SF-TAP: Scalable and Flexible Traffic Analysis Platform Running on Commodity Hardware

Yuuki Takano, Ryosuke Miura, and Shingo Yasuda, ***National Institute of Information and Communications Technology and Japan Advanced Institute of Science and Technology;***
Kunio Akashi, ***Japan Advanced Institute of Science and Technology;***
Tomoya Inoue, ***Japan Advanced Institute of Science and Technology and National Institute of Information and Communications Technology***

https://www.usenix.org/conference/lisa15/conference-program/presentation/takano

This paper is included in the Proceedings of the 29th Large Installation System Administration Conference (LISA15).

November 8–13, 2015 • Washington, D.C.

ISBN 978-1-931971-270

Open access to the Proceedings of the 29th Large Installation System Administration Conference (LISA15) is sponsored by USENIX

SF-TAP: Scalable and Flexible Traffic Analysis Platform running on Commodity Hardware

Yuuki Takano[1,2] ytakano@wide.ad.jp
Ryosuke Miura[1,2] myu2@nict.go.jp
Shingo Yasuda[1,2] s-yasuda@nict.go.jp
Kunio Akashi[2] k_akashi@jaist.ac.jp
Tomoya Inoue[2,1] t-inoue@jaist.ac.jp

National Institute of Information and Communications Technology, Japan[1]
Japan Advanced Institute of Science and Technology[2]

Abstract

Application-level network traffic analysis and sophisticated analysis techniques such as machine learning and stream data processing for network traffic require considerable computational resources. In addition, developing an application protocol analyzer is a tedious and time-consuming task. Therefore, we propose a scalable and flexible traffic analysis platform (SF-TAP) that provides an efficient and flexible application-level stream analysis of high-bandwidth network traffic. Our platform's flexibility and modularity allow developers to easily implement multicore scalable application-level stream analyzers. Furthermore, SF-TAP is horizontally scalable and can therefore manage high-bandwidth network traffic. We achieve this scalability by separating network traffic based on traffic flows, forwarding the separated flows to multiple SF-TAP cells, each of which consists of a traffic capturer and application-level analyzers. In this study, we discuss the design and implementation of SF-TAP and provide details of its evaluation.

1 Introduction

Network traffic engineering, intrusion detection systems (IDSs), intrusion prevention systems (IPSs), and the like perform application-level network traffic analysis; however, this analysis is generally complicated, requiring considerable computational resources. Therefore, in this paper, we propose a scalable and flexible traffic analysis platform (SF-TAP) that runs on commodity hardware. SF-TAP is an application-level traffic analysis platform for IDSs, IPSs, traffic engineering, traffic visualization, network forensics, and so on.

Overall, two problems arise in application-level network traffic analysis. The first is the difficulty in managing various protocols. There are numerous application protocols, with new protocols being defined and implemented every year. However, the implementation of application protocol parsers and analyzing such programs is a tedious and time-consuming task. Thus, a straightforward implementation of a parser and analyzer is crucial for application-level network traffic analysis. To enable such a straightforward implementation, approaches using domain-specific languages (DSLs) have been previously proposed. For example, BinPAC [24] is a parser for application protocols that are built into Bro IDS software. Wireshark [38] and Suricata [34] are binding Lua languages, with analyzers that can be implemented in Lua. Unfortunately, DSLs are typically not sufficiently flexible because there is often the requirement that researchers and developers want to use specific programming languages for specific purposes such as machine learning.

The second problem with application-level network traffic analysis is the low scalability of conventional software. Traditional network traffic analysis applications such as tcpdump [35], Wireshark, and Snort [33] are single threaded and therefore cannot take advantage of multiple CPU cores when performing traffic analysis. With the objective of improving the utilization of CPU cores, several studies have been conducted and software solutions have been proposed. For example, for high-bandwidth and flow-based traffic analysis, GASPP [36] exploits GPUs and SCAP [25], which utilizes multiple CPU cores, implements a Linux kernel module. Although it is important to reconstruct TCP flows efficiently, the efficiency of a parser or analyzing programs is more critical because they require more computational resources for performing such deep analysis as pattern matching or machine learning. Therefore, multicore scaling is required for both TCP flow reconstruction and traffic-analyzing components to enable the analysis of high-bandwidth traffic. In addition to multicore scalability, horizontal scalability is important for the same reason. To support the deep analysis of high-bandwidth network traffic to be performed easily and cost effectively, the corresponding application-level analysis plat-

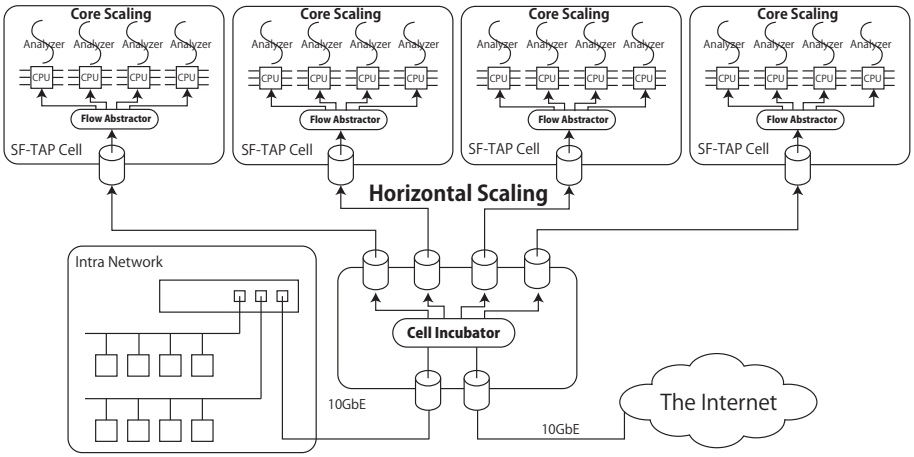


Figure 1: High-level Architecture of SF-TAP

form must have horizontal scalability.

Given the abovementioned issues, in this study, we discuss the design and implementation of SF-TAP for high-bandwidth application-level traffic analysis. SF-TAP adopts a flow abstraction mechanism that abstracts network flows by files, much like Plan 9 [28], UNIX's /dev, or the BSD packet filter (BPF) [19]. Using the interfaces, analyzing logic developers can rapidly and flexibly implement application-level analyzers in any language. Furthermore, L3/L4-level controlling and application-level analyzing components are separate given the modularity of the architecture. As a result of this design, analyzing components can be flexibly implemented, dynamically updated, and multicore scalable.

Note that our proof-of-concept implementation is distributed on the Web (see [32]) under BSD licensing for scientific reproducibility, and thus, it is freely available for use and modification.

2 Design Principles

In this section, we discuss the design principles of SF-TAP, i.e., the abstraction of network flows, multicore scalability, horizontal scalability, and modularity.

First, we describe the high-level architecture of SF-TAP, which is shown in Figure 1. SF-TAP has two main components: the cell incubator and flow abstractor. We call a group that consists of a flow abstractor and analyzers a cell. The cell incubator provides horizontal scalability; thus, it captures network traffic, separates it on the basis of the flows, and forwards separated flows to specific target cells. Conventional approaches using pcap or other methods cannot manage high-bandwidth network traffic, but our approach has successfully managed 10 Gbps network traffic using netmap [30], multiple threads, and lightweight locks.

Furthermore, by separating network traffic, we can manage and analyze high-bandwidth network traffic using multiple computers. By providing multicore scalability, the flow abstractor receives flows from the cell incubator, reconstructs TCP flows, and forwards the flows to multiple application-level analyzers. The multicore and horizontally scalable architectures enable application-level traffic analysis, which requires considerable computational resources, to be performed efficiently.

2.1 Flow Abstraction

DSL-based approaches have been adopted in several existing applications, including the aforementioned Wireshark, Bro, and Suricata. However, these approaches are not always appropriate because different programming languages are suitable for different requirements. As an example, programming languages suitable for string manipulation, such as Perl and Python, should be used for text-based protocols. Conversely, programming languages suitable for binary manipulation, such as C and C++, should be used for binary-based protocols. Furthermore, programming languages equipped with machine learning libraries should be used for machine learning.

Therefore, we propose an approach that abstracts network flows into files using abstraction interfaces, much like Plan 9; UNIX's /dev; and BPF, to provide a flexible method for analyzing application-level network traffic. Using these abstraction interfaces, various analysts such as IDS/IPS developers or traffic engineers can implement analyzers using their preferred languages. Flexibility is of particular importance in the research and development phase of traffic analysis technologies.

2.2 Multicore Scalability

To analyze high-bandwidth network traffic efficiently, many CPU cores should be utilized for the operation of both TCP/IP handlers and analyzers. We achieve multicore scalability through our modular architecture and threads. More specifically, the flow abstractor is multithreaded with modularity that allows analyzers to be CPU core scalable.

2.3 Horizontal Scalability

Application-level analyzers require substantial computational resources. For example, string parsing is used to analyze HTTP messages, pattern matching via regular expressions is used to filter URLs in real time, and machine learning techniques are applied to extract specific features of network traffic. In general, these processes consume a considerable amount of CPU time.

Accordingly, we propose a horizontally scalable architecture for high-bandwidth application-level traffic analysis. The horizontal scalability allows the analyzers, which consume considerable computational resources, to be operated on multiple computers.

2.4 Modular Architecture

The modularity of our architecture is an important factor in providing multicore scalability and flexibility. In addition, it offers some advantages. In the research and development phase, analyzing components are frequently updated. However, if an update is required, traditional network traffic analysis applications, which are monolithic, such as Snort or Bro, must halt operation of all components, including the traffic capturer.

We therefore propose a modular architecture for network traffic analysis that allows network traffic capturing and analyzing components to be separate. Thus, modularity allows traffic analysis components to be easily updated without impacting other components. Furthermore, bugs in applications still under development do not negatively affect other applications.

2.5 Commodity Hardware

Martins et al. [18] indicated that hardware appliances are relatively inflexible and the addition of new functions is not easily accomplished. Furthermore, hardware appliances are very expensive and not easily scaled horizontally. To address these problems, software-based alternatives running on commodity hardware, including network function virtualization (NFV) [22], are now being developed. We propose a software-based approach that runs in commodity hardware environments to achieve flexibility and scalability similar to those of NFV.

3 Design

In this section, we describe the design of SF-TAP.

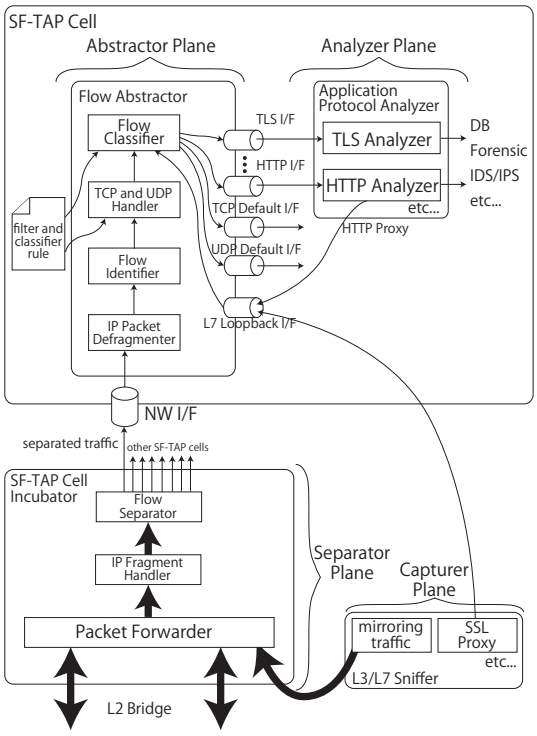


Figure 2: Architecture of SF-TAP

3.1 Design Overview

Figure 2 shows the architecture of SF-TAP. SF-TAP consists of four planes: the capturer, separator, abstractor, and analyzer planes. Each is described below. The capturer plane is a plane for capturing network traffic. More specifically, this plane consists of a port-mirroring mechanism of L2/L3 switches, an SSL proxy to sniff plain text, and so on. The separator plane is a plane that provides horizontal scalability for high-bandwidth network traffic analysis. This plane separates network traffic into L3/L4 levels, forwarding flows to multiple cells, each of which consists of the abstractor and analyzer planes.

The abstractor plane is a plane for network flow abstraction. This plane defragments IP fragmentations, identifies flows at the L3/L4 level, reconstructs TCP streams, detects the application protocol using regular expressions, and outputs flows to the appropriate abstraction interfaces. Traffic analyzer developers can develop analyzers by accessing the interfaces provided by this plane. Finally, the analyzer plane is a plane for analyzers developed by SF-TAP users. Users can implement analyzers in any programming language.

In this study, we focus on the separator and abstractor planes because the components of the capturer plane are well known and analyzers of the analyzer plane are developed by SF-TAP users. In the subsections that follow,

```
1  http:
2    up:      '^[-a-zA-Z]+ .+ HTTP/1\.(0\r?\n|1\r?\n([-a-zA-
   Z]+: .+\r?\n)+)'
3    down:    '^HTTP/1\.[01] [1-9][0-9]{2} .+\r?\n'
4    proto:   TCP  # TCP or UDP
5    if:      http # path to UNIX domain socket
6    nice:    100  # priority
7    balance: 4    # balanced by 4 IFs
8
9  torrent_tracker: # BitTorrent Tracker
10   up:      '^GET .*(announce|scrape).*\?.*info_hash=.+&.+
    HTTP/1\.(0\r?\n|1\r?\n([-a-zA-Z]+: .+\r?\n)+)'
11   down:    '^HTTP/1\.[01] [1-9][0-9]{2} .+\r?\n'
12   proto:   TCP
13   if:      torrent_tracker
14   nice:    90  # priority
15
16 dns_udp:
17   proto:   UDP
18   if:      dns
19   port:    53
20   nice:    200
```

Figure 3: Configuration Example for Flow Abstractor

we describe the design of the flow abstractor and cell incubator and show an example of an analyzer for HTTP.

3.2 Flow Abstractor Design

In this section, we describe the design of the flow abstractor, as shown in Figure 2, and explain its key mechanisms using the example configuration file shown in Figure 3. For human readability, the flow abstractor adopts YAML [39] to describe its configuration. Using a top–down approach, the flow abstractor consists of four components: the IP packet defragmenter; flow identifier; TCP and UDP handler; and flow classifier.

3.2.1 Flow Reconstruction

The flow abstractor defragments fragmented IP packets and reconstructs TCP streams. Thus, analyzer developers need not implement the complicated reconstruction logic required for application-level analysis.

The IP packet defragmenter shown in Figure 2 is a component that performs IP packet defragmentation. Defragmented IP packets are forwarded to the flow identifier, which identifies the flow as being at the L3/L4 level. We identify flows using 5-tuples consisting of the source and destination IP addresses, source and destination port numbers, and a hop count, which is described in Section 3.2.2. After the flows have been identified, the TCP streams are reconstructed by the TCP and UDP handler, and then, the flows are forwarded to the flow classifier.

3.2.2 Flow Abstraction Interface

The flow abstractor provides interfaces that abstract flows at the application level. For example, Figure 2 shows the TLS and HTTP interfaces. Furthermore, in Figure 3, the HTTP, BitTorrent tracker [2], and DNS interfaces are defined.

```
1  $ ls -R /tmp/sf-tap
2  loopback7=       tcp/             udp/
3
4  /tmp/sf-tap/tcp:
5  default=         http2=           ssh=
6  dns=             http3=           ssl=
7  ftp=             http_proxy=      torrent_tracker=
8  http0=           irc=             websocket=
9  http1=           smtp=
10
11 /tmp/sf-tap/udp:
12 default=         dns=             torrent_dht=
```

Figure 4: Directory Structure of Flow Abstraction Interface

The flow classifier classifies flows of various application protocols, forwarding them to flow abstraction interfaces, which are implemented using a UNIX domain socket, as shown in Figure 4. The file names of the interfaces are defined by items of *if*; for example, on lines 5, 13, and 18 in Figure 3, the interfaces of HTTP, BitTorrent tracker, and DNS are defined as `http`, `torrent_tracker`, and `dns`, respectively. By providing independent interfaces for each application protocol, any programming language can be used to implement analyzers.

Further, we designed a special interface for flow injection called the L7 loopback interface, i.e., *L7 Loopback I/F* in Figure 2. This interface is convenient for encapsulated protocols such as HTTP proxy. As an example, HTTP proxy can encapsulate other protocols within HTTP, but the encapsulated traffic should also be analyzed at the application level. In this situation, a further analysis of encapsulated traffic can easily be achieved by re-injecting encapsulated traffic into the flow abstractor via the L7 loopback interface. The flow abstractor manages re-injected traffic in the same manner. Therefore, the implementation of the application-level analysis of encapsulated traffic can be simplified, although, in general, it tends to remain rather complex.

Note that the L7 loopback interface may cause infinite re-injections. To avoid this problem, we introduce a hop count and corresponding hop limitation. The flow abstractor drops injected traffic when its hop count exceeds the hop limitation, thus avoiding infinite re-injection.

In addition to the flow abstraction and L7 loopback interface, the flow abstractor provides default interfaces for unclassified network traffic. Using these default interfaces, unknown or unclassified network traffic can be captured.

3.2.3 TCP Session Abstraction

An example output is shown in Figure 5, with the flow abstractor first outputting a header, which includes information on the flow identifier and abstracted TCP event; the flow abstractor then outputs the body, if it exists.

```
1  ip1=192.168.0.1,ip2=192.168.0.2,port1=62918,port2=80,hop=0,l3=ipv4,l4=tcp,event=CREATED
2  ip1=192.168.0.1,ip2=192.168.0.2,port1=62918,port2=80,hop=0,l3=ipv4,l4=tcp,event=DATA,from=2,match=down,len=1398
3
4  1398[bytes] Binary Data
5
6  ip1=192.168.0.1,ip2=192.168.0.2,port1=62918,port2=80,hop=0,l3=ipv4,l4=tcp,event=DESTROYED
```

Figure 5: Example Output of Flow Abstraction Interface

Line 1 of Figure 5 indicates that a TCP session was established between 192.168.0.1:62918 and 192.168.0.2:80. Line 2 indicates that 1398 bytes of data were sent to 192.168.0.1:62918 from 192.168.0.2:80. The source and destination addresses can be distinguished by the value of *from*, and the data length is denoted by the value of *len*. Lines 3–5 indicate that transmitted binary data are outputted. Finally, line 6 indicates that the TCP session was disconnected.

In the figure, *match* denotes the pattern, i.e., *up* or *down*, shown in Figure 3, that is used for protocol detection.

Managing a TCP session is quite complex; thus, the flow abstractor abstracts TCP states as three events, i.e., *CREATED, DATA, and DESTROYED*, to reduce complexity. Accordingly, analyzer developers can easily manage TCP sessions and keep their efforts focused on application-level analysis.

3.2.4 Application-level Protocol Detection

The flow classifier shown in Figure 2 is an application-level protocol classifier, which detects protocols using regular expressions and a port number. Items *up* and *down*, shown in Figure 3, are regular expressions for application-level protocol detection, and when upstream and downstream flows are matched by these regular expressions, flows are outputted to a specified interface. There are several methods for detecting application-level protocols, including Aho–Corasick, Bayesian filtering, and regular expressions. However, we adopt regular expressions because of its generality and high expressive power. In addition, a port number can be used to classify flows as application-level flows. As an example, line 19 of Figure 3 indicates that DNS is classified by port number 53.

Values of *nice*, which is introduced to remove ambiguity in Figure 3, are used for priority rules; here, the lower the given value, the higher the priority. For example, because BitTorrent tracker adopts HTTP for its communication, there is no difference in terms of protocol formats between HTTP and BitTorrent tracker. Accordingly, ambiguity occurs if rules for HTTP and BitTorrent tracker have the same priority; however, this ambiguity is removed by introducing priorities. In Figure 3, the priority of BitTorrent tracker is configured as being higher than that of HTTP.

3.3 Load Balancing using the Flow Abstraction Interface

In general, the number of occurrences of each application protocol in a network is biased. As such, if only one analyzer process is executed for one application protocol, the computational load will be concentrated in a particular analyzer process. Therefore, we introduce a load-balancing mechanism into the flow abstraction interfaces.

The configuration of the load-balancing mechanism is shown on line 7 of Figure 3. Here, the value of *balance* is specified as 4, indicating that HTTP flows are separated and outputted to four balancing interfaces. Interfaces *http0=, http1=, http2=,* and *http3=* in Figure 4 are the balancing interfaces. By introducing one-to-many interfaces, analyzers that are not multithreaded are easily scalable to CPU cores.

3.4 Cell Incubator Design

The cell incubator shown in Figure 2 is a software-based network traffic balancer that mirrors and separates network traffic based on the flows, thus working as an L2 bridge. The cell incubator consists of a packet forwarder, an IP fragment handler, and a flow separator.

The packet forwarder receives L2 frames and forwards them to the IP fragment handler. Furthermore, it forwards frames to other NICs, such as the L2 bridge, if required. Consequently, SF-TAP can be applied without hardware-based network traffic mirroring.

An IP fragment handler is required for the flow separation of fragmented packets because these packets do not always include an L4 header. This component identifies packets based on the given flows even if the packets are fragmented, forwarding the packets to the flow separator.

The flow separator forwards the packets to multiple SF-TAP cells using flow information that consists of the source and destination IP addresses and port numbers. The destination SF-TAP cell is determined by the hash value of the flow identifier.

3.5 HTTP Analyzer Design

In this subsection, we describe the design of an HTTP analyzer, which is an example of an application-level analyzer. The HTTP analyzer reads flows from the abstraction interface of HTTP provided by the flow abstractor

```
{
  "client": {
    "port": "61906",
    "ip":"192.168.11.12",
    "header": {
      "host": "www.nsa.gov",
      "user-agent":"Mozilla\/5.0 (Macintosh; Intel Mac OS X 10.9; rv:31.0) Gecko\/20100101 Firefox\/31.0",
      "connection": "keep-alive",
      "pragma": "no-cache",
      "accept": "text\/html,application\/xhtml+xml,application\/xml;q=0.9,*\/*;q=0.8",
      "accept-language": "ja,en-us;q=0.7,en;q=0.3", 11 "accept-encoding": "gzip, deflate",
      "cache-control": "no-cache"
    },
    "method": {
      "method": "GET",
      "uri": "\/",
      "ver": "HTTP\/1.1"
    },
    "trailer": {}
  },
  "server": {
    "port": "80",
    "ip": "23.6.116.226",
      "header": {
      "connection": "keep-alive",
      "content-length":"6268",
      "date": "Sat, 16 Aug 2014 11:38:25 GMT",
      "content-encoding": "gzip",
      "vary": "Accept-Encoding",
      "x-powered-by": "ASP.NET",
      "server": "Microsoft-IIS\/7.5",
      "content-type": "text\/html"
    },
    "response": {
      "ver": "HTTP\/1.1",
      "code": "200",
      "msg": "OK"
    },
    "trailer": {}
  }
}
```

Figure 6: Example Output of HTTP Analyzer

and then serializes the results into JSON format to provide a standard output. Figure 6 shows an example output of the HTTP analyzer. The HTTP analyzer reads flows and outputs the results as streams.

4 Implementation

In this section, we describe our proof-of-concept implementation of SF-TAP.

4.1 Implementation of the Flow Abstractor

We implemented the flow abstractor in C++; it depends on Boost [3], libpcap [35], libevent [14], RE2 [29], and yamp-cpp [40] and is available on Linux, *BSD, and MacOS X. The flow abstractor is multithreaded, with traffic capture, flow reconstruction, and application protocol detection executed by different threads. For simplicity and clarity, we applied a producer–consumer pattern for data transfer among threads.

The flow abstractor implements a garbage collector for zombie TCP connections. More specifically, TCP connections may disconnect without an FIN or RST packet because of PC or network troubles. The garbage collector collects this garbage on the basis of timers, adopting a partial garbage collection algorithm to avoid locking for a long time period.

In general, synchronization among threads requires much CPU loads. Thus, in the flow abstractor, we implemented bulk data transfers among threads. More specifically, bulk data transfers are performed among threads if the specified amount of data is in the producer's queue or the specified time has elapsed.

The performances of netmap [30] and DPDK [8] are better than libpcap; however, we did not adopt them because of their higher CPU resource consumption and less flexibility. Note that netmap [1] and DPDK require significant CPU resources because they access network devices via polling to increase throughput. Accordingly, they take away CPU resources from application-level analyzers, which require a substantial amount of CPU resources. Furthermore, netmap and DPDK exclusively attach to NICs; thus, other programs such as tcpdump cannot attach to the same NICs. This is an annoyance for network operations and for developing or debugging network software. High throughput, if required, can be accomplished with the help of netmap–libpcap [21].

4.2 Implementation of the Cell Incubator

The cell incubator must be able to manage high-bandwidth network traffic, but conventional methods such as pcap cannot manage high bandwidth. Therefore, we took advantage of netmap for our cell incubator implementation to provide packet capturing and forwarding. Consequently, we could implement a software-based high-performance network traffic balancer, i.e., the cell incubator.

The cell incubator is implemented in C++ and is available on FreeBSD and Linux. It has an inline mode and a mirroring mode. The inline mode is a mode in which the cell incubator works as an L2 bridge. On the other hand, the mirroring mode is a mode in which the cell incubator only receives and separates L2 frames, i.e., it does not bridge among NICs (unlike the L2 bridge). Users can select either the inline or mirroring mode when deploying the cell incubator.

The separation of network traffic is performed using hash values of each flow's source and destination IP addresses and port numbers. Thus, an NIC to which a flow is forwarded is uniquely decided.

Because we adopted netmap, we require that the NICs used by the cell incubator are netmap-available. In general, receive-side scaling (RSS) is enabled on NICs that are netmap-available, and there are multiple receiving

[1] netmap can manage packets by blocking and waiting, but this increases latency.

and sending queues on the NICs. Thus, the cell incubator generates a thread for each queue to balance the CPU load and achieve high-throughput packet managing. However, sending queues are shared among threads; thus, exclusive controls, which typically require a heavy CPU load, are needed. Therefore, to reduce the CPU load for exclusive controls, we adopted a lock mechanism that takes advantage of the compare-and-swap instruction.

4.3 Implementation of the HTTP Analyzer

For demonstration and evaluation, we implemented an HTTP analyzer, comprising only 469 lines, in Python. In our implementation, TCP sessions are managed using Python's dictionary data structure. The HTTP analyzer can also be easily implemented in other lightweight languages. Note that the Python implementation was used for performance evaluations presented in Section 5.

5 Experimental Evaluation

In this section, we discuss our experimental evaluations of SF-TAP.

5.1 HTTP Analyzer and Load Balancing

A key feature of the flow abstractor is its multicore scalability of application protocol analyzers. In this section, we show the effectiveness of the load-balancing mechanism of the flow abstractor through various experiments. In our experiments, the HTTP analyzer was used as a heavy application-level analyzer. Experiments were executed using a PC with DDR3 1.5 TB memory and an Intel Xeon E7-4830v2 processor (10 cores, 2.2 GHz, 20 MB cache) $\times$ 4 and the Ubuntu 14.10 (Linux Kernel 3.16) operating system.

The CPU loads of the HTTP analyzer and flow abstractor when generating HTTP requests are shown in Figure 7. In the figure, 50 HTTP clients were generated per second, with a maximum of 1,000 clients; on average, 2,500 HTTP requests were generated per second. Figures 7(a), (b), and (c) show CPU loads when load balancing was executed using one, two, and four HTTP analyzer processes, respectively.

When only one HTTP analyzer process was used, it could manage approximately 2,500 requests per second because of CPU saturation. However, when two processes were used, each process consumed only approximately 50% of CPU resources (i.e., it was not saturated). Moreover, when four processes were used, only approximately 25% of CPU resources were consumed. Consequently, we conclude that the load-balancing mechanism is remarkably efficient for multicore scalability. In our experiments, although the HTTP analyzer was implemented in Python (a relatively slow interpreted language), we could completely manage C10K using four HTTP analyzer processes.

Total memory usage of the HTTP analyzer is shown in Figure 8. When executing one, two, and four processes, approximately 12, 23, and 43 MB memory were allocated to them, respectively. Consequently, we conclude that memory usage proportionally increases with the number of processes; however, it is probably sufficiently small to allow application in the real world.

5.2 Performance Evaluation of the Flow Abstractor

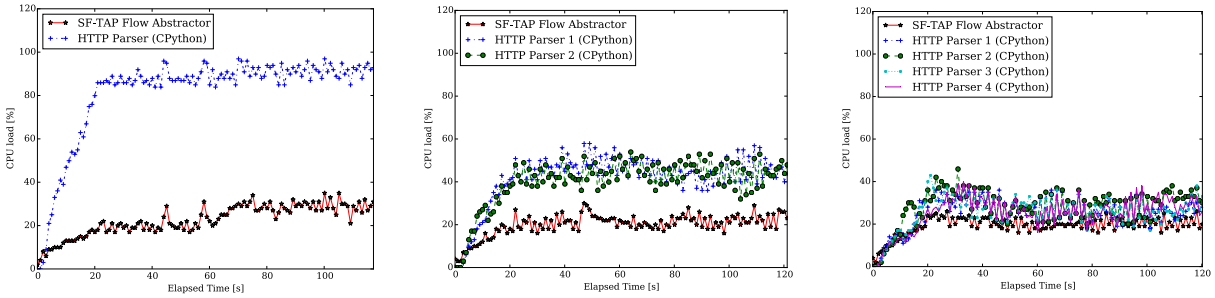
In this subsection, we show our experimental results for the flow abstractor. These experiments were conducted using the same PC described in Section 5.1.

Figure 9, in which CPS implies connections per second, shows packet-dropping rates when many TCP connections were generated. Here, one TCP session consisted of five packets: a 3-way handshake and two 1400-byte data packets. For comparison, we also determined the performances of tcpdump and Snort. We specified a 1 GB buffer for the flow abstractor and tcpdump; furthermore, we specified the maximum values possible for the elements of Snort's configuration, such as the limitation of memory usage and the number of TCP sessions. Experimental results showed that tcpdump, Snort, and the flow abstractor can manage approximately 3,200, 10,000, and 50,000 CPS, respectively. We achieved these performances because of multithreading and bulk data transfers described in Section 4.1. The flow abstractor completely separates capturing and parsing functions into different threads. Furthermore, bulk data transfers mitigated the performance overhead caused by spin lock and thread scheduling.

Figures 10(a), (b), and (c) show CPU loads when generating 1K, 5K, and 10K TCP CPS for up to 10 M connections, respectively. Because our implementation maintains TCP sessions by std::map of C++, the number of TCP connections affects the CPU load of the flow abstractor. For 10K CPS, the average CPU load exceeded 100%. This shows that the flow abstractor scales up to multiple CPU cores because of multithreading.

In Figure 10(c), after approximately 400 s, the CPU load slightly decreased from approximately 150% to 120%. This was probably caused by our garbage collection algorithm for TCP sessions. In our implementation, when the number of TCP sessions maintained by the flow abstractor is sufficiently small, the garbage collector scans all TCP sessions; on the other hand, when the number of TCP sessions is large, it partially scans TCP sessions to avoid a lock being caused by the garbage collector over a long time period.

Figure 11 shows CPU loads for traffic volumes of 1, 3, 5, and 7 Gbps. Here, we generated only one flow per measurement. Given these results, we conclude that the flow abstractor can manage high-bandwidth network



(a) HTTP Analyzer x 1 (b) HTTP Analyzer x 2 (c) HTTP Analyzer x 4

generate 50 clients / sec, 1000 clients maximum, 2500 requests / sec on average

Figure 7: CPU Load of HTTP Analyzer and Flow Abstractor

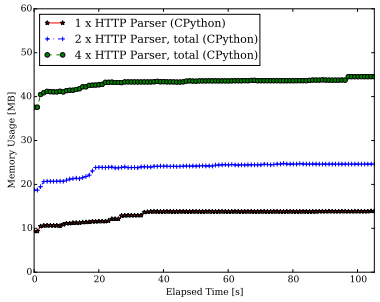


Figure 8: Total Memory Usage of HTTP Analyzer

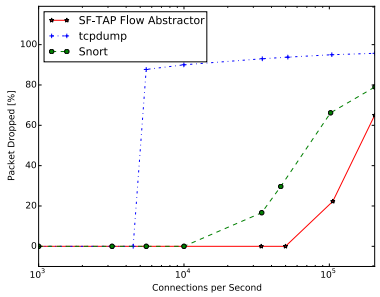


Figure 9: Packet Drop against CPS

traffic when the number of flows is small.

Figure 12 shows physical memory usage of the flow abstractor when generating 10K TCP CPS. The amount of memory usage of the flow abstractor primarily depends on the number of TCP sessions. More specifically, the amount of memory usage increases proportionally with the number of TCP sessions.

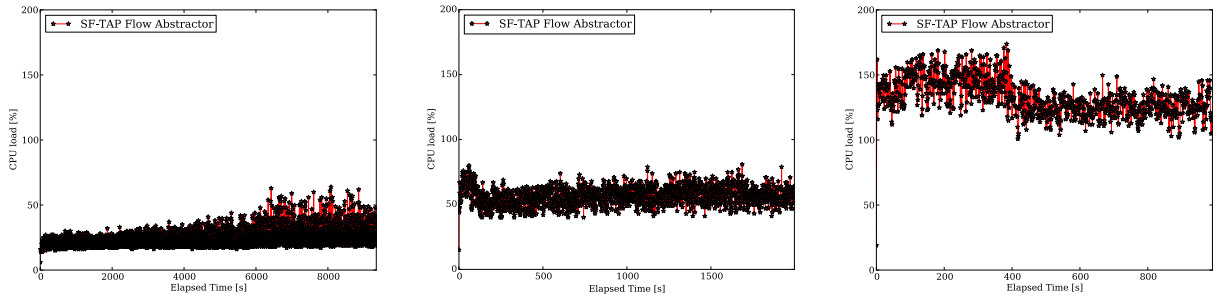
5.3 Performance Evaluation of the Cell Incubator

In the experiments involving the cell incubator, we used a PC with DDR3 16 GB Memory and an Intel Xeon E5-2470 v2 processor (10 cores, 2.4 GHz, 25 MB cache) and FreeBSD 10.1. The computer was equipped with four Intel quad-port 1 GbE NICs and an Intel dual-port 10 GbE NIC. We generated network traffic consisting of short packets (i.e., 64-byte L2 frames) on the 10 GbE lines for our evaluations. The cell incubator separated traffic based on the flows, with the separated flows forwarded to the twelve 1 GbE lines. Figure 13 shows our experimental network.

We conducted our experiments using three patterns: (1) the cell incubator worked in the mirroring mode using port mirroring on the L2 switch; in other words, it captured packets at α and forwarded packets to γ; (2) the cell incubator worked in the inline mode but did not forward packets to 1 GbE NICs, instead only α to β; and (3) the cell incubator worked in the inline mode, capturing packets at α and forwarding to both β and γ.

Table 14 shows the performance of the cell incubator. For pattern (1), i.e., the mirroring mode, the cell incubator could manage packets up to 12.49 Mpps. For pattern (2), i.e., the cell incubator working as an L2 bridge, it could forward packets up to 11.60 Mpps. For pattern (3), i.e., forwarding packets to β and γ, the cell incubator could forward packets to β and γ up to 11.44 Mpps. The performance of the inline mode was poorer than that of the mirroring mode because packets were forwarded to two NICs when using the inline mode. However, the inline mode is more suitable for specific purposes such as IDS/IPS because the same packets are dropped at β and γ. In other words, all transmitted packets can be captured when using the inline mode.

Table 15 shows the CPU load averages of the cell incubator when in the inline mode and forwarding 64-byte frames. At 5.95 and 10.42 Mpps, packets were not dropped when forwarding. At approximately 10.42 Mpps, the upper limit of dropless forwarding was reached. This indicates that several CPUs were used for forwarding, but the 15th CPU's resources were especially consumed.



(a) 1K CPS (b) 5K CPS (c) 10K CPS

generate 1K, 5K and 10K CPS, 10M connections maximum

Figure 10: CPU Loads of Flow Abstractor versus CPS

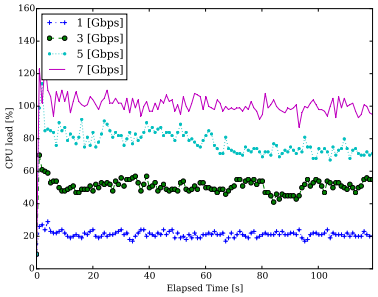


Figure 11: CPU Load of Flow Abstractor versus Traffic Volume

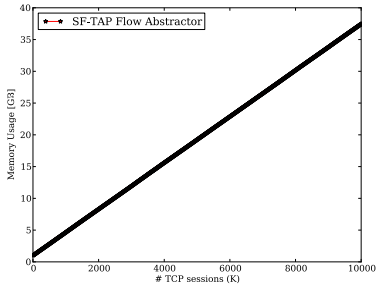


Figure 12: Physical Memory Usage of Flow Abstractor (10K CPS)

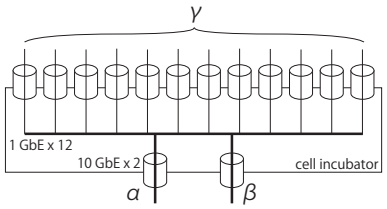


Figure 13: Experimental Network of Cell Incubator

Figure 16 shows the CPU loads of the 15th CPU. At 5.95 Mpps, the load average was approximately 50%, but at 10.42 Mpps, the loads were close to 100%. Moreover, at 14.88 Mpps, CPU resources were completely consumed. This limitation in forwarding performance was probably caused by the bias, which in turn was due to the flow director [10] of Intel's NIC and its driver. The flow director cannot currently be controlled by user programs on FreeBSD; thus, it causes bias depending on network flows. Note that the fairness regarding RSS queues is simply an implementation issue and is benchmarked for future work.

Finally, the memory utilization of the cell incubator depends on the memory allocation strategy of netmap. The current implementation of the cell incubator requires approximately 700 MB of memory to conduct the experiments.

6 Discussion and Future Work

In this section, we discuss performance improvements and pervasive monitoring.

6.1 Performance Improvements

We plan to improve the performance of the flow abstractor in three aspects.

(1) The UNIX domain socket can be replaced by another mechanism such as a memory-mapped file or cross-memory attach [6]; however, these mechanisms are not suitable for our approach, which abstracts flows as files. Thus, new mechanisms for high-performance message passing, such as the zero-copy UNIX domain socket or zero-copy pipe, should be studied.

(2) The flow abstractor currently uses the malloc function for memory allocation, which has some overhead. Here, malloc can be replaced by another lightweight

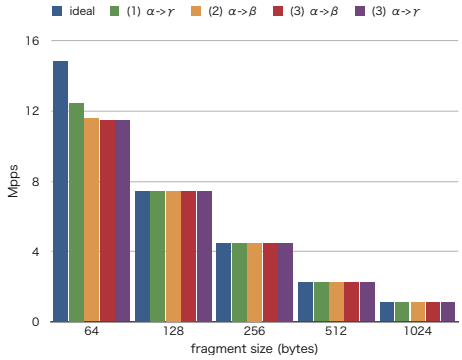


Figure 14: Forwarding Performance of Cell Incubator (10 Gbps)

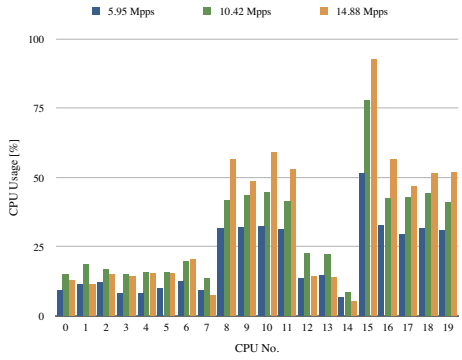


Figure 15: CPU Load Average of Cell Incubator (64-byte frames)

mechanism such as a slab allocator. The replacement of malloc by a slab allocator therefore constitutes an aspect of our future work.

(3) The flow abstractor adopts regular expressions for application protocol detection. We profiled the flow abstractor, but at present, this is not critical. Nonetheless, it potentially requires high computational resources. Thus, high-performance regular expressions should be studied in the future. Some studies have taken advantage of GPGPUs for high-performance regular expressions [5, 37, 41]. The implementation of regular expressions using GPGPUs is therefore another aspect of our future work.

6.2 Pervasive Monitoring and Countermeasures

Pervasive monitoring [9] is an important issue on the Internet. Countermeasures against pervasive monitoring include using cryptographic protocols such as SSL/TLS instead of traditional protocols such as HTTP and FTP, which are insecure. However, cryptographic protocols invalidate IDS/IPS, and consequently, other security risks are incurred.

Host-based IDS/IPS is a solution to the problem, but it is not suitable for mobile devices, which are widely used in today's society, because of the lack of machine power. Therefore, new approaches such as IDS/IPS cooperating with an SSL/TLS proxy should be studied to support the future of the Internet. The L7 loopback interface of the flow abstractor may also help future IDS/IPS implementations to be more robust against cryptographic protocols.

7 Related Work

Wireshark [38] and tcpdump [35] are widely used traditional packet-capturing applications, and libnids [15] is a network traffic-capturing application that reassembles TCP streams. The execution of these applications is essentially single threaded. Thus, they do not take advantage of multiple CPU cores and are therefore not suitable for high-bandwidth network traffic analysis.

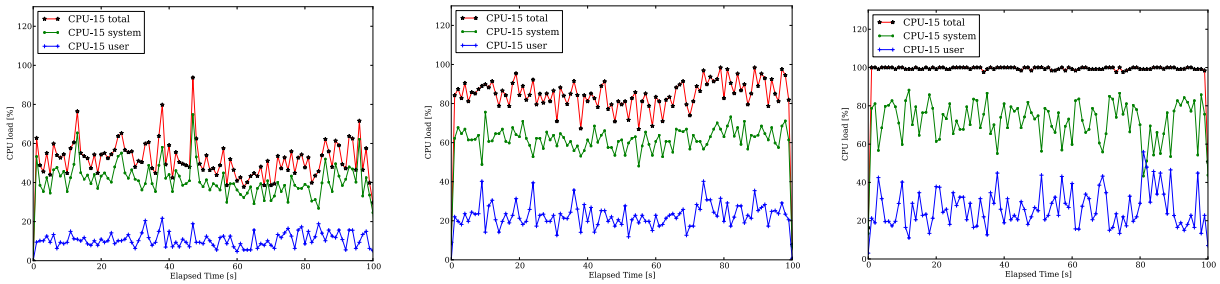
SCAP [25] and GASPP [36] were proposed for flow-level and high-bandwidth network traffic analyses. SCAP is implemented within a Linux kernel, taking advantage of the zero-copy mechanism and allocating threads for NIC's RX and TX queues to achieve high throughput. In addition, SCAP adopts a mechanism called subzero-copy packet transfer using analyzers that can selectively analyze required network traffic. GASPP is a GPGPU-based flow-level analysis engine that uses netmap [30]; thus, GASPP achieves high-throughput data transfers between the NIC and CPU memory.

DPDK [8], netmap [30], and PF_RING [27] were proposed for high-bandwidth packet-capture implementations. In traditional methods, many data transfers and software interrupts occur among the NIC, kernel, and user, thus making it difficult to capture 10 Gbps network traffic using traditional methods. Our proposed method achieved wire-speed traffic capture by effectively reducing the frequency of memory copies and software interrupts.

L7 filter [13], nDPI [20], libprotoident [16], and PEAFLOW [7] have been proposed for application-level network traffic classification implementations. These methods use Aho–Corasick or regular expressions to detect application protocols. PEAFLOW uses a parallel programming language called FastFlow to achieve high-performance classification.

IDS applications such as Snort [33], Bro [4], and Suricata [34] reconstruct TCP flows and application-level analysis. BinPAC [24] is a DSL used by Bro for protocol parsing; however, Snort and Bro are single threaded and cannot manage high-bandwidth network traffic. On the other hand, Suricata is multithreaded and manages high-bandwidth network traffic.

Schneider et al. [31] proposed a horizontally scalable



(a) 5.95 Mpps (b) 10.42 Mpps (c) 14.88 Mpps

Figure 16: 15th CPU's Load of Cell Incubator (64-byte frames)

architecture that separates 10 Gbps based on the flows, much like SF-TAP. They verified their architecture using only 1 Gbps network traffic and are yet to verify it using 10 Gbps network traffic.

Open vSwitch [23] is a software switch that can control network traffic based on the flows, much like our cell incubator does in our system, but its OVF CTRL cannot manage IP fragmentation. Some filtering mechanisms such as iptables [11] and pf [26] can also control network traffic based on the flows, but these mechanisms cannot manage IP fragmentation. Furthermore, these methods are less scalable and characterized by performance issues.

In Click [12], SwitchBlade [1], and ServerSwitch [17], modular architectures were adopted to provide flexible and programmable network functions for network switches. In SF-TAP, we adopted these ideas and proposed a modular architecture for network traffic analysis.

BPF [19] is a well-known mechanism for packet capturing that abstracts network traffic as files, much like UNIX's/dev. In SF-TAP, we adopted this idea, abstracting network flows as files to achieve modularity and multicore scaling.

8 Conclusion

Application-level network traffic analysis and sophisticated analysis techniques such as machine learning and stream data processing for network traffic require considerable computational resources. Therefore, in this paper, we proposed a scalable and flexible traffic analysis platform called SF-TAP for sophisticated high-bandwidth real-time application-level network traffic analysis.

SF-TAP consists of four planes: the separator plane, abstractor plane, capturer plane, and analyzer plane. First, network traffic is captured at the capturer plane, and then, captured network traffic is separated based on the flows at the separator plane, thus achieving horizontal scalability. Separated network traffic is forwarded to multiple SF-TAP cells, which consist of the abstractor and analyzer planes.

We provided cell incubator and flow abstractor implementations for the separator and abstractor planes, respectively. Furthermore, we implemented an HTTP analyzer as an example analyzer at the analyzer plane. The capturer plane adopts well-known technologies, such as port mirroring of L2 switches, for traffic capturing.

The flow abstractor abstracts network traffic into files, much like Plan9, UNIX's /dev, and BPF; the architecture of the flow abstractor is modular. The abstraction and modularity allow application-level analyzers to be easily developed in many programming languages and be multicore scalable. We showed experimentally that the HTTP analyzer we implemented as an example using Python can be easily scaled to multiple CPU cores.

The flow abstractor takes advantage of multithreading and bulk data transfers among threads. Thus, from our experiments, we found that the flow abstractor can manage up to 50K connections per second without dropping packets; tcpdump and Snort can manage only up to 4K and 10K connections per second, respectively.

In addition, we showed that the flow abstraction interfaces can help scale the HTTP analyzer to multiple CPU cores. Our experiments showed that our HTTP analyzer written in Python as a single process consumed 100% of CPU resources, but with four processes, each process only consumed 25% of CPU resources.

The cell incubator is a component that provides horizontal scalability. To manage high-bandwidth network traffic, the cell incubator separates network traffic based on the flows, forwarding separated flows to cells that consist of a flow abstractor and application-level analyzers. We experimentally showed that the cell incubator can manage approximately 12.49 Mpps and 11.44 Mpps when in the mirroring and inline modes, respectively.

Acknowledgments

We would like to thank the staff of StarBED for supporting our research and WIDE Project for supporting our experiments.

References

[1] Anwer, M. B., Motiwala, M., Tariq, M. M. B., and Feamster, N. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010* (2010), S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, Eds., ACM, pp. 183–194.

[2] BitTorrent. http://www.bittorrent.com/.

[3] Boost C++ Library. http://www.boost.org/.

[4] The Bro Network Security Monitor. http://www.bro.org/.

[5] Cascarano, N., Rolando, P., Risso, F., and Sisto, R. iNFAnt: NFA pattern matching on GPGPU devices. *Computer Communication Review 40*, 5 (2010), 20–26.

[6] Cross Memory Attach (index : kernel/git/torvalds/linux.git). http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=fcf634098c00dd9cd247447368495f0b79be12d1.

[7] Danelutto, M., Deri, L., Sensi, D. D., and Torquati, M. Deep Packet Inspection on Commodity Hardware using FastFlow. In *PARCO* (2013), M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. J. Peters, Eds., vol. 25 of *Advances in Parallel Computing*, IOS Press, pp. 92–99.

[8] Intel@ DPDK: Data Plane Development Kit. http://www.ntop.org/products/pf_ring/.

[9] Farrell, S., and Tschofenig, H. Pervasive Monitoring Is an Attack. RFC 7258 (Best Current Practice), May 2014.

[10] Intel®, High Peformance Packet Processing. https://networkbuilders.intel.com/docs/network_builders_RA_packet_processing.pdf.

[11] netfilter/iptables project homepage - The netfilter.org "iptables" project. http://www.netfilter.org/projects/iptables/.

[12] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (2000), 263–297.

[13] L7-filter — ClearFoundation. http://l7-filter.clearfoundation.com/.

[14] libevent. http://libevent.org/.

[15] libnids. http://libnids.sourceforge.net/.

[16] WAND Network Research Group: libprotoident. http://research.wand.net.nz/software/libprotoident.php.

[17] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011* (2011), D. G. Andersen and S. Ratnasamy, Eds., USENIX Association, pp. 15–28.

[18] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V. A., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), R. Mahajan and I. Stoica, Eds., USENIX Association, pp. 459–473.

[19] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993* (1993), USENIX Association, pp. 259–270.

[20] nDPI. http://www.ntop.org/products/ndpi/.

[21] The netmap project. http://info.iet.unipi.it/~luigi/netmap/.

[22] Leading operators create ETSI standards group for network functions virtualization. http://www.etsi.org/index.php/news-events/news/644-2013-01-isg-nfv-created.

[23] Open vSwitch. http://openvswitch.github.io/.

[24] Pang, R., Paxson, V., Sommer, R., and Peterson, L. L. binpac: a yacc for writing application protocol parsers. In *Internet Measurement Conference* (2006), J. M. Almeida, V. A. F. Almeida, and P. Barford, Eds., ACM, pp. 289–300.

[25] Papadogiannakis, A., Polychronakis, M., and Markatos, E. P. Scap: stream-oriented network traffic capture and analysis for high-speed networks. In *Internet Measurement Conference, IMC'13, Barcelona, Spain, October 23-25, 2013* (2013), K. Papagiannaki, P. K. Gummadi, and C. Partridge, Eds., ACM, pp. 441–454.

[26] PF: The OpenBSD Packet Filter. http://www.openbsd.org/faq/pf/.

[27] PF_RING. http://www.ntop.org/products/pf_ring/.

[28] Pike, R., Presotto, D. L., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. Plan 9 from bell labs. *Computing Systems 8*, 2 (1995), 221–254.

[29] RE2. https://github.com/google/re2.

[30] Rizzo, L., and Landi, M. netmap: memory mapped access to network devices. In *SIGCOMM* (2011), S. Keshav, J. Liebeherr, J. W. Byers, and J. C. Mogul, Eds., ACM, pp. 422–423.

[31] Schneider, F., Wallerich, J., and Feldmann, A. Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware. In *PAM* (2007), S. Uhlig, K. Papagiannaki, and O. Bonaventure, Eds., vol. 4427 of *Lecture Notes in Computer Science*, Springer, pp. 207–217.

[32] SF-TAP: Scalable and Flexible Traffic Analysis Platform. https://github.com/SF-TAP.

[33] Snort :: Home Page. https://www.snort.org/.

[34] Suricata — Open Source IDS / IPS / NSM engine. http://suricata-ids.org/.

[35] TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/.

[36] Vasiliadis, G., Koromilas, L., Polychronakis, M., and Ioannidis, S. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.* (2014), G. Gibson and N. Zeldovich, Eds., USENIX Association, pp. 321–332.

[37] Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E. P., and Ioannidis, S. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings* (2009), E. Kirda, S. Jha, and D. Balzarotti, Eds., vol. 5758 of *Lecture Notes in Computer Science*, Springer, pp. 265–283.

[38] Wireshark - Go Deep. https://www.wireshark.org/.

[39] The Official YAML Web Site. http://yaml.org/.

[40] A YAML parser and emitter in C++. https://github.com/jbeder/yaml-cpp.

[41] Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K., and Dong, Q. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012* (2012), J. Ramanujam and P. Sadayappan, Eds., ACM, pp. 129–140.