In-Network Caching Assisted Error Recovery For File Transfers

Nagmat Nazarov and Engin Arslan
Computer Science and Engineering
University of Nevada Reno
{nnazarov,earslan}@unr.edu

***Abstract*—Silent data corruption poses permanent data loss risk for file transfers as receiver server can accept and store corrupted data as genuine. Researchers proposed end-to-end integrity verification to recover from silent errors but recovery process requires entire file to be transmitted, which degrades the performance of transfers significantly. This paper takes advantage of programmable network devices to implement in-network caching for file transfers such that silent errors can be recovered quickly and scarce network bandwidth can be used more efficiently. In the proposed design, data packets are mirrored to cache server as they pass through programmable devices such that when the receiver detects an error, it can retrieve the file from in-network cache instead of downloading from the source again. Our preliminary results show that in-network caching can shorten error recover duration by 50% and improve overall transfer performance by around 20%.**

I. Introduction

Wide area file transfers are susceptible to silent data corruptions that escape existing error detection mechanisms such as TCP checksum. Although some system components have built-in integrity verification mechanisms, they are either weak or available only in a subset of computing facilities. For example, TCP uses 16-bit checksum to capture data corruption, but it fails to detect errors once in 16 million to 10 billion packets [1], which results in frequent missed errors in today's high-speed networks that operate at hundreds of gigabit-per-second speeds. In addition to network, data can also be corrupted during disk read and write operations as disk drives suffer from a significant number of silent data corruptions (aka undetected disk errors) that occur mainly due to firmware or hardware malfunctions in disk drives [2], .

Researchers proposed application-layer integrity verification process to detect silent data corruptions which involves, upon the completion of transfer, the use of secure hash functions such as SHA256 to calculate and compare the checksum of files on destination nodes against the checksum of original files on source nodes. An earlier study showed that nearly 5% of all file transfers in research networks are exposed to silent errors that would have gone unnoticed if application-layer integrity verification was not implemented [3]. Hence, integrity verification plays an important role in protecting file transfers against silent errors.

On the other hand, integrity verification can degrade the performance of file transfers significantly since it requires entire file to be retransmitted even if only one bit is corrupted. Researchers proposed block-level integrity verification that involves checking the integrity of file transfers in smaller segments to avoid transferring very large files in the case of integrity verification failure. Despite reducing the overhead, this approach has a risk of missing silent errors while writing data to disk [4]. Moreover, it still requires the transfer of nonnegligible amount of data from source to destination when integrity verification fails.

In this paper, we propose to take advantage of caching and computing resources deployed inside the network to expedite the error recovery for file transfers. Network service providers (e.g., ESnet) are deploying storage and compute resources to the network alongside network devices that can be allocated by users to optimize their workflows. These resources can be leveraged to cache data in-transit until its integrity is verified. Storing files in cache servers in network along the transfer path between source and destination nodes (i) reduces the overhead on network resources by not transferring large volume of data over the long distances when files need to be retransferred due to silent errors, and (ii) lower the duration of error recovery when available bandwidth between caching site and destination is larger than available bandwidth between source and destination. Programmable network devices offer a unique opportunity for the implementation of in-network caching as we can selectively choose which network flows to mirror and save in the caching server. Hence, in this work, we demonstrate that programmable switches can be utilized to mirror file transfer packets to a nearby cache server which can then construct files and save them until the integrity of transfers is verified. In case error is detected due to checksum mismatch of file at source and destination nodes, files can be streamed from the cache server to destination node to increase the speed of recovery and minimize the impact of recovery process on limited network resources. In summary, we make following contributions:

- We develop an in-network caching mechanism for file transfers. We leverage the programmability of switches to mirror data packets as they pass through the switches to save them in the cache server. We then process the captured packets in the cache server to reconstruct the files.
- We introduce in-network caching assisted error recovery to retrieve files from cache servers when an error is

detected to minimize error recovery time and reduce overhead on network resources.
- We conduct experiments using Tofino EdgeCore P4 switch and show that the proposed method leads to 50% decrease in error recovery process and 20% decrease in total transfer time of file transfers.

II. Related Work

In network caching has been extensively studied by researchers mainly in the context of web data caching to enhance user experience [5]. Content Distribution Network (CDN) achieve this goal by deploying many cache servers close to end users to store popular content such that they can be delivered to users quickly. Named Data Networking (NDN) proposed content-centric networking design to store popular data on network devices (e.g., routers) such that users can access them quickly without the need to contact end hosts [6]. Since cache replacement policy is critical for efficient execution of NDN, researchers investigated various caching policies [7], [8], [9].

Thomas et al. analyzed the feasibility of using NDN to improve the recovery time for network transmission errors [10]. They presented expected improvement rates for error recovery based on network configuration and the location of errors between source and destination. As anticipated, the error recovery time can be significantly reduced when errors happen near receiver nodes since cached packets can be retrieved from one of the nearby network devices. However, they show that existing memory technologies (e.g., SRAM and DRAM) used in network devices are unable to support packet-level caching at high speeds due to capacity and access time limitations. Moreover, packet-level caching cannot be used to recover from silent errors since it requires data packets to processed before caching them to ensure that they are not exposed to silent errors, which may not be feasible using network devices due to lack of computational power.

Globus transfer service implements integrity verification for file transfers and overlaps transfer and checksum compute operations to minimize the overhead [11]. Liu et al. proposed dividing large files into blocks to improve pipelining (i.e., block-level pipelining) of transfer and compute tasks for mixed-size datasets [12]. Despite reducing the execution time considerably, it requires careful tuning of block size to perform well. Moreover, it tries to overlap the transfer operation of one file with the checksum computation of another file, thus incurring extra I/O overhead due to reading files twice; one for transfer and the other for checksum computation. In a previous work, we proposed the Fast Integrity VERification (FIVER) algorithm to pipeline the transfer and checksum operations for the same file to enable I/O sharing between thereby reducing system overhead [13], [14]. FIVER outperforms Globus and block-level pipelining by reducing the overhead of integrity verification from up to 60% to less than 10%.

In another work, we introduced Robust Integrity Verification Algorithm (RIVA) to enhance the reliability of integrity verification process by detecting and recovering from receiver-side disk errors that can cause corrupted data to be accepted [4]. RIVA enforces checksum operations to read files directly from the disk to capture undetected disk write errors. We compared RIVA against state-of-the-art end-to-end integrity verification algorithms in terms of robustness to capture error injections during disk write operations for various file size when using transfer nodes with a 16GB RAM. Both FIVER [13] and BlockLevelPpl [12] failed to detect injected errors for all file sizes as they always read files from page cache during checksum calculations. On the other hand, RIVA captured all fault injections regardless of file size by invalidating cache copies of file pages before starting checksum computation. However, all of existing integrity verification solutions require data to be retrieved from the source, which can induce significant overhead to network in addition to slowing down the transfer completion. Hence, this work takes advantage of in-network storage capacity to cache the file content until its integrity is verified.

III. Background

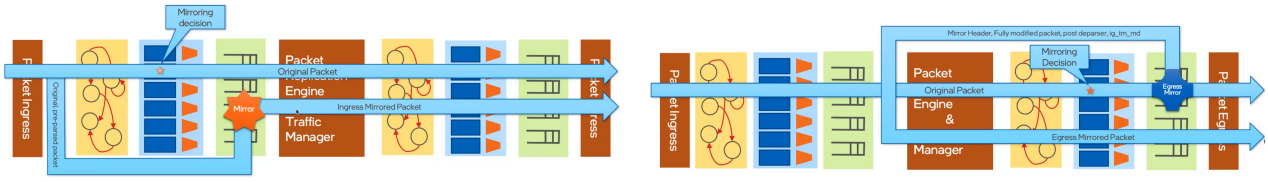
A. Programmable Switches

Software Defined Networking aimed at separating control and data planes to allow complex routing decisions to be taken without the need for modifying switch software. However, data-control plane communication overhead was introduced because switches in the data plane are dependent on the controller for forwarding decisions[15]. Moreover, OpenFlow switches are equipped with a fixed set of functionalities which requires new ASIC to be manufactured even for minor changes in the packet processing architecture. To overcome these limitations, programmable switch architecture is introduced to execute custom actions through match-action pipelines. P4 (Programming Protocol-independent Packet Processors) is a domain-specific language used to implement processing pipelines in programmable devices [16], [17].

The high-speed packet processing on P4 is composed of four main components [18]. First is programmable which identifies and parses packet header fields. Second is programmable match-action pipeline where packet header fields are checked against match table and corresponding actions are performed on the header. Third is deparser which recreates the packet by attaching updated header to payload make it ready for forwarding. Fourth and perhaps the most important point of this whole architecture is a programmable header and metadata bus which carries packet header and intermediate results across processing stages. Parsing and processing only affects the header field, so packet payload is not affected by this process.

B. Mirroring

A packet may need to be mirrored to a port in addition to the set of targets intended by the packet's source. The packets can be mirrored both from ingress or egress ports as shown in Figure 1. If we mirror the packet at ingress port, then the mirrored packet will not be affected by the processing pipeline that the original packet traverses. On the other hand, if the mirroring is done it at egress port, then the mirrored packet



(a) Ingress Mirroring (b) Egress Mirroring

Fig. 1. Illustration of ingress and egress mirroring in programmable switches.

will be the same as the original packet since mirroring is done upon the completion of all processing by the switch.

Ingress Mirroring: On the ingress side, the packet header passes through the ingress pipeline before being appended to the payload to form the modified packet. Although mirroring decision is given during ingress match action pipeline, mirrored packet contains unmodified header. The Mirror_ID field in metadata field is used to indicate whether or not a packet will be mirrored at the ingress pipeline. The mirrored packet is then placed into Mirror Buffer after appending Mirror_ID and some optional metadata fields [19].

Egress Mirroring: Egress pipeline also has a metadata field called Mirror_ID. If this field is valid then the egress pipeline will copy the egress packet together with the Mirror_ID and additional optional metadata fields into the Mirror Buffer. The egress pipeline deparser must be programmed to specify the set of metadata that will accompany the mirrored packet. As a result, the Mirror Buffer can contain an ingress mirrored packet or an egress mirrored packet together with a Mirror_ID field and some additional metadata. The Mirror Buffer then copies packets to Queuing Subsystem. Since Queuing Subsystem needs to know where each packet will be forwarded to, the Mirror Buffer builds a table that maps the Mirror_ID to a tuple that includes the data needed by the Queuing Subsystem as follows:

- Unicast Egress port ID
- Multicast Group ID (1 and 2)
- Congestion Group ID
- Class of Service (Cos)

The packet and its associated metadata is transferred from the Mirror Buffer to the Queuing Subsystem when the ingress pipeline is not transferring a packet to the Queuing Subsystem. In other words, the Mirror Buffer steals unused cycles from the Input Pipeline Deparser in order to copy its packets into the Queuing Subsystem. Mirrored packets enter the Queuing Subsystem with the Is_Mirrored_Packet metadata bit set, which allows egress pipeline to differentiate between original packets and mirrored packets. Deparser terminates the parsed headers and brings original headers from headers and metadata bus, adds the payload of the original packet body, and sends them to the traffic manager. Inside the deparser, we can instantiate mirror extern which has the special method called "emit". It has a special parameter called mirror_session id. Among intrinsic metadata for ingress deparser, there is a special field called MirrorType as shown in Figure 2.

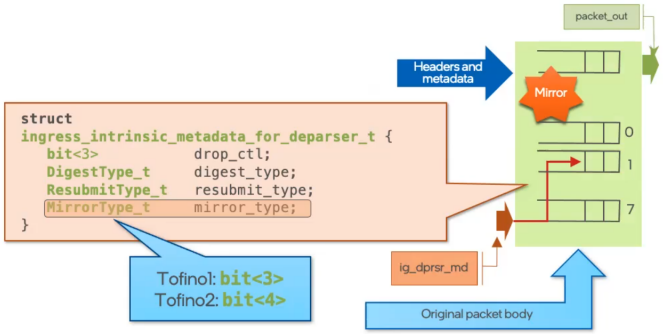


Fig. 2. Ingress Mirror deparser scheme.

We adopted ingress mirroring since we are not interested in any modifications done at match-action pipeline. In Tofino-1 architecture mirrored packets are configured via BRI (Barefoot Runtime Interface) "mirror.cfg" table. It allows up to seven mirror sessions to be created with different configurations, thus we created a custom mirroring session. After creating a session, we add source server's port id along with output port (i.e., cache server's port id) into "p4.Ingress_acl.entry_with_acl_mirror" with "mirror_session_id". Doing so will check the packets in the ingress deparser and mirror them if they match a specified rule. We then update IP header checksum as packet header is updated during the mirroring.

IV. Proposed Model and Preliminary Results

Figure 4 demonstrates the proposed in-network caching architecture. As files are being transferred from the sender to receiver, data packets are mirrored to cache server located near the programmable switch. The packets are captured at the mirror site and processed to reconstruct the file. After the file transfer operation is completed, the receiver servers calculates the checksum for the transferred file and compare it against the checksum sent by the sender. If the checksums do not match, then the receiver sends a request to sender to resend the file. This message is then intercepted by the programmable switch and forwarded it to the cache server to transfer the file from the cache server.

Since cache servers receives packets from programmable switch in raw format, we implemented high-performance data capturing and processing pipeline to reconstruct files from mirror traffic. We use `tcpdump` [20] to capture mirrored

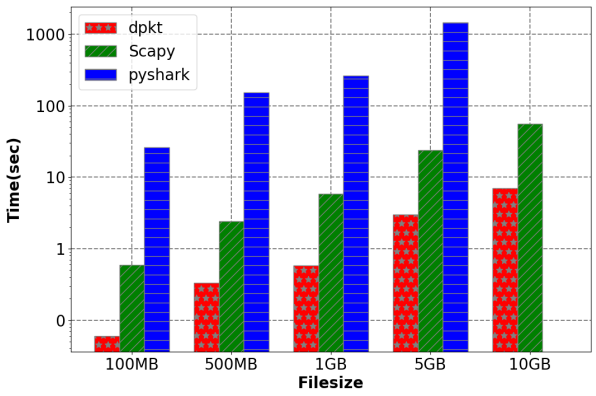


Fig. 3. `dpkt` outperforms `pyshark` and `Scapy` libraries by $10-400\times$ when processing captured traffic (i.e., pcap files) on cache servers to extract original files.

packets on the cache server. Upon the completion of the mirroring, we execute pcap parser to remove packet headers and handle out of order and duplicate packets. Since mirrored data must be ready as soon as an error is detected by the receiver to take advantage of caching, the speed of pcap processing is critical to observe a benefit from using the cache data for error recovery. We evaluated the performance of multiple pcap processing tools for various pcap sizes in Figure 3. Specifically, we tested `pyshark` [21], `Scapy` [22], and `dpkt` [23] libraries. We observe that `pyshark` is the slowest when it comes to parsing pcap files as its performance is more than $400\times$ slower than that of `dpkt`. Also, while attaining better performance than `pyshark`, `Scapy` is $8-10\times$ times slower than `dpkt`.

In addition to yielding the best performance, `dpkt` can also be used to process raw packets as they are being captured by `tcpdump` to avoid incurring additional delay between packet capture and file reconstruction. However, we find that `dpkt` is not fast enough to process raw packets as they arrive to cache server, thus we write the packets into a pcap file and let the `dpkt` to process the pcap file.

To evaluate the proposed in network caching assisted error recovery method for file transfers, we used three servers and a switch (EdgeCore Wedge100BF-32QS) that comes with $32 \times 100G$ ports, 16 Core Intel x86 Broadwell-DE, Pentium D-1517 processor, and 128 GiB SSD storage. The source, destination, and cache servers are connected to the EdgeCore switch with $1G$, $10G$, and $10G$ interfaces, respectively as illustrated in Figure 4. We intentionally kept the bandwidth between source and the switch lower compared to other links to highlight the performance of in-network assisted error recovery method in the presence of bandwidth difference between source-destination and cache server-destination pairs. As in-network resources are widely deployed, this is expected to be a common scenario as wide-area network bandwidth are highly likely to be lower compared to bandwidth between end hosts and nearby network switches.

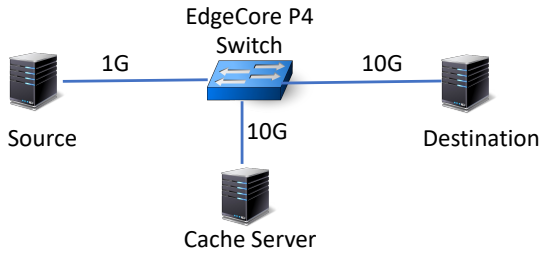


Fig. 4. Network topology of test environment. While the network capacity between source and destination nodes are limited to 1 Gbps, network bandwidth between the cache server and destination node is 10Gbps. Caching files at in-network cache servers helps to recover from silent errors quickly by transferring file from nearby cache servers with possibly higher bandwidth.

TABLE I
COMPARISON OF ERROR RECOVERY TIMES WITH AND WITHOUT (REGULAR) IN NETWORK CACHING IN THE PRESENCE OF SILENT ERRORS.

File Size	Regular	In Network Caching	Improvement (%)
100MB	2.08	1.66	55
1GB	21.29	16.31	48
5GB	105.47	81.56	50
10GB	233.43	176.45	46
20GB	468.89	357.27	47

We first measure transfer times in the absence of cache servers. In this scenario, we first transfer files from source to destination and in the case of checksum mismatch, we resend files from the source to destination to recover from silent errors. We use MD5 to calculate file hashes and run integrity check between source and destination nodes. That is, both sender and receiver calculates file hashes based on their copy of file and exchange them after files are transferred. We measured error recover time and total transfer times when in network caching is implemented to recover from silent errors for different file sizes. Total transfer time includes times of first transfer, checksum calculation, and second transfer of files with the assumption that the checksum values do not match. Recovery time, on the other hand, only contains time to transfer a file second time to recover from checksum mismatch issue. Table I and Table II presents values and improvement ratios of in network caching method in comparison to traditional approach (i.e., regular). We observe that, for all file sizes, recovery time is shortened by around 50% and total transfer time is reduced by 20% with the help of in-network caching. Although the bandwidth between the cache server and destination node is $10x$ higher than the bandwidth between the source and destination nodes, we only observe $2x$ improvement in transfer times. This is due to the fact that file transfers are limited to around 2.2 Gbps disk write speed at the destination node, thus cache server to destination transfer speed cannot reach to 10 Gbps.

V. CONCLUSION AND FUTURE WORK

Silent error detection is critical for file transfers to avoid permanent data losses. However, current recovery process which involves the retransfer of files from source node in-

TABLE II
COMPARISON OF TOTAL TRANSFER TIMES WITH AND WITHOUT (REGULAR) IN NETWORK CACHING IN THE PRESENCE OF SILENT ERRORS.

File Size	Regular	In Network Caching	Improvement (%)
100MB	2.08	1.66	20
1GB	21.29	16.31	23
5GB	105.47	81.56	22
10GB	233.43	176.45	24
20GB	468.89	357.27	23

creases transfer times considerably in addition to consuming significant network bandwidth. In this paper, we presented an in-network caching assisted error recovery approach that takes advantage of in-network resources to store files until their transfers are completed successfully. If an error is detected, then files can be retrieved from a nearby cache server instead of downloading them from the original source. This provides significant speed up in the recovery time as well as reducing the load on network resources. Programmable devices lend themselves for the efficient implementation of in-network caching as they provide a configurable traffic mirroring scheme. Hence, we implemented the proposed in-network caching assisted error recovery approach using a programmable Tofino switch. The preliminary results show that in network caching reduces error recovery times by 50% and total transfer times by 20% for varying file sizes. It is important to note that the improvement rates are highly dependent on network configurations and cache server settings, thus the proposed solution has a potential to yield higher gains through the use of more customized networking and caching configurations.

As a future work, we plan to extend the proposed method with more automation such that clients can demand only certain files to be cache instead of all files. This way, the load on cache servers will be minimized. Similarly, cache servers can register themselves to programmable devices to support the implementation of multiple cache servers. Finally, we will explore various cache eviction policies to avoid overwhelming cache servers' storage space with stale date.

VI. ACKNOWLEDGMENTS

This project is in part sponsored by the National Science Foundation (NSF) under award numbers 2019164 and 2145742.

REFERENCES

[1] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *ACM SIGCOMM computer communication review*, vol. 30, no. 4. ACM, 2000, pp. 309–319.

[2] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 8, 2008.

[3] R. Kettimuthu, Z. Liu, D. Wheeler, I. Foster, K. Heitmann, and F. Cappello, "Transferring a petabyte in a day," *Future Generation Computer Systems*, vol. 88, pp. 191–198, 2018.

[4] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyoul, M. H. Gunes, and E. Arslan, "Towards securing data transfers against silent data corruption," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 262–271.

[5] S. Vural, P. Navaratnam, N. Wang, C. Wang, L. Dong, and R. Tafazolli, "In-network caching of internet-of-things data," in *2014 IEEE international conference on communications (ICC)*. IEEE, 2014, pp. 3185–3190.

[6] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[7] Z. Li, G. Simon, and A. Gravey, "Caching policies for in-network caching," in *2012 21st International conference on computer communications and networks (ICCCN)*. IEEE, 2012, pp. 1–7.

[8] I. Psaras, W. K. Chai, and G. Pavlou, "Probabilistic in-network caching for information-centric networks," in *Proceedings of the second edition of the ICN workshop on Information-centric networking*, 2012, pp. 55–60.

[9] N. Choi, K. Guan, D. C. Kilper, and G. Atkinson, "In-network caching effect on optimal energy consumption in content-centric networking," in *2012 IEEE international conference on communications (ICC)*. IEEE, 2012, pp. 2889–2894.

[10] Y. Thomas, G. Xylomenos, and G. C. Polyzos, "In-network packet-level caching for error recovery in icn," in *2020 18th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT)*, 2020, pp. 1–8.

[11] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Communications of the ACM*, vol. 55:2, pp. 81–88, 2012.

[12] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, "Towards optimizing large-scale data transfers with end-to-end integrity verification," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3002–3007.

[13] E. Arslan and A. Alhussen, "Fast integrity verification for high-speed file transfers," *arXiv preprint arXiv:1811.01161*, 2018.

[14] A. Alhussen and E. Arslan, "Avoiding data loss and corruption for file transfers with fast integrity verification," *Journal of Parallel and Distributed Computing*, vol. 152, pp. 33–44, 2021.

[15] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[17] H. Stubbe, "P4 compiler & interpreter: A survey," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, vol. 47, 2017.

[18] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, "P4-compatible high-level synthesis of low latency 100 gb/s streaming packet parsers in fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 147–152.

[19] Barefoot-Intel, "Switch architecture specification," *10k Device Family*, vol. Product Specification, pp. 45–47, 2020.

[20] P. Goyal and A. Goyal, "Comparative study of two most popular packet sniffing tools-tcpdump and wireshark," in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 2017, pp. 77–81.

[21] "pyshark", 2022, accessed = 2022-10-05. [Online]. Available: https://github.com/KimiNewt/pyshark

[22] "Scapy", 2022, accessed = 2022-10-05. [Online]. Available: http://www.secdev.org/projects/scapy/

[23] D. Song, "Dpkt - python module for fast, simple packet parsing," 2006. [Online]. Available: https://github.com/kbandla/dpkt