BITS Darshini: A Modular, Concurrent Protocol Analyzer Workbench

Prasad Talasila
Department of CS & IS
BITS, Pilani - KK Birla Goa Campus
Goa, India 403726
Email: tsrkp@goa.bits-pilani.ac.in

Mihir Kakrambe
Department of CS & IS
BITS, Pilani - KK Birla Goa Campus
Goa, India 403726
Email: mrkakrambe@gmail.com

Sebastin Santy
Department of E & I
BITS, Pilani - KK Birla Goa Campus
Goa, India 403726
Email: f2015357@goa.bits-pilani.ac.in

Anurag Rai
Department of CS & IS
BITS, Pilani - KK Birla Goa Campus
Goa, India 403726
Email: f2013693@goa.bits-pilani.ac.in

Neena Goveas
Department of CS & IS
BITS, Pilani - KK Birla Goa Campus
Goa, India 403726
Email: neena@goa.bits-pilani.ac.in

Bharat Deshpande
Department of CS & IS
BITS, Pilani - KK Birla Goa Campus
Goa, India 403726
Email: bmd@goa.bits-pilani.ac.in

***Abstract*—Network measurements are essential for troubleshooting and active management of networks. Protocol analysis of captured network packet traffic is an important passive network measurement technique used by researchers and network operations engineers. In this work, we present a measurement workbench tool named *BITS Darshini* (*Darshini* in short) to enable scientific network measurements.**

We have created Darshini as a modular, concurrent web application that stores experimental meta-data and allows users to specify protocol parse graphs. Darshini performs protocol analysis on a pipeline architecture, persists the analysis to a database and provides the analysis results via a REST API service. We also formulate the problem of mapping protocol parse graph to a pipeline as a graph embedding problem. Our tool, Darshini, performs protocol analysis up to transport layer and is suitable for the study of small and medium-sized networks. Darshini enables collaboration and consultations with experts.

***Index Terms*—Network measurements, measurement workbench, packet analyzer, collaborative analysis, concurrent packet analysis, protocol parse graph, graph embedding**

I. Introduction

Network packet capture and protocol analysis is an integral part of modern network management [1], [2]. A major concern in network measurements community is the lack of emphasis on the application of scientific (repeatable, verifiable and falsifiable) measurement principles [3]. Researchers and operations engineers often wish to control / restrict the packet analysis to scenarios of interest as implied by the experimental objectives [4]. Thus user-directed protocol analysis is an important requirement on packet capture and analysis tools. The network measurement community needs *collaboration* and *user-directed protocol analysis* features together in one measurement tool. An ideal measurement tool would also enable *scientific measurements*.

Centralized data repositories such as Crawdad [5] and DataCat [6] maintain useful network measurement datasets created using scientific measurement principles. In most of the network traffic data sets placed in public domain, the process of creating and documenting experimental design is adhoc. Having a packet capture tool that facilitates experimental design, documentation and collaboration would be useful in creating templates for measurement data exchange. In addition, the longitudinal evolution of network traffic mix [7], [8] requires user-directed protocol analysis. None of the existing tools include all three features – scientific measurements, collaboration and user-defined protocol analysis.

Popular protocol analysis tools like Wireshark [9] are developed for the scenario of lone engineer analyzing the captured packet stream on a local machine. Hence the concept of collaborative analysis is not a standard feature in these tools. Persistence is not a standard feature of these tools; thus collaborative analysis becomes repetitive. The experts / reviewers are asked to look at a pcap file without the associated experimental meta-data. Another limitation is the fixed configuration in measurement tools denying users the ability to select protocols of their interest for further analysis.

We overcome the above mentioned limitations in BITS Darshini. Our major contributions are:

1) Support measurement strategies and experimental workbench functionality to foster sound network measurements.
2) Support for collaboration between measurement engineers by enabling sharing of experimental workbench consisting of experimental setup and analysis results.
3) Allow experimenters to create user-defined protocol parse graph and perform protocol analysis as per this parse graph.
4) Create a flexible concurrent pipeline from one generic

TABLE I
A comparison of Darshini with other packet processing tools.

Parameter	BITS Darshini	tshark	Wireshark	ntop	BroIDS
Maintenance of measurement meta-data	✓	✗	✗	✗	✗
Collaboration	share analysis, pcaps hidden	⟵ share pcaps ⟶		share analysis	share logs
Persistence	Elastic Search (ES) DB	✗	✗	disk in HTML/RRD[a] format	logs
Protocol selectivity for analysis	user-defined parse graph	✗	✗	✗	BroScript
Concurrency	multi-threaded, multi-core	✗	✗	distributed capture	✗
Addition of new protocols	P4 protocol headers	⟵ WSGD[b]/ Lua / C ⟶			C++ and BroScript
Packet filters	BPF[c] for capture filters; ES REST API for display filters	⟵ BPF ⟶			BroScripts

[a] RRD - Round Robin Database, [b]WSGD - WireShark Generic Dissector, [c]BPF - Berkeley Packet Filter,

analyzer cell (GAC). The created pipeline protocol analyzer performs protocol analysis as per user-defined parse graph.

5) Support for persistence of analysis results in database with REST API access to data.

The protocol analyzer pipeline of Darshini is able to perform protocol analysis with a maximum throughput of 606 Mbps. This throughput is sufficient for most offline packet analysis scenarios. In-memory protocol analysis tools have difficulty with analyzing large pcap files; for example, Wireshark has difficulty analyzing pcap files larger than 100MB [10]. Darshini does not have any limitations on the input pcap size; the packet analysis rate of Darshini is independent of the input pcap file size. A comparison of Darshini with other packet processing tools is available in Table I.

II. Related Work

Darshini requires specification of the following elements: protocol headers, protocol parse graph, protocol analysis pipeline and persistence module. This section describes previous work on each of the above mentioned sub-areas.

A. Parse Graph

A protocol parse graph can be implemented in hardware, software or a mix of both hardware and software. One popular form of hardware implementation is the synthesis of the parse graph state machine onto ASICs with TCAM [11], and onto the commercially available FPGA architectures [12]. A survey of the hardware implementations of the protocol parse graph techniques is available in [13]. Examples of software implementation for the parse graphs are: Wireshark [9] and tcpdump [14].

Software / hardware implementations of the protocol parse graphs can either be a fixed or a programmable kind. A fixed parse graph can only parse the protocol sequences that are part of the given parse graph. On the other hand, a programmable parse graph can dynamically select a parse graph at the run time. In Darshini, we implement a programmable parse graph.

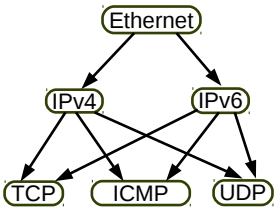


Fig. 1. Protocol parse graph.

The idea of protocol parse graph is a very old [15], [16]. A variation of protocol parse graph called Berkeley Packet Filter (BPF) is used in the tcpdump [14] and Linux socket filter [17]. In these instances, the parse graph is typically used to select packets of interest.

Our purpose is to parse the incoming packets in order to analyze the protocol stack of the packet. P4 language [18] presents a parse graph notation that is suitable for both hardware and software implementations. Hence we use P4 language to express the protocol parse graph.

B. Packet Parsers

There have been attempts to implement a pipeline protocol parse graph to enhance the protocol analysis throughput. The architectural solutions proposed by [11], [12], [19] and [20] are some of the pure hardware implementations of the packet parsers.

We adopt the Generic Protocol Parser Interface (GPPI) of Benáček et al. [19], [21] for our design of generic analyzer cell (GAC). The high frequency extractor (HFE) M2 and the GPPI discussed by Benáček et al. [19], [21] together form a serially connected pipeline implemented in hardware. Wireshark [9] implements parse graph in software without pipelines. Our analyzer pipeline is a complete

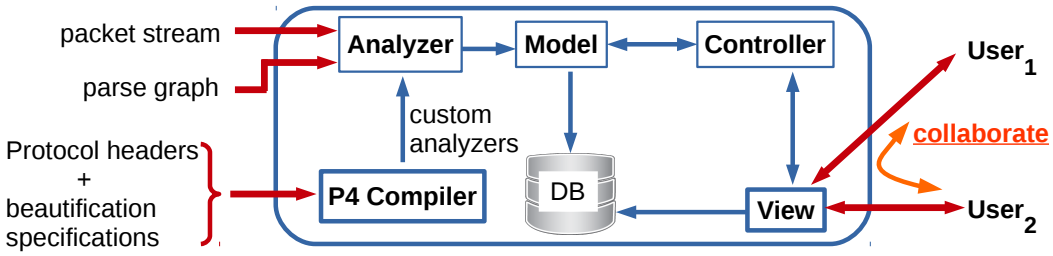


Fig. 2. System architecture of Darshini.

software implementation with support for concurrency. In addition, we introduce a logical bus connectivity to this pipeline by using feedforward / feedback line between all stages of the pipeline. We support multiple protocols per pipeline stage; The end result is an architecture that is flexible enough to support load balancing across pipeline stages.

C. System Architecture

Packet capture and protocol analysis software tcpdump, tshark and Wireshark have been developed as stand alone packet processing utilities. Ntop [1], [22] is a web application with dynamic plug-in system for customization of analysis and view. Darshini has also been designed as a web application with support for adding new protocols. One major difference between Ntop and Darshini is the user-defined protocol analysis. Darshini allows users to specify parse graph for protocol analysis while Ntop performs basic protocol analysis for all supported protocols.

III. Architecture

A. Preliminaries

We use P4 language [18] for specifying the header format of protocols and for specifying the protocol parse graph. An example protocol parse graph is given in Figure 1.

Darshini uses two different parse graphs. One is the parse graph of all supported protocols; prior literature refers to this graph as *union parse graph* [11]. The second parse graph is the user-specified parse graph indicating the protocols of interest to the user. For all practical purposes, user-specified parse graph is a sub-graph of the union parse graph.

B. System Overview

An outline of the system architecture is shown in Figure 2. Darshini takes in a stream of packets to operate on. The protocol parse graph, protocol header specification and beautification files are also given as input to Darshini. The protocol headers and parse graph are expressed in P4 language.

The major building blocks of Darshini are: analyzer pipeline, P4 compiler, database (DB) and MVC components. Analyzer pipeline receives custom analyzers created by P4 compiler. Analyzer pipeline uses user-defined parse graph and custom analyzers to parse the packet stream. Analyzer pipeline internally follows pipes-and-filters architectural pattern.

Model module is a part of MVC architecture that stores all the data of analyzers; model interacts with controller and DB. Controller module is also a part of MVC architecture and glues the model with views. Controller is responsible for handling all requests from view module and making method calls to models module. Controller returns data to view module which presents formatted data to user.

View module can directly interact with database over REST API. View module facilitates collaboration between users.

C. Analyzer Pipeline

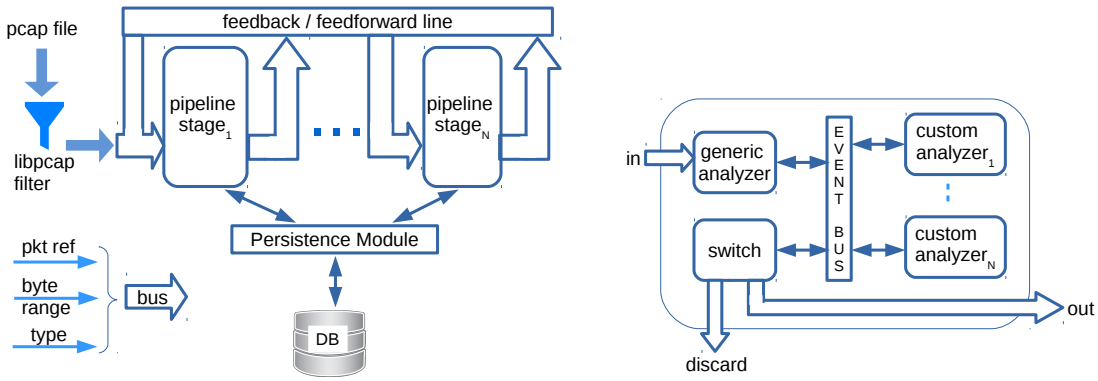
Analyzer pipeline forms the backbone of Darshini. Analyzer pipeline is responsible for taking in a filtered stream of packets and analyzing these packets as indicated by the user-specified parse graph. Analyzer pipeline stores the analysis results in DB via the persistence module. The analyzer pipeline architecture is shown in Figure 3a.

Each pipeline stage receives a packet, completes analysis for the protocols the stage is responsible for. It then forwards the packet to next stage. A packet is sent to next stage using the feedforward / feedback line. In order to accommodate tunneling scenarios, a feedback line connects a stage to itself or to one of the former pipeline stages. It is possible to encounter packets that do not have PCI header for a protocol layer (ex: raw packets with only TCP header would obviously miss the IP header). Feed forward line enables skipping of a pipeline stage where necessary.

D. Generic Analyzer Cell (GAC)

All the pipeline stages are created from the same template named GAC. The block diagram of the GAC is shown in Figure 3b.

All the incoming packets of a GAC are received by the generic analyzer. Generic analyzer collects statistical / flow information from the packet for record keeping purposes. Generic analyzer pushes the collected information to the persistence module. After this, the generic analyzer informs all the registered custom analyzers of the analyzer cell about the available packet. An appropriate custom analyzer picks up the packet to extract protocol headers.



(a) Analyzer module has parser pipeline and it interfaces with persistence module.

(b) Generic analyzer cell (GAC)

Fig. 3. Protocol analyzer pipeline. Each stage of the pipeline consists of one GAC which is illustrated in part-(b).

As soon as the protocol header extraction is done in a custom analyzer, next protocol is determined. The processed packet is then forwarded to next analyzer cell; extracted protocol headers are forwarded to the persistence module.

Generic and custom analyzers complete their work quickly; the data path from cell input to custom analyzers forms *fast path* of execution. Data path from custom analyzers to persistence module works at much slower rate and is called *slow path*.

Analyzer cells decouple the fast and slow execution paths. As soon as the output of the faster execution path is ready, the processed packet is passed to the next pipeline stage. The slower execution path of a pipeline stage can have long input and output queues to adjust to the packet processing throughput disparity between the fast and slow paths.

E. Parse Graph to Pipeline Mapping: A Graph Embedding Problem

Analyzer pipeline is the platform on which the parse graph gets executed. Hence, each node of a parse graph (each node is really a protocol) needs to be assigned to one pipeline stage. Because of the feedforward / feedback line, all stages of the pipeline are interconnected. From graph theoretic framework, the pipeline shown in Figure 3a forms a complete graph K_N where N indicates the number of stages in the pipeline. Since we wish to assign vertices of the parse graph to the pipeline stages (vertices of K_N), the problem is equivalent to graph embedding problem. More details about the applicability of the graph embedding problem to Darshini are available in Appendix.

IV. Implementation

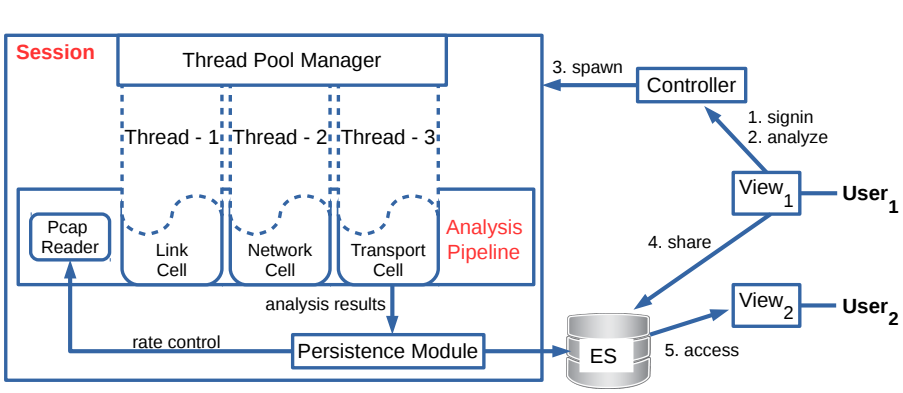
A. Realization of Architecture

Darshini has been implemented as a model-view-controller (MVC) architecture based web application.

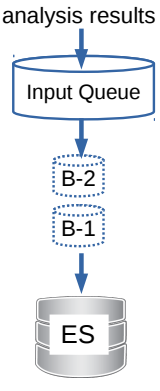
Figure 4 shows the modules of Darshini.

Model Consists of all the Java objects representing the data (persistence package) as well as analyzer, protocol and utils packages. All model objects get saved in Elastic Search.

View Consists of client-side (web-browser) code. We use backbone.js Model-View framework to implement client-side functionality for Darshini in the browser.



(a) Session creation for one run of protocol analysis



(b) Persistence queue. B-1, B-2 etc are batched queries.

Fig. 4. Implementation details of the system architecture for Darshini.

TABLE II
API END POINTS FOR DARSHINI. THE BASE URL OF HOST AND ES URLS ARE NOT SHOWN.

API URL	Service
/	home page
/signup	user signup
/signin	user login
/session/validate	validate parse graph
/session/analyze	start protocol analysis

Based on context, view either interacts with server-side controller or with Elastic Search (ES). The API URLs accessed by the client-side code are listed in Table II.

Controller Controller is responsible for authenticating the users. Controller is also responsible for an on-demand launch of a session to manage analyzer pipeline of a packet analysis experiment.

Session Corresponds to one independent protocol analysis. The pipeline protocol analysis itself is performed using Java Threads. Each analyzer cell of an analyzer pipeline is run on a dedicated thread. All custom analyzers persist the protocol analysis results to ES. Details of session and analyzer pipeline are illustrated in Figure 4a.

Elastic Search (ES) A plug-and-play module that provides base for persistence of application data, especially the packet analysis data.

Persistence Responsible for managing the speed mismatch between fast analyzer pipeline and the slow ES. The speed mismatch is managed using a two-stage queue as illustrated in Figure 4b. Analysis data from custom analyzers is put into batches and handed over to Elastic Search.

B. Measurement Workbench

We created a system of measurement workbench where the Internet measurements can go through the measurement cycle (Objective → Strategies → Measure → Analyze → Refine Objective).

Darshini facilitates the measurement cycle using the measurement strategies suggested by Vern Paxson [3]. Further details are available in Table III.

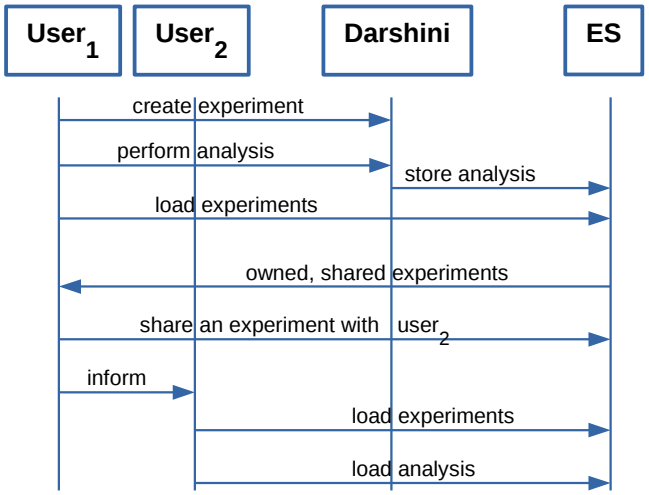


Fig. 5. Sequence diagram illustrating collaboration inside BITS Darshini.

C. Collaborative Analysis

Users of Darshini can share owned experiments with other users. User have access to two categories of experiments: owned and shared. *Owned* experiments are the experiments created by self; *Shared* experiments are the experiments shared with a user by the other users. A typical sequence of actions taken for collaboration within Darshini are illustrated in Figure 5.

D. User-Defined Protocol Analysis

Darshini comes pre-configured with a union parse graph that acts as a base graph from which experimenter selects a sub-graph. The user-specified parse graph must be a

TABLE III
SUPPORT FOR MEASUREMENT STRATEGIES IN DARSHINI

Measurement Strategy	Implementation
Maintain meta-data	Experimental description page
Error detection	Auto-detection possible with Elastic Search queries
Reproducible analysis	Experiment history
Sub-sample large data	Sub-sampling in protocol domain using parse graph; time domain sub-sampling possible via REST API queries
Periodic analysis	Available as a service
Data reduction scripts	Elastic Search as a service to execute dynamic queries from users
Outlier detection	Supported through REST API interface
Comparing multiple measurements	Supported through REST API interface
Public data sets	Share experiment with other users; avoids sharing pcap files

strict sub-graph of the union parse graph available on the analyzer software. In this work, we show results for a static mapping from union parse graph to the analyzer pipeline. User has complete freedom to specify any sub-graph of union parse graph for each experiment.

V. Experimental Results

A. Data Set

We measure the performance of Darshini by using offline pcap files. These offline pcap files contain unfiltered network traffic captured on an edge computer connected to a mid-level enterprise network having approximately 5000 users. Since Darshini is better suited to perform offline protocol analysis on the traffic of small to medium-scale networks, mid-level enterprise traffic is a representative test scenario for Darshini.

In this section, we compare the performance of Darshini with tshark tool. Darshini is run in two modes – persistent mode and non-persistent mode. In *persistent mode*, analysis results are saved to Elastic Search. In *non-persistent mode*, analysis results are not saved. We use non-persistent mode to demonstrate the performance of the analyzer pipeline.

B. User-defined Protocol Analysis

We consider two protocol parse graphs, namely P_1 and P_2 for demonstrating the user-defined protocol analysis capability of Darshini. P_1 contains protocols $eth, ipv4, tcp$ and P_2 contains just eth. We complete the user-defined protocol analysis using parse graphs P_1 and P_2 on Darshini. The execution time and run-time memory consumption results of these two experiments are shown in Table IV.

TABLE IV
User-defined protocol analysis on a pcap file with 1,508,352 packets. The size of pcap file is 955MB.

Selected Protocols	Execution time (sec)			Memory consumption (MB)		
	A[a]	B[b]	C[c]	A[a]	B[b]	C[c]
$eth, ipv4, tcp$	11.3	40.2	394.8	770	128	220
eth	11.3	25.6	115.9	770	128	238

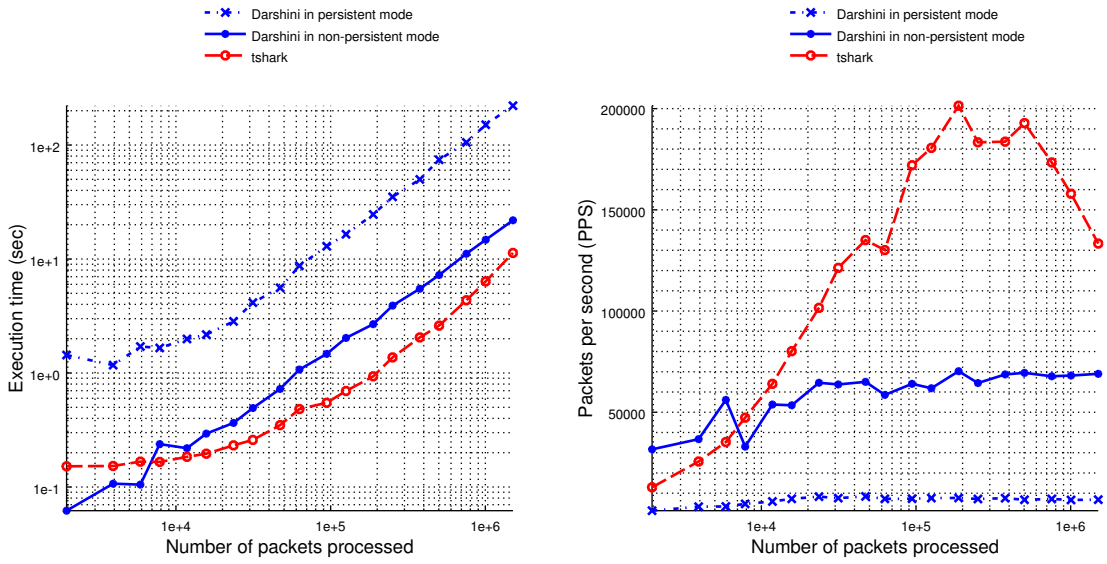
[a] A - tshark
[b] B - Darshini in non-persistent mode
[c] C - Darshini in persistent mode

We draw three conclusions from this experiment. First, the run time performance of Darshini is inversely proportional to size of parse graph. Darshini would be able to complete analysis of few selected protocols very quickly. Thus Darshini becomes suitable for user-defined protocol analysis. Second, the fast path (from input to custom analyzers of generic analyzer cell) is order of magnitude faster than the slow path (from custom analyzers to Elastic Search). We can extract better run time performance from Darshini by optimizing the database storage performance.

Third, Darshini uses 128MB for processing a pcap size of 955MB, where as tshark consumes 770MB for processing the same file. The reported number of 128MB include the memory allocation to Java Virtual Machine (JVM) and Tomcat Servlets. Thus Darshini is more memory efficient when compared with tshark.

C. Memory Management

We control batch sizes of queries sent to Elastic Search. Batch size is a configurable parameter for Darshini. With a batch size of 20,000 queries, Darshini processes 1.5 million



(a) Execution time

(b) Packet throughput

Fig. 6. Cell and application performance data on execution time and packet throughput parameters.

TABLE V
THROUGHPUT NUMBERS ACHIEVABLE BY DARSHINI.

frame size	PPS*			Throughput (Mbps)		
	A[a]	B[b]	C[c]	A[a]	B[b]	C[c]
1514 bytes	49,391	51,371	6,638	583	606	78.4
74 bytes	2,30,023	66,157	6,550	189.5	54.5	5.4

* PPS – packets per second
[a] A - tshark
[b] B - Darshini in non-persistent mode
[c] C - Darshini in persistent mode

packets and saves analysis data to Elastic Search in 224 seconds with a maximum memory consumption of 414 MB. With a batch size of 5,000 queries, Darshini processes the same pcap file in 908 seconds with a maximum memory consumption of 235 MB. Thus we can use the batch size to make trade offs between run time vs memory consumption.

D. Throughput

Figure 6 shows the relative performance of persistent and non-persistent modes of Darshini. Figure 6a shows the difference between persistent mode and non-persistent mode.

The range of throughput numbers achievable by Darshini are shown in Table V. Since the application and analyzer pipeline performance is packet size invariant, a meaningful performance metric is the packets processed per second (PPS) by Darshini which stands at approximately 66,000 PPS.

VI. CONCLUSION

BITS Darshini is a modular and concurrent protocol analysis tool. Darshini facilitates scientific network measurements done in a collaborative manner. Users can share experiments within Darshini.

Darshini enables users to select protocols of interest for analysis. The protocols of interest are specified using protocol parse graph. This user-defined parse graph directs user-defined protocol analysis in Darshini. We map the parse graph onto an analysis pipeline. Each protocol analysis request from a user launches a custom analyzer pipeline as per the parse graph.

Each custom analyzer pipeline is executed in a completely concurrent mode there by taking advantage of the multi-core processor architectures. In Appendix, we provide the mathematical formulation for mapping user-defined parse graph to analyzer pipeline as a graph embedding problem.

The results of protocol analysis are persisted in a database (Elastic Search) instance which in turn makes the results data available over REST API service interface.

REFERENCES

[1] *Ntopng – high-speed web-based traffic analysis and flow collection*, http://www.ntop.org/products/traffic-analysis/ntop/, Accessed: 2017-05-08.

[2] C. Williamson, "Internet Traffic Measurement," *Internet Computing, IEEE*, vol. 5, no. 6, pp. 70–74, 2001.

[3] V. Paxson, "Strategies for Sound Internet Measurement," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '04, Taormina, Sicily, Italy: ACM, 2004, pp. 263–271, ISBN: 1-58113-821-0. DOI: 10.1145/1028788.1028824. [Online]. Available: http://doi.acm.org/10.1145/1028788.1028824.

[4] ——, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23–24, pp. 2435–2463, 1999, ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/S1389-1286(99)00112-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128699001127.

[5] *CRAWDAD: Community Resource for Archiving Wireless Data at Dartmouth*, http://crawdad.org/, Accessed: 2017-05-07.

[6] *DataCat: Home*, http://imdc.datcat.org/, Accessed: 2017-05-07.

[7] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger, "Distilling the internet's application mix from packet-sampled traffic," in *International Conference on Passive and Active Network Measurement*, Springer, 2015, pp. 179–192.

[8] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, ACM, 2010, pp. 267–280.

[9] *Wireshark*, https://www.wireshark.org/, Accessed: 2017-05-08.

[10] *Performance – Wireshark Wiki*, https://wiki.wireshark.org/Performance, Accessed: 2017-05-08.

[11] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design Principles for Packet Parsers," in *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, Oct. 2013, pp. 13–24. DOI: 10.1109/ANCS.2013.6665172.

[12] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, Oct. 2011, pp. 12–23. DOI: 10.1109/ANCS.2011.12.

[13] V. A. P. Kumar, V. Thiyagarajan, and N. Ramasubramanian, "A Survey of Packet Classification Tools and Techniques," in *Computing Communication Control and Automation (ICCUBEA), 2015 International Conference on*, Feb. 2015, pp. 103–107. DOI: 10.1109/ICCUBEA.2015.26.

[14] *TCPDUMP/LIBPCAP public repository*, http://www.tcpdump.org/, Accessed: 2017-05-08.

[15] J. Mogul, R. Rashid, and M. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Net-

work Code," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 39–51, Nov. 1987, ISSN: 0163-5980. DOI: 10.1145/37499.37505. [Online]. Available: http://doi.acm.org/10.1145/37499.37505.

[16] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93, San Diego, California: USENIX Association, 1993, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267303.1267305.

[17] *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*, https://www.kernel.org/doc/Documentation/networking/filter.txt, Accessed: 2017-05-08.

[18] *P4*, http://p4.org/, Accessed: 2017-05-08.

[19] P. Benáček, V. Puš, and H. Kubátová, *Automatic generation of 100 Gbps packet parsers from P4 description*, www.beba-project.eu/papers/CESNET_p4.pdf, Accessed: 2017-05-08.

[20] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, "Leaping multiple headers in a single bound: wire-speed parsing using the Kangaroo system," in *INFOCOM, 2010 Proceedings IEEE*, Mar. 2010, pp. 1–9. DOI: 10.1109/INFCOM.2010.5462139.

[21] V. Puš, L. Kekely, and J. Kořenek, "Low-Latency Modular Packet Header Parser for FPGA," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ACM, 2012, pp. 77–78.

[22] L. Deri and S. Suin, "Effective Traffic Measurement using ntop," *IEEE Communications Magazine*, vol. 38, no. 5, pp. 138–143, May 2000, ISSN: 0163-6804. DOI: 10.1109/35.841838.

Appendix

Let $G_1 = (V_1, E_1)$ be a weighted and directed graph used to denote a protocol parse graph. The vertices of G_1 represent protocols and the edges represent a layer interface / provider-to-user service relationship from a parent node to a child node. The weight of an edge represents the number of packets estimated to pass along the edge. The edge weights allows us to perform load balancing across pipeline stages. One such parse graph is shown in Figure 7.

For now, ignore the cross-generational edges, back edges and errors. Each custom protocol analyzer (vertex of G_1) also has a processing cost associated with it. Now, our pipeline with a Pub-Sub architecture can be thought of as complete graph. In general, we can have the following properties for one stage of a pipeline.

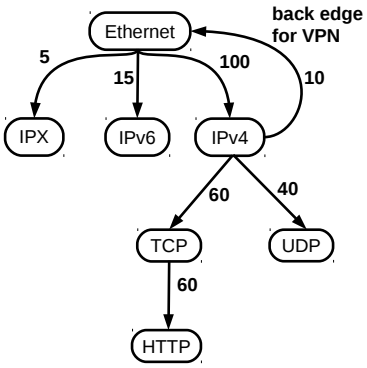


Fig. 7. Sample parse graph with edge weights

C_{v_i} Cost of processing for a protocol node v_i.

C_{p_j} Capacity of a pipeline stage j. We use the notion of capacity to restrict the number of vertices of G_1 that can reside in a pipeline stage.

w_x Cost of communication between two vertices that are part of the same pipeline stage.

w_y Cost of communication between two vertices that are part of different pipeline stages.

Thus a pipeline with N stages is equivalent to a complete graph with N vertices (K_N). Let's represent the complete graph using G_2. Thus

$$G_2 = (V_2, E_2)$$
$$E_2 = \{w_{x_i}, w_{y_j}\} \quad \forall i, j \in N$$

Implementing a parse graph on a pipeline is equivalent to graph embedding of G_1 in G_2. We wish to minimize the number of stages in a pipeline and the communication cost of the created pipeline. We also have a constraint on our ability to assign protocols (nodes in G_1) to a pipeline stage (node in G_2). The constraint is placed by the capacity of the pipeline itself.

The equivalent optimization problem can be formulated as:

Objectives:
$$min\ N = |V_2|$$
$$min \sum_{\forall i,j} (w_{x_i} + w_{y_j})$$
$$\text{OR } min \sum_i w(e_i)$$

Constraints:
$$f_v : v_i \to v_j \quad \forall v_i \in V_1 \text{ and } v_j \in V_2$$
$$f_e : e_i \to e_j \quad \forall e_i \in E_1 \text{ and } e_j \in E_2$$
$$\sum C_{v_i} \leq C_{p_j} \quad \forall f_v(v_i) = v_j \text{ and } v_i \in V_1, v_j \in V_2$$

In simple terms, we wish to map all vertices and edges of graph G_1 to a minimal complete graph G_2 with minimum cumulative edge cost.

This problem has network embedding and bin packing properties. This problem is similar to virtual network embedding (VNE) problem. The bin packing problem is known to be NP-hard. Hence this problem must also be NP-hard.