2021 IEEE International Performance, Computing, and Communications Conference (IPCCC) | 978-1-6654-4331-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/IPCCC51483.2021.9679436

PUFF: A Passive and Universal Learning-based Framework for Intra-domain Failure Detection

Lianjin Ye*†, Qing Li†‡, Xudong Zuo*†, Jingyu Xiao* Yong Jiang*†, Zhuyun Qi†, Chunsheng Zhu‡†
*Tsinghua Shenzhen International Graduate School, Shenzhen, China
†Peng Cheng Laboratory, Shenzhen, China
‡Southern University of Science and Technology, Shenzhen, China

***Abstract*—The increasing amount of network devices brings significant improvement to network quality but is inevitably prone to various failures. The frequent occurrence of link failures and node failures in the real-world network, causing packet losses and delay, calls for more accurate and fast detection methods. Existing network failure detection systems focus on probes and end-to-end metrics, but are limited by overhead on bandwidth or storage. Reliance on specific deployment of monitoring systems on devices like hosts also limits the feasibility and compatibility in general network topology, ignoring the potential of transferring monitoring tasks from hosts to switches. In this paper, we propose *PUFF*, a passive and data-driven network failure detection system based on in-network feature collection in programmable switches and machine learning algorithms. First, *PUFF* explores the potential use of continuous traffic changes to detect node and link failures instead of end-to-end metrics. Second, *PUFF* offers a software-based prototype and compares its performance with the latest passive failure detection methods. Evaluation based on simulation on real-world topology shows that PUFF can detect nearly 90% node failures and 80% link failures with less overhead in a shorter time.**

I. Introduction

The Internet is facing the significant challenge of robustness and reliability due to the ever-increasing scale where devices are inevitably prone to failures which could significantly weaken network QoS and incur revenue penalties. An operating network may suffer 302 failures per link in 30 days[1] and the packet loss caused by network failures seriously degrades the overall throughput and transmission delay[2]. However, the time cost of fault localization for operators is relatively huge, which calls for real-time and faster failure detection.

Troubleshooting intra-domain network failures is non-trivial and requires both *low overhead*, *full failure coverage* and *high accuracy*. Unfortunately, off-the-shelf solutions are far from satisfactory, which can be specified into proactive and passive failure detection according to whether the probe packet is needed. *Proactive failure detection*[3], [4], [5], [6], [7] can detect failures by periodically sending probe packets but strongly relies on moderate number of probe packets. In detail, considerable probes may exert the bandwidth and few probes cannot reflect network quality timely. *Passive failure detection*[8], [9], [10] handles failure detection by collecting end-to-end metrics under failures to infer the hidden failures. However, end-to-end feature collections call for specific monitoring systems and some of them strongly relies on structure of topology[9], leading to limited failure coverage. In addition, end-to-end feature collections also heavily rely on the total sampling period, which limits the speed of detection.

To address the above requirements of overhead, failure coverage and speed, we propose PUFF, a Passive and Universal learning-based Framework for quick intra-domain failure detection. PUFF is based on two key designs that make innovation on data collection and classification methods. First, to achieve *high failure coverage* with *low overhead*, PUFF shifts the monitoring functions from hosts to switches with the help of programmable switches, which speeds up fetching data by requesting switches instead of many hosts. Second, just like other applications of machine learning in network[11], [12], PUFF adopts machine learning algorithm and performs elaborate feature design instead of directly using metrics and successfully reduces the overhead of recording all packet history in switches to compute retransmissions or end-to-end delay. Motivated by NetPoirot[8], which uses the decision tree to troubleshoot failures, PUFF explores the potential of using in-network feature changes before and after failure happens and finds the latent relationship between nodes to infer the network failures by machine learning. In addition, the deployment of monitors (switches with monitoring function) and feature design also empower PUFF to perform well with few monitors and many nodes. Thereby, PUFF performs analysis of effects of machine learning and different parameters in failure detection.

Our contributions are as follows.

- *PUFF* is a passive network failure detection system with machine learning and feature collection on programmable switches instead of end-to-end metrics (§V).
- We build a prototype of *PUFF* and deploy it on BMv2 (§VI). Our code has been released on github[13]. The evaluation shows that our prototype has low memory and bandwidth for in-network data collection (§VII).
- We evaluate *PUFF* in node and link failure in three topologies[14], [15] with two testbeds built on Mininet[16] and find that *PUFF* outperforms ML-LFIL[10] in link failure detection and troubleshoots more than 90% node failures with 0.2 seconds of data and one-tenth of the switches to install monitoring function.

Corresponding author: Qing Li (liq@pcl.ac.cn)

978-1-6654-4331-9/21/$31.00 ©2021 IEEE

Table I
COMPARING PUFF WITH EXISTING METHODS OF NETWORK FAILURE DETECTION

	Type	Failure Coverage	Sampling Period	Bottleneck
BFD	proactive	general	ms	bandwidth
Pingmesh	proactive	data center	10s	storage
007	passive	data center	s	deployment
ML-LFIL	passive	general	s	collection
PUFF	passive	general	ms	-

II. RELATED WORK

Proactive Failure Detection. Based on injected probes, device malfunctions can be traced or detected by the probe packet. Classical solutions such as OSPF [3] and BFD [4] establish periodic dialogues and treat peer as failure if not received packet at the agreed time. There also exists failure detection by analyzing probe packets. Pingmesh[5] keeps periodic packet to detect failures by measuring latency, but cannot locate the specific device and relies on non-trivial deployment overhead. Everflow[6] mirrors the traffic to traceroute failures, while exerting enormous pressure on bandwidth. NetBouncer[7] keeps round-trip packets between the server and the top-layer switch to solve the packet loss rate of the link and detect link failures. In conclusion, proactive failure detection all faces the problem of keeping balance between the time granularity and the overhead of probes injection.

Passive Failure Detection. Passive failure detection collects TCP statistics to locate the failure by abnormal metrics without probes. NetPoirot[8] trains decision-tree algorithms to infer failures based on TCP statistics such as DupACKs. [17] collects metrics from various layers and network-I/O system call delays at end hosts every 10 seconds and applies statistical tests to identify failures. 007[9] votes for the most likely failed device by end-to-end TCP retransimissions in 30s in datacenter network. ML-LFIL[10] adopts machine learning methods to locate link failures based on volume, delay and drop rates between nodes. However, collecting all end-to-end metrics like volume and drop rates strongly relies on the existence of flows between every nodes, which cannot react to the nonstationary network status and the cost of collection heavily limits the speed of passive failure detection. In summary, the current network failure detection systems either introduce large overhead or is limited in general topology, which inevitably compromises their practicality. Table I presents a brief comparison between PUFF and the above representative methods in *type*, *failure coverage*, *sampling period* and *bottleneck*.

III. MOTIVATION

Comprehensive and in-network packet history helps locate the malfunctions. Figure 1 shows how comprehensive packet history helps locate the malfunctions. To simplify illustration, the number i in the circle denotes node i and all hosts only connected to node i. Let $< i, j >$ be the flow from i to j, N_i be the node i and $L_{m,n}$ be the link between m and n.

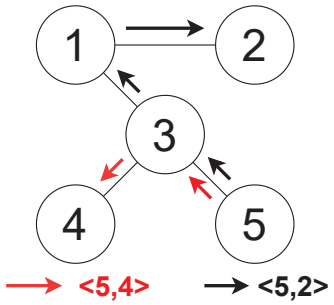


Figure 1. Using packet history to locate failures

If $L_{3,4}$ fails, the combination of $< 5, 4 >$ and $< 5, 2 >$ can correctly locate $L_{3,4}$ because the correctly forwarding of $< 5, 2 >$ excludes the possibility of $L_{3,5}$ failures while in 007[9], $L_{3,4}$ and $L_{3,5}$ have the same retransmission vote, making it different to tell the correct failure. If N_2 fails, N_1 continuously receives the retransmitted packet of $< 5, 2 >$ from N_3 but does not receive the response from N_2. The failure of N_2 can also be easily deduced from N_3 or N_2 if there exists flow of $< 5, 1 >$ or $< 3, 1 >$. Therefore, unlike end-to-end failure detections, moving collection from ends to switches helps find the correlation between shared failures and locate them easily.

Continuous changes in traffic of TCP reflect network failure without resource-consuming end-to-end metrics. Figure 2 shows an example of how traffic changes when one node is down. Let T_i be different moments of one TCP stream, w_i be a series of time windows. At time F, some nodes of the

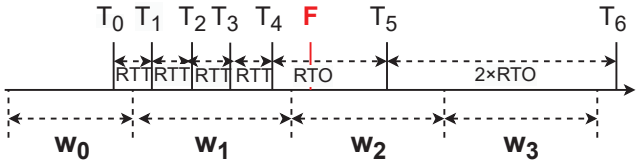


Figure 2. Example of a given TCP stream when one node is down

forwarding path fail. Here we set 4 continuous time windows to illustrate this change. Assuming T_0 represents the very beginning of a TCP stream, From T_0 to T_4, due to TCP's slow start, the volume of packets recorded at the monitors grows exponentially in w_0 and w_1. At T_5, the sender does not receive an ACK within the Retransmission Time Out (RTO) because some nodes fail at F. Then, the sender retransmits packet and doubles its RTO and the volume of packets decreases sharply in w_2 and w_3. From Figure 2, a reasonable setting of w is critical to whether the change in throughput of one TCP stream can be observed. Compared with counting volume, the required time and memory of collecting end-to-end metrics is huge for its reliance on recording Seq (Sequence ID of the packet, 4Bytes) in a sampling period, which requires $\frac{5MB \times 4}{MTU} \approx$ 14KB/s of a flow of 5MB/s. Therefore, changes in traffic imply the changes in end-to-end metrics and may help reduce overhead without sacrifice in effect of detection.

IV. Design of PUFF

A. Key ideas of PUFF

Passive detection of network failures requires *representative and real-time features*, *high failure coverage* and *low collection overhead of bandwidth and storage*. To meet those requirements, we propose two key ideas of PUFF.

In-network metrics collection. Most of existing passive failure detection focuses on end-to-end metrics such as RTTs, lacking information on the forwarding path and can only tell whether failure exists and cannot accurately locate failures. Compared with the overhead of installment of monitoring program on hosts, programmable switches performs lightweight in-network metrics collection by storing the value in registers. However, each Match-Action Units (MAU) of a Tofino switch contains up to two 256KB registers[18], making it hard to keep the complete information to get traditional passive indicators like retransmission.

Flow statistics-based machine learning for Detection. The controller perceives the network status by collecting traffic changes on all monitors. However, the dynamics of network status, such as the arrival or the departure of flow, limits the effectiveness of using volume of counting. PUFF solves this problem by adopting machine learning methods to learn about failures under various network conditions by collecting a large amount of labeled traffic data in continuous time windows.

B. Architecture of PUFF

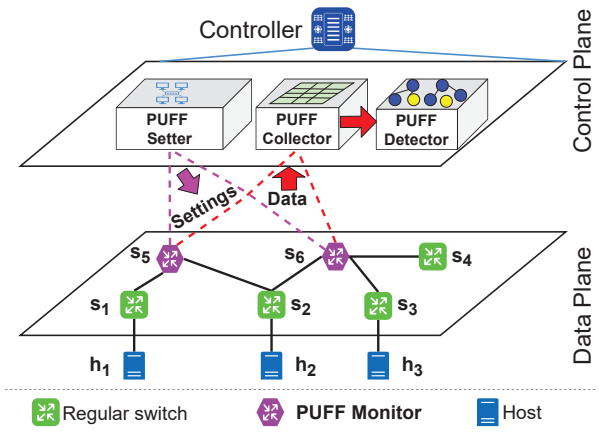


Figure 3. Overview of PUFF

Workflow of PUFF. Overall, the workflow of PUFF consists of four steps in Figure 3. (1) **PUFF setter** deploys the P4 code of the monitoring function to PUFF monitors. (2) **PUFF monitor** executes the compiled P4 code and periodically reports the traffic data to PUFF collector. (3) **PUFF collector** transforms the unstructured original data into tabular data fed to PUFF detector. (4) **PUFF detector** performs various machine learning algorithms to detect network failures.

Settings of PUFF. PUFF settings involve two aspects of functions. Firstly, the PUFF setter needs to set the size of time window and the amount of total time windows according to the topology and the classification effects of labeled data. Secondly, PUFF setter needs to distribute the compiled P4 code to the specific monitors. It seems feasible to send the compiled P4 code to all programmable switches while in reality, programmable switches are limited and the full deployment of compiled P4 codes demands higher cost in network composed of programmable switches only. Therefore, dynamic deployment makes the PUFF mechanism compatible and scalable with existing and increasing network devices.

Table II
Preprocess for each time window t

Feature	Definition
ts_t^i	the volume of TCP packets whose source is h_i
td_t^i	the volume of TCP packets whose destination is h_i
tc_t	the volume of recorded TCP packets
fw_t	=1 when t is the first time window
bs_t^i	$(ts_t^i)/(ts_{t-1}^i)$ if fw_t!=1
bd_t^i	$(td_t^i)/(td_{t-1}^i)$ if fw_t!=1
lw_t	=1 when t is the last time window
as_t^i	$(ts_t^i)/(ts_{t+1}^i)$ if lw_t!=1
ad_t^i	$(td_t^i)/(td_{t+1}^i)$ if lw_t!=1

V. Feature Design and Detection Algorithm in PUFF

A. Feature Design

Step 1: Collecting Features. In this step, the monitor needs to collect node features based on collected flow statistics. Motivated by continuous change of volume, we deduce network situation from the changes in traffic between node and node.

Feature Model. To ensure high accuracy of the failure detection, for each monitored node, we extract traffic features at each time window based on original traffic, as shown in figure II. Let h_i be the hosts connected to switch i, therefore, in each time window, the monitor computes the corresponding value of features for each node. Therefore, node features in different time windows on one monitor can be represented as vector ${N_i}^K$ as defined below:

$$ {N_i}^K = [{N_{i,0}}^K, ..., {N_{i,T}}^K] \tag{1} $$

where K is the monitor ID, i is the monitored node and T is the total window count while ${N_{i,T}}^K$ are the vector composed of features in Table II. We also set other dummy variables for bs_t^i, bd_t^i, as_t^i and ds_t^i to avoid division by zero errors when ts_{t-1}^i, td_{t-1}^i, ts_{t+1}^i or td_{t+1}^i equals zero.

Illustrative Example. We illustrate the changes of our features on one monitor in Figure 2 in the following analysis. We assume that the monitor forwards TCP stream with h_i as the destination or source address, which means that h_i can either be source or destination in Figure 2. Here we also assume node i fails in w_2. In w_1, we can see the increase in bs_t^i and ad_t^i and the decrease in as_t^i and ad_t^i. When the fault occurs at w_2, the flow with h_i as the destination or source experiences timeout retransmission, and the number of packets of these two flows decreases. Therefore, node i's feature changes as

follows: ts_t^i, td_t^i , tc_t^i, bd_t^i and bs_t^i decrease while ad_t^i and as_t^i increase in w_2. In w_3, since node i has failed in w_2, there will not be packets with i as the source and the volume of packets whose destination is i decreases sharply. Therefore, ts_t^i and bs_t^i become 0, while td_t^i , tc_t^i and bd_t^i decrease to 0.

Step 2: Generating Node Feature and Link Feature. Step 2 is to generate node features and link features based on the result of Step 1. In this way, we provide a universal modeling method for both links and nodes to detect failures.

Node Feature. The features collected by each monitor are rearranged in descending order of the distance from the monitor to the monitored node. The principle of this rearrangement is that the closer monitor reduces interference of irrelevant traffic, making its collection of monitored traffic more representative. Therefore, these node features collected on different monitors are fed to machine learning algorithms as vector N_i:

$$N_i = [N_i{}^1, ..., N_i{}^{K_2}] \tag{2}$$

where K_2 is the index of the sorted K monitors while $N_i{}^{K_2}$ represents the result of Step 1.

Link Feature. We get the link features by concatenating the node features of two ends of this link. This is inspired by the probably different behaviors of nodes when link failure occurs. Thus, the link features are fed to classifier as L_i:

$$L_i = [N_{i1}, N_{i2}] \tag{3}$$

where $i1$ and $i2$ means the two ends of this link while N_{i1} and N_{i2} are the two node features.

B. Detection Algorithm

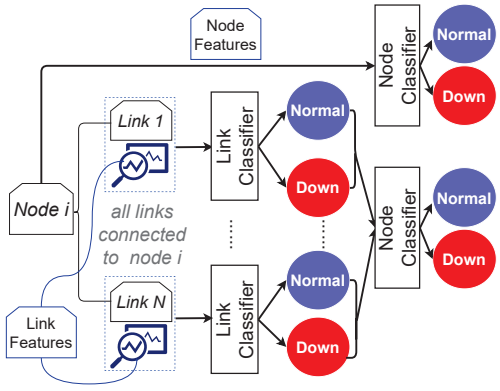


Figure 4. Failure detection model of nodes and links

Figure 4 shows the framework of link detection and node detection. For full failure coverage, PUFF provides **link classifier** and **node classifier** for both two tasks.

Link Classifier. The traffic features collected from the monitors are fed to the machine learning-based link classifier firstly.

Node Classifier. Except for directly using node features for node classifier by machine learning, PUFF also provides another threshold-based node classifier by aggregating the result of link classifier. In this way, we first compute the average value of links that are connected to this node. If this value is greater than the given threshold η, the node is regarded to be a failed node. This method is motivated by two properties: a). When one end of the link fails, its behavior is completely different from the other end. b). all links connected to the broken node are equivalent to failure.

VI. Implementation

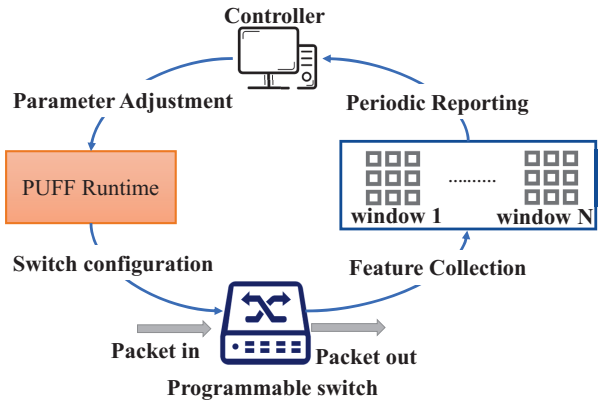


Figure 5. PUFF Implementation

PUFF Control plane. PUFF uses Ryu[19] controller to realize functions including collector and detector. Among them, the machine learning-based detector is realized by scikit-learn[20]. In collector, we implement a SQL-based database to store the processed data. Thereby, the control plane provides interfaces for parameter adjustment in monitoring based on the comparison between various algorithms.

PUFF Runtime. Once the parameters of monitoring are set or updated, PUFF runtime generates the complied P4 codes and distributes them to specific programmable switches.

PUFF Data plane. We implement PUFF using P4[21] (400 lines of code). The overall time synchronization between switches is reached by DPTP[22]. Once packet arrives, PUFF hashes its five tuples to update the corresponding register by computation of ingress_global_timestamp.

VII. Evaluation

Setup. We perform experiments on a server equipped with 2 Intel CPUs (Xeon E5-2643, v4, 3.40 GHz, 6 physical cores) while running Ubuntu 18.04.1. We establish two testbeds on Mininet [16] to test the performance of PUFF. In Testbed A, the continuous traffic between each host is generated with the rates randomly chosen in [0,30Mbps]. In Testbed B, the size of the discontinuous traffic between random hosts obeys the Pareto distribution with 1MB as the 80% quantile and 20MB as the maximum and the packets in the stream conform to the Poisson Pareto Burst Process model[23]. We randomly disconnect one link and let one node down to simulate link and node failures. The setting of Testbed A is roughly the same as [10] to compare the link classification effects. The data of evaluation is simulated under these settings and divided into training set and test set according to the ratio of 4:1.

Objectives. We evaluate PUFF with five objectives.

Table III
TOPOLOGY SETTINGS

Topology	Node	Link	RTT Median
GEANT	40	61	21ms
Tinet	53	88	72ms
AS1221	104	306	28ms

Table IV
COVER FLOW INDEX

	Monitor Counts							
Topology	1	2	3	4	5	6	7	8
GEANT	67.1	85.6	87.6	89.4	89.4	90.4	90.4	90.4
Tinet	64.5	74.6	76.3	76.7	80.4	81.7	82.2	88.0
AS1221	15.8	21.6	26.9	31.6	33.2	36.2	38.5	40.6

- We evaluate the effectiveness of PUFF with few monitors.
- We verify the effectiveness of two-stage feature design and four well-known machine learning algorithms in link failure detection.
- We analyze the effects of different parameters in link failure detection and compare PUFF with the latest end-to-end passive link failure detection in terms of accuracy.
- We evaluate PUFF in accuracy and time in node failure detection in various topologies.
- We evaluate the resource usage of software-based prototype under different traffic settings.

Topology Settings. The basic settings about the topologies are depicted in Table III. The evaluation is completed on three heterogeneous topologies including GEANT, AS1221 and Tinet[14], [15]. We also report the median value of all end-to-end RTTs, which is the unit of the time window for one topology because in RFC6298[24], RTO is closely related with RTT after the first RTT is measured.

Parameter Settings. The specific parameter settings involve K (the number of monitors), w (the size of time window) and T (the amount of time window). The detail of the evaluation can be found in github[13].

Evaluation Metrics. We use the following metrics[25] to evaluate failure detection: (a) Precision: The ratio of the correct classified failures over the total samples classified as failures. (b) Recall: The ratio of the correctly classified failures over the total real failures. (c) Accuracy: The ratio of the correctly classified samples over the total samples. (d) F1-score: The harmonic average of precision and recall values. (e) Fault Localization time: The time to detect failures.

A. Evaluation of monitor deployment

Table IV shows the evaluation of monitor deployment by using the ratio of the amount of flow observed on monitors(%) to the amount of the total flow named COVER FLOW INDEX as an indicator in GEANT, Tinet and AS1221 in Testbed B when K ranges from 1 to 8. When K is 8, the COVER FLOW INDEX has reached 90.4%, 88.0% and 40.6% respectively. This result reveals that few monitors are sufficient to obtain the global view of full traffic and our method of monitor deployment performs well on different topologies while for topologies like AS1221, where most nodes are respectively clustered in some single clusters and connected by finite switches, adding monitors provides less of a benefit. However, the necessary amount of monitors is also small (8), compared to 104 nodes in AS1221 to capture 40% of the flow.

Table V
EXAMPLE OF FEATURE IN GEANT WHEN W=42MS

Node Type	Broken Node			Normal Node		
Position	-1	0	+1	-1	0	+1
ts_t^i	16.4	12.9	9.6	57.3	57.1	59.5
td_t^i	16.6	16.3	11.7	64.2	62.8	65.5
bs_t^i	0.52	0.56	0.51	2.20	1.93	2.72
bd_t^i	0.3	2.3	1.0	2.48	2.54	2.52
as_t^i	1.7	1.9	1.4	3.0	2.4	1.5
ad_t^i	0.64	3.1	1.2	3.4	2.4	1.8

Table VI
EXAMPLE OF FEATURE IN GEANT WHEN W=105MS

Node Type	Broken Node			Normal Node		
Position	-1	0	+1	-1	0	+1
ts_t^i	34.6	25.7	23.9	128.8	149.2	154.3
td_t^i	39.5	33.0	30.0	140.5	156.3	162.5
bs_t^i	1.8	1.7	2.8	6.3	5.2	3.9
bd_t^i	0.7	4.8	6.2	7.7	5.2	3.8
as_t^i	7.8	4.9	6.3	3.4	2.7	2.8
ad_t^i	6.6	7.9	2.6	2.4	2.8	4.7

B. Evaluation of two-stage feature design

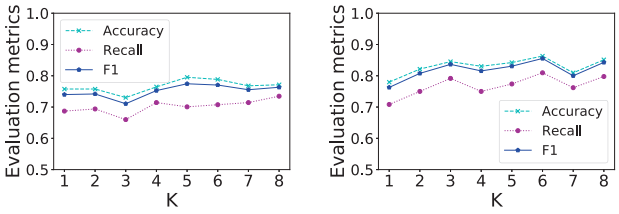
1) Evaluation of feature design: Tables V, VI and VII respectively show the average value of the node features of the broken node and the normal node collected at the nearest monitor when w is $2\times RTT_{median}$(42ms), $5\times RTT_{median}$(105ms) and $10\times RTT_{median}$(210ms). These results are performed in GEANT at Testbed B when the node fails. The position of 0, -1 or +1 indicates that it is the failure window, the last window before or the first window after the failure occurrence window. In Table V and Table VI, both the ts_t^i and td_t^i of the broken node decrease sharply, which conforms to derivation in the illustrative example. However, in Table VII, the ts_t^i and td_t^i of the broken node do not decrease sharply because the larger w also means more possibility of undisturbed packet forwarding.

Table VII
EXAMPLE OF FEATURE IN GEANT WHEN W=210MS

Node Type	Broken Node			Normal Node		
Position	-1	0	+1	-1	0	+1
ts_t^i	68.1	120.4	37.1	194.3	315.9	394.9
td_t^i	69.6	117.4	41.6	209.2	342.5	436.9
bs_t^i	34.9	10.8	2.4	57.5	35.8	12.3
bd_t^i	32.0	13.4	3.2	60.9	36.9	12.5
as_t^i	4.8	53.3	22.8	2.1	5.5	2.6
ad_t^i	2.6	10.3	8.1	1.4	2.5	2.7

Table VIII
COMPARISON OF MACHINE LEARNING ALGORITHMS

Machine Learning Method	Failures	F1-score	Time Per link (in μs)
Logistic Regression	Link	0.75	2.5
SVM	Link	0.80	808
GBDT	Link	0.81	4.4
Random Forest	Link	0.79	12.3
Logistic Regression	Node	0.71	1.3
SVM	Node	0.63	305
GBDT	Node	0.76	7.8
Random Forest	Node	0.73	8.9



(a) GEANT (b) Tinet

Figure 6. Effects of the amount of monitors

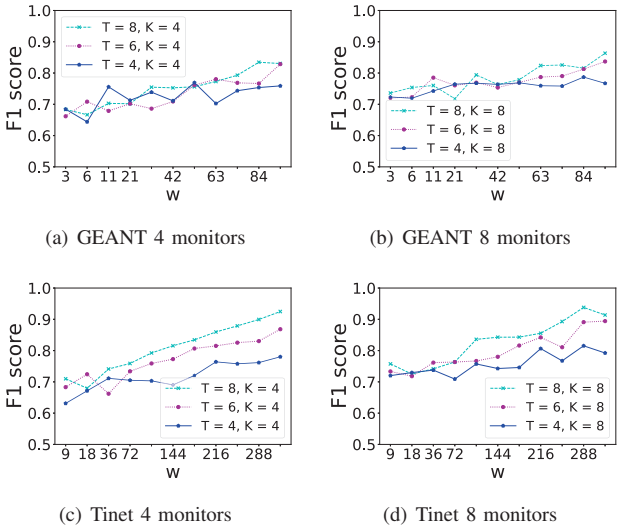
In addition, when w becomes larger, bs_t^i, bd_t^i, as_t^i, and ad_t^i all perform in accordance with the illustrative example. In contrast, this mutation in bs_t^i, bd_t^i, as_t^i, and ad_t^i is not obvious in Table V and VI. The changes of the node features of the normal nodes illustrate the randomness due to the arrival or end of the flow and perform completely different from the broken node, verifying the feasibility of feature design.

2) Evaluation of classifier in detection algorithm: Table VIII shows the F1-score and time per link of PUFF using Logistic Regression[26], SVM[27], GBDT[28] and Random Forest[29] in Testbed A on GEANT[14] when link fails. Here we set K as 4, w as 21ms and T as 8 on nearly 100000 samples. We adopt GBDT as link classifier in PUFF for its feasibility without GPU and performance to handle nonlinear features when link or node fails while SVM needs a longer time to find nonlinear kernels. GBDT also performs better in link failures than node failures, enlightening using threshold-based classifier in node failure detection.

C. Evaluation of link failure detection

1) Analysis of parameters: Figure 6, 7 and 8 show the effect of parameters on the link failure detection of 100000 samples. Figure 6 shows the effects of K on accuracy, recall and F1-score of link failure detection when w is $2 \times RTT_{median}$ in GEANT and Tinet. PUFF exceeds 75% only using one-tenth of the total switches as monitors. K improves link failure detection while this benefit is not obvious in GEANT and becomes smaller when K is larger than one-tenth.

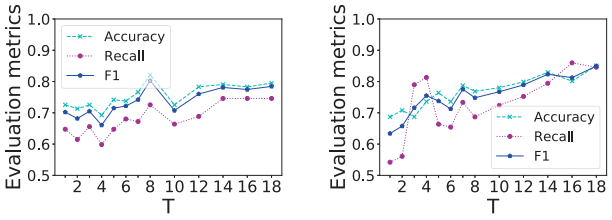
Figure 7 shows the effect of w on link failure detection when K is 4 or 8, T is 4, 6 or 8 and w ranges from $\frac{1}{8} \times RTT_{median}$ to $4.5 \times RTT_{median}$. The F1-score is significantly improved as w increases when w is larger than $1 \times RTT_{median}$. If w



(a) GEANT 4 monitors (b) GEANT 8 monitors

(c) Tinet 4 monitors (d) Tinet 8 monitors

Figure 7. Effects of the size of time window



(a) GEANT (b) Tinet

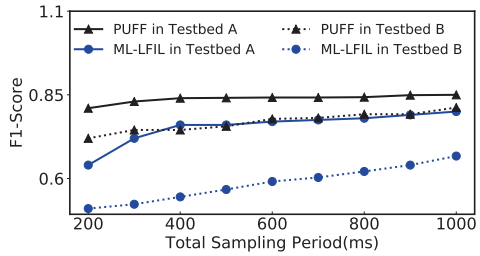
Figure 8. Effects of the amount of time windows

is too small, the flow may not experience retransmission and changes in traffic is too minimal for classifier to detect failures.

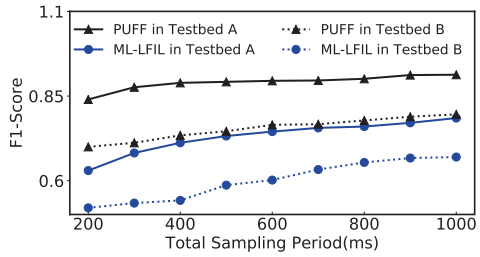
Figure 8 shows the effect of T on the F1-score of link failure detection when K is 8 and w is $1 \times RTT_{median}$ when T ranges from 1 to 18. When T is 1, we only collect the failure window. The F1-score increases with T and becomes slower after T is bigger than 8. In conclusion, enlarging w and T is not always a good choice for the extra cost of capturing. PUFF is more sensitive to w and T and the effect of K varies from topologies.

2) Comparison with end-to-end passive detection: Figure 9 shows the comparison between PUFF and ML-LFIL with Random Forest (the best version) in link failure detection in two testbeds within the same sampling period ranging from 200ms to 1000ms with 100000 datapoints. Here we set K and T as 8, thus w is equal to the sampling period divided by T. In Figure 9, PUFF outperforms ML-LFIL in all sampling periods in both GEANT and Tinet. In Tinet, the F1-score has reached 0.91 when the total sampling period is 800ms. The extra experiment shows that the proposed result of ML-LFIL cannot be reached until the total sampling period is larger than 5s. Therefore, without long-time monitoring of end-to-end traffic, PUFF provides quicker failure detection.

D. Evaluation of node failure detection



(a) GEANT



(b) Tinet

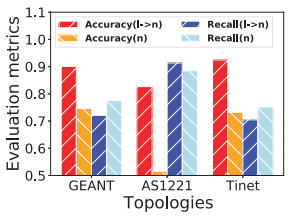
Figure 9. Comparison vs ML-LFIL

Table IX
PROPORTION OF LINKS CONNECTED TO NORMAL NODE BEING LABELED AS BROKEN LINK

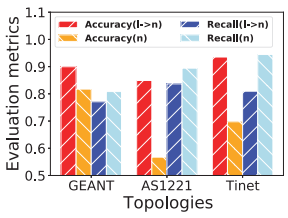
Node Type		Normal Node			
Topology	w(ms)	[0,0.25]	(0.25,0.5]	(0.5,0.75]	[0.75,1]
GEANT	21	0.8	0.1	0.02	0.08
GEANT	42	0.77	0.1	0.03	0.1
Tinet	72	0.75	0.13	0.05	0.07
Tinet	144	0.82	0.10	0.02	0.05
AS1221	28	0.70	0.10	0.03	0.17
AS1221	56	0.73	0.10	0.03	0.14

Table X
PROPORTION OF LINKS CONNECTED TO BROKEN NODE BEING LABELD AS BROKEN LINK

Node Type		Broken Node			
Topology	w(ms)	[0,0.25]	(0.25,0.5]	(0.5,0.75]	(0.75,1]
GEANT	21	0.23	0.04	0.01	0.72
GEANT	42	0.19	0.03	0.01	0.77
Tinet	72	0.22	0.04	0.03	0.71
Tinet	144	0.15	0.02	0.03	0.80
AS1221	28	0.07	0.01	0.01	0.91
AS1221	56	0.10	0.05	0.01	0.84



(a) $w = 1 \times RTT_{median}$,T=8



(b) $w = 2 \times RTT_{median}$,T=8

Figure 10. Results of Node Failure Detection

Table XI
COMPARISON OF FAULT LOCALIZATION TIME

Methods	Task	Time (in μs)
Ping-based approach	link failures	1638000
ML-LFIL	link failures	202
PUFF	link failures	224
Ping-based approach	node failures	114500
PUFF	node failures	249

1) Analysis of threshold: Table IX and Table X show the proportion of the links connected by the broken or normal node as broken link by the link classifier (the output of the threshold-based node classifier) where K is 8, T is 8 and w is $1 \times RTT_{median}$ or $2 \times RTT_{median}$ in Testbed B in GEANT. [a, b) represents that the value is $\geq$ a, and $<$ b. More than 75% of normal nodes' value is in [0, 0.25] in Table X and more than 70% of the broken nodes is in (0.75,0.1] in Table IX. The ratio of (0.25, 0.75] is relatively small in both tables and this value of broken node is lower than the one of normal node. Further research shows that most of the misclassification happens when there is no flow whose source or destination fails in the total sampling period and a smaller or larger threshold causes higher false negative rate or higher false positive rate. In conclusion, the threshold-based node classifier successfully reduces the impact of randomness and the recommended setting of threshold is 0.7.

2) Comparison in accuracy and failure localization time: Figure 10 shows the results of direct and threshold-based node classifiers of PUFF in testbed A in GEANT, AS1221 and Tinet in node failure detection when w is 1 or $2 \times RTT_{median}$, K is 8 and T is 8. The result with (l->n) or (n) respectively represents the result of threshold-based or direct node classifier and shows that the former performs better. The accuracy and recall (l->n) exceeds 80% in Figure 10 (b) in AS1221 with 8 monitors. We can also see decrease in recall(l->n) in AS1221 and accuracy(n) in Tinet, showing that increasing w may also introduce noises in classifiers. Table XI depicts the fault localization time in GEANT of every mentioned method. In ping-based approach, we randomly let half or one of the hosts ping the remaining hosts to detect failures. Table XI shows that PUFF is slower than ML-LFIL while ML-LFIL needs time to collect data from hosts and the time cost of PUFF does not scale up with the size of network.

E. Resource Usage

Figure 11 shows the resource usage of the monitoring function of PUFF when the transmission rate of per flow ranges from 1 to 10MB/s and the count of flows ranges from 1 to 10 in Testbed A. The black line and the yellow bar show that the memory usage of the monitoring function remains roughly stable at 7MB and the bandwidth usage remains at 73KB/s respectively even when the transmission rate exceeds 5MB/s and the total sampling period is 1s. In addition, the average size of reporting file is 52KB, 66KB and 70KB when the total sampling period is 200ms, 400ms and 1000ms while the one

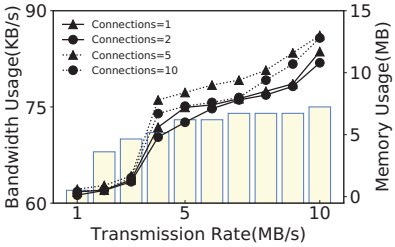


Figure 11. Resource usage of PUFF

captured by tcpdump from one host in ML-LFIL is 1.6MB, 3.1MB and 7.6MB. Obviously, the feature design based on counting empowers monitors to meet with high throughput and successfully reduce overhead of in-network feature collection.

VIII. Conclusion

We present PUFF, a passive and universal learning-based framework for intra-domain failure detection. PUFF successfully explores the potential of shifting network monitoring from ends to switches, reducing the overhead of collection by refined feature design based on TCP and attaining high performance by machine learning algorithms. The experiments of PUFF in various parameters and tasks show that PUFF achieves high failure coverage, fast detection speed and low overhead. Port-level feature, more effective monitor deployment, performance under congestion and deep learning algorithms need to be further studied.

Acknowledgement

This work is supported by the National Key Research and Development Program of China under Grant No. 2020YFB1804704, National Natural Science Foundation of China under grant No. 61972189 and the Shenzhen Key Lab of Software Defined Networking under grant No. ZDSYS20140509172959989.

References

[1] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage, "California fault lines: understanding the causes and impact of network failures," in *SIGCOMM '10*, New Delhi, India, Aug. 2010.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4:experience with a globally-deployed software defined wan," in *ACM SIGCOMM'13*, Hong Kong,China, Aug. 2013.

[3] J. Moy, "Ospf version 2," Internet Requests for Comments, RFC Editor, STD 54, April 1998.

[4] D. Katz and D. Ward, "Bidirectional forwarding detection (bfd)," Internet Requests for Comments, RFC Editor, RFC 5880, June 2010.

[5] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. L. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *ACM SIGCOMM'15*, London, United Kingdom, Aug. 2015.

[6] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM'15*, London, United Kingdom, Aug. 2015.

[7] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "Netbouncer: Active device and link failure localization in data center networks," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19, 2019.

[8] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, Aug. 2016.

[9] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 419–435.

[10] S. M. Srinivasan, T. Truong-Huu, and M. Gurusamy, "Machine learning-based link fault identification and localization in complex networks," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6556–6566, 2019.

[11] G. Xie, Q. Li, and Y. Jiang, "Self-attentive deep learning method for online traffic classification and its interpretability," *Computer Networks*, vol. 196, p. 108267, 2021.

[12] H. Jiang, Q. Li, Y. Jiang, G. Shen, R. Sinnott, C. Tian, and M. Xu, "When machine learning meets congestion control: A survey and comparison," *Computer Networks*, vol. 192, p. 108033, 2021.

[13] L. Ye, J. Xiao, and X. Zuo, "Puff," 2021. [Online]. Available: https://github.com/yelianjin/PUFF

[14] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 9, pp. 1765 –1775, october 2011.

[15] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring isp topologies with rocketfuel," *IEEE/ACM Transactions on Networking*, vol. 12, no. 1, pp. 2–16, 2004.

[16] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX, 2010.

[17] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17, 2017.

[18] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang, "Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 991–1010.

[19] N. Telegraph and T. Corporation, "Ryu network operating system," 2012. [Online]. Available: Available: http://osrg.github.com/ryu/

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[22] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise time-synchronization in the data-plane using programmable switching asics," in *Proceedings of the 2019 ACM Symposium on SDN Research*, ser. SOSR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–20.

[23] M. Zukerman, T. D. Neame, and R. G. Addie, "Internet traffic modeling and future technology implications," in *INFOCOM'03*, 2003.

[24] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing tcp's retransmission timer," Internet Requests for Comments, RFC Editor, RFC 6298, June 2011.

[25] N. Chinchor, "Muc-4 evaluation metrics," in *Proceedings of the 4th Conference on Message Understanding*, ser. MUC4 '92. USA: Association for Computational Linguistics, 1992, p. 22–29.

[26] H. David, W, L. Jr, Stanley, and X. S. Rodney, *Multiple Logistic Regression*. John Wiley & Sons, Ltd, 2000, ch. 2, pp. 31–46.

[27] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, May 2011.

[28] J. Friedman, "Greedy function approximation: A gradient boosting machine," *The Annals of Statistics*, vol. 29, 11 2000.

[29] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, p. 5–32, Oct. 2001.