BITS Darshini: A Modular, Concurrent Protocol Analyzer Workbench

Prasad Talasila, Mihir Kakrambe, Anurag Rai,
Sebastin Santy, Neena Goveas, Bharat M. Deshpande
BITS Pilani, KK Birla Goa Campus
Goa, India
{tsrkp,f2011071,f2013693,f2015357,neena,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 concurrent pipeline architecture, persists the analysis to a database and provides the analysis results via a REST API service. We formulate the problem of mapping protocol parse graph to a concurrent 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 secure collaboration and consultations with experts.

CCS CONCEPTS

• **Networks** → **Network measurement**;

KEYWORDS

Network measurements, concurrent packet analysis, graph embedding, measurement workbench, packet analyzer, collaborative analysis, protocol parse graph

ACM Reference Format:
Prasad Talasila, Mihir Kakrambe, Anurag Rai, Sebastin Santy, Neena Goveas, Bharat M. Deshpande . 2018. BITS Darshini: A Modular, Concurrent Protocol Analyzer Workbench. In *ICDCN '18: 19th International Conference on Distributed Computing and Networking, January 4–7, 2018, Varanasi, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3154273.3154316

ICDCN '18, January 4–7, 2018, Varanasi, India

ACM ISBN 978-1-4503-6372-3/18/01…$15.00
https://doi.org/10.1145/3154273.3154316

1 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 data sets 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, 9] 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 [10] 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.

Table 1: A comparison of Darshini with other packet processing tools.

Parameter	BITS Darshini	tshark	Wireshark	ntopng	BroIDS
Maintenance of measurement meta-data	✓	✗	✗	✗	✗
Collaboration	share analysis, pcaps hidden	⟵ share pcaps ⟶		share analysis	share logs
Persistence	Elastic Search (ES) DB	✗	✗	HTML/MySQL/ES	logs
Protocol selectivity for analysis	user-defined parse graph	✗	✗	only app-layer protocols	BroScript
Concurrency	multi-threaded, multi-core	✗	✗	✗	✗
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,

(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 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 1.

2 RELATED WORK

In Darshini, we utilize the graph embedding for creating an implementation of protocol parse graph. Graph embedding is a familiar concept in the context of virtual networks [11]. In virtual networks, user networks are embedded in the substrate network offered by the network service providers. The problem of virtual network embedding (VNE) as it appears in data networks has been proven to be an $\mathcal{NP}$-hard[11]. However, researchers have produced heuristics-based algorithms for solving the VNE problem in real-time [12]. We use a similar approach to formulate the graph embedding problem for packet parsers. Even though there are significant overlaps between VNE and graph embedding problem for packet parsers, there are quite a few differences as well. One major difference is in the mapping of links. In VNE, the links of user networks are mapped onto the physical paths of provider networks. In packet parsers implemented in a machine / cluster, an edge of protocol parse graph get mapped to another edge of the provider graph.

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

2.1 Parse Graph

The idea of protocol parse graph is a very old [13, 14]. 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 [15], and onto the commercially available FPGA architectures [16]. Examples of software implementation for the parse graphs are: Ntop[1], Wireshark [10] and tcpdump [17].

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. A variation of the programmable protocol parse graph called Berkeley Packet Filter (BPF) is used in the tcpdump [17]. In tcpdump the parse graph is typically used

to select packets of interest. Unlike previous implementations which use fixed parse graph for packet analysis, in Darshini we implement a programmable parse graph for packet analysis.

Darshini parses the incoming packets in order to analyze the protocol stack of the packet. P4 language [18, 19] presents a parse graph notation that is suitable for both hardware and software implementations. We use P4 language to represent the protocol parse graph.

2.2 Packet Parsers

There have been attempts to implement a pipeline protocol parse graph to enhance the protocol analysis throughput. The architectural solutions proposed by [15], [16], [20] and [21] are some of the pure hardware implementations of the packet parsers.

We adopt the Generic Protocol Parser Interface (GPPI) of Benáček et al. [20, 22] for our design of generic analyzer cell (GAC). The high frequency extractor (HFE) M2 and the GPPI discussed by Benáček et al. [20, 22] together form a serially connected pipeline implemented in hardware. Wireshark [10] implements parse graph in software without pipelines. Our analyzer pipeline is a complete 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.

2.3 System Architecture

Packet capture and protocol analysis software tcpdump, tshark and Wireshark have been developed as stand alone packet processing utilities. Ntop [23] is a web application with dynamic plug-in system for customization of persistence, 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.

3 PROTOCOL PARSE GRAPH

A sample protocol parse graph is shown in Figure 1. The labels of vertices represent protocols and the direction of edges represent the provider-user relationship between protocols. The arrow points away from the provider protocol towards the user protocol. The protocol data units (PDUs) of user protocol become payload of provider protocol. The edge weights represent the number of user protocol PDUs expected as payload of provider protocol PDUs.

Let $G_1 = V_1, E_1$ be a rooted, labeled tree. G_1 is used to represent protocol parse graph. We use the following notations to represent different aspects of the protocol parse graph in terms of weighted version of G_1.

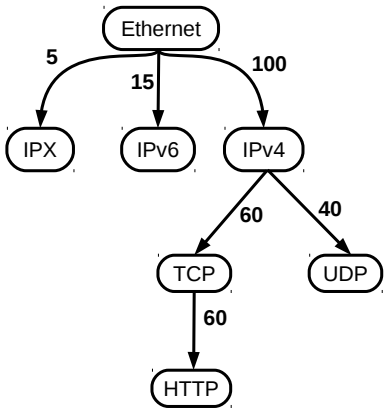


Figure 1: Sample protocol parse graph with edge weights

$$G_1 = V_1, E_1 \quad (1)$$

where,

v_a = a vertex in V_1, $a \in 1, |V_1|$

C_{v_a} = weight of the vertex $v_a \in V_1$

e_b = simple, directed and weighted edge of G_1

w_b = weight of the edge $e_b \in E_1$

The weight function, $f_{w1} : E_1 \rightarrow N$

The set intervals are on the set $\mathbb{N}$.

4 PARSING PIPELINE

4.1 Motivation

We are interested in faster processing of captured packets by increasing the throughput of a packet analyzer. One of the ways of achieving higher throughput is to use concurrency. We can strive for concurrency at the level of protocols. If we have enough processing resources in the form of parallel processors, all the protocols of the protocol parse graph can be processed in a concurrent mode. If not, we strive for maximum possible concurrency. Since the protocol parse graph is a weighted and rooted tree, we architect the concurrent solution to respect the partial ordering of protocols implied by the protocol parse graph. The incoming packet traffic need to be processed in the strict ordering of protocols implied by the packet headers. Thus our concurrent solution will end up as a pipeline of protocol parsers. The interconnections among the pipeline stages are controlled by the edges of the protocol parse graph. One distinct advantage of the concurrent solution is the concurrent analysis of multiple packets.

In a simple sequential execution, one packet is handled at a time; packet analyzer tools such as tshark, tcpdump take this approach. At the other extreme is the concurrent solution in which one protocol is allocated to one concurrent pipeline stage and the pipeline stages are connected as per the edges of the protocol parse graph. But that would result in too many pipeline stages and is not ideal for software environments

where cost of implementing a pipeline stage is high. In order to assign the task of parsing a protocol to a pipeline stage, the processing capacity of the pipeline stage must be greater than or equal to the requirements of the protocol. The aim is to minimize the number of pipeline stages to complete the work; Within this constraint, we need to minimize the cost of distributing packets between the pipeline stages.

4.2 Pipeline as Graph

Consider a complete graph of all pipeline stages with self-loops for each of the stages. A graphical representation is shown in Fig. 2. We use this graph to represent the pipeline stages.

We implement the communications / connections among the pipeline stages in software using *Pub-Sub* design pattern. In the subsequent discussion, we consider mapping of protocol parse graph to K_N pipeline with self-loops.

The problem to be solved is mapping of the protocol parse graph onto a completely connected pipeline. Thus we need to map one or more protocols onto each pipeline stage. If two protocols mapped to a single pipeline stage have a parent-child relation, then we need to facilitate communication within one stage of a pipeline; such a communication is enabled by self-loops of a pipeline stage. If the same two protocols sharing parent-child relation are mapped to different pipeline stages, then we need to facilitate communication between pipeline stages. We can define a pipeline as follows.

$$
\begin{aligned}
G_2 &= P, E_2 && (2)\\
&= \text{Complete graph } K_N \text{ with self-loops for all vertices}\\
&= \text{graph representing all pipeline stages}\\
\textit{where},&\\
N &= \text{order of the graph } G_2 = |P|\\
&= \text{number of stages in the pipeline}\\
p_i &= \text{a vertex of } G_2,\ p_i \in P\\
&= i^{th} \text{ stage of a pipeline, } i \in 1, N\\
C_{p_i} &= \text{weight carrying capacity of a vertex } p_i \in P\\
&= \text{processing capacity of } i^{th} \text{ pipeline stage.}\\
e_{ij} &= \text{weighted, simple edge between vertices } p_i \text{ and } p_j,\ \forall i, j \in 1, |P| \text{ and } i \neq j\\
w_{ij} &= \text{weight of } e_{ij}\\
e_{ii} &= \text{weighted, self-loop at vertex } p_i,\ i \in 1, |P|\\
w_{ii} &= \text{weight of } e_{ii}
\end{aligned}
$$

For the sake of simplicity, assume

$$w_{ij} = w_1 \quad \forall i, j \in 1, |E_2| \text{ and } i \neq j \qquad (3)$$

$$w_{ii} = w_2 \quad \forall i \in 1, |E_2| \qquad (4)$$

The weight function,
$f_{w_2} : E_2 \rightarrow \{w_1, w_2\}$, where $\{w_1, w_2\} \subset \mathbb{R}^+$

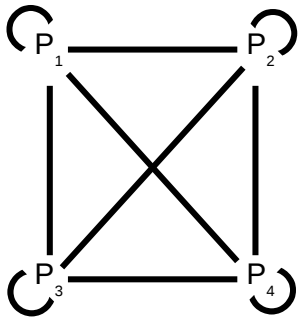


Figure 2: Completely connected (K_N graph with Self-loops) Pipeline. Here N indicates the number of vertices / pipeline stages. In this figure, we illustrate K_4.

5 GRAPH EMBEDDING: PARSE GRAPH TO PIPELINE MAPPING

In this section, we represent the graph embedding problem as manifest in parse graph to pipeline mapping. We can represent the details of mapped protocols as follows.

$$
\begin{aligned}
\text{Let } {v_a}^i &= \text{vertex } v_a \in V_1 \text{ of } G_1 \text{ mapped to } p_i\\
&= \text{a protocol assigned to } i^{th} \text{ pipeline stage}\\
{V_1}^i &= \text{all vertices of } V_1 \text{ that are mapped to } p_i\\
&= \text{all protocols that are assigned to } i^{th} \text{ pipeline stage}\\
\text{then,}&\\
\sum_{\forall v_a \in {V_1}^i} C_{v_a} &= \text{cumulative weight of all vertices of } {V_1}^i\\
&= \text{total cost of processing for all the protocols assigned to } i^{th} \text{ pipeline stage}
\end{aligned}
$$

For realistic mapping, the cost of processing for all the protocols assigned to one pipeline stage must be less than the processing capacity of that pipeline stage.

$$\sum_{\forall v_a \in {V_1}^i} C_{v_a} \leq C_{p_i} \qquad (5)$$

The cumulative weight of all vertices assigned to p_i of G_2 must be less than or equal to the weight bearing capacity of p_i. Extending the same reasoning to all the vertices of G_2, we can say that,

$$\sum_{\forall v_a \in V_1} C_{v_a} \leq \sum_{\forall p_i \in P} C_{p_i} \qquad (6)$$

If all pipeline stages have the same weight bearing capacity C_p, then N – the order of G_2 – is indicated by,

$$N \geq \lceil \frac{\sum_{\forall v_a \in V_1} C_{v_a}}{C_p} \rceil$$

The upper limit on the number of pipeline stages is imposed by the order of G_1, i.e., $N \leq |V_1|$.

Thus the limits on N are:

$$\lceil \frac{\sum_{\forall v_a \in V_1} C_{v_a}}{C_p} \rceil \leq N \leq |V_1| \qquad (7)$$

Implementing a parse graph on a pipeline is equivalent to graph embedding of G_1 onto G_2. When we embed G_1 onto G_2, we need to transform the edge weights of G_2. Let the pipeline graph after edge weight transformation be identified as G_2'. The only difference between G_2 and G_2' is their edge weights; in all other aspects, G_2 and G_2' are identical. Therefore,

$$f_v : V_1 \rightarrow P \tag{8}$$

$$f_e : E_1 \rightarrow E_2' \tag{9}$$

$$f'_{w_2} : \{w_b\} \times \{w_1, w_2\} \rightarrow \mathbb{R}^+ \quad \text{where } b \in 1, |E_1| \tag{10}$$

Both f_v and f_e represent *onto* functions.

The weight of an edge in E_1 (of G_1) indicate the frequency of using an edge; the weight of an edge in E_2 (of G_2) indicate the cost of using an edge. When we map $e_b \in E_1$ onto $e_{ij} \in E_2'$, we indicate the use of e_{ij} exactly w_b times, each use incurring a cost of w_{ij}. For the case of multiple edges of E_1 being mapped onto one edge in G_2', the effective edge weights get added up. We can express the edge weights of G_2' as follows.

$$w'_{ij} = w_{ij} \times w_b \qquad \forall\ e_b \in E_1 \cap {V_1}^i \times {V_1}^j \text{ and } i, j \in 1, |E_2|$$

In protocol analysis, we wish to minimize the number of pipeline stages and the overall communication cost in the pipeline. On G_2', we wish to minimize both the order of the graph (N) and the sum of all edge weights ($_{\forall i,j \in 1,|E_2|} w_{ij}$). Obviously, any mapping has to satisfy the capacity constraints expressed by the Equation 7.

In summary, our graph embedding problem can be formulated as the following optimization problem.

Problem Statement: Embed a rooted, labeled, weighted tree G_1 onto a weighted, complete graph with self-loops G_2' such that

Objectives:

- min N = order of G_2'
- min $_{\forall i,j \in 1,|E_2|} w'_{ij}$ = sum of edge weights of G_2'

Constraints

- capacity limits, $_{\forall v_a \in V_1} C_{v_a} \leq {}_{\forall p_i \in P} C_{p_i}$
- node mapping, $f_v : V_1 \rightarrow P$, f_v is an *onto* function
- edge mapping, $f_e : E_1 \rightarrow E_2'$, f_e is an *onto* function

We wish to embed G_1 onto G_2' with the aim of G_2' having the smallest order and least cumulative edge weight.

In this paper, we describe Darshini which demonstrates an implementation of G_1 onto G_2' graph mapping. We allow for modification of protocol parse graph (G_1). Removal of a vertex from G_1 leads to automatic removal of the mapped element from G_2'. Sections 6 to 8 describe an implementation G_2' as a concurrent (multi-threaded) pipeline.

6 SYSTEM ARCHITECTURE

6.1 Preliminaries

In this work, 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* [15]. 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.

6.2 System Overview

An outline of the system architecture is shown in Figure 3. 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 beautification files are used to represent human preferences for representation of protocol header fields (for example, the use of dotted decimal notation for IPv4 addresses). 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. Our 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

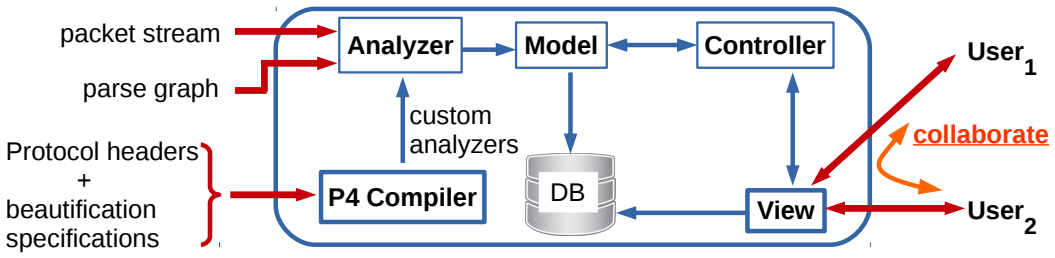
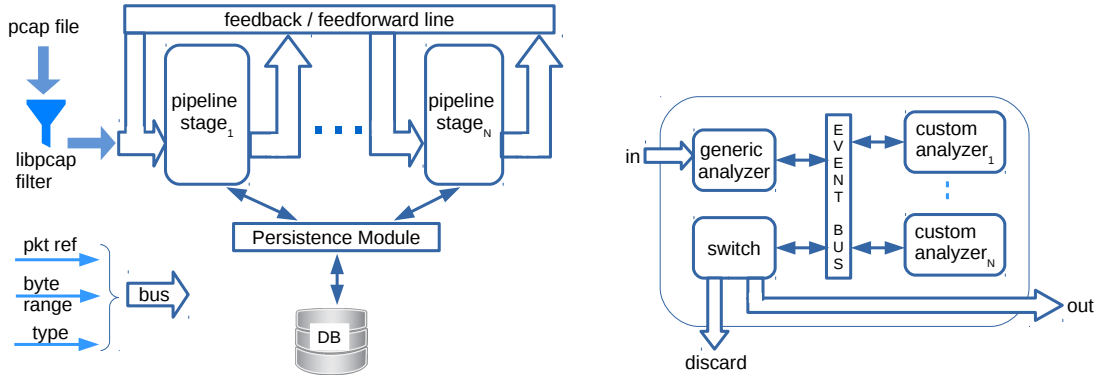


Figure 3: System architecture of Darshini.



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

(b) Generic analyzer cell (GAC)

Figure 4: Protocol analyzer pipeline. Each stage of the pipeline consists of one GAC which is illustrated in part-(b).

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.

6.3 Analyzer Pipeline

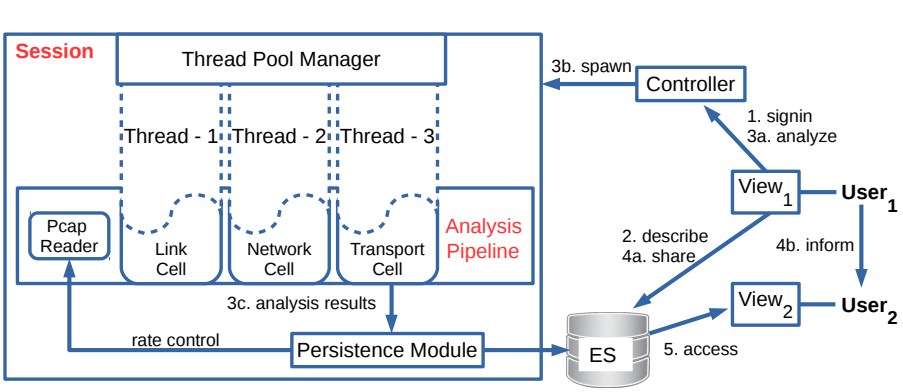
Analyzer pipeline forms the backbone of Darshini. Analyzer pipeline is responsible for taking in a filtered stream of packets and analyzing these packets as indicated by the user-specified parse graph. Analyzer pipeline stores the analysis results in DB via the persistence module. The analyzer pipeline architecture is shown in Figure 4a.

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.

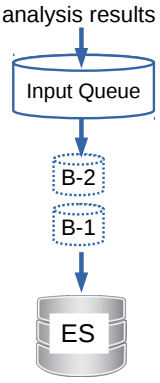
6.4 Generic Analyzer Cell (GAC)

All the pipeline stages are created from the same template named GAC. The block diagram of the GAC is shown in Figure 4b.

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



(a) Session creation for one run of protocol analysis.



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

Figure 5: Implementation details of the system architecture for Darshini. ES is an acronym for Elastic Search database.

the available packet. An appropriate custom analyzer picks up the packet to extract protocol headers. 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.

7 IMPLEMENTATION

Darshini has been implemented as a model - view - controller (MVC) architecture based web application. Figure 5 shows the modules of Darshini. If we compare the proposed architecture with graph G_2', then the following parallels exist.

$$\begin{aligned} p_i &\leftrightarrow \text{GAC} \\ e_{ij} &\leftrightarrow \text{feedback / feedforward line} \\ e_{ii} &\leftrightarrow \text{Event Bus in GAC} \end{aligned}$$

A brief description of different modules of Darshini is given below.

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. 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 2.

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 5a. The sequence of steps involved in one protocol analysis request are illustrated in Figure 5a.

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 5b. Analysis data from custom analyzers is put into batches and handed over to Elastic Search.

Table 2: 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

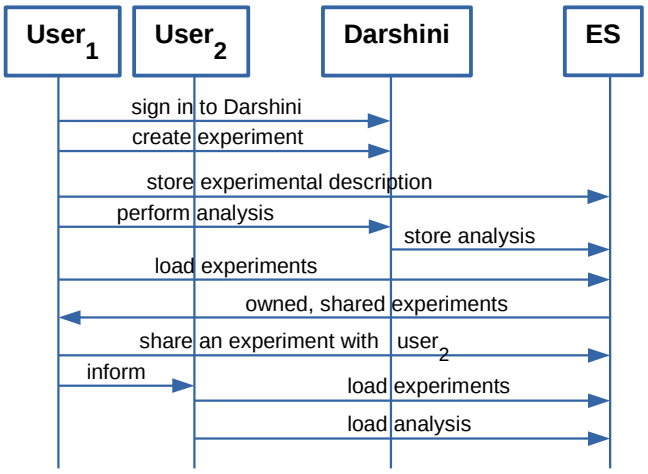


Figure 6: Sequence diagram illustrating collaboration inside BITS Darshini.

7.1 Collaborative Analysis

Darshini enables users to share experimental results with other users. A typical sequence of actions taken for collaboration within Darshini are illustrated in Figures 5a and 6.

User signs into Darshini and creates a new experiment; User is also responsible for supplying experimental description (meta-data). The experimental description gets stored in Elastic Search (ES) database. User then initiates experimental analysis; controller component of Darshini receives user's command and spawns an analysis pipeline to complete protocol analysis. The analysis pipeline persists the analysis results in ES. ES makes the analysis results available to clients via REST API. A user can preview the results of all the analyses done previously. Users 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.

Table 3: 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

7.2 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. 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.

7.3 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 (see Table 3) using the measurement strategies suggested by Vern Paxson [3].

8 EXPERIMENTAL RESULTS

8.1 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. Wireshark and tshark tools show similar performance results, hence for the sake of brevity, only the performance of tshark is discussed in this section. 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.

8.2 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 limited 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 4.

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 limited 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 improve the run time performance from Darshini in persistent mode by optimizing the database storage performance of Elastic Search.

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

Table 4: 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

Table 5: 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

memory allocation to Java Virtual Machine (JVM) and Tomcat Servlets. Thus Darshini is more memory efficient when compared with tshark.

8.3 Memory Management

We can 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 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.

8.4 Throughput

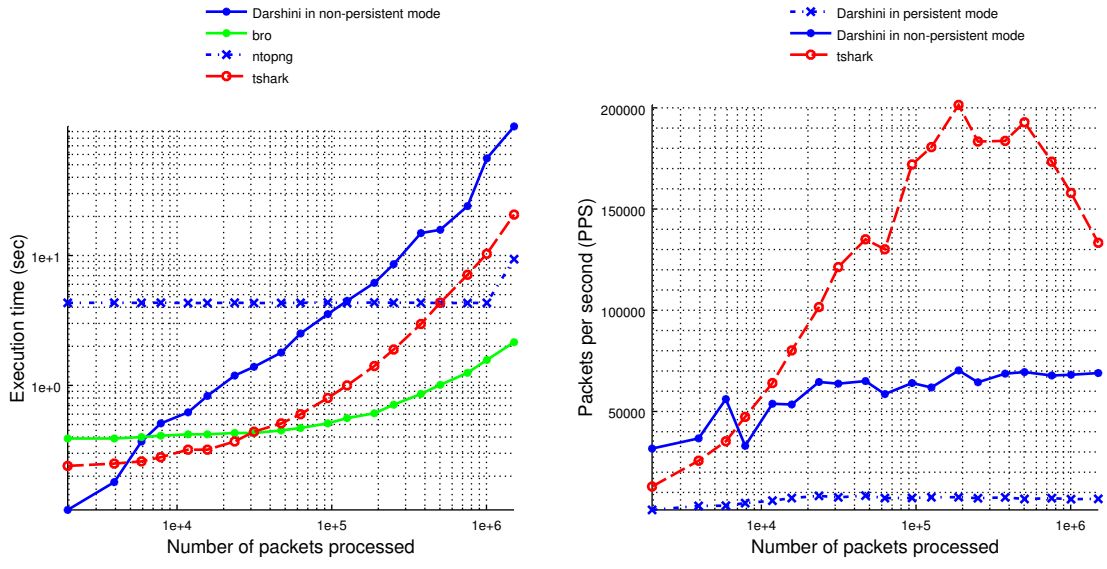
Figure 7 shows the relative performance of packet parsing tools. We compare the tools based on their execution time and memory consumption. Fig 7a shows the relative execution times of all the packet parsing tools for different pcap files. Figure 7b compares packet processed per second (PPS) metric of Darshini with tshark. The PPS performance of ntopng and Bro are significantly better than Darshini and tshark. If shown, the results of ntopng and Bro would have compressed the now visible dynamic range between Darshini and tshark.

As expected, Darshini performs better in non-persistent mode when compared with persistent mode. Another way of looking at this performance differential is the number of packets processed by the analyzer pipeline vis-á-vis complete application. While analyzer cells consistently process around 51,000 to 66,000 packets per second (PPS) for a range of pcap file sizes, application performance consistently hovers around 6,500 PPS. These performance numbers are independent of packet sizes.

The range of throughput numbers achievable by Darshini are shown in Table 5. 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.

9 CONCLUSION

We propose a concurrent, modular and scalable solution to packet processing. We start with a mathematical formulation of packet processing as a graph embedding problem. We propose a modular, scalable architecture as a heuristic solution to the graph embedding problem. We realize the architecture in our software tool named BITS Darshini with the help of a modular, concurrent architectural element named generic analyzer cell (GAC). The GAC itself has been implemented



(a) Execution time

(b) Packet throughput

Figure 7: A comparison of execution time and packet throughput parameters for different packet parsing tools.

using software threads to provide the necessary concurrency. Multi-threaded implementation of BITS Darshini is capable of scaling up/down with the availability of processing resources, namely processor and memory resources.

Darshini enables users to select protocols of interest for analysis. The protocols of interest are specified using protocol parse graph. 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.

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. Users can share experiments within Darshini. Darshini facilitates scientific network measurements done in a collaborative manner.

SOURCE CODE

The complete source code of BITS Darshini application is available at *https://github.com/prasadtalasila/BITS-Darshini*. The source code is available under GNU General Public License (GPL)-2.0.

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*, 5(6):70–74, 2001.

[3] V. Paxson. Strategies for Sound Internet Measurement. In *ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 263–271, Taormina, Sicily, Italy. ACM, 2004. ISBN: 1-58113-821-0.

[4] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[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] A. Arora and S. K. Peddoju. Minimizing Network Traffic Features for Android Mobile Malware Detection. In *International Conference on Distributed Computing and Networking (ICDCN)*, ICDCN '17, 32:1–32:10, Hyderabad, India. ACM, 2017.

[8] 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*, pages 179–192. Springer, 2015.

[9] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM SIGCOMM Conference on Internet Measurement*, pages 267–280. ACM, 2010.

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

[11] A. Fischer, J. F. Botero, M. T. Beck, H. De Meer, and X. Hesselbach. Virtual network embedding: a survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[12] M. Chowdhury, M. R. Rahman, and R. Boutaba. Vineyard: Virtual Network Embedding Algorithms with Coordinated Node and Link Mapping. *IEEE / ACM Transactions on Networking (TON)*, 20(1): 206–219, 2012.

[13] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. *SIGOPS Oper. Syst. Rev.*, 21(5):39–51, Nov. 1987.

[14] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter 1993 Conference*, USENIX'93, pages 2–2, San Diego, California. USENIX Association, 1993.

[15] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ACM / IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 13–24, Oct. 2013.

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

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

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

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: http://doi.acm.org/10.1145/2656877.2656890.

[20] 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.

[21] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In *IEEE INFOCOM*, pages 1–9, Mar. 2010.

[22] V. Puš, L. Kekely, and J. Kořenek. Low-Latency Modular Packet Header Parser for FPGA. In *ACM / IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 77–78. ACM, 2012.

[23] L. Deri and S. Suin. Effective Traffic Measurement using ntop. *IEEE Communications Magazine*, 38(5):138–143, May 2000.