# WebScreen+: Web Traffic Screening Performance Enhancement with eBPF Filtering

Neminath Hubballi and Pratibha Khandait
Department of Computer Science and Engineering
Indian Institute of Technology Indore, India
{neminath, phd1801101001}@iiti.ac.in

***Abstract*—Web applications attract different cyber attacks as they often deal with sensitive information like banking transactions. Web Application Firewalls are used to screen incoming requests and detect such attacks. A busy web server may receive large number of requests and hence screening can become performance bottleneck. Our contributions in this paper are two-fold. First, we present a novel Deep Packet Inspection (DPI) based method for detecting attacks against a web server. This method employs attack-specific signatures encoded as regular expressions, enabling precise screening of incoming HTTP requests. We describe an approach to semi-automatically generate these signatures using Jaccard index, which measures the similarity between invariant character sets extracted from known malicious HTTP requests. Second, we filter non-essential traffic at the network interface card level to achieve scalability. We evaluate the effectiveness of both the detection method and performance enhancement strategy rigorously through experiments conducted in a live network environment.**

***Index Terms*—Intrusion Detection, Regular Expression, Deep Packet Inspection, Packet Filtering.**

## I. Introduction

Web applications are the lucrative targets for cyber exploitation as they are the most widely used applications on the Internet and also deal with sensitive information like credit card and bank transactions. Attacks like brute-forcing, script injection, cross-site forgery, SQL-Injection, etc. are very common [1]. Considering the importance of protecting these applications, there is a wealth of techniques developed for detecting and mitigating such attacks. Detection and mitigation techniques appear in the form of intrusion detection systems and web application firewalls, respectively. Both the methods rely on Deep Packet Inspection (DPI) for detecting/filtering HTTP requests pertaining to attacks. However, DPI is a computationally expensive operation and particularly regular expression matching which is commonly used as part of signatures. Unfortunately, the DPI-based matching itself can become the target for a DoS attack. For example, even identifying a packet is carrying a HTTP request itself is computationally expensive. Moreover, as these packets go through kernel space, there are multiple context switches, data copy operations, etc are involved. As a result, the detection engines spend considerable amount of computational resources to detect/filter bad traffic.

Drawing motivation from this, we describe a method to filter non-relevant packets from reaching the user space application. We consider a software-based IDS (Intrusion Detection System) engine (like Snort [2]) running as an application in the user space that uses rules/signatures for detecting the attacks. We propose to filter non-relevant packets at the Network Interface Card (NIC) and send only those packets which require screening to the software IDS engine. In this direction, our contributions in this paper are as follows.

(i) We propose a method to generate signatures for detecting a few common attacks against web servers using frequent character combination.

(ii) We use the Extended Berkeley Packet Filter (eBPF) to filter large set of non relevant packets to improve performance.

(iii) We experiment in a production network showing that the IDS engines have to screen many orders less number of packets resulting in substantial resource savings.

## II. Prior Work

Prior works in web application security mainly fall into two categories as below.

`(i) Web Application Firewall`: Web Application Firewall (WAF) is an application layer filtering method that protects web servers from attacks. It works by having a set of positive or negative rules (policies) for filtering. Krueger et al., in their work TokDoc [3] proposed to sanitize the HTTP requests by overwriting the content that matches known malicious patterns before forwarding them to the server. FlowWatcher [4] is another such WAF that screens both GET and POST requests coming from a client and also the response from the server to identify data disclosure attacks. Zhao et al. [5] observed similar data disclosure vulnerabilities in the mobile applications. However, many of these firewalls do not provide adequate protection owing to incomplete set of rules, misconfiguration of firewall, and implementation issues [6] and can also be evaded by adversarial machine learning [7] techniques. Applet et al. [8] propose to address these issues by automating rule analysis with a combination of machine learning and multi-objective genetic algorithms. On similar lines, Sepczuk [9] proposes to add a dynamic set of rules for detecting issues.

`(ii) Intrusion Detection Systems`: Intrusion detection systems are passive monitoring techniques that screen the network traffic (HTTP packet payload) for detecting attacks. Broadly, they either use regular expression based patterns or break the entire HTTP payload into short sequences to ascertain their chance of appearing in a normal request. PAYL [10], McPAD [11], Layergram [12], Rangegram [13]

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

use such short sequences generally known as n-grams to generate an anomaly score which is an indicator of whether the request is anomalous or normal. PAYL [10] measures the distance between n-grams generated from malicious and test payloads using Mahalanobis distance to detect attacks. McPAD [11] uses multiple one class SVMs trained with sequences of different length and perform a majority voting to detect attacks. On the similar lines Layergram [12] generates training models with n-grams of different length to detect attacks. Rangegram [13] identifies the upper and lower range of frequencies of different n-grams to detect rare combinations falling outside the range as attacks.

While both WAF and IDS engines are effective for detecting and sanitizing malicious HTTP requests, a software implementation may require significant computational resources owing to the complexity of screening. In this paper, we leverage the features of eBPF [14] for reducing the complexity and also evaluate it with signatures generated using a novel semi-automatic signature generation method.

## III. Proposed Web Attack Detection and Filtering

In this section, we outline the procedure for signature generation and also technique to improve detection engine's performance with eBPF.

### A. Attack Detection

We use regular expression based signatures for detecting different attacks against HTTP servers. In the following discussion, we outline the procedure for generating such signatures.

`(i) Identifying Useful Characters:` Attacks generated against web servers are application layer attacks and appear in the form of SQL injections, cross-site scripting, cross-site forgery, etc. Our aim is to generate unique signatures which can detect these attacks. As a first step in generating the signatures for these attacks, we identify useful patterns that appear in these attacks. For example, in a JavaScript injection attack `<script>.....</script>`, may appear as part of the HTTP request. This denotes the start and end of the injected code, which is executed on the server. Along the same lines, in a SQL injection attack, some characters like '+' may appear frequently as part of SQL commands. The character '+' is used there for concatenating different words within the command. A partial command of such a SQL injection attack is shown below where an attacker is trying to drop a customer table from the database (shown in the encoded form)

`modo=entrar%27%3B+DROP+TABLE+CUSTOMER%3B`

We use similar syntactical observations for extracting useful set of characters and generating the signatures for detecting such attacks. However, completely relying on manually generating signatures is error-prone and cumbersome. Hence, we propose to partially automate signature generation by identifying such useful patterns by processing packets from known attacks. As a first step, we identify invariant characters

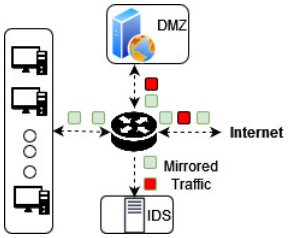


Fig. 1: Network Setup

which appear in the packets corresponding to known attacks. We take a collection of packets belonging to an attack and select a set of characters that are frequently appearing. Subsequently, we screen these characters to identify useful character sets for detecting attacks and ignore the rest of the characters.

`(ii) Generating Regular Expression based Signatures:` After selecting a set of useful characters, we identify their association and co-occurrence within web (HTTP) requests to craft detection signatures. Utilizing the Jaccard similarity coefficient across various characters, we create attack signatures. The premise is to pinpoint invariant characters that frequently appear together in attack payloads. Upon identifying a potential set of co-occurring characters, we iteratively expand this set by progressively incorporating additional characters that maintain a high similarity or co-occurrence index. For instance, if characters $C_1$ and $C_2$ exhibit a notable degree of co-occurrence, we include character $C_3$ (considering $C_1$ and $C_2$ as one entity) only if its addition does not diminish the co-occurrence degree below a predefined threshold value. This iterative expansion process enables the generation of comprehensive signatures tailored for detecting various types of attacks, effectively enhancing the robustness of our detection mechanisms against evolving threat landscapes.

Jaccard similarity coefficient for two characters $C_i$ and $C_j$ is calculated using Equation 1.

$$J(C_i, C_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|} \tag{1}$$

where

$J(C_i, C_j)$- is the Jaccard Similarity Coefficient,

$|C_i \cap C_j|$ - is the size of intersection set of the two variables $C_i$ and $C_j$,

$|C_i \cup C_j|$ - is the size of union set of the two variables $C_i$ and $C_j$.

As a result of this expansion, those character combinations which show high Jaccard index will eventually be part of the signature. Details of actual signatures generated are in Section IV.

### B. Improving Performance with eBPF based Filtering

Once the signatures are generated, we use a software application working in the user space like any other IDS or WAF matching the signatures in the incoming HTTP requests. We

consider a network setup commonly used in practice as shown in Fig. 1. In this configuration, an organization's network infrastructure interfaces with the wider internet through a gateway or router, establishing connectivity to the Wide Area Network (WAN). Internally, the network is partitioned into two distinct segments. The first segment encompasses end-user workstations and desktop computers, constituting the domain where employees or authorized personnel interact with the organization's resources and services. This segment typically operates within a more secure perimeter, safeguarding sensitive data and internal systems from external threats.

The second segment is designated as a demilitarized zone (DMZ), a buffer zone positioned between the internal network and the external internet-facing services. Within this DMZ reside services that are exposed to both internal users, such as employees accessing corporate resources, and external users, including clients or customers accessing publicly available services. Crucially, the web server, being a critical component of the organization's online presence, is located within this DMZ to facilitate external access while maintaining a degree of isolation from the internal network. To fortify the security

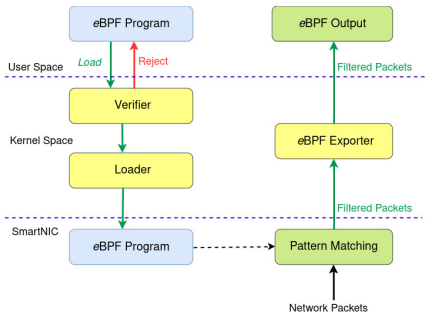


Fig. 2: Filtering Non-HTTP Packets with eBPF

posture of this setup, an Intrusion Detection System (IDS) is deployed. This IDS operates in software, leveraging algorithms and detection mechanisms to identify and respond to potential threats targeting the web server. As part of its operation, the IDS intercepts network traffic by receiving mirrored packets from the router. These mirrored packets are essentially duplicates of the original network traffic, enabling the IDS to scrutinize and analyze network activity without directly impacting the flow of data.

The IDS is equipped with a repository of attack signatures generated with method outlined in the preceding discussion. These signatures serve as patterns or fingerprints indicative of known attack vectors or malicious activities. By comparing the characteristics of incoming network traffic against these signatures, the IDS can swiftly detect and flag suspicious behavior, allowing security personnel to take appropriate remedial action to mitigate potential threats before they escalate into security incidents. Thus, the IDS plays a pivotal role in securing the organization's network infrastructure, particularly in safeguarding web server housed within the DMZ.

As mentioned in the Section I, this IDS engine will get enormous amount of traffic, which may not be relevant for screening. Such traffic will consume unnecessary CPU cycles and to minimize this, we filter those packets at the NIC level using eBPF [14]. It is worth noting that this filtering is only on the mirrored traffic and original packets will reach the web server.

eBPF is a technology which allows to load and execute custom code in the Linux kernel. It allows programs to be written in restricted C language and to attach such code to kernel hooks. This hook can be any event, for example, the arrival of a new packet. Recent developments also allow to completely offload such programs to a programmable network interface known as SmartNIC, and a tool Express Data Path (XDP) [15] allow to load eBPF programs to these NICs. By virtue of custom built architecture of these SmartNICs for packet processing, they bring considerable compute cycle savings for the general-purpose CPU. Even without completely offloading computation to SmartNICs, the context switches from kernel space to user space and associated data copy operations bring significant cost savings.

In our context, we select HTTP packets using a set of keywords (GET, POST, HEAD, etc.) and send only packets containing HTTP requests to the IDS engine (running in user space) for screening to detect attacks. This keyword or pattern-matching operation is done in the NIC. The packets which carry relevant keywords will be sent to the user space IDS engine directly bypassing the kernel(this does not impact applications). Fig. 2 shows an example of how eBPF code can be loaded into the kernel. Loading operation is done after a eBPF verifier checks that the code will not impact the kernel operations.

## IV. Evaluation

Here we provide the details of evaluation done for attack detection and filtering operation. We give details of signature generation and evaluation results in live network in the next two subsections.

`(i) Signature Generation:` To generate signatures, we used a publicly available dataset, namely HTTP CSIC 2010 dataset [16], which provides HTTP requests containing different attacks such as CRLF, SQL injection, server-side scripting, cross-site scripting, etc. These requests were collected by launching attacks against a non operational e-commerce (vulnerable) website. As the dataset contains both normal and different attack requests, to begin with, we separated all the malicious requests to enable signature generation. We divided these malicious requests into two parts in the ratio of 50% each. Table I shows the distribution of HTTP requests pertaining to different attack types. Using the part-1, we extracted different character frequencies appearing in all those HTTP requests of that part. Fig. 3 shows the frequency of different characters extracted from the malicious HTTP requests. From this character set, we filtered unwanted characters and selected those characters that appeared in at least 70% of malicious requests. Among those characters, we calculated the Jaccard Index between pairs of characters and gradually expanded them to generate desired signatures as outlined in Section III.

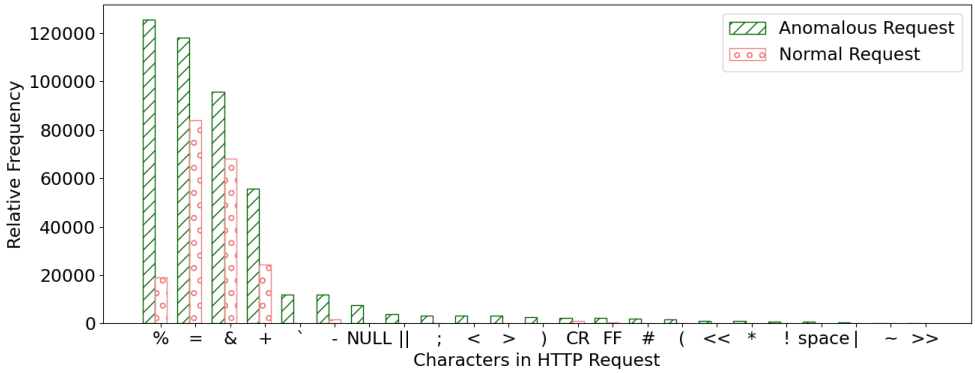


Fig. 3: Frequency Distribution of Characters

Table II shows the signatures generated for different attack types in both text format (S1-T) and hexadecimal format (S1-H)[1].

TABLE I: Summary of Attack Dataset.

| Attack Type | Malicious URL Count | |
|---|---|---|
| | Signature Generation | Detection |
| Cross Site Scripting (XSS) | 485 | 505 |
| SQL Table Modification | 92 | 112 |
| Blind SQL Injection | 532 | 572 |
| Server Side Include(SSI) | 354 | 370 |

TABLE II: Signatures Generated with Jaccard Index.

| Attack Type | Signature | Jaccard Similarity Coefficient |
|---|---|---|
| Cross Site Scripting | **S1- T:** r"[^<]+<[^>]+>[^<>]+<\/ [^>]+>" | 1.0 |
| | **S1- H:** r"[^3C]+3C[^3E]+3E[^3C3E]. +?3C%2F[^3E]+3E" | |
| SQL Table Modification | **S1- T:** r"[^+]+\+*\+*" | 0.98 |
| | **S1- H:** r"[^+]+\+*\+*" | |
| Blind SQL Injection | **S1- T:** r"[^;]+;[^']+'+[^']+';–*" | 1.0 |
| | **S1- H:** r"[^%3B].+%3B[^%27].+%27+[^%27].+%27%3B–*" | |
| Server Side Include | **S1- T:** r"[^<]*<!- -#[^-]+- ->*" | 1.0 |
| | **S1- H:** r"[^3C]*3C%21- -%23[^-].+- -%3E" | |

(ii) Filtering Packets in Real-Time: We wrote an application program(IDS engine) in Python to detect the attacks against the web server using the signatures generated. As shown in Table II, signatures have patterns containing regular expressions, we used the Python regular expression library to search for these patterns. We also wrote an eBPF program to identify HTTP requests. This was done by searching keywords commonly found in HTTP packets (not attack) and loaded into the kernel of the server running the IDS engine. We mirrored the traffic of our institute network to this server for live screening purposes. The server has Nvidia ConnectX-6 Ethernet NIC which supports eBPF offloading. This eBPF based screening serves two purposes. First, it allows us to identify the false alarms generated as there were no vulnerabilities in the web server application (at least to the ones for which signatures are written). Second, it allows to benchmark the performance for the scale of operations.

In order to establish the ground truth about the filtering process that our program is performing, we began with testing the whole setup by collecting traffic with tcpdump [17] and also running the eBPF program concurrently. This test case spanned for 20 minutes. Table III shows the statistics pertaining to number of packets processed by the user space IDS engine. We can notice that in the absence of eBPF filtering, all the packets will be taken to the user space and processed. However, with the eBPF filtering, only packets carrying HTTP requests are taken to the user space for screening. Approximately, there is a 403-fold reduction in the number of packets handled by the IDS engine. This is evident in the CPU utilization graph pertaining to packet processing on the IDS server machine, as shown in Fig. 4. This graph shows the CPU utilization for a period of 20 minutes with and without eBPF filtering, with utilization measured every 15 seconds. We can notice that without eBPF filtering, the user space IDS is utilizing 60-100% of the CPU, and with filtering, it is between 15-25%.

[1]These are identical signatures

TABLE III: Packets Processed With and Without Filtering.

| | HTTP | Others |
|---|---|---|
| Tcpdump (actual) Packet Count | 394585 | 159.124 Million |
| Packets Processed by IDS Without eBPF Filtering | 394585 | 159.124 Million |
| Packets Processed by IDS With eBPF Filtering | 394585 | 0 |

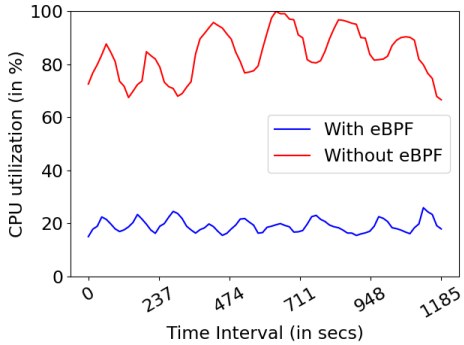


Fig. 4: CPU Utilization With and Without eBPF

After the ground truth was established, we run this filtering program for live screening for five days along with signatures

TABLE IV: Statistics of HTTP Requests and Other Packets Across Five Days in One Hour Duration.

| | Total Packets Captured | # HTTP Req Packets | # Other Packets | # HTTP Req Packets Processed With eBPF | # HTTP Req Packets Processed Without eBPF | # Other Packets Processed With eBPF | # Other Packets Processed Without eBPF |
|---|---|---|---|---|---|---|---|
| Day 1 | 335379455 | 736925 | 334642494 | 736925 | 736925 | 0 | 334642494 |
| Day 2 | 323680675 | 792770 | 322887905 | 792770 | 792770 | 0 | 322887905 |
| Day 3 | 419540644 | 928463 | 418612181 | 928463 | 928463 | 0 | 418612181 |
| Day 4 | 418080940 | 766650 | 417314290 | 766650 | 766650 | 0 | 417314290 |
| Day 5 | 402997320 | 596022 | 402401298 | 596022 | 596022 | 0 | 402401298 |

so generated. Although the program was able to perform screening continuously, owing to the very large number of packets passing through the core switch, we could not get the actual number of packets processed as statistics were reset to zero after reaching a limit by the eBPF. Thus for reporting purpose, we chose statistics generated in a span of one-hour duration each for every day. Table IV shows this distribution across five days. We can notice from the table that there is a substantial reduction in the number of packets processed and, hence, overhead associated with the IDS engine. For e.g on the Day-1 out of the 335379455 total packets captured in an hour period, only 736925 packets were of HTTP requests and exactly those many were processed by the IDS application after eBPF identified them and remaining packets were not sent to the application.

`(iii) Attack Detection Performance`: While the previous screening study established that screening operation can be done in real-time at line speed with eBPF. We conducted a study to establish the detection performance of the signatures generated. We performed a performance evaluation by generating malicious HTTP requests against our institute web server using part-2 of the CSIC 2010 dataset. As this dataset has HTTP requests in plain text format, we first converted them into valid requests carried in packets using the Scapy library and invoked through a Python script. The requests so generated were collected as a pcap file with tcpdump, and subsequently, this file is replayed with tcpreplay [18] to the SmartNIC interface. This evaluation was done for a period of one hour, along with regular traffic from the institute network. Table V shows the detection performance of those requests. We can notice that the signatures generated are robust and are able to detect majority of the attacks resulting in nearly 99% detection performance without any false alarms.

TABLE V: Detection Performance.

| Attack Type | Detection Rate | False Positive Rate |
|---|---|---|
| Cross Site Scripting | 99.60 % | 0.0 % |
| SQL Table Modification | 99.10 % | 0.0 % |
| Blind SQL Injection | 99.12 % | 0.0 % |
| Server Side Include | 98.91 % | 0.0 % |

## V. Conclusion

Web applications are subjected to different cyber attacks. Detecting these attacks requires expensive DPI techniques. In this paper, we proposed a semi-automatic technique for regular expression based signature generation using the Jaccard index of frequent character sets taken from known malicious HTTP requests. We also showed significant performance improvements by filtering non-useful packets at the NIC level by offloading this operation to the NIC.

## VI. Acknowledgment

Second author is financially supported through a generous fellowship by Tata Consultancy Services Foundation, India. Authors thankfully acknowledge the funding received.

## References

[1] B. Rajić, Z. Stanisavljević, and P. Vuletić, "Early web application attack detection using network traffic analysis," *International Journal of Information Security*, vol. 22, pp. 77–91, 2022.
[2] "https://www.snort.org/ (accessed on 10-sep-2024)."
[3] T. Krueger, C. Gehl, K. Rieck, and P. Laskov, "TokDoc: A self-healing web application firewall," in *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 1846–1853.
[4] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch, "FlowWatcher: Defending Against Data Disclosure Vulnerabilities in Web Applications," in *CCS'15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM Press, 2015.
[5] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1106–1120.
[6] D. Appelt, C. D. Nguyen, and L. Briand, "Behind an application firewall, are we safe from sql injection attacks?" in *ICST'15: Proceedings of 8th International Conference on Software Testing, Verification and Validation*, 2015, pp. 1–10.
[7] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "Waf-a-mole: Evading web application firewalls through adversarial machine learning," in *SAC '20: Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1745–1752.
[8] D. Appelt, A. Panichella, and L. Briand, "Automatically repairing web application firewalls based on successful sql injection attacks," in *ISSRE'17: Proceedings of IEEE 28th International Symposium on Software Reliability Engineering*, 2017, pp. 339–350.
[9] M. Sepczuk, "Dynamic web application firewall detection supported by cyber mimic defense approach," *Journal of Network and Computer Applications*, vol. 213, p. 103596, 2023.
[10] K. Wang and S. J. Stolfo, "PAYL: Anomalous payload-based network intrusion detection," in *RAID '04: Proceedings of Recent Advances in Intrusion Detection*, 2004, pp. 203–222.
[11] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee, "McPAD: A multiple classifier system for accurate payload-based anomaly detection," *Computer Networks*, vol. 53, no. 6, pp. 864–881, 2009.
[12] N. Hubballi, S. Biswas, and S. Nandi, "Layered higher order n-grams for hardening payload based anomaly intrusion detection," in *FARES '10: Proceedings of the workshop on Frontiers of Availability, Reliability and Security*, 2010, pp. 1–6.
[13] M. Swarnkar and N. Hubballi, "Rangegram: A novel payload based anomaly detection technique against web traffic," in *ANTS '15: Proceedings of the 9th IEEE International Conference on Advanced Networks and Telecommunications Systems*, 2015, pp. 1–6.
[14] "https://ebpf.io/ (accessed on 10-sep-2024)."
[15] "https://www.iovisor.org/technology/xdp (accessed on 15-sep-2024)."
[16] C. T. Gimenez, A. P. Villegas, and G. A. Maranon, "Http dataset csic 2010," http://www.isi.csic.es/dataset/ (accessed on 01-Sep-2024), 2010.
[17] "https://www.tcpdump.org/ (accessed on 15-aug-2024)."
[18] "https://tcpreplay.appneta.com/wiki/tcpreplay (accessed on 10-sep-2024)."