FILP: filtering packets in logarithmic time

Andrea Palazzo*

Department of Computer Science, University of Pisa, Pisa

E-mail: andrea.palazzo89@gmail.com

Abstract

During the first decades of their existence, computer networks were primarily used by university researchers for sending e-mail and by corporate employees for sharing printers. Under these conditions, security did not get a lot of attention. But now, as millions of people are using networks, network security is looming on the horizon as a potentially massive problem. There are several ways to guarantee the security in a LAN network. In the present work only one of the different typology has been taken into account, i.e. the **filtering packets**. Packet filter is a part of software which looks at the header of network packets as they pass through, and decides the life of the entire packet. It might decide to drop the packet (e.g. discard the packet as if it had never received it) or accept the packet (e.g. let the packet go through). Packet filters are managed by firewalls: a **firewall** is a system that controls the incoming and outgoing network traffic and it builds a bridge between the internal network or computer it protects, upon securing that the other network is secure and trusted, usually an external network that is not assumed to be secure and trusted. The firewall works as described hereinafter: first of all it takes a set of filter packets, then, for each packet that passes on the network, it takes one packet and controls if the packet 'satisfies' a filter (a packet 'satisfies' a filter if his information's header are equal to those of the filter). Finally the firewall decides if a packet can be removed from the network or sent to a specific receiver. Normally, the speed of the packets which go

*To whom correspondence should be addressed

through the network is quite high, so that the firewall have to control as rapidly as possible if the packets 'satisfies' a filter. For that reason, the control cannot be execute sequentially, but it must be done using efficient algorithms.

Introduction

FILP is a software used to manage filter packets logically (not sequentially). It allows the firewall (or a common computer) to organize all filter's packet using two types of data structure: ***AVL tree*** and ***Compact Trie***. These structures guarantee an **O(log n)** search time (where n is the filters number); in this case, these two structures have been combined (one *AVL*'s node could have more *AVLs* or *tries*) and search is used to check if the packet 'satisfies' a filter. However, the search of this structure guarantees an O(log n) search time, so it's useful for fast filtering.

Problems

In general, FILP views filters and packets like a tuple composed by 5 information:

1. Protocol (transport layer)
2. Destination IP address
3. Destination port
4. Sender IP address
5. Sender port

In a filter the information are often not specified (packets must specify all these information); for example, if a filter has only specified Protocol='abc' and Destination Port='123', FILP returns all packets that have Protocol and Destination IP address equals to these values (others information are ignored). Another problem is the use of **subnet mask** for IP addresses. Filters can have subnet masks for IP addresses (packets have no subnet masks). In this case, IP address is written as the

first address of a network, followed by a slash character (/), and ending with the bit-length of the prefix. For example, 192.168.1.0/24 specify a subnet where /24 notation indicates that the leftmost 24 bits (of the 32-bit quantity) define the subnet address.

Data Structure

FILP uses a data structure obtained combining *AVL* and *trie* to resolve all problems. In order to manage the subnet mask FILP uses the *trie*, while the management of all types of filter is carried out using the following main data structures:

- Protocol tree: all filters having protocol defined are inserted in this tree
- Destination port tree: all filters having protocol undefined but destination port defined are inserted in this tree
- Sender port tree: all filters having protocol and destination port undefined but sender port defined are inserted in this tree
- Destination IP tree: all filters having protocol, destination port and sender port undefined but destination IP address defined are inserted in this tree
- Sender IP tree: all filters having only sender IP address defined are inserted in this structure

For the sake of simplicity only the first main structure has been considered in the present document, since the other structures are similar (some structures are showed in the next figures). Protocol tree is an *AVL*, where nodes are ordered by the protocol. Here, every node has the following attributes:

- Data: it specifies the name of protocol filter's
- Others are none: it specifies if there is a filter that has only protocol defined (with the same value of Data) and others information undefined.
- Tree PD: it is an *AVL* ordered by destination port and in this tree are inserted filters having defined protocol (with the same value of Data) and destination port.

- Tree PS: it is an *AVL* ordered by sender port and in this tree are inserted filters having defined protocol (with the same value of Data) and sender port but destination port is undefined.
- Tree IPD: : it is a *trie* used to insert filters having defined protocol (with the same value of Data) and destination IP but sender port and destination port are undefined.
- Tree IPS: : it is a *trie* used to insert filters having defined protocol (with the same value of Data) and sender IP but sender port, destination port and destination IP are undefined.

For the sake of simplicity, only destination port tree has been take into account in this node because others trees have a similar behavior. The PD tree of this node has the same behavior of the second main data structure (where are inserted filters with protocol undefined). The destination port tree in this node is an *AVL* where nodes are ordered by the destination port and every node has following attributes:

- Data: it specifies the number of destination port filter
- Others are none: it specifies if there is a filter that has only destination port (and protocol) defined (with the same value of Data) and others information undefined
- Tree PS: it is an *AVL* ordered by sender port and in this tree are inserted filters having defined destination port (with the same value of Data) and sender port (and protocol).
- Tree IPD: : it is a *trie* used to insert filters having defined destination port (with the same value of Data) and destination IP (and protocol) but sender port is undefined.
- Tree IPS: : it is a *trie* used to insert filters having defined destination port (with the same value of Data) and sender IP (and protocol) but sender port and destination IP are undefined.

Obviously, sender port tree, destination IP tree and sender IP tree have the same structure and the same behavior (change only the order of nodes). FILP looking for a filter that is satisfied by the packet using this algorithm:

1. Search the packet's protocol into Protocol tree

2. If protocol hasn't been found (there isn't a node with data equals to packet's protocol) FILP continues the search in other main 4 data structures (Destination port tree, Sender port tree, Destination IP tree and Sender IP tree)

3. If protocol has been found, check if 'others are none'=1 and, if it's true, FILP specifies that the packet 'satisfies' a filter

4. If 'others are none'=0, FILP search (in the same way) into destination port tree node

5. If the search in this tree return a positive reply, FILP specifies that the packet 'satisfies' a filter; else FILP searches, in the same way, in other trees of the protocol tree's node

6. If the search in one of trees node return a positive reply, FILP specifies that the packet 'satisfies' a filter, else it searches in other main data structures

7. If the search returns a negative reply, in all main data structures, FILP specifies that the packet doesn't 'satisfy' any filter

The time complexity of search algorithm, in the worst case, depends on the filters number, indicated with n, and has the same protocol tree time complexity (because it's more complex than other main trees). Once the packet's protocol has been found in O(log n) time, there are 4 possible trees to be used for a further search; in this node the most complex tree is 'tree PD' and we consider this tree. After the packet's destination port has been found in O(log n) time, there are 3 possible trees be used for a further search; in this node the most complex tree is 'tree PS' and we taken this tree into account. Once the packet's sender port has been found in O(log n) time, there are 2 possible trees to be used for a further search; in this node the most complex tree is 'tree IPD' and we consider this tree. After the packet's destination IP address has been foundin O(log n) time, there is 1 possible tree to scan, that is 'tree IPS' (when the search requires O(log n) time). So complete algorithm time complexity is approximately: $T(n) \approx (4 \times 3 \times 2 \times 1)O(\log n) \approx O(\log n)$

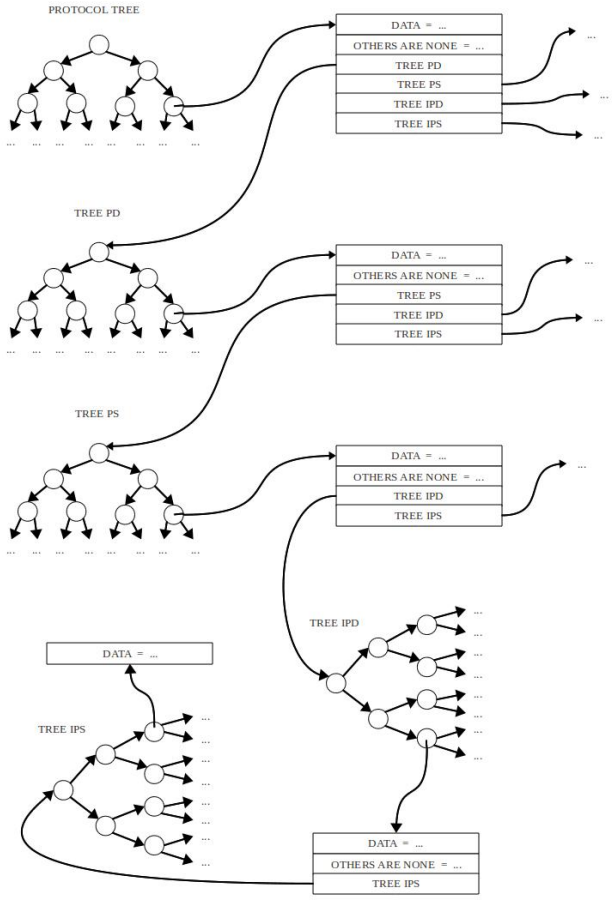

Figure 1: Protocol Tree, first main data structure

To Do

The work on the FILP is currently in progress but it still needs to be tested more.Below it can be found a list of improvements needed:

- Test on more architectures: currently it's tested on Linux/x86, so make this work on Windows and MAC
- Support more internet layer protocols, it supports only IPv4
- Support port's range for filters, for example the filter (null,null,'>30',null,null) considers all packets that have destination port > 30
- Need more testing for all possible situations

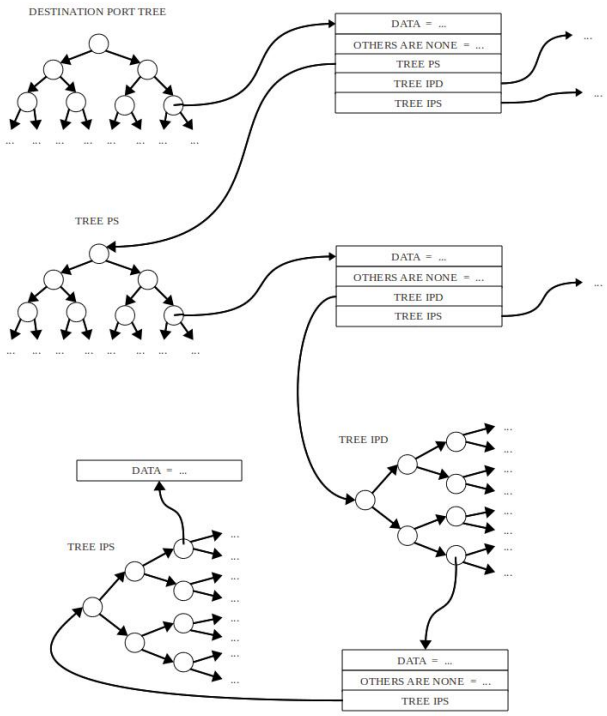

Figure 2: Destination Port Tree, second main data structure

How To Use

FILP is available from https://github.com/dimunix/filp. It requires Python (version 2.6.5) and superuser access, for packets capture. The Use of FILP requires downloading and extracting all files then executing the following command on shell:

```
~$ sudo ./filp.py filtersfile.txt optionIGMP
```

filtersfile.txt is a file composed by all user filters, to apply on FILP. In this file, every filter must be written in a row without spaces, as follows:

```
(UDP, 1.2.3.4,2, 5.6.0.0/16, 3)
(TCP, 10.9.8.0/21, 20, None, None)
```

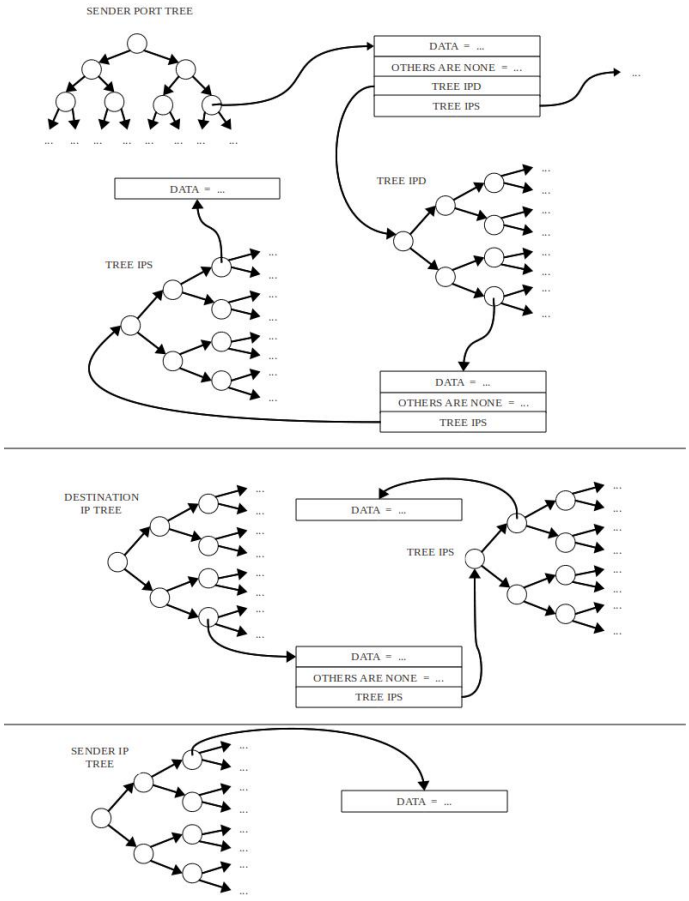

Figure 3: Others main data structure

. . .

In this file, protocols must be written in upper case and IP address must respect the standard dot-decimal notation, also with subnet mask (for example, 1.2.3.4 and 1.2.0.0/16). Moreover None is used to specify that an information is not specified in a filter. *optionIGMP* is an option setted by user, it's used to manage IGMP's packets (these packets haven't any port number, for more details see in references). This option has two possible values:

- 'yes': in this case FILP filters all IGMP's packets
- 'no': in this case FILP doesn't filter any IGMP's packets

After executing previous command, FILP starts taking packets from all network devices computer; stop filtering packets is possible with *CRTL-C* command. During execution, FILP prints, on the shell, if a packet 'satisfies' a filter using a string.

References

Informations about data structures used into FILP are available on Wiki:

- *AVL*: `http://en.wikipedia.org/wiki/AVL_tree`
- *trie*: `http://en.wikipedia.org/wiki/Radix_tree`
- *IP address*: `http://en.wikipedia.org/wiki/Ip_address`
- *subnet mask*: `http://it.wikipedia.org/wiki/Subnet_mask`
- *IGMP protocol*: `https://en.wikipedia.org/wiki/Internet_Group_Management_Protocol`

For further information or question about FILP please send me an email with a bug report.