time: Policy of 1000 rules

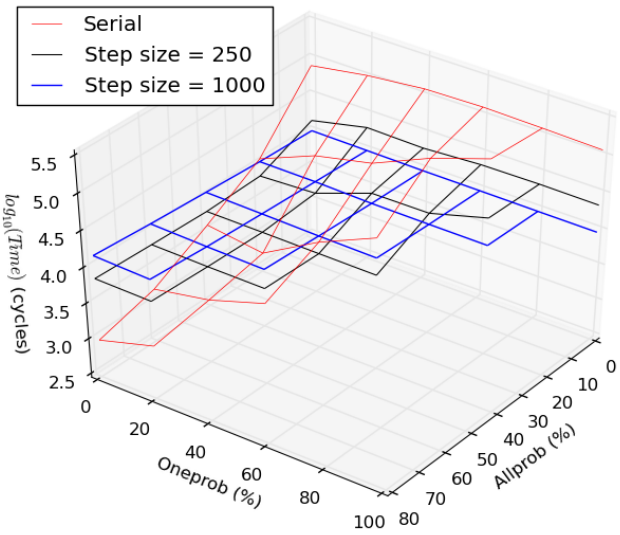


Fig. 4. Resolution time: Policy of 4000 rules

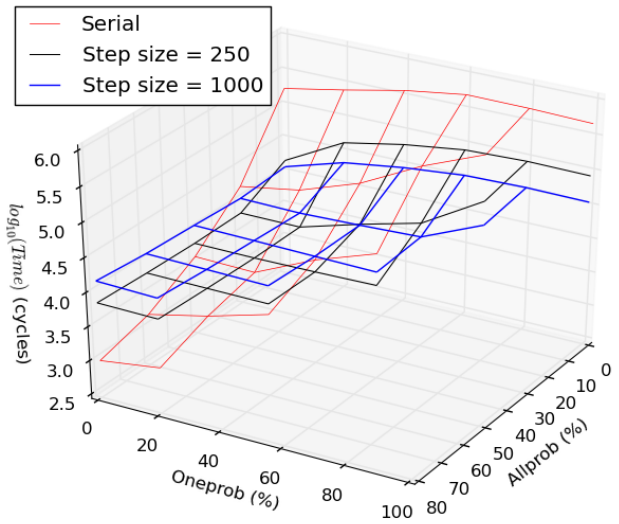


Fig. 5. Firewall Decision Diagrams: Size

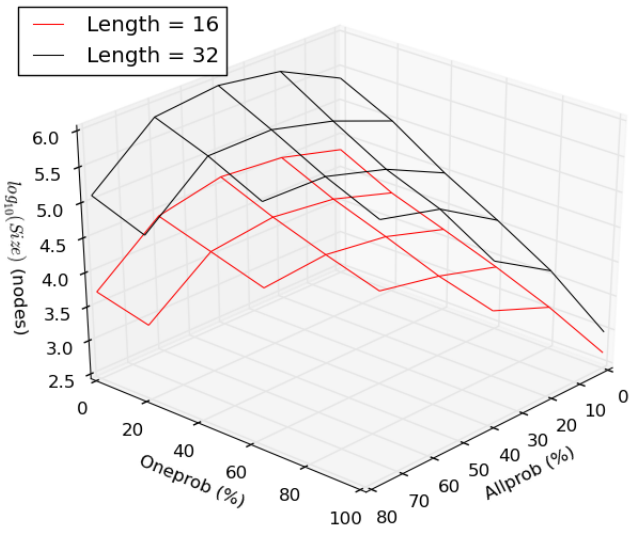
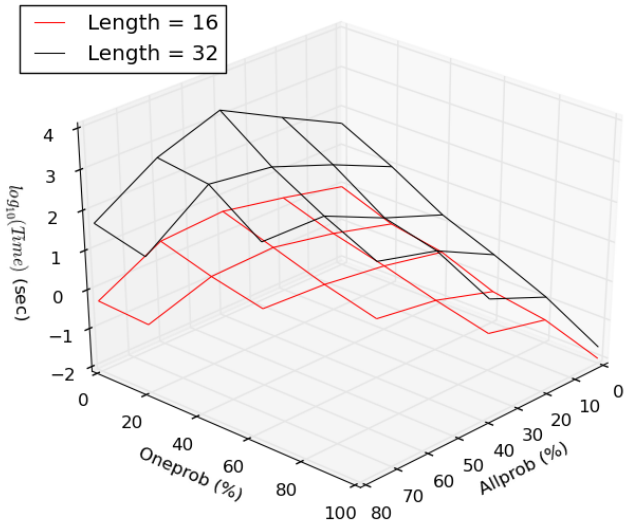


Fig. 6. Firewall Decision Diagrams: Time



on average need to be tried before one of them resolves a packet.

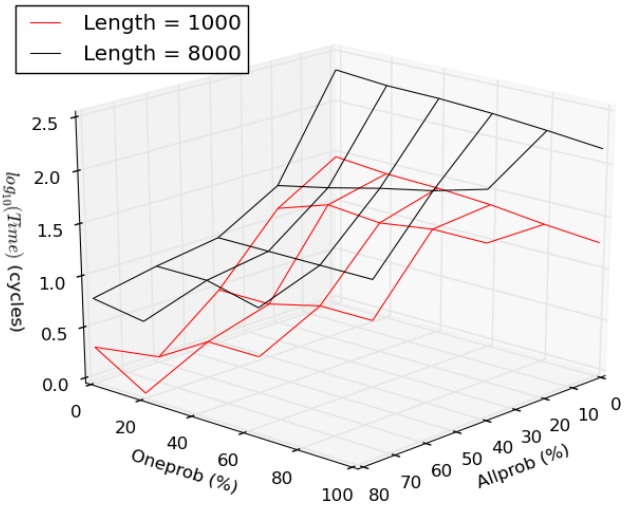
PaNeL shows a relatively flat response. There is some initial decrease in resolution time as $allprob$ increases and also as $oneprob$ decreases, but it becomes quite flat for values of $allprob$ beyond 20. Also, PaNeL with a large step size shows a flatter response than with a small step size. Indeed, for the smallest policies, with length 1000, PaNeL is perfectly flat (exactly as expected, as the step size is 1000, and the time required is constant: the time to process one batch of 1000 rules in parallel, and find the first match).

B. Experiment 2: Firewall Decision Diagrams

Our second set of experiments involved developing an implementation of Firewall Decision Diagrams, and checking the size as well as the time required. We initially started with policy sizes of $100, 200..1000$ but our python-2.7 implementation consistently ran out of memory and crashed on 8 GB workstations. To be able to complete our experiment for all values of width index, i.e. $(0,0),..(0,100),(20,0)..(80,20)$, we used small policies of length 8, 16, and 32 rules. For each data point, we took the average values for FDDs generated from 100 random policies.

Our results are presented in Figure 5 and Figure 6. As may be expected, the two figures are extremely similar, and follow a clearly visible pattern. The space and time requirement curves rise to a maximum at $allprob = 40$, and then fall for higher values. Also, both graphs fall steadily with increasing $oneprob$; the drop seems gentle and linear, but the z axis in the figures is logarithmic, so in practice there is an exponential increase

Fig. 7. Probe: Time



in FDD size as *oneprob* becomes smaller.

We conclude that most likely, in the earlier successful studies on FDD [8] which use policies of as much as 660 rules, the value of *oneprob* was quite high for the policies used in the experiment. For the sake of completeness, we then performed a small test with a width index of $(0, 90)$ for policies of $100, 200..800$ rules, taking average values over 10 policies of each length.

Our results are as follows.

Rules	Size (nodes)	Time (sec)
100	15981	0.52
200	78249	4.77
300	185237	20.9
400	401176	116
500	670664	182
600	934651	299
700	1454927	1516
800	2124602	2504

These results are in reasonable agreement with the values reported by earlier researchers, when we adjust for implementation differences (Python is approximately 200 to 250 times slower than C).

C. Experiment 3: Probe

Our final set of experiments were with the Probe algorithm for policy verification, implemented in plain C. (As Probe is very fast and runs in milliseconds, we used the non-ANSI **clock()** function rather than **time()**, which has only second level resolution.) As in all our experiments, width index was varied from $(0, 0), (0, 20)..(0, 100), (20, 0)..(80, 20)$. The policy length was varied as $1000, 2000..10000$ rules. We took the average of $10,000$ readings for every point in our experiments.

Our results can be seen in Figure 7. The general shape of the graph seems quite similar to the graph for serial resolution in Figure 3; there is a very pronounced effect of the time decreasing as *allprob* increases.

oneprob does not have such a clear effect; increase in *oneprob* causes a decrease in the running time when *allprob* is very high or very low, but affects the time more irregularly for moderate values of *allprob*. We were surprised that *oneprob* has such a small effect, and investigated the working of the algorithm in more detail. We suggest that the reason for the observed behavior is that *oneprob* can only become large when *allprob* is small; in these cases, it is quite rare that two rules intersect with themselves and also with the given policy - a very large majority of projected slices are extremely small, and have only 1 to 3 rules. Making the rules "thin" by increasing *oneprob* has very little additional effect.

We believe the experimental evidence clearly demonstrates the importance of the width index in the study of policies; in particular, *allprob* has a powerful effect on packet resolution as well as Probe, and *oneprob* on Firewall Decision Diagrams.

VII. Related Work

Our work in this paper contributes to the study of policies, their operation and verification. Owing to the practical importance of policies, there exists a considerable body of work devoted to their study; in this section, we discuss how this paper fits into the context of this research.

The first and most obvious area related to our work is the design of algorithms and data structures for fast packet processing (i.e. resolution) as well as for verification of policies. A wide variety of specialized data structures have been used to represent policies, notably tries [9] and lookup-table based solutions by Waldvogel [10] and Gupta [11]. As preprocessing can be expensive, solutions have been developed by Suri [12] using B-trees, and by Sahni and Kim [13] using red-black trees and skip lists; these solutions allow fast update, and also perform (longest prefix) matching in $O(\log n)$ time.

We demonstrate that the width index has a dramatic effect on the size of one such structure, the firewall decision diagram [5]. Indeed, we argue that it is possible to predict from the width index (and the length) of a policy, whether it is likely to have a decision diagram of tractable size. (Exceptions are possible; for example, a policy consisting entirely of repetitions of the same rule will have a very simple decision diagram - a linked list - irrespective of its width index and its length. However, we believe that such exceptions are very unlikely to occur in practice.) The question immediately follows whether width index is also an important metric for predicting the complexity of other specialized data structures, as mentioned above. We intend to explore this direction in future work.

This paper grew out of our work on fast resolution, discussed in Section III, and therefore also contributes to another area of research: the study of high-performance architecture and its application to packet resolution. Several current systems, such as backbone routers, make use of special hardware - ternary content addressable memory [3], ordinary RAM, pipelining systems [14], and so on; the goal is to improve the performance of policies. Our experiments with PaNeL show that width index can be the determining factor in how fast a

policy implementation performs; while PaNeL is very good for policies with low values of $allprob$, the serial implementation is superior as $allprob$ becomes high. In the current literature, testing is performed by varying the length of the policy, but our results show that this is insufficient; which of two approaches is better, may depend on the width rather than the length of the policy.

This result is not limited to choosing an implementation for a policy; our experiments demonstrate that the width index of the policy is an important factor in choosing an algorithm for verification or optimization, as well. More generally, the analysis of policies includes the study of anomalies [15], inter-rule conflicts [16], redundancies, and so on. For example, Frantzen [17] provides a framework for understanding the vulnerabilities in a firewall, and Blowtorch [18] is a framework to generate packets for testing. It may be noted that Probe [7], one of the algorithms we show to be strongly affected by width index, has also been used for policy optimization by removing redundant rules [19]. It seems very probable that the width index of the analyzed policy is an important factor in the complexity of these algorithms also. We suggest that width index is a metric of considerable general interest, and plan to extend our study of its impact on policy specifications and implementations in our future work.

VIII. Conclusion

Policies are used in both routing and filtering of packets, and are thus critical components of network infrastructure. In this paper, we make two contributions to the theory of policies and their complexity. Our first contribution is to define the "width index", a new metric for the complexity of a policy. Width index is very easy to compute (one $O(nd)$-time pass through the policy is sufficient to count the number of fields in rules set to single values or to "All"), and can, as we demonstrate experimentally, have a dramatic effect on the behavior of algorithms that deal with policies. In particular, algorithms such as FDD, which are not only well studied but widely used in practice, are intractable for policies with poor values of width index. Thus, for our second contribution, we propose an explanation for the "unreasonable effectiveness" of such algorithms in practice (despite their $O(n^d)$ running time): practical policies have tractable values of width index.

Our work with the width index suggests several problems for further study. The idea of the index can be extended to be more precise - for example, by specifying the width index field by field (a policy with width index $(100, 0)$ for a 128-bit field like IPv6 address, and $(0, 0)$ for some 1-bit field, say protocol, may behave differently than one with $(0, 0)$ for address and $(100, 0)$ for protocol). On the other hand, perhaps it can be made more simple; the index consists of the two separate measures $allprob$ and $oneprob$, which are important for different cases, and it would be interesting to see if it can be expressed as a single one. Another possible line of attack would be to examine other metrics, such as how many other rules the average rule in a policy overlaps, and their relationship to the width index.

As our own next step, we intend to study the impact of width index on further algorithms and data structures in policy research, such as FDD compression by Bit Weaving [8]. We hope that, by making use of the width index, we will be able to develop tighter theoretical bounds than the current $O(n^d)$ bound for policy verification and optimization algorithms.

References

[1] H. B. Acharya, "Panel : The parallel next-step lookup system," 2014.

[2] X. Wen and U. Vishkin, "Pram-on-chip: first commitment to silicon," in *SPAA*, 2007, pp. 301–302.

[3] D. Shah and P. Gupta, "Fast updating algorithms for tcams," *IEEE MICRO*, vol. 21, no. 1, p. 3647, 2001.

[4] G. E. Blelloch, "Prefix sums and their applications," Synthesis of Parallel Algorithms, Tech. Rep., 1990.

[5] A. X. Liu and M. G. Gouda, "Firewall policy queries," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 6, pp. 766–777, 2009.

[6] ——, "Diverse firewall design," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1237–1251, 2008.

[7] H. B. Acharya and M. G. Gouda, "Projection and division: Linear-space verification of firewalls," *Distributed Computing Systems, International Conference on*, pp. 736–743, 2010.

[8] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, vol. 20, no. 2, pp. 488–500, 2012.

[9] K. Sklower, "A tree-based routing table for berkeley unix," in *Technical report, University of California*, 1993.

[10] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed ip routing lookups," in *Proceedings of ACM SIGCOMM*, 1997, p. 2536.

[11] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings of IEEE INFOCOM*, 1998.

[12] S. Suri, G. Varghese, and P. Warkhede, "Multiway range trees: Scalable ip lookup with fast updates," in *GLOBECOM*, 2001.

[13] S. Sahni and K. Kim, "O(log n) dynamic packet routing," in *IEEE Symposium on Computers and Communications*, 2002.

[14] A. Basu and G. Narlika, "Fast incremental updates for pipelined forwarding engines," in *Proceedings of IEEE INFOCOM*, 2003.

[15] H. H. Hamed, E. S. Al-Shaer, and W. Marrero, "Modeling and verification of ipsec and vpn security policies," in *ICNP*, 2005, pp. 259–278.

[16] D. Eppstein and S. Muthukrishnan, "Internet packet filter management and rectangle geometry," in *SODA*, 2001, pp. 827–835.

[17] M. Frantzen, F. Kerschbaum, E. E. Schultz, and S. Fahmy, "A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals," *Computers & Security*, vol. 20, no. 3, pp. 263–270, 2001.

[18] D. Hoffman and K. Yoo, "Blowtorch: a framework for firewall test automation," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 96–103.

[19] H. B. Acharya and M. G. Gouda, "Firewall verification and redundancy checking are equivalent," in *INFOCOM*, 2011, pp. 2123–2128.