Libtrace: A Packet Capture and Analysis Library

Shane Alcock
University of Waikato
Hamilton, New Zealand
salcock@cs.waikato.ac.nz

Perry Lorier
University of Waikato
Hamilton, New Zealand
perry@cs.waikato.ac.nz

Richard Nelson
University of Waikato
Hamilton, New Zealand
richardn@cs.waikato.ac.nz

ABSTRACT

This paper introduces libtrace, an open-source software library for reading and writing network packet traces. Libtrace offers performance and usability enhancements compared to other libraries that are currently used. We describe the main features of libtrace and demonstrate how the libtrace programming API enables users to easily develop portable trace analysis tools without needing to consider the details of the capture format, file compression or intermediate protocol headers. We compare the performance of libtrace against other trace processing libraries to show that libtrace offers the best compromise between development effort and program run time. As a result, we conclude that libtrace is a valuable contribution to the passive measurement community that will aid the development of better and more reliable trace analysis and network monitoring tools.

Categories and Subject Descriptors: C.2.0 [Computer Communications Networks]: General

General Terms: Measurement, Performance

Keywords: Trace Analysis, Packet Capture, Protocol Decoding

1. INTRODUCTION

Network packet traces are a widely used source of Internet measurement data, as they provide a detailed and comprehensive record of the Internet traffic traversing a link. Packets are captured and stored exactly as they appeared on the network (although some truncation may be performed) and they cannot be easily processed and analysed using simple scripts or statistical languages. Even basic research tasks such as counting the amount of traffic observed on a given TCP port can require complicated code to decode the packet headers and extract useful values from the raw binary data.

As a result, researchers typically utilise a software library that can decode captured packets and process them in an abstracted fashion to analyse packet traces. The most widely used of these libraries is libpcap [1], which can be used to read packet traces captured using the pcap format (such as captures taken with the tcpdump tool).

However, libpcap has several weaknesses. Firstly, there is no support for capture formats other than pcap, making conversion necessary to analyse any traces captured using a different format. This can lead to the loss of important information and a reduction in timestamp resolution. Secondly, libpcap has no support for decoding the packet contents. Instead, the user must write their own decoding functions. This is often an error-prone and time consuming process, even for experienced programmers, due to the edge-cases and unexpected behaviour that can be observed in real Internet traffic. Finally, most trace files are stored compressed due to their size but libpcap does not include any native support for reading or writing compressed files. Instead, any input or output must be piped through the gzip tool but the pipe can often act as a bottleneck, reducing the performance of the analysis program.

In response, we have developed libtrace, an open-source software library for reading, processing and writing packet traces that addresses the weaknesses of libpcap and other trace processing libraries. Libtrace has been written using the C programming language and has been actively developed and used for over five years.

The principal features of libtrace can be summarised as follows:

API: The libtrace programming API has been designed to be streamlined, consistent and comprehensive. The specifics of compression, capture formats and protocol headers are handled by the library rather than the programmer, making development of analysis tools both faster and simpler. A libtrace program typically requires 40% fewer lines of code than an equivalent libpcap program.

Capture format agnostic: A libtrace program can read from and write to any supported capture format without code changes, making format conversion unnecessary. Libtrace supports most common packet trace formats and can also read from and write to live capture interfaces and Endace DAG cards [2]. Internally, each capture format is a separate module so adding support for new capture formats is trivial.

Protocol decoding: The libtrace API enables users to directly access the protocol header for any layer at or below the transport layer, automatically decoding and skipping any intermediate headers. Libtrace includes full support for protocols that older tools and libraries may not support, such as IPv6, VLAN, MPLS and PPPoE headers. Furthermore, edge-cases such as IP fragments, incomplete headers or tunnelling are handled correctly by libtrace rather than relying on the user to detect and handle them.

Compression: Libtrace supports reading and writing compressed trace files using both the gzip and bzip2 formats. Compression and decompression are performed using a separate thread, providing additional parallelism. For analyses where I/O is the limiting factor, this offers improved performance over piping data through a gzip process.

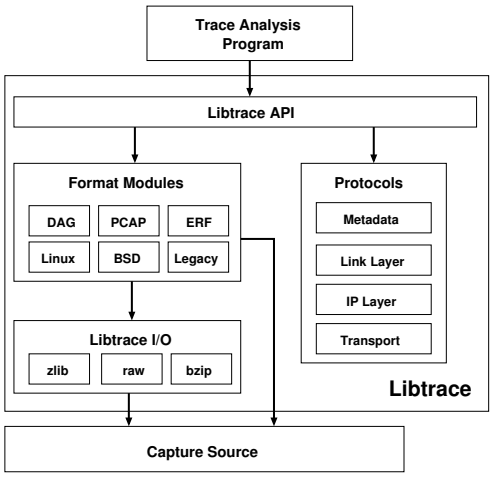

Figure 1: The architecture of libtrace.

Performance: In addition to performance improvements from the threaded I/O, libtrace is optimised to avoid copying packets in memory wherever possible and to cache packet properties to avoid decoding the same packet header multiple times. In practice, libtrace is notably faster than other protocol header decoding libraries.

Libtrace is not the first trace processing library that attempts to address the weakness of libpcap; previous efforts include libcoral [3], libnetdude [4] and scapy [5]; but we shall demonstrate that libtrace offers a better compromise between the effort required to develop a working analysis program and the subsequent performance of that program.

2. DESIGN AND ARCHITECTURE

Over the past five years, libtrace has evolved primarily to support the research activities conducted by the University of Waikato, with an emphasis on both ease of development (especially for student projects) and performance (as we work with large trace sets). A high-level view of the current architecture of the libtrace library is depicted in Figure 1. The arrows describe the communication between the user program, library components and the capture source. The libtrace library consists of 12,842 lines of C and C++ code and comes bundled with a set of tools that perform common trace manipulation and analysis tasks including splitting, anonymising and merging trace files.

In the following subsections, we shall describe each of the library components in turn and discuss the reasoning behind their current implementation.

2.1 Capture Formats

Many capture formats exist and each differs slightly from the others. The de-facto standard is the pcap format, primarily due to the ubiquity of both the libpcap library (for writing analysis software) and the tcpdump tool (for capturing and examining trace files). Another common format is the ERF format employed by Endace high-performance capture hardware [2], which we use to capture many of the trace files we use for our research.

Traces using formats other than pcap must be converted to pcap before they can be analysed using libpcap or tcpdump and this can have unintended consequences. For instance, converting an ERF trace to pcap loses timestamp precision (the pcap timestamp format only has microsecond precision whereas ERF timestamps have nanosecond resolution [6]), the loss counter and the interface that the packet was captured on, which often describes the packet direction. While direction can sometimes be inferred from MAC or IP addresses within the recorded packet itself, this requires that the addresses have not been modified or anonymised subsequent to the original capture.

Format	Input	Output
Pcap file	✓	✓
Pcap interface	✓	✓
ERF file	✓	✓
DAG capture hardware	✓	✓
Native Linux interface	✓	✓
Native BSD interface	✓	✗
TSH / FR+	✓	✗
Legacy DAG files	✓	✗
ATM cell header file	✓	✗
RT network protocol	✓	✗

Table 1: Trace formats supported by libtrace.

Therefore, we have designed libtrace to support multiple trace formats, enabling the user to analyse a trace in its native format and removing the need for potentially lossy format conversions. Each supported capture format is implemented as a separate module within libtrace. The module provides all of the capture format-dependent functionality, such as opening capture sources, reading packet records and accessing information stored within the capture header such as timestamps, wire lengths and loss counters.

The libtrace API provides format-neutral functions that developers can call within their program to act upon a capture source or a packet record. When the API function is called, the matching function for the capture format associated with the trace or packet parameter is then invoked. For example, if `trace_read_packet` is called on a pcap trace file, libtrace will call the 'read packet' function provided by the pcapfile format module and return the result. This is entirely transparent to the programmer; they simply call the libtrace API function and the library handles the rest.

Because the libtrace API is entirely format-neutral, a libtrace program is "capture format agnostic", i.e. the trace format and location can be specified as a command-line argument and adjusted by the user as needed, but the same code will read and write any of the supported trace formats without modification and recompilation. Libtrace can automatically detect the format for many input sources, so specifying the capture format is often unnecessary.

One benefit of this approach is that a program can be developed and tested against off-line trace data before being deployed (without any code changes) on a live capture device, such as a DAG card. Libtrace will also implicitly convert packets from one format to another if the user selects a different format when writing packets to an output trace. The conversion is performed internally by libtrace and converts between formats as faithfully as possible.

Table 2.1 lists the trace formats that are supported by libtrace. Output is not supported for many of these formats because the formats are either obsolete (in the case of the

legacy DAG formats) or have not been requested by users. The structure of libtrace enables programmers to easily add support for new trace formats and capture devices to future versions of the library without having to change the libtrace API. Once the code for the new format is added (typically as a separate source file), the only other necessary changes are to ensure the format constructor is called by the `trace_init` function inside libtrace and to update the build system to compile and link the new module into libtrace.

2.2 File Compression and Libtrace I/O

Packet trace files used in Internet measurement research are often large and are usually compressed to reduce storage requirements. The typical approach for reading compressed trace files is to use a separate process to decompress the trace, piping the decompressed packets into the analysis program via standard input. This allows the decompression to occur in parallel with the analysis, improving performance on multi-core systems. However, the size of the pipe buffer (64 KB in Linux) limits the rate at which data can be passed between the decompression and analysis processes.

Instead, I/O in libtrace is implemented using separate threads rather than separate processes. For example, when decompressing a trace, the I/O thread writes the decompressed data into one of fifty 1 MB buffers. Once a buffer is full, a signal is sent to the main thread to indicate there is data available for processing. The main thread reads the packets from the filled buffer as they are requested by the analysis, while the I/O thread moves on to filling the next buffer. When the main thread has emptied the buffer it is reading from, it sends a signal to the decompression thread to indicate that the buffer is available for writing again.

Libtrace supports three compression formats: raw (i.e. uncompressed files), zlib and bzip2. When a trace file is first opened for reading by a capture format module, the compression format that was used to create the file is determined by examining the magic number at the start of the file and the appropriate I/O module is then associated with the file. Writing compressed trace files is similar, except that the compression format must be specified by the caller when the file is opened.

2.3 Protocol Decoding

When using libpcap, the programmer must develop code to decode the protocol headers and find the data that they are interested in. For example, an analysis of TCP source and destination ports must include code to search for the TCP header, starting from the beginning of the packet and skipping over any other headers which may be present. Because of the extra development effort required, tools written using libpcap are seldom capable of parsing header combinations that were not encountered by the author, limiting their usefulness to a wider audience.

Another problem that arises when decoding protocol headers is the variety of edge-cases that occur when examining real Internet traffic. If not detected and handled correctly, these can have a major (and often undetectable) effect on the analysis results. For example, we commonly encounter incomplete packet headers in trace files, where a captured packet is truncated mid-way through a header. If care is not taken, it can be easy to read data from beyond the end of the capture record. This may lead to program crashes or invalid data being incorrectly treated as packet contents. Similar problems can arise with IP fragmentation, as novice programmers often assume that the bytes following an IPv4 header will always be a transport header.

The libtrace API provides functions which allow direct access to the packet headers at the metadata, link, IP and transport layers. These functions will implicitly decode and skip any preceding headers. For example, the `trace_get_tcp` function will find and return the TCP header for a packet (or NULL if no TCP header is present), without the user having to concern themselves with the rest of the packet. The Libtrace protocol decoders also detect and handle truncated headers and IP fragments appropriately.

Libtrace recognises many protocol headers, including (but not limited to) Ethernet, 802.11, VLAN, MPLS, PPPoE, IPv4, IPv6, TCP, UDP and ICMP. Libtrace can also decode meta-data headers, such as RadioTap, Linux SLL and Prism. New libtrace releases often add support for new headers and link types. Libtrace programs linked against new versions of the library will automatically be able to decode the new headers without code changes.

2.4 API

The libtrace programming API is the means by which developers access the trace processing capabilities of libtrace and is the only component of libtrace which they interact with directly. The C program shown in Listing 1 is a libtrace program that reads packets from a capture source and extracts the standard flow 5-tuple fields (IP addresses, ports and the transport protocol) and the IP total length. The main function opens the capture and reads each packet in turn using `trace_read_packet`. For brevity, we have omitted from this example much of the error checking that should normally be performed when opening and reading the capture. Each packet is passed to the `per_packet` function which finds the IPv4 header (skipping any packets that are not IPv4) and extracts the desired information from the packet.

As we are only interested in data from the IP header onwards, we use `trace_get_ip` to skip directly to the IPv4 header. Libtrace handles any preceding link layer headers, so our only responsibility is to check that a valid IPv4 header has been returned. The IP addresses and ports are retrieved directly using the appropriate libtrace API functions. The IP length and the protocol are determined by reading the values from fields in the IP header returned by the call to `trace_get_ip`. If further analysis at the transport layer is required, `trace_get_transport` can be used to retrieve a pointer to the transport header.

Because the intricacies of capture formats and compression libraries are handled internally within libtrace itself, this program will cope equally well with any trace format supported by libtrace. Similarly, the differences between link layer protocols are dealt with transparently by the call to `trace_get_ip`, so the properties of the network that the input trace was captured from does not matter. By using libtrace, this is all achieved using only 41 lines of code.

The libtrace API also allows for asynchronous reading from input sources using the `trace_event` function, meaning that the main program will not block while waiting for a packet to arrive. Instead, the libtrace API will return either a time to wait or a file descriptor to wait on if no packets are available. This is particularly useful for developing interactive or GUI-based measurement applications.

In addition, there are Ruby bindings for many of the libtrace API functions that can be used to quickly script prototypes of trace analysis tools which use all the features of libtrace. However, trace processing with Ruby takes significantly longer than an equivalent C libtrace program (as shown in Section 3.1), so Ruby is not suitable for large or complicated analysis tasks.

2.5 Performance

The trace data sets that we deal with in our research activities are typically large and can take multiple days to process. For example, the Waikato I trace set in the WITS archive is 1.3 TB compressed and spans 620 days of capture [7]. When dealing with such large datasets, even minor performance enhancements can save hours of processing time. As a result, libtrace incorporates a number of techniques designed to maximise the performance of the library.

Firstly, as mentioned earlier, all I/O operations in libtrace are conducted using a separate thread. This allows any compression and decompression operations to be off-loaded onto another CPU core, without the bottleneck that is created by piping to and from separate gzip processes. Any analysis that is I/O-bound, i.e. where the I/O operations require more processing power than the analysis itself, will benefit most from this approach.

When reading from live capture devices, libtrace utilises zero-copy behaviour to improve performance. For instance, instead of copying packet data from the DAG memory buffer into a local buffer, libtrace operates directly on a pointer to the memory in the DAG buffer. As most trace processing tasks deal with each packet in sequence and then effectively discards them, this approach saves an expensive memory copy for each captured packet. If a packet does need to be retained, the libtrace API provides a function that will copy a packet into a buffer allocated by libtrace so that it will not be inadvertently overwritten by the capture device.

Finally, libtrace caches the location of headers and the values for length fields for each packet, saving time on subsequent look-ups. For example, a call to `trace_get_layer3` will result in the location of the IP header being cached. If `trace_get_transport` is then called on the same packet, libtrace can resume from the cached layer 3 header rather than searching from the beginning of the packet again.

3. EVALUATION

In this section, we evaluate the impact of the design decisions made during the development of libtrace in terms of both ease of development and performance. We have compared libtrace against four other trace processing libraries: libpcap, libcoral, libnetdude and scapy, as well as the Ruby libtrace bindings. The feature set of each library is briefly summarised in Table 2.

Libcoral [3], a C library included in the CoralReef software suite, bears the closest resemblance to libtrace. Libcoral supports reading from multiple trace formats (including pcap and ERF) and has native support for compressed pcap traces, although this is implemented using a forked gzip process rather than the threaded approach employed by libtrace. Libcoral has API support for writing trace files using both the native CRL format and pcap, although each format requires different API functions. CoralReef includes a tool that is capable of writing ERF traces, but the conversion is not part of the software library.

Listing 1: An example libtrace program.

```
1  #include <libtrace.h>
2
3  void per_packet(libtrace_packet_t *pkt) {
4          uint16_t src_port, dst_port, ip_len;
5          struct sockaddr_storage src_stor, dst_stor;
6          struct sockaddr *src_ip, *dst_ip;
7          uint8_t proto;
8          libtrace_ip_t *ip;
9
10         ip = trace_get_ip(pkt);
11         if (ip == NULL)
12                 return;
13
14         src_ip = trace_get_source_address(pkt,
15                         (struct sockaddr *)&src_stor);
16         dst_ip = trace_get_destination_address(pkt,
17                         (struct sockaddr *)&dst_stor);
18         src_port = trace_get_source_port(pkt);
19         dst_port = trace_get_destination_port(pkt);
20
21         ip_len = ntohs(ip->ip_len);
22         proto = ip->ip_p;
23
24         /* Analysis code goes here... */
25 }
26
27 int main(int argc, char *argv[]) {
28         libtrace_t *trace;
29         libtrace_packet_t *packet;
30
31         packet = trace_create_packet();
32         trace = trace_create(argv[1]);
33         if (trace_start(trace)) return 1;
34
35         while (trace_read_packet(trace,packet)>0) {
36                 per_packet(packet);
37         }
38         trace_destroy(trace);
39         trace_destroy_packet(packet);
40         return 0;
41 }
```

Libnetdude [4] is also written in C and is designed to perform the underlying trace inspection and analysis for the Netdude GUI. The library has been released separately to allow users to develop their own trace analysis tools. Libnetdude supports reading from uncompressed pcap trace files only and cannot read from stdin or a named pipe, so the entire trace must be decompressed prior to opening it with libnetdude. Protocol decoding is done using plugins that describe how to decode each supported protocol header, enabling the library to be easily extended to accommodate new protocols. Libnetdude does not include a plugin for decoding IPv6 by default.

Scapy [5] is a Python module that can be used to decode and analyse packets captured using the pcap format. Users can use the Python language to quickly develop packet analysis scripts without having to deal with the memory management and strong typing that is required in C programming. Like libnetdude, scapy only supports the pcap format.

For the evaluation, we selected two analysis tasks and implemented programs to perform the tasks using each evaluated library. The first task, which we refer to as "Ports", was to count both packets and bytes observed on each TCP and UDP port. The byte count will be based on application level bytes, i.e. from the end of the transport header. The second task, "Scans", was designed to replicate the data gathering performed by the Bro IDS [8] for [9]. This analysis required state to be maintained for each TCP flow observed in the traces. To do this, we implemented a separate flow maintenance library using 472 lines of C code. Each evaluation program extracted the necessary information from each TCP packet before passing it into the flow maintainer.

In developing the evaluation programs, we were careful to ensure that all programs for a given analysis task produced matching results and dealt with truncated packets, IP frag-

	Libtrace	Libpcap	Libcoral	Libnetdude	Scapy
First Released	2004	1994	1999	2003	2005
Language	C	C	C	C	Python
Version Evaluated	3.0.12	1.1.1	3.8.6	0.12	2.1.0
Released	Sep 2011	Apr 2010	Jun 2009	Mar 2010	Dec 2009
Supports Pcap	✓	✓	✓	✓	✓
Other Formats	✓	✗	✓	✗	✗
Reads Stdin	✓	✓	✓	✗	✓
Native Gzip	✓	✗	✓	✗	✓
Decodes IPv4	✓	✗	✓	✓	✓
Decodes IPv6	✓	✗	✓	✗	✓
Decodes VLANs	✓	✗	✓	✓	✓

Table 2: A summary of the evaluated trace analysis libraries.

Name	Format	Duration	Packets	TCP Flows
Waikato	ERF	7 days	1,122 M	76.5 M
ISP	ERF	1 day	1,860 M	67 M
MAWI	Pcap	4 hours	523 M	23 M

Table 3: Trace sets used for the evaluation experiments.

	Libtrace	Libpcap	Libcoral
Ports	158	261	220
Scans	125	204	167

	Libnetdude	Scapy	Ruby
Ports	221	96	120
Scans	161	97	99

Table 4: Lines of code required for each evaluation program.

ments and other edge-cases in the same fashion. Every effort was also made to use the same coding style throughout, so the lines of code would not be inflated for some programs compared to others. We have made the source for the evaluation programs publicly available for reader evaluation [1].

The traces used for the evaluation are summarised in Table 3. The Waikato traces came from the Waikato V trace set and the ISP traces came from the Local ISP C-I trace set, which are both part of the WITS traffic archive [7]. The MAWI traces came from the MAWI traffic archive (the 2010 DITL set) [10]. The evaluation traces are only a subset of the data available in their respective trace sets to ensure that we could complete the evaluation in a timely manner. All trace files were compressed using gzip.

Each of the trace sets presented challenges beyond simply reading conventional Ethernet/IPv4 traffic. All three trace sets contained IPv6 traffic tunnelled over IPv4. Also, the ISP and MAWI traces contained native IPv6 traffic and all of the ISP traffic was VLAN-tagged. The evaluation programs had to be able to parse all of those headers correctly. Finally, the ISP traces contained wireless traffic, necessitating careful error checking due to the increased likelihood of packet corruption.

As some of the evaluated libraries did not support either compressed files or trace formats other than pcap, we required a standard method to convert the traces into a suitable format. We selected the `zcat` and `dagconvert` tools for the decompression and format conversion respectively and piped the output of the tools directly into the evaluation programs via standard input. Note that the tools were only used if needed; neither `zcat` nor `dagconvert` were used when running the libtrace evaluation program, for example.

Reading the ERF traces with libnetdude was particularly problematic: the libnetdude evaluation programs could not read input from standard input or a named pipe without crashing [2]. Instead, we decompressed (and converted to pcap, if necessary) the input traces and wrote them to disk, producing a trace file that libnetdude could read successfully. This approach was only feasible because we were only using short trace sets: doing this with the entire Waikato I trace set would have required over 4 TB of free disk space.

3.1 Results

Table 4 shows the number of lines of code (LOC) required to develop each of the evaluation programs, as measured by `sloccount` [11]. As expected, the scripted programs needed the least development effort, with scapy requiring less than 100 LOC for each analysis task. The libtrace programs used the least LOC of all the C programs, needing 40% less lines of code than an equivalent libpcap program. The libpcap programs required the most development effort, due to the need to develop code to parse each protocol header encountered in the traces. Libcoral and libnetdude were similar in terms of lines of code, although an additional 161 lines of C code were required to add IPv6 support to libnetdude.

Table 5 shows the amount of time taken to run each evaluation program against the three trace sets, measured using the Linux `time` tool. We also ran the libtrace programs with threading disabled (using `zcat` and pipes to decompress the traces) to demonstrate the effect of the threaded I/O. All evaluation tests were run on a lightly loaded Debian Linux server. Tests with a duration less than an hour were run multiple times and the fastest time reported. We have not included any results for scapy because the scapy Ports program took over 52 hours to process a single 24 hour Waikato trace. As a result, it was not feasible to run all of the tests and we concluded that scapy was therefore unsuitable for long duration trace analysis. Ruby libtrace performed better but was still much slower than any of the C libraries.

The libtrace program completed the Ports analysis fastest for all three trace sets, principally as a result of the threaded I/O. This is because the Ports analysis requires relatively little processing power; the major limitation is the speed

[1] `http://wand.net.nz/trac/libtrace/wiki/PaperEval`

[2] This also occurred with an example program included with libnetdude, so this was not an error on our part.

	Waikato	ISP	MAWI
Libtrace	00:08:51	00:20:01	00:04:21
Libtrace (no threads)	00:12:15	00:25:54	00:06:12
Libpcap	00:12:09	00:25:45	00:06:12
Libcoral	00:12:12	00:29:50	00:04:24
Libnetdude	00:30:43	00:49:19	00:14:41
Ruby	03:53:39	06:00:44	01:43:45

(a) Ports Analysis.

	Waikato	ISP	MAWI
Libtrace	00:19:39	00:32:49	00:09:09
Libtrace (no threads)	00:22:45	00:35:05	00:09:52
Libpcap	00:19:15	00:31:07	00:08:35
Libcoral	00:21:35	00:35:35	00:09:42
Libnetdude	00:47:23	01:16:34	00:22:10
Ruby	04:33:16	06:43:05	02:07:39

(b) Scans Analysis.

Table 5: Time taken to run the evaluation programs (HH:MM:SS).

that packets can be read from disk. The difference between libtrace and other libraries was significant for the Waikato and ISP traces: libpcap, which was next fastest, required 40% more time to complete the same analysis. However, libcoral was only three seconds slower than libtrace when processing the MAWI traces; libcoral forks an additional process to read pcap trace files, which improves the I/O speed, but this does not occur when reading ERF trace files.

When performing the Scans analysis, we found that libpcap was slightly faster than libtrace. Due to the processing-heavy task of having to maintain a flow table, the Scans analysis was not I/O-bound and therefore libtrace's threaded I/O offered little advantage over the piped I/O used by the libpcap program. Instead, the speed of the protocol decoders was the critical factor: profiling showed that the increased function call overhead from using the libtrace API functions was sufficient to give libpcap a slight performance edge. However, the caching of header locations enabled libtrace to outperform the libcoral analysis.

4. CONCLUSION

In this paper we have described libtrace, an open-source library for reading, processing and writing packet traces that we believe offers the best compromise between the effort required to develop a trace analysis program and the subsequent performance of that program. We have discussed the problems typically encountered when working with passive network traces, especially when using libpcap, and the methods used by libtrace to solve or alleviate them.

We have evaluated our library by implementing two trace analysis tasks using several trace analysis libraries, including libtrace and libpcap, and measuring both the development effort required and the time taken to run the analysis over some sample trace sets. We found that libtrace offered the best performance for I/O-bound analysis tasks (40% faster than an equivalent libpcap program) and was second-fastest when running an analysis-heavy task, where libpcap proved slightly faster due to less function call overhead.

However, the real strength of libtrace lies in the programming API, which allows easy development of analysis programs. Libtrace deals with the intricacies of capture formats, network protocols and edge-cases internally, leaving the programmer free to concentrate on measuring the relevant network properties instead. Using libtrace required less development effort (measured using lines of code) than any of the C libraries that we evaluated. The scripted language libraries were faster to develop with, but the performance of the resulting programs proved to be unsuitable for large-scale trace analysis. By contrast, the libpcap programs required the most lines of code, due to the lack of protocol decoding functions in libpcap and the need to detect and handle special cases explicitly.

A libpcap program may perform better in some instances, but significant expertise and time are required to develop a libpcap program that delivers accurate and reliable results. Libtrace allows researchers to quickly and easily develop passive network analysis tools that "just work" regardless of the capture format or network configuration, which we regard as a valuable contribution to the measurement community.

Libtrace remains under continuous development and support for new protocols and capture formats is added on a frequent basis. There are also ongoing efforts aimed at improving performance; for instance, we have developed an experimental JIT compiler that may allow for BPF filtering that is faster than the standard libpcap implementation.

The latest version of libtrace can be freely downloaded from `http://research.wand.net.nz/`.

5. REFERENCES

[1] "Libpcap," http://www.tcpdump.org/.

[2] Endace Measurement Systems Ltd., http://www.endace.com/.

[3] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and k. claffy, "The Architecture of CoralReef: an Internet Traffic Monitoring Software Suite," in *Passive and Active Network Measurement Workshop (PAM)*, Apr 2001.

[4] "Netdude," http://netdude.sourceforge.net/.

[5] "Scapy," http://www.secdev.org/projects/scapy/.

[6] W. John, S. Tafvelin, and T. Olovsson, "Review: Passive Internet Measurement: Overview and Guidelines based on Experiences," *Comput. Commun.*, vol. 33, pp. 533–550, March 2010.

[7] WAND Network Research Group, "WITS: Waikato Internet Traffic Storage," http://www.wand.net.nz/wits/index.php.

[8] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[9] M. Allman, V. Paxson, and J. Terrell, "A Brief History of Scanning," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, ser. IMC '07, 2007, pp. 77–82.

[10] "MAWI Working Group Traffic Archive," http://tracer.csl.sony.co.jp/mawi/samplepoint-F/2010/201004261400.html.

[11] D. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/.