The Static DPF Simulator (v.2)*

Ali A. Selçuk Kihong Park Heejo Lee

Network Systems Lab
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{selcuk,park,hlee}@cs.purdue.edu

CSD-TR 02-008
September 7, 2002

Abstract

The Static DPF Simulator is a tool for evaluating the performance of the Route-Based Distributed Packet Filtering (DPF) protocol [4, 3] in a static network environment. In this document, we describe the operation of the DPF simulator and the structure of the programs in the package, and discuss certain issues related to installing and running the DPF simulator tool suite.

*This work was supported in part by grants from DARPA ATO FTN (AFRL F30602-01-2-0539) and CERIAS.

Contents

1 Introduction

The Static DPF Simulator is a tool for evaluating the performance of the Route-Based Distributed Packet Filtering (DPF) protocol [4, 3] in a static network environment where the nodes and edges do not change and transient time dynamics of protocols—e.g., route table convergence—are ignored. Given a network topology with a list of nodes and edges, and also given a list of the filter nodes in the network, the DPF simulator first calculates, or uploads, the routing tables for pairwise routes in the network; then sets the DPF filters to allow only the source addresses that are legitimate according to these routes; and finally computes according to these filters which nodes are able to launch an attack on which nodes using which spoofable addresses, and it outputs the results. The proactive and reactive performance measures computed by the Static DPF Simulator are exact for a given input. Hence, in this sense, it is different from a typical "simulator" where approximate solutions are the norm.

In this document, we describe the operation of the DPF simulator and the structure of the programs in the package, and discuss certain issues related to installing and running the DPF simulator tool suite. The static DPF simulator is written in C, and runs on Linux and Unix operating systems. It can be easily ported to Windows platforms.

1.1 Changes in the Second Version

The main differences of the second version of the simulator from the first version are as follows:

- Egress filtering. In the first version of the simulator, if egress filtering was to be performed, it was performed by all the DPF filter nodes in the network. In the current version, the egress and DPF filtering functionalities are completely decoupled. The list of nodes to perform egress filtering must be supplied as a separate file.
- Trace option. A new binary command-line argument is added which specifies whether a trace file will be generated which contains a summary of the distribution of the nodes that can launch spoofed attacks.
- New auxiliary tools. Two new auxiliary tools are included in the package: One tool collects various statistics over the paths in the network, the other executes a randomized routing algorithm which can be used as an alternative to the shortest-path routing.

2 General Structure of the Simulator

The operation of the DPF simulator consists of the following stages:

1. **Input** The input files are read in, the nodes and edges are created, and the filters and routing tables are allocated. The input files are a file specifying the network graph topology in adjacency list format, another file that lists the filter nodes on the network, and a third file that lists the transit nodes.

   The function that is in charge of the input stage is `GetInputGraph` in the init.c file.

2. **Route Computation** Pairwise routes are computed and the route tables are filled. By default, the routes are shortest path routes computed by Dijkstra's algorithm, where Dijkstra's algorithm is run starting at the destinations rather than the more common way of starting at the sources. This is necessary to guarantee that the path from the next hop to a destination is a sub-path of the original path from the original source.

   In addition to shortest path routing, the DPF simulation tool supports routes computed by any other algorithm through a route upload option. If this option is selected, first the routes are uploaded from a specified file and checked for cycles. If the routes are cycle-free, the simulation proceeds to Step 3, optionally after complementing the routing tables with shortest path routes for any missing source-destination pairs in the uploaded file. The usage of routing options is discussed further in Section 5.3

   The main function in charge of the route computation stage is the `ComputeRoutes` function in the route.c file.

3. **Filter Computation** The filters at the edges of the DPF filter nodes are constructed to allow only the source addresses that can traverse that edge according to the routes computed in stage 2. This is done by sending a filter update packet between every source-destination pair and updating all the filters on the path to allow that source address.

   This operation is performed by the `TraverseRoute_SetFilters` function in the filter.c file.

4. **Set Computation** An $S_{a,t}$ set is the set of spoofable addresses that can be sent from the attack source a to the target t escaping filtering. The fourth stage of the program computes the $S_{a,t}$ sets for all (a,t) pairs. The set computation algorithm is based on the recursive property that, for $a \neq t$ and b denoting the next hop from a to t,

$$S_{a,t} = \begin{cases} S_{b,t}, & \text{if } b \text{ is not a filter node} \\ F_{a,b} \cap S_{b,t}, & \text{if } b \text{ is a filter node} \end{cases}$$

   where $F_{a,b}$ is the set of permissible source addresses on the filter edge on b coming from a.

   Given the property above, the algorithm starts with $S_{t,t} = V$, V denoting the set of all nodes, and computes the $S_{a,t}, a \neq t$, sets in a recursive fashion. At the end,

$S_{t,t}$ is set to $\emptyset$ since t is not considered an attack node on itself. Similarly, if egress filtering is in effect, $S_{a,t}$ is set to $\{a\}$ at the end for all filter nodes a.

The main function in charge of this stage is `ComputeSets` in the filter.c file.

5. Output Once the $S_{a,t}$ sets are computed for a given t, their cardinalities $|S_{a,t}|, a \in V$, are written out to a specific file.[1] At the same time, the dual sets $C_{s,t}$—which list the possibilities for the actual source of a packet received at target t with the apparent source address s—are derived from the $S_{a,t}$ sets and their cardinalities are also written out. The outputting is done for a target t as soon as the computation is completed for that target in order to reduce the space requirement of the program.

The function in charge of computing and writing out the cardinalities is `SaveStats` in the output.c file.

3 Data Structures

The main data structures in the simulator are the nodes, edges, routing tables, and filter tables.

The nodes are represented by the `NODE` data structure which consists of a list of the edges incident on the node and a routing table. The definition of the `NODE` structure is as follows:

```
typedef struct {
    EDGE          *edge;
    RTABLE_ENTRY  *rtable;
} NODE;
```

The edges of a node are represented as an `EDGE` array of size equal to the degree of the node; the degree of the nodes are stored in the central array `Degree`. An `EDGE` structure includes the node ID of the neighbor node. It also includes a pointer to the filter on the neighbor, if the neighbor node is a filter node.[2] If the neighbor is not a filter node, this pointer is set to nil. The definition of the `EDGE` structure is as follows:

```
typedef struct {
    NODE_ID  to;
    FTR_ID   ftr;
}  EDGE;
```

[1] Optionally, the explicit $S_{a,t}$ and $C_{s,t}$ sets can be written out instead of just their cardinalities, as explained in the discussion of the *sets* argument variable in Section 5.3.

[2] Keeping a pointer to the next filter is in order to speed up the filter discovery process during the filter and $S_{a,t}$ computations.

The other component of a node is its routing table, which is an array of `EDGE_ID`s indexed by the destination address. That is, the ith entry of the route table gives the edge to be taken to reach the AS node with node ID i.

The last major data structure in the DPF simulator is the filter table. A filter table includes the list of source addresses that are permissible at the edge where it is located. There are two different representations of a filter table utilized in the program:

1. A bit array representation, where a 1 at the ith entry denotes that the source address i is allowed to pass this filter and a 0 denotes that it is not allowed.

2. A dynamic table representation, which includes only the allowed source addresses.

The bit array representation has the advantage of enabling direct access to the filtering value of a certain address. The dynamic table representation has the advantage of enabling fast enumeration of the allowable addresses where the bit array representation would be too sparse. The simulator program performs both of these operations quite often. Therefore the filters are maintained in both formats, as a bit array and as a dynamic table, at the same time.

The filter tables are maintained in two global arrays, `Filter1` and `Filter2`. The former holds the filter tables in bit array format; the latter holds them in dynamic table format. The filter tables are pointed to by the `ftr` field of the `EDGE` data structure.

4 Directory Structure

The DPF simulator's home directory includes a number of subdirectories. These subdirectories and their contents are as follows:

dpf/ This is the directory that includes the simulator code. All files discussed so far are located in this directory.

cover/ This directory includes an auxiliary tool which computes a filter cover on a given graph. The computed cover can be either a vertex cover or a cover with a specified size. Vertex cover is shown to be an effective choice for the placement of the filter nodes [4, 3].

measure/ This directory also includes an auxiliary part of the simulator package. It provides the programs to compute the two main performance measures of the DPF tool, Φ_2 and Ψ_1, as discussed in [4, 3].

examples/ This directory includes a sample set of the simulator's input and output files, discussed in Section 5, with $ID = 19971108$. The data are based on the Route Views

Project's BGP dump file for November 8, 1997; ASmap.19971108.879009857.gz at http://moat.nlanr.net/Routing/rawdata/ .

misc/ This directory includes the auxiliary tools which are not directly related to the simulator but can be useful for experimentation. Currently two tools are included: One for generating randomized routes, included in the random_rt subdirectory; the other for collecting various statistics about the paths in the network, included in the path_stat subdirectory. Further information on these tools are provided in README files in the respective directories.

5 Using the Simulator

5.1 Installation

Running *make* at the simulator's home directory will generate all the executables for the simulator as well as for the auxiliary tools.

One point that needs care is the path information on the first line of the script files. Currently, the perl script files start with the line

```
#!/usr/bin/perl
```

The path for perl may be different on different systems. It is the user's responsibility to make sure that the path information at the beginning of the script files is updated according to the host system.

5.2 Preparation of the Simulation Input

The DPF simulator takes four files as input:[3] One file specifying the AS graph topology in an adjacency list representation, one file listing the transit nodes in the network[4], one file listing the DPF filter nodes, and a fourth file listing the nodes which perform egress filtering[5]. The following four-step procedure describes a basic way of preparing these input files. These steps are not an integral part of the DPF tool and can be skipped if the input files are obtained by other means.

[3]Optionally, a fifth file that includes the pre-computed routing entries is needed if route uploading is chosen. This option is specified by the *route* argument variable, discussed in Section 5.3.

[4]It is possible to run the simulator without making any distinction between stub and transit nodes by simply including all AS numbers of the network in the file specifying the transit nodes. The file `examples/AStrans.all`, which lists all numbers $x, 0 \leq x \leq 65535$, can be used for this purpose.

[5]The egress nodes may be taken to be the same as the DPF filter nodes.

Before executing the following procedure, a BGP dump table in the format of the Oregon Route Views[6] Project's dump tables must be present in the simulator's home directory. The name of this file must be in the form of "ASmap.*ID*", where *ID* is the identifier of the data, usually the date of the dump, such as 19990101.

Step 1: Extracting the AS paths from the BGP dump file
Command: % get_ASpaths.pl *ID*
Product: ASpaths.*ID*

Step 2: Deducing the graph topology from the AS paths
Command: % get_ASgraph.pl *ID*
Product: ASgraph.*ID*

Step 3: Identifying the transit nodes
Command: % get_AStrans.pl *ID*
Product: AStrans.*ID*

Step 4: Computing a vertex cover for the filter placement
Command: % cover/vc ASgraph.*ID* AStrans.*ID* > ASfilter.*ID*
Product: ASfilter.*ID*

By default we assume that the filters will be placed at the transit nodes only, hence the VC selection is restricted to the transit nodes. If all nodes are to be allowed in the VC selection, the `cover/vc` program must be run with the `examples/AStrans.all` file. When non-transit nodes are allowed in the selection process, the cover is usually slightly smaller but the effectiveness of the filtering tends to decrease as well.

5.3 Running the Simulator

The dpf program has seven command-line arguments:

% `dpf/dpf` *graph trans cover egress max route sets trace*

graph: This variable is the name of the file that contains an input AS graph topology in adjacency list format. (In Section 5.2, this file is named ASgraph.*ID*, generated in Step 2.) Every line of the file includes an AS node and the list of its neighbors. An example line from an input *graph* file is as follows:[7]

```
22      ->      5       :668:7170:5855:5303:5881
```

[6]For information on the Route Views Project, see http://www.routeviews.org/. The BGP dump files can be obtained through http://archive.routeviews.org/.

[7]More examples of these files can be found as the ASconnlist.* files at the http://moat.nlanr.net/Routing/rawdata/ site.

In this example, "22" is the AS number of the node which this adjacency list refers to. The second entry on this line, "5", is the degree of AS 22. The five numbers that follow are the neighbors of this AS.

trans: This file lists the transit nodes in the network, one node per line. In Section 5.2, this file is named AStrans.*ID*, generated in Step 3. If the user wishes to include all nodes in the routing process without stub-transit categorization, he can use the AStrans.all file for *trans*, which is provided in the examples/ directory.

cover: This file lists the filter nodes in the network, one node per line. In Section 5.2, this file is named ASfilter.*ID*, generated in Step 4.

egress: This file lists the nodes that perform egress filtering. Typically, it can be taken as the same as the *cover* file.

max: This variable determines whether maximal or semi-maximal filtering is to be performed. A value of 1 shows the filtering to be performed is maximal, a 0 shows it is semi-maximal. A maximal route-based filter performs the filtering based on both the destination and the source addresses in a packet. (see [4, 3] for additional details.)

route: This variable determines the type of routing to be carried out. A 0 means that shortest-path routing will be used. A 1 or 2 means that pre-computed routes will be uploaded and used. If *route* = 1, the uploaded routes will be completed by running Dijkstra's algorithm for the source-destination pairs missing from the uploaded data. If *route* = 2, no completion is done and only the pre-computed routes are used. In both cases, a *graph*.rt file must be present in the directory of the *graph* file. This file must include the source-destination-hop triplets of `unsigned short` in binary for the pre-routed source-destination pairs. An example program that generates a route file of this format can be seen at the `DownloadRoutes` function in the `preroute.c` file.

sets: This variable determines whether the output will be the cardinality of the $S_{a,t}$ and $C_{s,t}$ sets or the sets themselves explicitly. If *sets* = 0, the output includes the set cardinalities only; if *sets* = 1, explicit sets are written out. In either case, the simulator outputs two files; *graph*.Sta and *graph*.Cts. With *sets* = 0, these files are binary files starting with the integer N, the number of nodes in the input graph, followed by an $N \times N$ matrix, where the (i, j)th entry of the matrix is $|S_{a,t}|, t = i, a = j$, in *graph*.Sta, and $|C_{s,t}|, t = i, s = j$, in *graph*.Cts. With *sets* = 1, these files are ASCII files listing the $S_{a,t}$ and $C_{s,t}$ sets, respectively, one set per line.

Performance analysis of the DPF protocol typically deals with the size of the $S_{a,t}$ and $C_{s,t}$ sets; hence outputting the cardinality of these sets usually suffices. The analysis tools provided with this package, described in Section 5.4, work with the set cardinalities and require that the dpf tool be run with *sets* = 0.

trace: This variable determines whether a trace file named *graph*.tr will be generated which contains the relative location[8] of the nodes that can launch spoofed attacks. A 1 shows the trace file will be generated; a 0 shows it will not.

5.4 Analysis of the Output

There are numerous ways to measure the effectiveness of the filtering protocol. We provide the tools for calculating four of the measures discussed in [4, 3]: Φ_1, Φ_2, Φ_3, and Ψ_1. The call structure for these functions is

```
% measure/function x graph
```

where *function* is one of `phi1, phi2, phi3, psi1`, x is the value of the function to be computed, and *graph* is the name of the graph file whose output is to be used. For example, to compute $\Phi_2(1)$ on the simulation output of *graph*, the command is

```
% measure/phi2 1 graph
```

and the command for computing $\Psi_1(5)$ is

```
% measure/psi1 5 graph
```

All of the analysis tools discussed here work on the binary cardinality files and require that the dpf tool be run with *sets* = 0. The `phi1` program works on the *graph*.Sta file and the `psi1` program works on the *graph*.Cts file. The `phi2` and `phi3` programs work on *graph*.Sat which contains the transpose of the cardinality matrix in the *graph*.Sta file. The *graph*.Sat file can be obtained by the `transpose` program in the `measure` directory, as

```
% measure/transpose graph.Sta graph.Sat
```

The `transpose` program transposes the cardinality matrix in the first file and writes the output to the second file.

6 Algorithms of the Simulator

In this section, we give a high level pseudo-code of the major functions in the simulator program and an asymptotic analysis of its run time. V denotes the set of nodes, E denotes the set of edges, and F denotes the set of filter nodes.

[8] The relative location of an attack node "a" according to the spoofed address "s" and the target node "t" is taken as the distance of "a" to the point of intersection between the paths (s, t) and (a, t).

The Main Function

The simulator program consists of the five stages discussed in Section 2. The structure of the main function is as follows:

main()

```
/* Initialization Stage */
GetInputGraph()
OpenOutputFiles()

/* Route Computation Stage */
if (route = 1) or (route = 2)
    UploadRoutes()
if (route = 0) or (route = 1)
    for all d ∈ V
        Dijkstra(d)

/* Filter Computation Stage */
for all s, d ∈ V
    TraverseRoute_SetFilters(s, d)

/* Set Computation & Output Stages */
for all t ∈ V
    ComputeSets(t)
    SaveStats(t)

return
```

Initialization Stage

The input graph is read and the nodes, edges, routing tables and filters are created. The node creation takes $O(|V|)$ time. Creation of edges and filters takes $O(|E|)$ time. Creation and initialization of the route tables takes $O(|V|^2)$ time. The overall runtime of the initialization stage is therefore $O(|V|^2)$.

Route Computation Stage

Pairwise shortest path routes are computed by Dijkstra's algorithm, where the algorithm is run starting at the destination nodes, as discussed in Section 2.

Our Dijkstra implementation achieves a very efficient realization of the priority queue structure by assuming a constant limit on the maximum length of the paths in the AS

graph, denoted by MAXHOP.[9] The priority queue is implemented as an array of linked lists, indexed by the distance, with range [0...MAXHOP+1]. The list at the ith entry of the array includes the nodes with distance i to the destination. Initially, the destination node d is located in list 0 and all other nodes are in list MAXHOP+1 (practically ∞). As the edges are "relaxed" by Dijkstra's algorithm, the nodes are moved into the appropriate lists.

The DECREASE_KEY priority queue routine in Dijkstra's algorithm [1] is realized by cutting the node whose key is to be decreased from its current list and appending it to its new list according to the decreased key value. This operation takes $O(1)$ time. Similarly, the EXTRACT_MIN routine is also realized in $O(1)$ time per node. For extracting the minimum-distance unfinished node, the program begins at list 0 and proceeds through the lists by picking up one node at a time till it reaches the end of list MAXHOP. This operation is completed in $O(|V|)$ time in total, and in $O(1)$ time on average per node.

In this implementation of Dijkstra's algorithm, initialization takes $O(|V|)$ time, relaxation of edges takes $O(|E|)$ time, and processing of the nodes, not including the edge relaxations, takes $O(|V|)$ time; giving a total runtime of $O(|V| + |E|)$. Running the algorithm for all source-destination pairs takes $O(|V|^2 + |V||E|)$ in total.

The Internet AS graphs are typically sparse and the average degree of the nodes tends to remain constant around 4.3 [2]. Hence $|E| \approx O(|V|)$ and the route computation time for the AS graphs is $O(|V|^2)$ time.

Filter Computation Stage

The filters are computed by calling the TraverseRoute_SetFilters function, given below, for all source-destination pairs. The runtime of this stage is $O(c \cdot |V|^2)$, for c denoting the average AS path length. The average path length in the AS graphs over the past few years is observed to be relatively stable around 3.7 [2], which may be taken as a constant factor in the asymptotic notation. Hence, the runtime of the filter computation stage is practically $O(|V|^2)$.

TraverseRoute_SetFilters(s, d)

for every hop x on the path from s to d
    if $x \in F$
        Update filter on x to allow source address s
return

[9] Currently, MAXHOP is set to 250 which is actually many times larger than the maximum distance in the January 2002 AS network, which is 12 hops [2].

To set the filter on x's link where it received a packet from s, the bit corresponding to s in the bit array representation of that filter is set to 1. Moreover, s is added to the dynamic table representation of the filter if it is not already there.

Set Computation Stage

The computation of the $S_{a,t}$ sets for a given target t is achieved by a call to the ComputeSets function, in which the ComputeSta function is called for every attacker a exactly once. The pseudo-codes for these two functions are as follows:

ComputeSets(t)

```
for all a ∈ V − {t}
    S[a, t] ← ∅
    Done[a] ← 0
S[t, t] ← V
Done[t] ← 1

/* The main set computation part */
for all a ∈ V
    if Done[a] = 0
        ComputeSta(t, a)

if egress filtering is on
    for all a ∈ F
        S[a, t] ← {a}
S[t, t] ← ∅
return
```

ComputeSta(t, a)

```
Done[a] ← 1
if a = t or there is no path from a to t
    return
b ← next hop from a to t
if Done[b] = 0
    ComputeSta(t, b)

/* The S[a,t] computation */
if b ∈ F
    S[a, t] ← S[b, t] ∩ Filter[b, a]
else
    S[a, t] ← S[b, t]

return
```

Regarding the runtime of these algorithms, the following can be said:

- One run of the ComputeSets function, not including the time spent in the ComputeSta calls, takes $O(|V|)$ time.
- During a call to ComputeSets, the ComputeSta function is called exactly $|V| - 1$ times.
- Not counting the time spent in the recursive call to ComputeSta, the runtime of one call of ComputeSta is dominated by the time spent in the set copying operation $S[a,t] \leftarrow S[b,t]$ or the set intersection operation $S[a,t] \leftarrow S[b,t] \cap Filter[b,a]$.
- The set copying operation takes time proportional to the size of the set copied. The intersection operation takes time proportional to the filter density (i.e., the number of allowed addresses in the filter).
- In general both the set size and the filter density are $O(|V|)$. Nevertheless, on the Internet AS graph with vertex cover filter placement, both tend to grow much less than linear in $|V|$.

Therefore, the overall set computation runtime is $O(|V|^3)$ and $\Omega(|V|^2)$. For the special case of the Internet AS graph with vertex cover filter placement, the runtime empirically tends to be much less than $\Theta(|V|^3)$ and actually closer to $\Theta(|V|^2)$.

Output Stage

The SaveStats function computes and saves the cardinalities of the $S_{a,t}$ and $C_{s,t}$ sets. The general structure of this function is as follows:

SaveStats(t)

for all $x \in V$
    $n\_St[x] \leftarrow 0$
    $n\_Ct[x] \leftarrow 0$

for all $a \in V$
    for all $s \in S[a,t]$
        $n\_St[a]{+}{+}$
        $n\_Ct[s]{+}{+}$

write n_St, n_Ct arrays
return

The runtime of this function for one target t is $O(c \cdot |V|)$, for c denoting the average size of the $S_{a,t}$ sets, and $O(c \cdot |V|^2)$ over all targets. In general, c is $O(|V|)$ and the overall runtime of this stage is $O(|V|^3)$. Nevertheless, for the Internet AS graphs between 1997-2002 with vertex cover filter placement, c remains quite stable in the interval (4.5, 4.8), giving a runtime of $O(|V|^2)$ for the output stage.

Total Runtime

In general, the overall runtime of the simulator is $O(|V|^3)$. For the Internet AS graphs with vertex cover filter placement, the runtime tends to be closer to $\Theta(V^2)$. We present the experimental timing results in Section 7.

6.1 Maximal Filtering

The default filtering performed by the simulator is semi-maximal—i.e., only source addresses are checked in route-based filtering—and the code discussed so far is for this type of filtering. If maximal filtering is in effect, the filters allow or drop packets with respect to their destination addresses as well as their source address. This effect is achieved in the simulator by making the filter computation destination-specific, as shown in the code below. The only change is in the main function; the rest remains the same.

```
main()
GetInputGraph()
OpenOutputFiles()
for all d ∈ V
    Dijkstra(d)
for all t ∈ V
    UnsetAllFilters()
    for all s ∈ V
        TraverseRoute_SetFilters(s, t)
    ComputeSets(t)
    SaveStats(t)
return
```

The only overhead in maximal filtering is invoking the UnsetAllFilters function for each node once. The number of filters in the network is $O(|E|)$. reseting a filter in general takes $O(|V|)$ time, giving a runtime of $O(|V||E|)$ for one t and $O(|V|^2|E|)$ overall. In Internet AS graphs, $|E| = O(|V|)$, hence the runtime becomes $O(|V|^3)$. In the case of a vertex cover filter placement, the average filter density tends to be approximately constant,

as discussed in the set computation stage above. In that case, the overall time spent in UnsetAllFilters is approximately $O(|V|^2)$.

7 System Requirements

7.1 Space Complexity

Table 1 shows the memory usage of the simulator for the Internet AS graphs of January 1998–2002. The results suggest an $O(|V|^2)$ increase in memory usage, dominated by the routing tables.

Year	1998	1999	2000	2001	2002
$\|V\|$	3213	4501	6582	8063	12517
Mem. (MBytes)	46	88	186	279	669

Table 1: The memory usage of the static DPF simulator for January 1998–2002 AS graphs.

7.2 Time Complexity

Table 2 shows the running time of the simulator on January 1998–2002 Internet AS graphs. The timings are taken on a 200 MHz Ultra-2 Sparc Station with 512 megabytes of physical memory. The results suggest a running time pattern slightly larger than $O(|V|^2)$.

Year	1998	1999	2000	2001	2002
$\|V\|$	3213	4501	6582	8063	12517
CPU Time (sec.)	97	201	449	720	1904
System Time (sec.)	1	3	6	12	59
Wall Time (sec.)	100	207	464	753	3623

Table 2: The running time of the static DPF simulator for January 1998–2002 AS graphs. All timing data are in seconds.

In the simulation experiments summarized in Table 2, about 2% of the simulation time is spent in the initialization stage, 14% in the route computation stage, 25% in the filter construction stage, 53% in the set computation stage, and 6% in the output stage. The large difference in the CPU time and the elapsed time in the 2002 data is due to the program's memory requirement exceeding the physical memory for the 2002 AS graph.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, 1990.

[2] Z. Ge, D. R. Figueiredo, S. Jaiwal, and L. Gao. On the hierarchical structure of the logical Internet graph. In *Proc. SPIE International Conference on Scalability and Traffic Control in IP Networks*, pages 208–223, 2001.

[3] Kihong Park and Heejo Lee. A proactive approach to distributed DoS attack prevention using route-based packet filtering. Technical Report CSD-TR 00-017, Department of Computer Science, Purdue University, December 2000.

[4] Kihong Park and Heejo Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proc. ACM SIGCOMM'01*, pages 15–26, August 2001.

A Multi-Path Routing

The routing scheme used by the dpf simulator is single-path—that is, at each node the next hop to forward a packet for a given destination is unique—which is also the type of routing BGP performs. Nevertheless, we provide an additional tool[10] that supports multi-path routing for those who are interested in experimenting with DPF in a multi-path routing environment where a node may have more than one alternative next hop for sending a packet to a destination.

In the rest of this section, we discuss the differences of the multi-path tool from the original single-path tool.

A.1 Data Structures

The only change in the data structures is in `RTABLE_ENTRY`, which is now a list of `EDGE_ID`s instead of a single `EDGE_ID`.

A.2 The Program Structure

The outline of the multi-path program is the same as its single-path counterpart, consisting of the five stages discussed in Section 2. The contents of these stages are quite similar

[10] Actually, single-path routing is just a special case of multi-path routing and it is possible to perform single path routing with the multi-path tool. However, this would result in an increase in the runtime and memory usage of the single-path simulations, which we chose to avoid by separating the two tools.

too. In particular, the input and output stages are just the same; whereas the route computation, filter computation, and set computation stages are slightly different.

The route computation stage in multi-path dpf is different in that the only routing option allowed is to upload and use pre-computed routes. This stage of the program is very similar to the single-path dpf run with the *route* = 2 option. The main difference between the two is that the multi-path version allows the uploaded routes to include more than one entry per source-destination pair.

The filter computation stage works very similarly to the single-path case as well: The path between every source-destination pair is traversed and the filters on that path are set to allow the given source address. The main difference from the single-path case is that the path traversal is done on multiple paths for each source-destination pair.

The set computation phase is a generalization of the single-path case: The set of spoofable addresses is computed for each next hop and then their union is taken. That is, if $H_{x,y}$ denotes the set of next hops from x for reaching the destination y, the $S_{a,t}$ sets are computed recursively as

$$S_{a,t} = \bigcup_{b \in H_{a,t}} S_{a,t}^{(b)}$$

where

$$S_{a,t}^{(b)} = \begin{cases} S_{b,t}, & \text{if } b \text{ is not a filter node} \\ F_{a,b} \cap S_{b,t}, & \text{if } b \text{ is a filter node} \end{cases}$$

for each $b \in H_{a,t}$.

A.3 Running the Simulator

The arguments of the dpf_mp program are identical to those of the original dpf program except for the *route* and *trace* variables. The *route* argument variable does not exist in dpf_mp since the only routing scheme supported is the uploading of pre-computed routes. As with the original dpf program with the route-uploading option specified, a *graph*.rt file must be present in the current directory which includes the source-destination-hop triplets of `unsigned short` in binary for the pre-routed source-destination pairs.

The command to run the tool from the home directory of the simulator is

```
% dpf/MP/dpf_mp graph trans cover egress max sets
```

where the argument variables *graph*, *trans*, *cover*, *egress*, *max*, *sets* are as explained in Section 5.3.

B Example Gnuplot Templates

In this section, we give two example gnuplot templates that can be used in analyzing the output of the DPF simulator.

B.1 Template for $\Phi_2(1)$

The following gnuplot template creates Figure 1 in file phi2.1.eps in EPS format.

```
set terminal postscript eps enhanced "Times-Roman" 20
set output 'phi2.1.eps'
set data style boxes
set nokey
set yrange [0:1]
set ytics 0,.1,1
set xtics 1997, 1, 2002
set xrange [1996:2003]
set ylabel '{/Symbol F}_2(1)'
set xlabel 'Year'
set boxwidth .4
plot 'phi2.1.dat'
```

The contents of an example phi2.1.dat file is as follows:

```
1997 0.969154
1998 0.969499
1999 0.967785
2000 0.974172
2001 0.972839
2002 0.979068
```

The output is shown in Figure 1.

B.2 Template for Ψ_1

The following gnuplot template creates Figure 2 in file psi1.eps in EPS format.

```
set terminal postscript eps enhanced "Times-Roman" 20
set output 'psi1.eps'
set data style linespoint
```

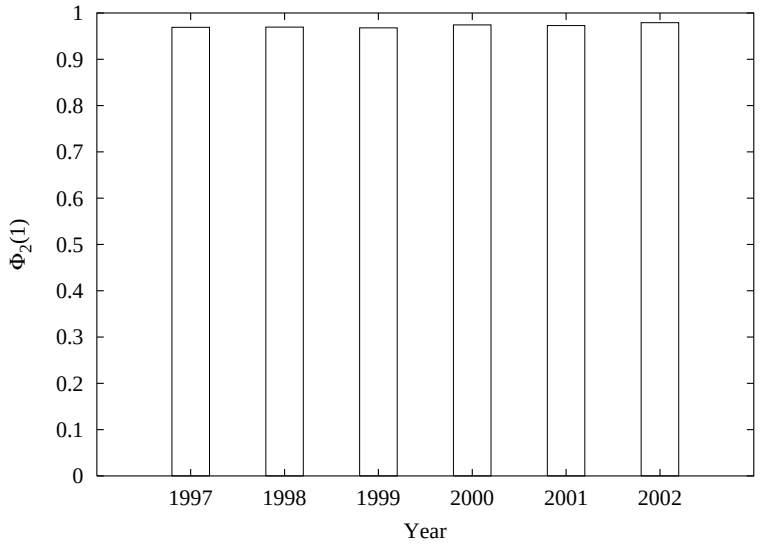

Figure 1: The graphics created by the gnuplot template for Φ_2.

```
set yrange [0:1]
set xrange [0:8]
set ytics 0,.2,1
set xlabel '{/Symbol t}'
set ylabel '{/Symbol Y}_1({/Symbol t})'
set key right bottom box
plot 'psi1.1997.dat' title '1997' \
, 'psi1.1998.dat' title '1998' \
, 'psi1.1999.dat' title '1999' \
, 'psi1.2000.dat' title '2000' \
, 'psi1.2001.dat' title '2001' \
, 'psi1.2002.dat' title '2002' \
```

The contents of a sample datafile, psi1.1998.dat, is shown below. The other data files must be in a similar format.

```
1  0.000000
2  0.791783
3  0.987862
4  0.999689
5  1.000000
6  1.000000
7  1.000000
```

```
 8  1.000000
 9  1.000000
10  1.000000
```

The output is shown in Figure 2.

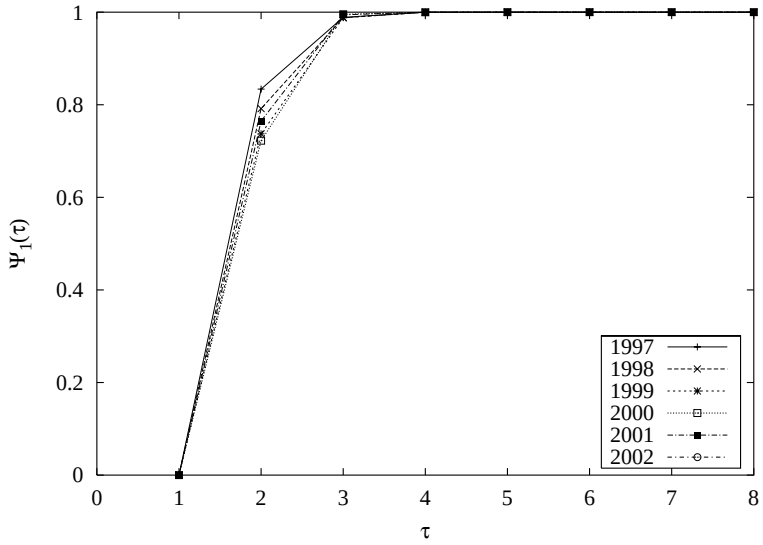

Figure 2: The graphics created by the gnuplot template for Ψ_1.