# Plab: a packet capture and analysis architecture

Alberto Dainotti and Antonio Pescapé

## 1 Introduction

We designed and implemented a software for packet capture and analysis and to extract samples of what we identified as random variables. We called it PLAB. We tried to design a program flexible enough but that needed as few processing resources as possible, to be capable of analyzing traffic traces of hundreds of millions packets. In this chapter we illustrate the main architecture of the software, motivate some design decisions and describe some features that were introduced.

## 2 Libpcap

As regards packet capture, PLAB is based on the Libpcap library[JLM], which is an open source C library offering an interface for capturing link-layer frames over a wide range of system architectures. It provides a high-level common Application Programming Interface to the different packet capture frameworks of various operating systems. The offered abstraction layer allows programmers to rapidly develop highly portable applications. Moreover it defines a common standard format for files in which captured frames are stored, also known as *tcpdump* format, which is currently a *de facto* standard and is largely used in public network traffic archives.

Modern kernel-level capture frameworks on Unix operating systems are mostly based on the BSD (or Berkeley) Packet Filter (BPF) [MJ93] [Ste95]. The BPF is a software device that "taps" network interfaces, copying packets into kernel buffers and filtering out unwanted packets directly in interrupt context. Definitions of packets to be filtered can be written in a simple human readable format using boolean operators and be compiled in a pseudo-code to be passed to the BPF device driver by a system call. The pseudo-code is interpreted by the BPF Pseudo-Machine, a lightweight, high-performance,

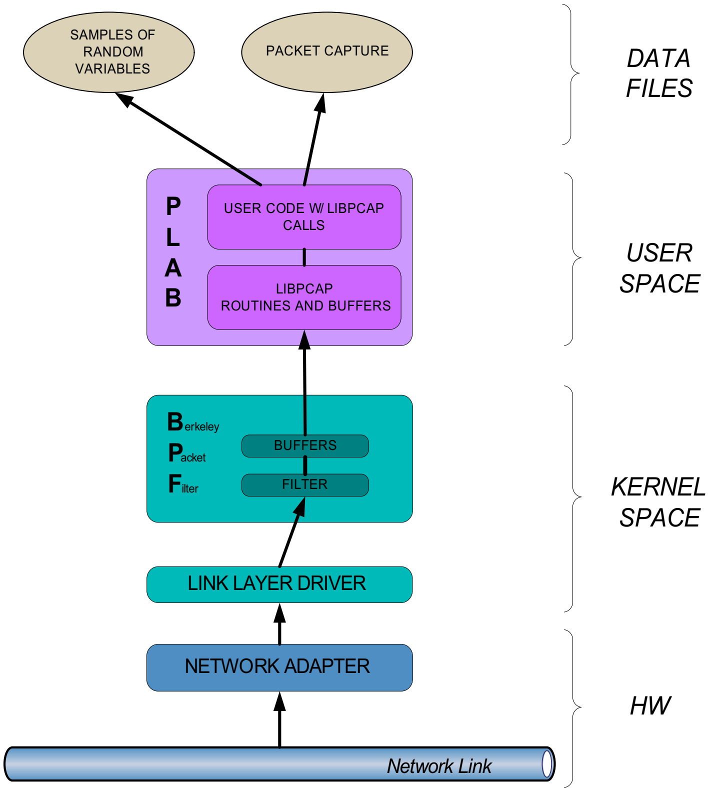


Figure 1: Flow of packet data from hardware to the application

state machine specifically designed for packet filtering. Libpcap allows programmers to write applications that transparently support a rich set of constructs to build detailed filtering expressions for most network protocols. By few Libpcap calls these boolean expressions can be read directly from user's commandline, compiled in pseudo-code and passed to the Berkeley Packet Filter. Figure 1 illustrates how PLAB, the Libpcap, and the BPF interact and how network packet data traverse several layers to finally be processed and transformed in capture files or in samples for statistical analysis.

Libpcap's feature of platform independence was practically useful in our work because we started our project on a FreeBSD platform at the UNINA site but only afterward we were given the opportunity to analyze traffic at the GECNR site, where a system running Linux had to be used. Indeed, while FreeBSD implements the original BPF, under Linux there is a similar framework called "Linux Socket Filtering" which is derived from the Berkeley Packet Filter but presents some distinct differences. Our software was capable of running under both systems without modifications.

Finally, Libpcap offers the possibility to read packets from files in *tcpdump* format rather than from network interfaces without modifications to the application's code but only with a different function call for source initialization. This allows to separate the process of packet capture from the packet analysis and to easily write a single application which can work both in realtime and offline conditions.

# 3 Architecture

In this section we will briefly illustrate how PLAB is organized. Figure 2 shows a simple, high-level, data flow diagram of the program, highlighting the main processing steps. After the interpretation of command line option and data initializations the Libpcap library is initialized for use by requesting a *pcap* descriptor. The invoked function is different if the source of packets is a network interface or a file of dumped packets. After that, the BPF filter is initialized with *pcap_set_filter()*, passing a pointer to a pseudo-compiled *bpf program*. From there, the main loop begins, which is executed for each packet that the library copies to the user space. The loop can be interrupted by setting a variable to zero. This happens when user sends a break on the terminal, when all packets from a dump file have been read or when a user-requested number of packets has been processed. Under error conditions a *cleanup*() function is called and the program exits.

Inside the main loop, a packet is first examined by the *filter_packet()* function which, based on user options and defaults, evaluates if the packet

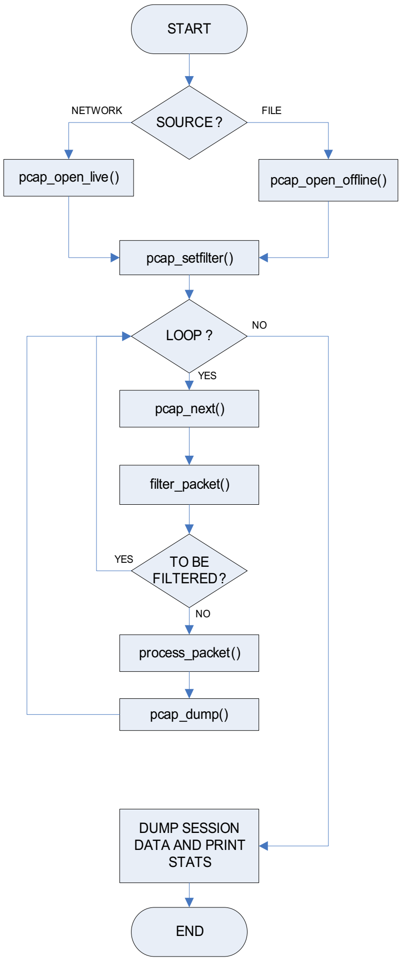


Figure 2: PLAB Data Flow Diagram

must be discarded or it can be further processed; there is a rich set of conditions that can influence the decision to discard a packet, we will see some of them in Section 6. This function also updates program statistics about filtered packets and calls subroutines that perform data extraction that must be done on some packets before discarding them. The *process_packet()* function is where all program's internal data structures are updated and where the calculations of traffic parameters that must be sampled are done. Also, inside this function, *micro-scale* samples are immediately dumped on data files because of memory constraints. Finally, the entire packet or part of it may be copied, if the corresponding option is activated by the user, in a capture file in *tcpdump* format. Outside of the main loop, when all packets have been processed, session data (*macro-scale*) are calculated and written in output files. A file of report is generated and few program statistics are printed on standard output.

# 4 Data structures

## 4.1 Hash tables

To keep track of traffic exchanged between each single couple of hosts there was the need of a data structure in which information for each different client-server pair could be dynamically stored. A client-server pair is obviously identified by the IP addresses of both hosts. An IPv4 address is constituted of 32 bits, so the maximum theoretical number of client-server couples that can be encountered on a link is $2^{64}$ , or $2^{63}$ if we suppose that two hosts never swap the client and server roles. Therefore, a static data structure, which would have yield the benefit of a $O(1)$ complexity for data insertion and search, was not feasible. Complexity for data lookup is very important in our software because an access is made for each single packet to evaluate inter packet times and to make other computations. Among dynamic data structures, balanced binary trees offer a worst-case complexity which is logarithmic with respect to the number of elements ($log_2(n)$) but we found that an implementation with hash tables could yield better results in all realistic situations. We used a *chained hash table* [Knuth98] [Morr98].

A hash table is basically made of a direct address table (a static array) which is addressed by an index obtained through an *hash function* applied to the original key of the element. Each position into the array is also named *slot*. A hash function performs a $m$ to $k$ mapping with $k < m$, $k$ equal to the length of the array, also called the hash table size, and $m$ equal to the number of values that elements' keys can assume ($2^{64}$ in our example). When multiple keys map onto the same integer we say that there is a *collision*. A hash function must be computationally fast and must be designed to reduce the number of collisions as much as possible by fairly exploiting all the slots in the table. In chained hash tables collisions are handled by chaining colliding elements into a linked list. This allows an unlimited number of collisions to be handled and does not require a priori knowledge of how many elements are contained in the collection. When the hash function is well-designed, and the number of slots is larger than the number of elements, collisions are rare and the average lookup and insertion time is $O(1)$. In our observations the number of different client-server pairs was of about 200 thousands after eight hours of traffic capture, allowing us to allocate a hash table with enough slots.

The condition to have an array length larger than the number of elements in the population might let think that there is a memory constraint which limits flexibility. But even when the length $k$ of the array is smaller than the number of elements, if the hash function is designed to generate indexes with uniform probability when applied to the population of the keys, then we can assume an average worst case complexity[1] of $O(n/k)$. This means that, for example, even for $n = 200$ millions of client-server pairs we could choose $k = 10$ millions, obtaining a complexity smaller than $log_2(2e + 08) \approx 27.5$ by allocating an array of pointers filling less than 40 MB of memory.

There are several strategies for maximizing the uniformity of the hash function and thereby maximizing the efficiency of the hash table [Gasch99]. One method, called the division method, operates by dividing a data item's key value by the total size of the hash table and using the remainder of the division as the hash function return value. Selecting an appropriate hash table size is an important factor in determining the efficiency of the division method. A good rule of thumb in selecting the hash table size for use with a division method hash function is to pick a prime number that is not close to any power of two.

In Figure 3 the simple hash function we used in our program is shown. The function has been written so that source and destination hosts' IP addresses can be swapped and still generate the same key.

## 4.2 Sessions

For each client-server pair it was necessary to keep track of the current session data and to update them whenever a new packet belonging to the same session was processed. Also, it was necessary to archive an expired session

[1] That is, the average length of the linked lists is n/k

and to allocate a new structure for a new opening session. We therefore associated to each item stored in the hash table a linked list of sessions structures. In other words, each element of the hash table, which represents a client-server pair, contains a pointer to a linked list of session structures, with the head associated to the current open session. Figure 4 shows a representation of how sessions are stored into the hash table, also the C structures that we implemented are shown.

The timestamps of last seen upstream and downstream packets are used to compute the inter packet time when a new packet is processed. The timestamp of the last seen packet, independently of flow direction, is used to check for session timeout and, together with the timestamp of session's first packet, allow us to compute the session duration ($T_{ON}$) and inter session time ($T_{OFF}$). There are four counters for the number of packets and bytes observed in both directions. Finally we have the client-server pair identifier, a session identifier, and a pointer to the previous expired session. We also store the *Maximum Segment Size* associated to a session, we will talk about it in Section 6.

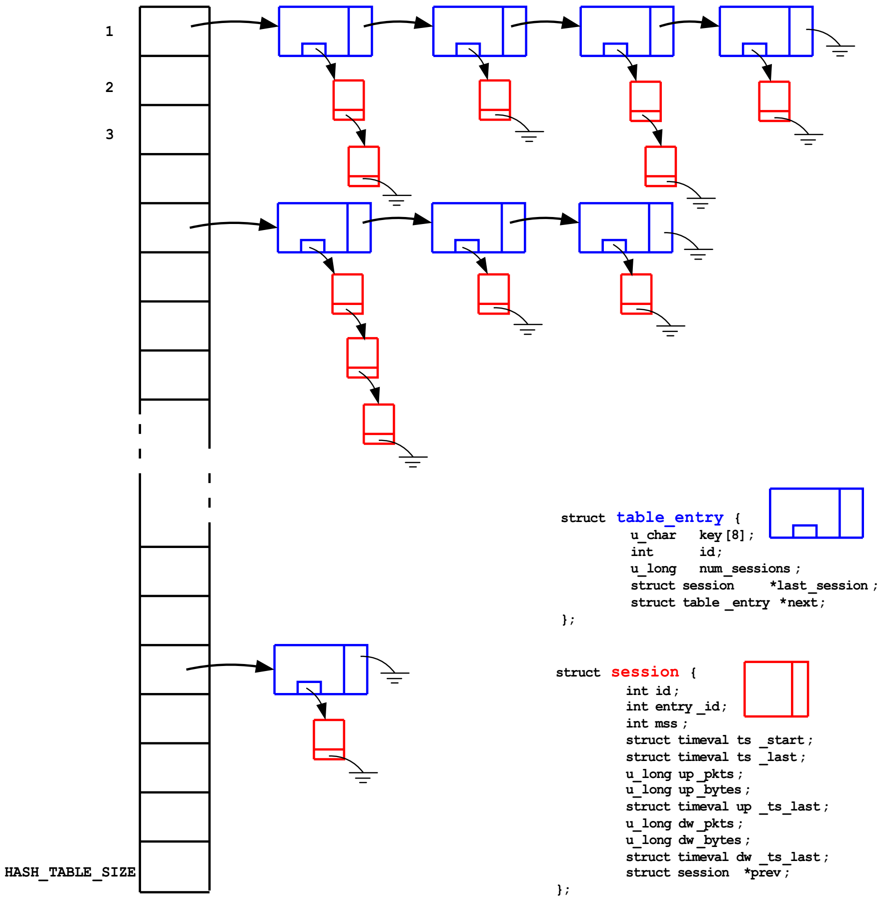
# 5 Packet manipulation

To analyze packets, it was necessary to read and interpret link-layer, IP and TCP protocol headers. Also, it was necessary to get timestamps of when packets were first seen on the interface. Those operations were easily accomplished because of the format of data returned by the Libpcap library for each captured packet. Each time *pcap_next()* returns with success it supplies a pointer to a *pcap_pkthdr* structure and a *u_char* pointer to a contiguous region of memory containing the captured portion of packet. The *pcap_pkthdr()* contains a timestamp that records the time in seconds and microseconds that

```
/* source ip */
for (i = 26, j = 0; i != 30; i++) {
        j = (j * 13) + packet[i];
}
/* dest ip */
for (i = 30, k = 0; i != 34; i++) {
        k = (k * 13) + packet[i];
}

PRINTDD("ht_hash: generated hash: %ld\n", (j + k) % HASH_TABLE_SIZE);
return ((j + k) % HASH_TABLE_SIZE);
```

Figure 3: Our simple hash function



```
struct table_entry {
        u_char   key [8] ;
        int      id;
        u_long   num_sessions ;
        struct session     *last_session ;
        struct table _entry *next;
};

struct session {
        int id ;
        int entry _id;
        int mss ;
        struct timeval ts _start ;
        struct timeval ts _last ;
        u_long up _pkts ;
        u_long up _bytes ;
        struct timeval up _ts_last ;
        u_long dw _pkts ;
        u_long dw _bytes ;
        struct timeval dw _ts_last ;
        struct session  *prev ;
};
```

Figure 4: Sessions stored into the chained hash table

the packet arrived on the interface, a *caplen* integer that records the length of the packet actually captured by the library, a *len* integer that stores the length of the packet as it appeared directly from the wire [Schif03]. The timestamp information is fundamental for the calculation of inter packet times and to evaluate sessions timeout, as explained in Section 4.2. The simple *u_char* pointer to packet data that is returned allowed us to treat the packet as an array of bytes, defining elementary macros to decode protocol fields. Figure 5 reports some snippets of the macros we implemented.

For example, in the *process_packet()* function, with these clean and fast operations we could get the size of TCP payload, one of our sampled parameters, independently from TCP headers variable size and by identifying TCP ports it was possible to distinguish between upstream and downstream packets. In several filtering sub-routines was also important to decode protocol fields, we will see it in Section 6. Also these macros were useful to easily print packets properties in debug contexts.

# 6 Filtering

In this section we briefly illustrate the packet filtering capabilities of PLAB.

## 6.1 Libpcap and BPF

We mentioned in Section 2 about how each program linked to the Libpcap library can easily compile filtering expressions to be passed to the Berkeley Packet Filter. The program can also interpret and use the same kind of expressions when packets are read from a capture file rather than from the network. Under PLAB it is possible to define a filtering expression from the command line or write it into a file to be read at run time. Figure 6 shows an example in which several conditions are imposed to accepted traffic: it catches only TCP packets in which one of the ports is set to 80 (line 1), where HTTP traffic is considered EGRESS as to the UNINA site (lines 2-8), and it explicitly filters out traffic related to the host *192.133.28.4*, which is the official Web cache at UNINA (last line).

## 6.2 PLAB filters

As regards the conditions to discard packets that are evaluated in the *packet_filter()* function and its called subroutines, these are of several kinds and most of them can be enabled or disable by the user with command line options.

```
#define PKT_ETH_LEN(a) ((a[12] << 8) + a[13])
#define PKT_IS_ETH_802(a) (PKT_ETH_LEN(a) <= 1500)

[...]

/*
 * Options
 */
/* pkt has no IP options */
#define         PKT_IS_NO_IP_OPTIONS(a) (a[14] == 0x45)
/* pkt has IP options */
#define         PKT_IS_IP_OPTIONS(a)            !PKT_IS_NO_IP_OPTIONS(a)
/* pkt has TCP options */
#define         PKT_IS_TCP_OPTIONS(a)           ((a[46] & 0xf0) != 0x50)

/*
 * Headers
 */
/* TCP header length in bytes */
#define         PKT_TCP_HLEN_B(a) ((a[46] & 0xf0) >> 2)
/* IP header length in bytes */
#define         PKT_IP_HLEN_B(a) ((a[14] & 0x0f) << 2)
/* IP total length in bytes */
#define         PKT_IP_TLEN_B(a) ((a[16] << 8) + a[17])

/*
 * Payloads
 */
/* IP payload - valid only for non-fragments */
#define PKT_IP_PAYLOAD_B(a) (PKT_IP_TLEN_B(a) - PKT_IP_HLEN_B(a))
/* TCP payload - valid only for non-fragments */
#define PKT_TCP_PAYLOAD_B(a) (PKT_IP_TLEN_B(a) - PKT_IP_HLEN_B(a) - PKT_TCP_HLEN_B(a))

/*
 * Ports
 */
/* TCP source port */
#define         PKT_TCP_SRC_PRT(a) ((a[34] << 8) + a[35])
/* TCP destination port */
#define         PKT_TCP_DST_PRT(a) ((a[36] << 8) + a[37])

/*
 * TCP Flags
 */
#define PKT_TCP_FLAG_FIN(a) (a[47] & 1)
#define PKT_TCP_FLAG_SYN(a) (a[47] & 2)
#define PKT_TCP_FLAG_RST(a) (a[47] & 4)
#define PKT_TCP_FLAG_PSH(a) (a[47] & 8)
#define PKT_TCP_FLAG_ACK(a) (a[47] & 16)
#define PKT_TCP_FLAG_URG(a) (a[47] & 32)

/*
 * HTTP
 */
#define PKT_IS_HTTP_UPSTREAM(a) (PKT_TCP_DST_PRT(a) == 80)
#define PKT_IS_HTTP_DOWNSTREAM(a) (PKT_TCP_SRC_PRT(a) == 80)
```

Figure 5: Some of the macros implemented to decode protocol fields

First of all we can skip the first $x$ captured packets if requested. This option can be useful when reading from a capture file and we know we want to start analyzing traffic only from a specific point. This option can be possibly used in conjunction with the option to stop analysis after that $y$ packets have been read.

Another option allows to specify a time range for packets' timestamps. For example we can capture packets from the network for one or several days and impose to analyze only traffic from 9:00 am to 01:00 pm. We used such kind of option for our analysis.

Other checks are fundamental for the correct execution of the program and can be disabled only together with packet processing, just in case we want to use PLAB only to capture and archive packets. Some checks, for example, verify packet integrity or discard IP fragments because, as said before, our program is not able to process them correctly. Also the IP checksum can be tested, but this is optional. A consistency check is made on the calculation of packet's TCP payload, to verify that it is in the allowed range (0-1460 for Ethernet-II frames).

It is possible to discard TCP packets without any payload bytes. Before this check, the *detect_mss()* subfunction is called for packets with the SYN flag set. We were interested in discovering the MSS requested by hosts that generated the observed traffic. This subroutine implements a simple state machine able to identify and decode the MSS optional TCP header.

# 7 Samples collection

After a packet has passed the filtering tests it is processed by the *process_packet()* function and its sub-routines. The first operation is to lookup into the hash table for a corresponding session associated to the packet. As said before, if an open session exists then it is placed at the head of the sessions' linked list for the specified client-server pair. So, we just need to identify into the hash table a corresponding item for the packet's source and

```
tcp and port 80 and not (
   (dst port 80 and dst net 143.225.0.0/16) or (src port 80 and src net 143.225.0.0/16)
or (dst port 80 and dst net 192.132.34.0/24) or (src port 80 and src net 192.132.34.0/24)
or (dst port 80 and dst net 192.55.101.0/24) or (src port 80 and src net 192.55.101.0/24)
or (dst port 80 and dst net 192.133.28.0/24) or (src port 80 and src net 192.133.28.0/24)
or (dst port 80 and dst net 192.135.165.0/24) or(src port 80 and src net 192.135.165.0/24)
or (dst port 80 and dst net 192.167.11.0/24) or (src port 80 and src net 192.167.11.0/24)
or (dst port 80 and dst net 192.167.33.0/24) or (src port 80 and src net 192.167.33.0/24)
) and not host 192.133.28.4
```

Figure 6: An example of a Libpcap filtering expression

destination IP combination. If there is no established session a new entry in the table and a session structure are allocated, otherwise the packet timestamp is subtracted to the timestamp of the last seen session's packet and compared to the user-defined session timeout. If a greater time has elapsed then the session is considered expired and a new session is created and initialized and put in the head of its linked list. The calculated time interval is stored as the inter session time ($T_{OFF}$). After these operations, the current session structure to work with is defined. The packet is identified as belonging to an upstream or downstream flow and the counters and timestamps cited in Section 4.2 are updated. The inter packet time is calculated if the packet was not the first one of its direction, the TCP payload is calculated and both values are dumped on the *micro-scale* samples files for the corresponding flow direction. Program's statistics are updated and control returns to the main loop.

*Macro-scale* sampled parameters are collected outside of the main loop, when all packets have been processed. A function designed to walk through all the hash-table and linked lists collects data about each session's packets and bytes counters and inter session times ($T_{OFF}$); it also evaluates session durations ($T_{ON}$) as the difference between the timestamps of session's first and last packet. These data is written into a samples collection file, ready to be opened by Matlab as the other samples files.

# 8 Capture files

PLAB can work with capture files in *tcpdump* format both as input and output files. We added functionalities to give flexibility to these options. An interesting feature was the ability to accept more than one file as input and read packets from them, in the same order given on command line, without interrupting traffic analysis. This obviously makes sense only if files represent contiguous parts of a single capture. Actually, because capture files can be very large in size, it can be easier to split a single captured traffic trace into several pieces to overcome space constraints. It is common, on public traffic archives, to find traces of several hours of traffic split into many files of just 15 minutes each.

While accepting capture files as input is useful to analyze archived traffic, writing packets on files in *tcpdump* format can be useful to exploit the filtering capabilities we introduced in PLAB. For the afore-mentioned advantages that can derive, we also added the ability for the user to dump packets into several files of approximately the same size.

We also added a specific optional feature to write to capture files only a portion of each packet. In traffic analysis there is often the concern to protect users' privacy. For this reason a commonly used technique, aside from scrambling the original IP addresses, is to capture only protocol headers without user payload, unless specifically motivated and authorized. Because when the kernel packet filter is initialized we can only set a fixed length of bytes to be captured for each frame, we can only be sure to not capture user payload only if we set the capture length equal to the minimum size of headers (which for Ethernet-II + IP + TCP is 14 + 20 + 20 bytes). This approach leads to the truncation of optional headers when they are present. Sometimes the concern is not about privacy but about disk space. Indeed, to be sure to capture also TCP optional headers we should almost double the capture length. We would then double the space used only to preserve headers in a minority of packets. In our traces, for example, we found that packets with optional TCP headers were approximately 20%. In PLAB we exploited its ability to decode TCP fields to dynamically change the number of bytes written for each packet to be dumped on file. A command line option lets the user set the number of bytes of TCP payload that must be written to disk, starting from zero. This option lets us satisfy privacy concerns without sacrificing legitimate information and allows a wiser usage of disk space.

# Acknowledgment

This work has been partially supported by the Italian Ministry for Education, University and Research (MIUR) in the framework of the the PRIN 2004 QUASAR project.

# References

[Gasch99] S. Gasch, *“Algorithm Archive - Data Searching and Storage”*. http://www.fearme.com/misc/alg/node7.html , 1999

[JLM] V. Jacobson, C. Leres, and S. McCanne, tcpdump, ftp://ftp.ee.lbl.gov/tcpdump.tar.Z

[Knuth98] D. E. Knuth, *“The Art of Computer Programming, Volume 3, Sorting and Searching”*. Addison-Wesley, Reading, Massachusetts. 1998

[MJ93] S. McCanne, V. Jacobson, *“The BSD packet filter: A new architecture for userlevel packet capture”*. Proceedings of the Winter 1993 USENIX Conference, pages 259–269. USENIX Association, January 1993

[Morr98] J. Morris, *“Data Structures and Algorithms”*. Electrical and Electronic Engineering, University of Western Australia, 1998 http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/

[p0f] Michal Zalewski, *“P0f v2. Passive OS fingerprinting tool”*. 2004

[RFC1191] J. Mogul, S. Deering, *“Path MTU Discovery”*. IETF RFC 1191, November 1990.

[RFC1323] V. Jacobson, R. Braden, D. Borman, *“TCP Extensions for High Performance”*. IETF RFC 1323, May 1992.

[RFC1945] T. Berners-Lee, R. Fielding, H. Frystyk, *“Hypertext Transfer Protocol – HTTP/1.0”*. IETF RFC 1495, May 1996.

[RFC2616] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *“Hypertext Transfer Protocol – HTTP/1.1”*. IETF RFC 2616, Jun 1999.

[Schif03] Mike D. Schiffman, *“Building Open Source Network Security Tools“*. Wiley 2003

[Ste94] W. Richard Stevens, *“TCP/IP Illustrated Vol. 1 The Protocols”*. Addison Wesley, 1994

[Ste95] W. Richard Stevens, Gary R. Wright, *“TCP/IP Illustrated Vol. 2 The Implementation”*. Addison Wesley