Received: 16 January 2024 | Revised: 13 November 2024 | Accepted: 26 December 2024

IET Communications

DOI: 10.1049/cmu2.12879

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH

Enhancing intrusion detection against denial of service and distributed denial of service attacks: Leveraging extended Berkeley packet filter and machine learning algorithms

Nemalikanti Anand[1,2] | **Saifulla M A**[2] | **Pavan Kumar Aakula**[2] | **Raveendra Babu Ponnuru**[3,4] | **Rizwan Patan**[5] | **Chegireddy Rama Prakasha Reddy**[6]

[1]Department of Computer Science and Engineering, BVRIT Hyderabad College of Engineering for Women, Hyderabad, India

[2]School of Computer and Information Sciences, University of Hyderabad, Hyderabad, India

[3]Dept of Computer Science, Virginia Tech, Blacksburg, Virginia, USA

[4]Dept. of Computer and Information Science, Virginia Military Institute, Lexington, Virginia, USA

[5]College of Computing and Software Engineering, Kennesaw State University, Marietta, Georgia, USA

[6]College of Engineering and Technology, Wollega University, Nekemte, Ethiopia

Correspondence
Chegireddy Rama Prakasha Reddy, College of Engineering and Technology, Wollega University, Nekemte, Ethiopia.
Email: prakashar@wollegauniversity.edu.et
Raveendra Babu Ponnuru, Dept of Computer Science, Virginia Tech, Blacksburg, Virginia, USA.
Email: raveendrababup@gmail.com

Abstract

As organizations increasingly rely on network services, the prevalence and severity of Denial of Service (DoS) and Distributed Denial of Service (DDoS) attacks have emerged as significant threats. The cornerstone of effectively addressing these challenges lies in the timely and precise detection capabilities offered by advanced intrusion detection systems (IDS). Hence, an innovative IDS framework is introduced that seamlessly integrates the extended Berkeley Packet Filter (eBPF) with powerful machine learning algorithms—specifically Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), and TwinSVM—enabling unparalleled real-time detection of DDoS attacks. This cutting-edge solution provides a robust and scalable IDS framework to combat DoS and DDoS threats with high efficiency, leveraging eBPF's capabilities within the Linux kernel to bypass typical user space constraints. The methodology encompasses several key steps: (a) Collection of data from the renowned CIC-IDS-2017 repository; (b) Processing the raw data through a meticulous series of steps, including transmission, cleaning, reduction, and discretization; (c) Utilizing an ANOVA F-test for the extraction of critical features from the preprocessed data; (d) Application of various ML algorithms (DT, RF, SVM, and TwinSVM) to analyze the extracted features for potential intrusion; (e) Implementing an eBPF program to capture network traffic and harness trained model parameters for efficient attack detection directly within the kernel. The experimental results reveal outstanding accuracy rates of 99.38%, 99.44%, 88.73%, and 93.82% for DT, RF, SVM, and TwinSVM, respectively, alongside remarkable precision values of 99.71%, 99.65%, 84.31%, and 98.49%. This high-speed, accurate detection model is ideally suited for high-traffic environments such as data centers. Furthermore, its foundational architecture paves the way for future advancements, including the potential integration of eBPF with XDP to achieve even lower-latency packet processing. The experimental code is available at the GitHub repository link: https://github.com/NemalikantiAnand/Project.

1 | INTRODUCTION

Researchers around the globe are working towards developing a strong and reliable IDS solution that can improve network security against cyber threats. It specifically focuses on creating an efficient and effective IDS that can detect and prevent DoS [1, 2] and DDoS attacks, which pose significant challenges for organizations that rely on network services. On the other side, researchers also address the need for timely and accurate responses to such attacks. To develop an IDS that captures and analyzes network traffic efficiently to detect and respond to security threats in real-time, this research will use the extended

Berkeley Packet Filter (eBPF) as well as machine learning algorithms such as Decision Tree, Random Forest, Support Vector Machine, and TwinSVM. By addressing these challenges, this research aims to contribute to the development of a robust and efficient IDS solution that can enhance network security and mitigate the impact of DoS and DDoS attacks on organizations. eBPF is a byte-code-based virtual machine that runs programs without modifying the kernel source code. It can implement various services, such as observability, security, and networking. Socket filters are an eBPF program attached to the socket in the Linux kernel that allows for efficient filtering and manipulation of network packets at the socket after packets are received from the network stack. Packets that are filtered at the socket level before entering the user space. In the past, practically all website content was written in a hypertext markup language (HTML) format. Browsing websites has evolved into a full-fledged application, and web-based technology has mostly supplanted traditional software. This development was made possible by programmability, which was made possible with the advent of JavaScript [3, 4].

The eBPF is the solution to use if we want to update the kernel dynamically. This solution is analogous to the one that JavaScript provides for HTML. The Linux kernel is undergoing a transformation brought on by eBPF, similar to how JavaScript altered the web. Users can execute programs in a secure setting by using eBPF, which permits the execution of sandboxed programs inside the context of privileged operations within the operating system [5]. Since the programs are run in the kernel, this results in a much-reduced amount of overhead compared to using native kernel modules. eBPF enables a wide variety of application cases, ranging from simple network monitoring to intricate performance optimization and security checks [6–10]. eBPF programs are event-driven programs that are activated when a hook point is passed by the kernel or an application. The terms system calls,"function entry and exit," "kernel trace points," and "network events" are all examples of pre-defined hook points. In the absence of a pre-established hook, the kernel probe (probe) and the user probe can attach eBPF programs almost anywhere in the kernel or user applications. The eBPF architecture is shown in Figure 1. When viewed through the lens of its underlying architecture, eBPF consists of several distinct components, including a virtual machine, a collection of libraries, and a set of maps. While the eBPF program is being run via the virtual machine, the libraries are providing a set of helper functions [11] for the eBPF program to make use to interact with the kernel and perform various tasks [11–15].

eBPF is a framework that allows users to load and execute their customized programs directly into the operating system's kernel. When an eBPF program is loaded into the kernel, a verifier checks to see whether it is safe to execute and decides whether to reject it if it is not safe. After it has been loaded, an eBPF program must be connected to an event for it to be activated whenever the associated event takes place. With the help of the Low-Level Virtual Machine (LLVM) compiler [16], we can transform pseudo-C code into eBPF byte code. This is necessary since the Linux kernel anticipates that eBPF programs will be loaded as byte code. It is possible to load an eBPF program into the Linux kernel by using the BPF system call. This is commonly accomplished by utilizing one of the eBPF libraries that are currently available. The program needs to be loaded into the Linux kernel and then go through the next two steps before it can be connected to the designated hook: Both the verification step and the Just-in-Time (JIT) [17] compilation step ensure that the eBPF program is safe to run. The JIT compilation step optimizes the execution speed of the program by translating the generic byte code of the program into the machine-specific instruction set. This allows eBPF programs to run as efficiently as natively compiled kernel code or as code loaded as a kernel module [3, 4] and [5, 11]. eBPF and ML integration, especially in high-traffic or large-scale network environments. It can manage load balancing, resource utilization, and real-time processing. The thorough access to network data that eBPF has, as it operates at the kernel level, can pose a security risk if not correctly managed. Writing a secure eBPF code and conducting routine audits can assist in reducing this risk. To ensure privacy, the IDS should only capture the data required for threat detection and anonymize it before incorporating it into machine learning algorithms to safeguard sensitive information.

ML models demand strong CPUs or GPUs and substantial memory during training, particularly for complicated models and big datasets. This offline phase does not affect real-time IDS performance. For low latency and optimal resource use, real-time inference requires optimized models. Maintaining performance and scalability requires model compression and effective eBPF program management. The IDS scales horizontally, transferring computing burden across numerous instances to ensure responsiveness in larger network settings.

eBPF programs can save and communicate information that they have gathered with one another. For this reason, eBPF programs may use eBPF maps, which are comparable to arrays or hash tables and enable eBPF programs to store and retrieve data in real-time. eBPF maps are similar to arrays and hash tables. Through the use of a system call, eBPF applications and programs running in user space can get access to eBPF maps. The following kinds of maps are supported in this list: eBPF programs are composable using the ideas of tail and function calls, allowing for the creation of hash tables, arrays, Least Recently Used (LRU), ring buffers, Longest Prefix Match (LPM), and so on. Within the context of an eBPF program, function calls enable the definition and invocation of functions. Akin to how the execve() system call functions for ordinary processes, tail calls [3–5, 11] can call and run another eBPF program while simultaneously replacing the execution environment. eBPF programs are efficient and compact computer instructions that may be run directly within the Linux kernel. They are written in C, however, a version that has restrictions is referred to as Restricted-C. This subset of C was chosen after much deliberation to make the environment in which eBPF programs run as safe and effective as possible. Because it only supports a subset of eBPF's functions and data types, it makes executing eBPF code in the kernel far less risky. The limitations placed on loops in Restricted-C and the use of floating point integers are two of the numerous constraints that are considered significant [18]. Real-time interaction with the system is made possible by

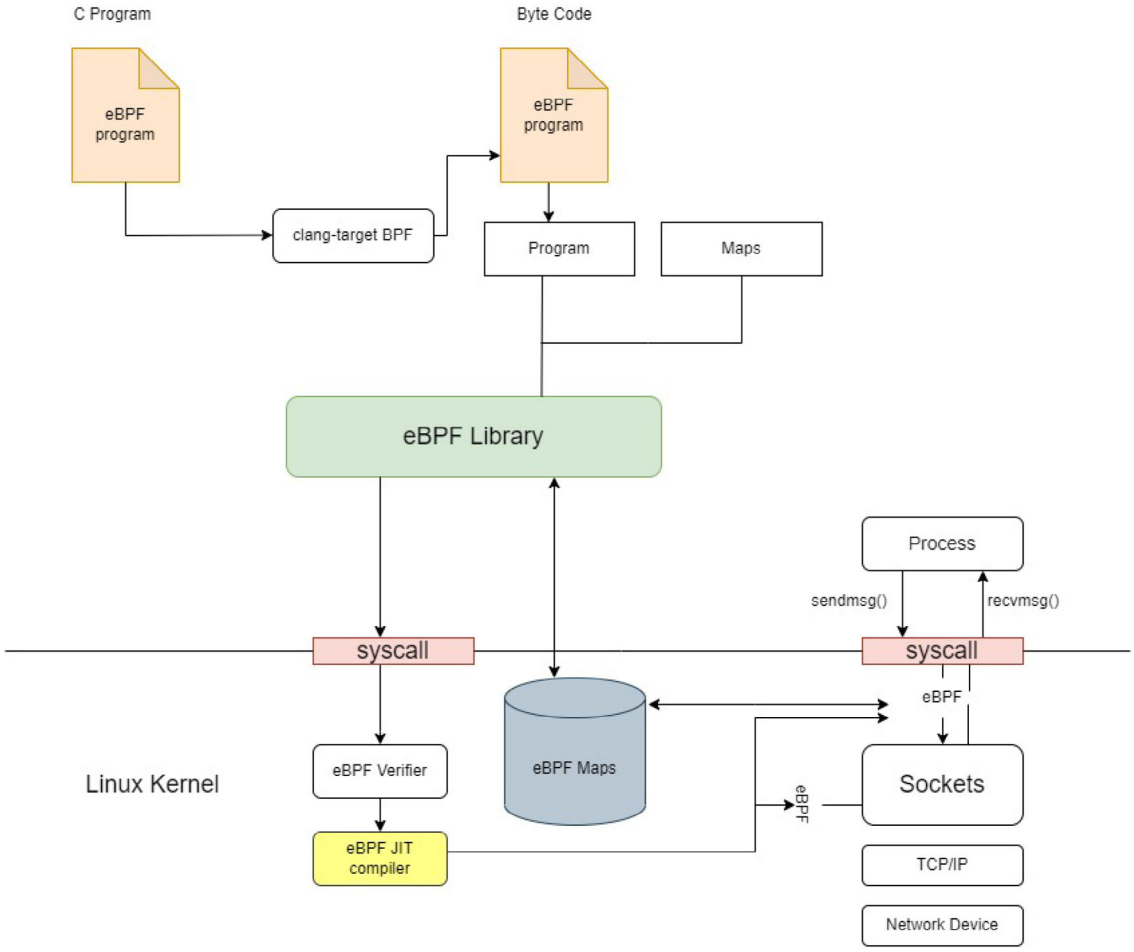


FIGURE 1 eBPF architecture.

the fact that eBPF programs may be tied to a wide variety of events in the system, including system calls, network packets, and others.

Let's examine a diagram that illustrates the typical workflow for building and deploying an eBPF program. The eBPF programming language is a restricted version of C that utilizes maps. Since the Linux kernel expects eBPF programs to be loaded in bytecode, the LLVM takes on the task of compiling this restricted C code into eBPF bytecode. With the help of the BPF system call, an eBPF byte code program may be loaded into the eBPF verifier that's included in the Linux kernel. In most cases, this is accomplished with the use of a library, the BPF Compiler Collection (BCC) [19]. After the verifier confirms that the program has not introduced any vulnerabilities and has been correctly written, it is sent on to the JIT compiler, which converts the byte code into the native machine code [20]. After being loaded into the kernel, an eBPF program has to be associated with an event before it can be used. Whenever the event takes place, the eBPF program (or programs) that are related to it are executed. In this scenario, Sockets enables the attachment of an eBPF program to a network interface [21]. As a result, data packets are routed via the eBPF process running in the kernel space before being sent to the real user process.

Our research focuses on combining machine learning algorithms with the extended Berkeley Packet Filter (eBPF) to develop a sophisticated intrusion detection system (IDS) that can manage the intricacies of contemporary network traffic. The system intends to enter high-quality, low-latency data into machine learning models by utilizing eBPF's ability to capture and analyze packet data at the kernel level. This will increase the precision and speed of intrusion detection. A variety of machine learning algorithms are utilized for classifying network traffic and recognizing potential security threats. These algorithms include Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), and TwinSVM. Each algorithm offers unique benefits: DT provides interpretable models with fast decision-making capabilities, RF enhances robustness and accuracy through ensemble learning, SVM is well-suited for high-dimensional spaces with clear margin separation, and TwinSVM further improves the classification process by building two non-parallel hyperplanes. The combination of these algorithms with eBPF allows for the development of an advanced IDS. This not only enhances detection accuracy but also decreases the computational workload usually linked with such systems. The synergy between eBPF's efficient data management at the kernel level and the predictive

abilities of machine learning algorithms represents a noteworthy progression in the field of network security [12–15].

1.1 | Contributions

Overall contribution to developing a high-performance IDS solution with effective DoS and DDoS attack detection and insights into integrating eBPF and machine learning for network security are given. The novel contributions are as follows:

- Proposal of an IDS integrating eBPF and machine learning algorithms (Decision Tree, Random Forest, Support Vector Machine, TwinSVM) for DoS and DDoS attack detection.
- Application of advanced preprocessing techniques (data transmission, cleaning, reduction, and discretization) to refine raw data from the CIC-IDS-2017 dataset.
- Use of ANOVA F-test for innovative feature selection, identifying specific features crucial for intrusion detection through various machine learning algorithms.
- Comprehensive performance evaluation of the proposed IDS, assessing accuracy, precision, recall, and F1-score, with comparisons to existing related work.
- Provision of open-source code for the proposed IDS, facilitating use by other researchers and practitioners for building and testing their own IDS solutions.

1.2 | Organization

Section 2 delves into the works of various authors related to this area. Section 3 comprises the proposed method and experimental results. Section 4 addresses performance analysis, while Section 5 encapsulates the conclusion and outlines future research scope.

2 | RELATED WORK

The authors in [22, 23] propose a flow-based IDS implemented with ML in eBPF, using the CIC-IDS-2017 dataset. A Decision Tree (DT) trained with scikit-learn achieved 90% accuracy, utilizing a training-to-testing ratio of 2:1. The same IDS was developed in user space and eBPF, with identical code aside from data structures due to eBPF's limitations. The user space version employed a simple hash map adapted from the Linux kernel, and both implementations ran sequentially for ten seconds. The user space version analyzed 125420 packets per second, while eBPF processed 152274, making eBPF nearly 20% faster.

The authors in [24] suggest the design and implementation of an IDS that makes use of eBPF inside the Linux kernel. To begin, they suggested using a method based on eBPF to design and deploy IDS systems. They develop and execute an IDS that comprises two components that collaborate. The initial portion of the code is executed in the Linux kernel. It does quick pattern matching with eBPF to pre-drop a very big part of packets that cannot match any rule. This is done to save bandwidth. The user's environment is the focus of the second component. It investigates the packets left behind by the previous portion to locate the rules corresponding to those packets. Under many measured conditions, an IDS system's maximum throughput may exceed Snort's by a factor of three.

The authors in [4] state that eBPF enables runtime modification, interaction, and kernel programmability. XDP (express Data Path) framework utilizes eBPF to write programs to process packets closer to the NIC for fast packet processing. He states that programs can be written in C or P4 [25] languages, compiled into eBPF instructions, and then loaded into the kernel, providing an eBPF runtime environment. His work will include eBPF and XDP rapid packet processing theory and practice. Theoretically, he covered BPF and eBPF machines and the Linux kernel's eBPF system. He demonstrated eBPF and the XDP hook with examples and tools. He thinks eBPF and XDP may advance new research initiatives since they process packets quickly.

Snort [26] and Suricata [27] utilize eBPF for packet filtering but are limited to parsing layer-3 headers and lack the ability to match patterns in packet content. Snort allows users to input filter expressions using the -F command-line option, which are then converted to eBPF commands. Suricata employs eBPF for XDP-based functions but does not support custom eBPF scripts for pattern matching in packet payloads, rendering DPI inapplicable. An eBPF-based DPI method exists [28] for identifying video frame types, but it can only handle a single packet format, making it insufficient for an IDS. ebpfH is an eBPF-based host-based IDS [29] that focuses on system rather than network anomalies.

A comprehensive analysis of eBPF was presented by the authors in [4], detailing its technical aspects and implementation locations. eBPF has transformed many fundamental network functionalities, including routing [30], switching [31], and firewalls [32]. Key features include load balancing, key-value storage [33], application-level filtering [34], and DDoS mitigation [35].

The In-KeV framework [36] allows for the development of kernel-based network services, enabling in-kernel Service Function Chains (SFCs) through tail calls. Limitations of eBPF and user experiences are detailed in [37]. Additionally, [38] analyzed the performance of eBPF in filtering packets based on header 5-tuple information. Moreover, [39] created an open-source 5G mobile gateway, while [40] used eBPF for monitoring InterVM communications. Finally, a framework for eBPF-based network functions for microservices was discussed in [41].

When implementing complex network functions, the authors of [4, 22–24, 37, 38, 42], and [43] highlighted several significant parameters that affect the performance of eBPF, as shown in Table 1. The article [38] focuses on eBPF-based firewalls, which are simpler than intrusion detection systems (IDS). Furthermore, [44] examined the performance of eBPF's eXpress Data Path (XDP) in virtual machines (VMs) and hardware offloading scenarios, revealing that XDP performance decreases within VMs.

String matching in the Snort intrusion detection system is performed using a combination of the Boyer–Moore (BM) [45]

TABLE 1 Comparison of related work.

Author	Contribution	Methodology adopted	Benefits
Vieira et al. [4]	eBPF and XDP limitations and usage in packet processing procedures.	To improve network security, eBPF and XDP can be used together.	Potential in network packet processing and security functionalities.
Hara et al. [22]	Proposed lightweight neural network and Decision Tree combined with eBPF, trained on CIC-IDS- 2017 dataset.	Decision Trees, lightweight neural network with eBPF, which decides if a packet is malicious.	Achieved NN 99% and DT 97.6% accuracy.
Gallego et al. [43]	Proposed MLP model with eBPF in comparison with a user space development approach.	An IoT scenario designed to assess the performance of a solution for detecting attacks in a 6LowPAN system.	The kernel implementation demonstrates a substantial decrease in execution time (by 97%) and a moderate reduction in CPU usage (by 6%).
Bachl et al. [23]	Proposed a NIDS using ML algorithms (e.g. Decision Trees) combined with eBPF, trained on CIC-IDS-2017 dataset.	ML-based NIDS using Decision Trees and eBPF, which decides if a packet is malicious.	24% performance increase compared to user space program, achieved 99% accuracy.
Wang et al. [24]	Proposed NIDS utilizing eBPF with kernel space and user space programs for packet filtering.	Combination of modified Snort ruleset with eBPF for packet filtering and kernel-based NIDS.	Higher network throughput, reduced CPU usage, less packet loss.
Caviglione et al. [42]	eBPF used for near real-time malware detection and anomaly identification by tracking software operations.	Analyzed two real-world attack strategies, tracking anomalies using eBPF algorithms for security monitoring.	Efficient malware detection with low overhead, adaptability in anomaly identification.
Scholz et al. [38]	Utilized eBPF and XDP in combination with SmartNICs to mitigate DDoS attacks and improve packet processing pipelines.	Combination of SmartNIC hardware filtering and XDP software filtering for effective packet processing and DDoS mitigation.	Reduced CPU consumption and packet dropping rates, effective mitigation of DDoS attacks.

and Aho–Corasick (AC) [46] algorithms. Additionally, [47] proposes using artificial intelligence (AI) to detect performance anomalies using eBPF. eBPF is also utilized in [48, 49], and [50] to develop countermeasures against denial-of-service (DoS) attacks, but these studies do not incorporate machine learning (ML) techniques at all.

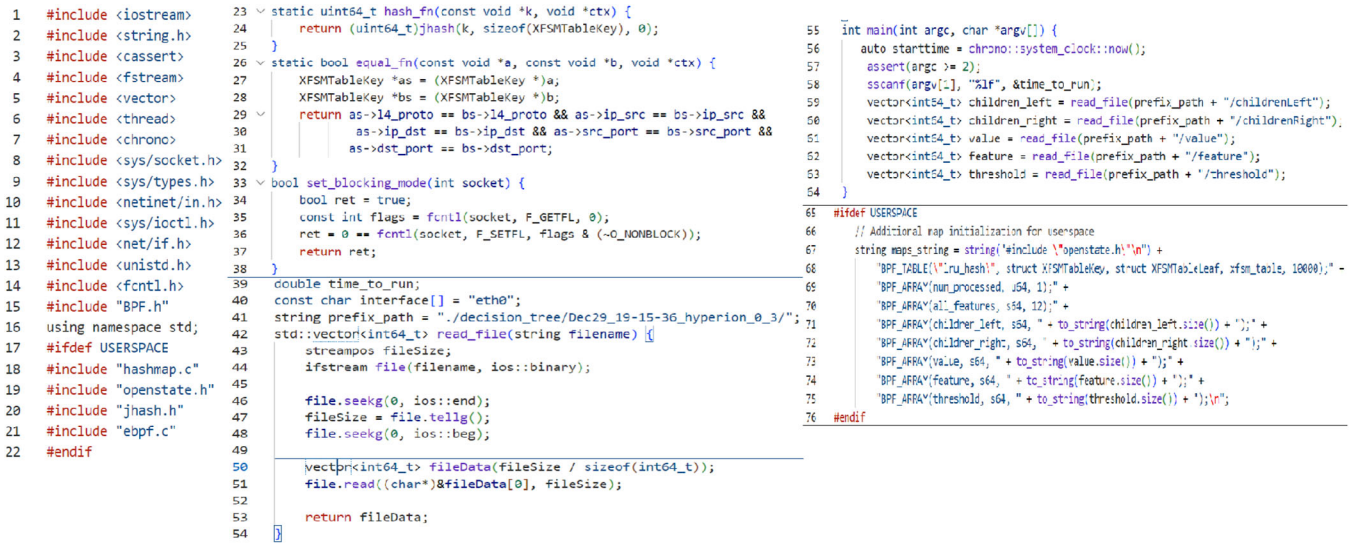
3 | PROPOSED METHOD AND EXPERIMENTAL RESULTS

This procedure in machine learning algorithms consists of the following steps:

- We are making use of data from the popular CIC-IDS-2017 dataset [51].
- Once the raw data is obtained, it undergoes pre-processing, which includes data transmission (reading data from files), cleaning (infinity, NULL values etc.), and reduction (size).
- Following the pre-processing step, to extract specific features from the pre-processed data by using the ANOVA F-test method.
- The extracted features are analyzed for intrusion detection using ML algorithms.
- After training and testing with RF and DT, export the model parameters, including the left child, right child, threshold, features, and value of each Decision Tree in the RF.
- After training and testing with SVM and TwinSVM, export the model parameters, including coefficients, intercepts, and features.
- Lastly, write an eBPF program that captures network traffic and utilizes trained model parameters to detect attacks within the kernel

To optimize eBPF-based IDS, preprocessing techniques are crucial because they guarantee data quality, minimize the amount of data processed, and facilitate data analysis. This leads to more accurate, efficient, and effective intrusion detection.

- **Data cleaning**: It performs major parsing and validation and filters noise. Clean data reduces the chances of misinterpretation and false positives/negatives in the IDS, leading to more reliable detection of intrusions. Parsing and validation ensure that the data captured by eBPF probes is correctly formatted and adheres to expected structures. This may involve parsing network packets and validating their integrity. Similarly, filtering noise removes irrelevant or erroneous data captured by eBPF, such as malformed packets or noise from non-relevant traffic. The primary advantage of data cleaning is that it improves accuracy and efficient processing.
- **Data reduction**: It performs major feature aggregation and sampling and filtering. The feature aggregation is aggregating related features captured by eBPF probes to reduce dimensionality. For instance, various network traffic metrics can be combined into summary statistics. The sampling and filtering techniques, such as packet sampling or applying filters to capture only relevant packets, reduce the volume of data processed. The primary advantage of data reduction is to enhance speed and resource efficiency.



```
#include <iostream>
#include <string.h>
#include <cassert>
#include <fstream>
#include <vector>
#include <thread>
#include <chrono>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <unistd.h>
#include <fcntl.h>
#include "BPF.h"
using namespace std;
#ifdef USERSPACE
#include "hashmap.c"
#include "openstate.h"
#include "jhash.h"
#include "ebpf.c"
#endif

static uint64_t hash_fn(const void *k, void *ctx) {
    return (uint64_t)jhash(k, sizeof(XFSMTableKey), 0);
}
static bool equal_fn(const void *a, const void *b, void *ctx) {
    XFSMTableKey *as = (XFSMTableKey *)a;
    XFSMTableKey *bs = (XFSMTableKey *)b;
    return as->l4_proto == bs->l4_proto && as->ip_src == bs->ip_src &&
           as->ip_dst == bs->ip_dst && as->src_port == bs->src_port &&
          as->dst_port == bs->dst_port;
}
bool set_blocking_mode(int socket) {
    bool ret = true;
    const int flags = fcntl(socket, F_GETFL, 0);
    ret = 0 == fcntl(socket, F_SETFL, flags & (~O_NONBLOCK));
    return ret;
}
double time_to_run;
const char interface[] = "eth0";
string prefix_path = "./decision_tree/Dec29_19-15-36_hyperion_0_3/";
std::vector<int64_t> read_file(string filename) {
    streampos fileSize;
    ifstream file(filename, ios::binary);

    file.seekg(0, ios::end);
    fileSize = file.tellg();
    file.seekg(0, ios::beg);

    vector<int64_t> fileData(fileSize / sizeof(int64_t));
    file.read((char*)&fileData[0], fileSize);

    return fileData;
}

int main(int argc, char *argv[]) {
    auto starttime = chrono::system_clock::now();
    assert(argc >= 2);
    sscanf(argv[1], "%lf", &time_to_run);
    vector<int64_t> children_left = read_file(prefix_path + "/childrenLeft");
    vector<int64_t> children_right = read_file(prefix_path + "/childrenRight");
    vector<int64_t> value = read_file(prefix_path + "/value");
    vector<int64_t> feature = read_file(prefix_path + "/feature");
    vector<int64_t> threshold = read_file(prefix_path + "/threshold");
}
#ifdef USERSPACE
    // Additional map initialization for userspace
    string maps_string = string("#include \"openstate.h\"\n") +
        "BPF_TABLE(\"lru_hash\", struct XFSMTableKey, struct XFSMTableLeaf, xfsm_table, 10000);" +
        "BPF_ARRAY(num_processed, u64, 1);" +
        "BPF_ARRAY(all_features, s64, 1);" +
        "BPF_ARRAY(children_left, s64, " + to_string(children_left.size()) + ");" +
        "BPF_ARRAY(children_right, s64, " + to_string(children_right.size()) + ");" +
        "BPF_ARRAY(value, s64, " + to_string(value.size()) + ");" +
        "BPF_ARRAY(feature, s64, " + to_string(feature.size()) + ");" +
        "BPF_ARRAY(threshold, s64, " + to_string(threshold.size()) + ");\n";
#endif
```

FIGURE 2 eBPF Wrapper Code part-1.

- **Data discretization**: It performs majorly binning and simplified analysis. Binning covers continuous variables (e.g. packet sizes, time intervals) into discrete bins or categories. For example, packet sizes can be categorized into ranges. Thresholding is setting thresholds to classify continuous data into discrete categories, such as flagging traffic as normal or suspicious based on predefined thresholds. The primary advantage of data discretization is that it simplifies analysis and improves detection.

The eBPF wrapper code functions as an additional layer that encapsulates and streamlines the use of eBPF programs. This code acts as a facilitator, providing a higher-level abstraction and simplifying the intricate processes involved in loading, attaching, and interacting with eBPF programs within the Linux kernel. The wrapper contributes to code reusability, efficient error handling, and heightened security by furnishing a more intuitive interface. Well-crafted wrappers often incorporate robust error-handling mechanisms, ensure compatibility across diverse kernel versions, and implement security checks to prevent the execution of malicious code. The eBPF wrapper code is given in Figures 2 and 3.

All things considered, the code appears to be a component of a network application, maybe incorporating eBPF and packet filtering, and conditional compilation to support user space and kernel implementations.

hash_fn function: (23-25) This function is a hash function used for generating a hash value from a given key (k). It utilizes the Jenkins hash function (jhash) with parameters including the key (k) and its size (sizeof(XFSMTableKey)).

equal_fn function: (26-32) This function is an equality comparison function used for checking if two keys (a and b) are equal. It compares individual fields of the XFSMTableKey structure (l4_proto, ip_src, ip_dst, src_port, dst_port). Returns true if all fields are equal, indicating that the keys are equal; otherwise, returns false. The resulting hash value is cast to a 64-bit unsigned integer and returned.

set_blocking_mode function: (33-38) This function sets a socket to blocking mode. It takes a socket file descriptor (socket) as input. Retrieves the current file status flags using fcntl with F_GETFL.Clears the O_NONBLOCK flag from the flags. Sets the modified flags using fcntl with F_SETFL. Returns true if the operation is successful, otherwise false.

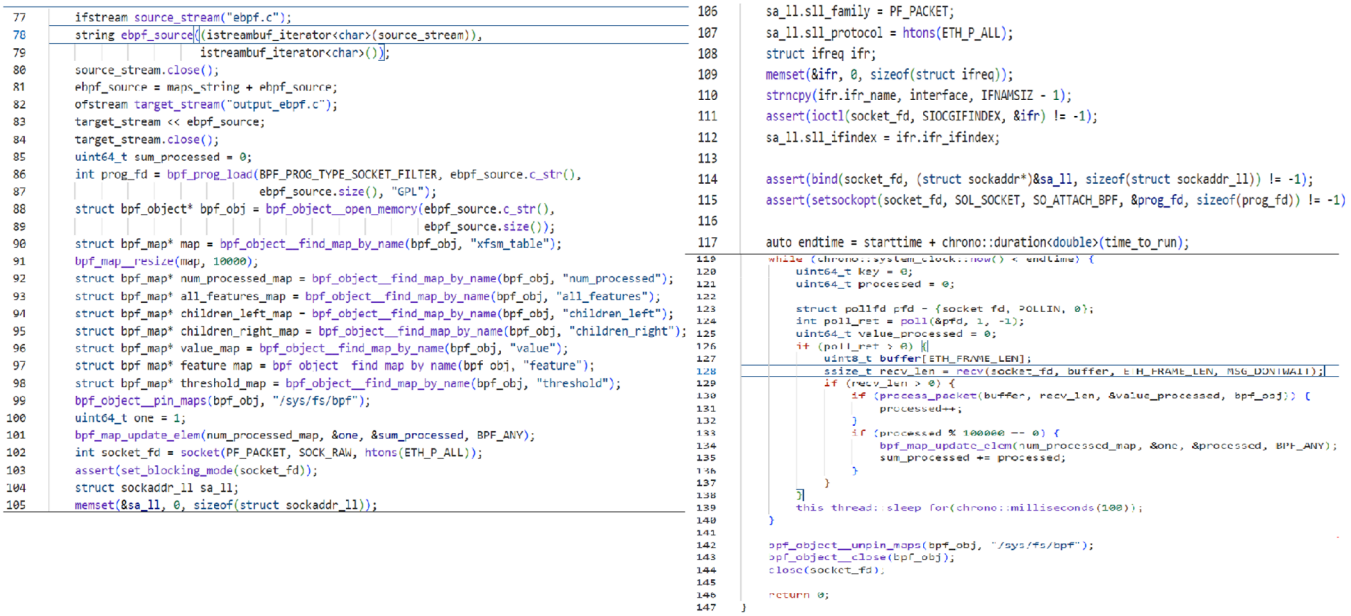
double time_to_run: (39) Declares a variable named time_to_run with a data type of double. It's likely intended to store a duration or timestamp related to the time taken to run some part of the program.

const char interface[] = "eth0": (40) Declares a constant character array named interface and initializes it with the string "eth0". It likely represents a network interface, possibly specifying the network connection for the program.

string prefix_path = "./decision_tree/Dec29_191536_hyperion_0_3/": (41) Declares a std::string named prefix_path and initializes it with a directory path. It represents a prefix path used in constructing file paths within the program.

std::vector <int64_t >read_file(string filename): (42-54) This declares a function named read_file that takes a std::string parameter filename and returns a vector of 64-bit integers (int64_t).

int main(int argc, char *argv[]): (55) This is the entry point of the program. It takes command-line arguments argc (argument count) and argv (argument vector). argc represents the number of command-line arguments, and argv is an array of strings containing those arguments.



```
77    ifstream source_stream("ebpf.c");
78    string ebpf_source((istreambuf_iterator<char>(source_stream)),
79                       istreambuf_iterator<char>());
80    source_stream.close();
81    ebpf_source = maps_string + ebpf_source;
82    ofstream target_stream("output_ebpf.c");
83    target_stream << ebpf_source;
84    target_stream.close();
85    uint64_t sum_processed = 0;
86    int prog_fd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER, ebpf_source.c_str(),
87                                ebpf_source.size(), "GPL");
88    struct bpf_object* bpf_obj = bpf_object__open_memory(ebpf_source.c_str(),
89                                                         ebpf_source.size());
90    struct bpf_map* map = bpf_object__find_map_by_name(bpf_obj, "xfsm_table");
91    bpf_map__resize(map, 10000);
92    struct bpf_map* num_processed_map = bpf_object__find_map_by_name(bpf_obj, "num_processed");
93    struct bpf_map* all_features_map = bpf_object__find_map_by_name(bpf_obj, "all_features");
94    struct bpf_map* children_left_map = bpf_object__find_map_by_name(bpf_obj, "children_left");
95    struct bpf_map* children_right_map = bpf_object__find_map_by_name(bpf_obj, "children_right");
96    struct bpf_map* value_map = bpf_object__find_map_by_name(bpf_obj, "value");
97    struct bpf_map* feature_map = bpf_object__find_map_by_name(bpf_obj, "feature");
98    struct bpf_map* threshold_map = bpf_object__find_map_by_name(bpf_obj, "threshold");
99    bpf_object__pin_maps(bpf_obj, "/sys/fs/bpf");
100   uint64_t one = 1;
101   bpf_map_update_elem(num_processed_map, &one, &sum_processed, BPF_ANY);
102   int socket_fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
103   assert(set_blocking_mode(socket_fd));
104   struct sockaddr_ll sa_ll;
105   memset(&sa_ll, 0, sizeof(struct sockaddr_ll));
106   sa_ll.sll_family = PF_PACKET;
107   sa_ll.sll_protocol = htons(ETH_P_ALL);
108   struct ifreq ifr;
109   memset(&ifr, 0, sizeof(struct ifreq));
110   strncpy(ifr.ifr_name, interface, IFNAMSIZ - 1);
111   assert(ioctl(socket_fd, SIOCGIFINDEX, &ifr) != -1);
112   sa_ll.sll_ifindex = ifr.ifr_ifindex;
113
114   assert(bind(socket_fd, (struct sockaddr*)&sa_ll, sizeof(struct sockaddr_ll)) != -1);
115   assert(setsockopt(socket_fd, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd)) != -1)
116
117   auto endtime = starttime + chrono::duration<double>(time_to_run);
118   while (chrono::system_clock::now() < endtime) {
119       uint64_t key = 0;
120       uint64_t processed = 0;
121
122       struct pollfd pfd = {socket_fd, POLLIN, 0};
123       int poll_ret = poll(&pfd, 1, -1);
124       uint64_t value_processed = 0;
125       if (poll_ret > 0) {
126           uint8_t buffer[ETH_FRAME_LEN];
127           ssize_t recv_len = recv(socket_fd, buffer, ETH_FRAME_LEN, MSG_DONTWAIT);
128           if (recv_len > 0) {
129               if (process_packet(buffer, recv_len, &value_processed, bpf_obj)) {
130                   processed++;
131               }
132               if (processed % 10000 == 0) {
133                   bpf_map_update_elem(num_processed_map, &one, &processed, BPF_ANY);
134                   sum_processed += processed;
135               }
136           }
137       }
138       this_thread::sleep_for(chrono::milliseconds(100));
139   }
140
141   bpf_object__unpin_maps(bpf_obj, "/sys/fs/bpf");
142   bpf_object__close(bpf_obj);
143   close(socket_fd);
144
145   return 0;
146
147
```

FIGURE 3 eBPF Wrapper Code part-2.

auto starttime = chrono::system_clock::now(): (56) This declares a variable starttime and initializes it with the current time using the <chrono >library.It's used to measure the start time of the program execution.

assert(argc >= 2): (57) This is an assertion that checks if there are at least two command-line arguments. If not, the program will terminate with an error message.

sscanf(argv[1], "%lf", &time_to_run): (58) This uses sscanf to parse the second command-line argument (argv[1]) as a double (%lf format specifier). The parsed value is stored in the time_to_run variable.

#ifdef USERSPACE (65-77) This is a preprocessor directive that checks if the macro USERSPACE is defined. If USERSPACE is defined, the code block following #ifdef will be included during preprocessing.

Map initialization block for userspace: The code block within #ifdef USERSPACE initializes BPF (Berkeley Packet Filter) maps for userspace.BPF maps are used for communication between the kernel and userspace in eBPF (extended Berkeley Packet Filter) programs. The maps_string variable is a string concatenation of BPF map definitions.

xfsm_table An LRU hash table (lru_hash) mapping XFSMTableKey to XFSMTableLeaf.

num_processed An array of 64-bit unsigned integers with size 1.all_features, children_left, children_right, value, feature, threshold: Arrays of 64-bit signed integers with sizes based on vectors obtained earlier.

String concatenation: The to_string function is used to convert the sizes of vectors (children_left.size(), children_right.size() etc.) to strings. These sizes are dynamically incorporated into the BPF array definitions in the maps_string.

End of #ifdef USERSPACE: The #endif statement marks the end of the conditional compilation block. If USERSPACE is not defined, this block will be excluded during preprocessing.

Read eBPF source code: (77-105) Opens the file "ebpf.c" and reads its content into a string (ebpf_source). Closes the file stream.

Append additional map initialization to eBPF source code: Concatenates the additional map initialization string (maps_string) to the beginning of the eBPF source code.

Write modified eBPF source code to a new file: Opens a new file "output_ebpf.c" and writes the modified eBPF source code to it.Closes the file stream.

Initialize variables and load eBPF program: Initializes a variable (sum_processed) to 0. Loads the eBPF program using bpf_prog_load and obtains a program file descriptor (prog_fd).

Open eBPF object from memory: Opens an eBPF object from memory using the eBPF source code and its size.

Find and resize BPF maps: Finds specific BPF maps by name in the eBPF object.Resizes the "xfsm_table" map to have a maximum of 10000 entries.Similar operations are performed for other maps.

Pin BPF maps to /sys/fs/bpf: Pins the BPF maps to the "/sys/fs/bpf" directory.

Update num processed map with initial value: Updates the "num_processed" map with the initial value of sum_processed (0).

Create raw socket for packet capture: Creates a raw socket for packet capture (socket_fd) with protocol ETH_P_ALL. Sets the socket to blocking mode using set_blocking_mode.

Initialize sockaddr_ll structure: Initializes a sockaddr_ll structure (sa_ll) and sets its memory to zero.

Setting socket address structure (sa_ll): (106-117) Initializes a sockaddr_ll structure (sa_ll) for a packet socket. sll_family is set to PF_PACKET, indicating the packet socket family. sll_protocol is set to ETH_P_ALL, indicating that the socket should capture all protocols.

Setting interface index: Declares a structure ifr of type ifreq to obtain information about the network interface.copies the interface name (interface) to ifr.ifr_name. Uses ioctl with SIOCGIFINDEX to get the interface index and stores it in sa_ll.sll_ifindex.

Binding socket: Binds the socket (socket_fd) to the specified interface using bind. The socket address structure (sa_ll) is cast to (struct sockaddr*).

Attaching BPF program to socket: Attaches the eBPF program (identified by prog_fd) to the socket using setsockopt.This step allows the eBPF program to filter and process incoming packets.

Setting end time: Calculates the end time by adding the specified duration (time_to_run) to the start time (starttime). This is used to define the time window during which the program captures packets.

while loop: (119-147) The loop continues as long as the current system time is before the specified end time (endtime). The purpose is to run packet capture and processing within a time window.

Packet reception using poll: Initializes a pollfd structure (pfd) for monitoring events on the socket (socket_fd). poll is used to wait for events on the socket. It blocks indefinitely (-1 timeout).

Packet reception and processing: Receives a packet into the buffer using the recv function. The MSG_DONTWAIT flag is used to make the recv call non-blocking.

Processing the received packet: Calls the process_packet function to handle the received packet. Increments the processed counter if the packet processing is successful.

Periodic update of processed packets: Periodically (every 100,000 processed packets), updates the value in a BPF map (num_processed_map) with the total processed count.

Sleeping between iterations: Introduces a 100 milliseconds sleep between iterations to avoid busy waiting.

Cleanup and closing: After the loop, it unpins the BPF maps and closes the BPF object and the socket.

3.1 | Testbed setup

The required system specifications are "Ubuntu OS 22.04 operating system with 4 GB RAM and Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz processor". The C program was written and compiled using GCC (GNU Compiler Collection) version 11.3.0. The experiments were conducted using Python 3.10.6.

TABLE 2 Overall dataset description of CIC-IDS-2017.

No.	Classes	No. of records	% of data
1	"BENIGN"	2273097	80.30%
2	"DoS Hulk"	231073	8.16%
3	"DDoS"	128027	4.52%
4	"DoS GoldenEye"	10293	0.36%
5	"DoS slowloris"	5796	0.20%
6	"DoS Slowhttptest"	5499	0.19%
7	"PortScan"	158930	5.61%
8	"FTP-Patator"	7938	0.28%
9	"SSH-Patator"	5897	0.21%
10	"Bot"	1966	0.06%
11	"Web Attack-Brute Force"	1507	0.05%
12	"Web Attack-XSS"	652	0.02%
13	"Infiltration"	36	0.001%
14	"Web Attack-Sql Injection"	21	0.0007%
15	"Heartbleed"	11	0.0003%

3.2 | Feature extraction

We are using the CIC-IDS-2017 dataset and given description in Table 2, a network traffic dataset containing labeled network traffic data, including benign traffic and various types of attacks. The dataset is designed to be used for intrusion detection and prevention purposes. By using this dataset, we can detect DoS/DDoS attacks. First, we should analyze the CIC-IDS-2017 dataset to get some essential features that can be trained into machine learning algorithms. Training is done by using the Analysis of Variance (ANOVA) F-test technique. The CIC-IDS-2017 dataset is a collection of network traffic data consisting of 78 features. These features are destination port, total forward packets, total backward packets, minimum packet length etc. This dataset is commonly used for testing intrusion detection systems. In previous work, the author [23] used 12 features of the overall packet inspection of the CIC-IDS-2017 dataset for the DT model, which is used in eBPF. So, we worked on the top 15 features of the comprehensive packet inspection of the CIC-IDS-2017 dataset and the DoS/DDoS attack of the CIC-IDS-2017 dataset for better performance for DT, RF, SVM, and Twin-SVM training. But for SVM and Twin-SVM, it takes more time and consumes the entire RAM for 15 feature training, so we reduced it to 10 features for training and testing. To detect DoS/DDoS attacks using the CIC-IDS-2017 dataset, we first analyze the dataset; then, by using the ANOVA F-test, we extract the top 15 features from 78 features for DT and RF and the top 10 features from 78 features for SVM and TwinSVM. ANOVA F-test determines features that have the most significant impact on the target variable. ANOVA is a statistical method that compares the means of multiple groups to determine if there are substantial differences among them. The selection process involves calculating the F-value for each feature using ANOVA. The F-value

TABLE 3 Extracted top 15 features description of CIC-IDS-2017 Dataset for DT and RF.

F.No	Feature	Description
1	"Idle Min"	"Minimum idle time observed in a network flow"
2	"Bwd Packet Length Min"	"Minimum length of the backward packets"
3	"Idle Mean"	"Average idle time in a network flow"
4	"Fwd IAT Total"	"Forward Inter-arrival Time Total"
5	"Bwd Packet Length Mean"	"Average length of the backward packets"
6	"Fwd IAT Mean"	"Average Inter-arrival time between consecutive forward packets"
7	"Min Packet Length"	"Smallest length observed among all the packets"
8	"Packet Length Mean"	"Average length of all the packets"
9	"Fwd IAT Max"	"Maximum Inter-arrival time between consecutive forward packets"
10	"Average Packet Size"	"Mean packet size observed in a network flow"
11	"Max Packet Length"	"Maximum length observed among all the packets"
12	"Packet Length Variance"	"Variance of packet lengths in a network flow"
13	"Avg Bwd Segment Size"	"Average size of backward segments in a network flow"
14	"Bwd Packet Length Max"	"Maximum length of the backward packets"
15	"Idle Max"	"Maximum idle time observed in a network flow"

represents the ratio of the variation between groups to the variation within groups, if the features with high F-values indicate strong dependence on the target variable. By using ANOVA F-test feature selection, we can reduce the dimensionality of the dataset.

Based on several considerations, we are using the top 15 or 10 features, extracted using an ANOVA F-test, instead of all features from the CIC-IDS-2017 dataset.

1. **Dimensionality reduction**: The CIC-IDS-2017 dataset contains 78 features; using too many features sometimes leads to overfitting, which will reduce the model's accuracy.
2. **Feature relevance**: All features in the dataset may not be equally informative or relevant for data analysis. Using the ANOVA F-test helps identify features that have a more significant impact on the target variable.
3. **Complexity and resource constraints**: Including all 78 features in the eBPF implementation might increase the complexity of the code. eBPF programs are typically designed to run efficiently within limited resource constraints. Using more features could lead to increased memory usage, longer processing times, and performance issues.

However, there are some drawbacks to using only the top 15 or top 10 features because the model might only detect the attack if the features of malicious packets are represented in the top 15 or top 10 features.

By using the ANOVA F-test technique on the CIC-IDS-2017 dataset, we extracted the top 15 features for Random Forest and Decision Tree algorithms as shown in Table 3. The ANOVA test on the top 15 features on the CIC-IDS-2017 dataset is shown in Figure 4 and on the CIC-IDS-2017 dataset with DoS/DDoS is shown in Figure 5. We extracted the top 10 features for SVM and TwinSVM algorithms as shown in Table 4.

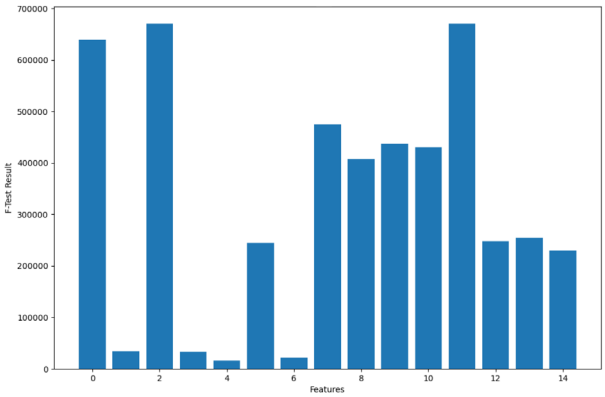


FIGURE 4 ANOVA F-test on top 15 features of overall (all packets inspection) dataset for Random Forest and Decision Tree.

The ANOVA test on the top 10 features on the CIC-IDS-2017 dataset is shown in Figure 6, and the CIC-IDS-2017 dataset with DDoS/DoS is shown in Figure 7.

3.3 | Decision Tree

We utilize the popular CIC-IDS-2017 dataset, which is available at [51], which contains the top 15 features for data analysis and prediction. We use sci-kit-learn to train the DT with a train/test split of 80% and 20%, respectively, with a maximum depth of 15 and a maximum number of leaves of 1,000. Data distribution where Label 0 is "Benign" and Label 1 is "Attack" is shown in Figure 8 and Data distribution for DoS/DDoS where Label 0 is "Benign" and Label 1 is "Attack" is shown in Figure 9. After training and testing with the DT, we export the model parameters (left children, right children, threshold, features, and

TABLE 4 Extracted top 10 features description of CIC-IDS-2017 dataset for SVM and TwinSVM.

No	Feature	Description
1	"Idle Mean"	"Average idle time in a network flow"
2	"Fwd IAT Max"	"Maximum Inter-arrival time between consecutive forward packets"
3	"Packet Length Mean"	"Average length of all the packets"
4	"Idle Min"	"Minimum idle time observed in a network flow"
5	"Bwd Packet Length Max"	"Maximum length of the backward packets"
6	"Max Packet Length"	"Maximum length observed among all the packets"
7	"Avg Bwd Segment Size"	"Average size of backward segments in a network flow"
8	"Packet Length Variance"	"Variance of packet lengths in a network flow"
9	"Idle Max"	"Maximum idle time observed in a network flow"
10	"Bwd Packet Length Mean"	"Average length of the backward packets"

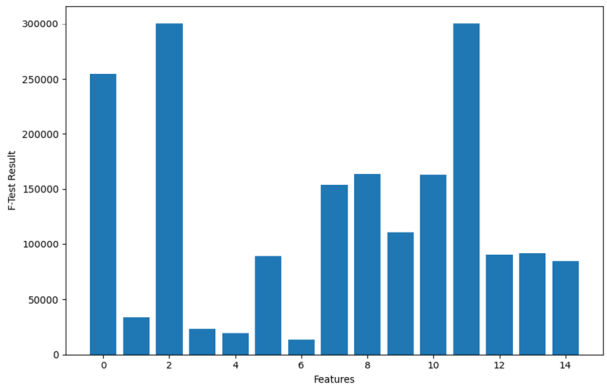


FIGURE 5 ANOVA F-test on top 15 features of DoS/DDoS dataset for Random Forest and Decision Tree.

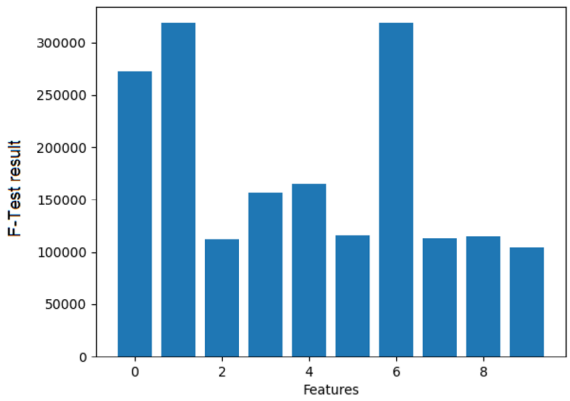


FIGURE 7 ANOVA F-test on top 10 features of DoS/DDoS dataset for SVM and TwinSVM.

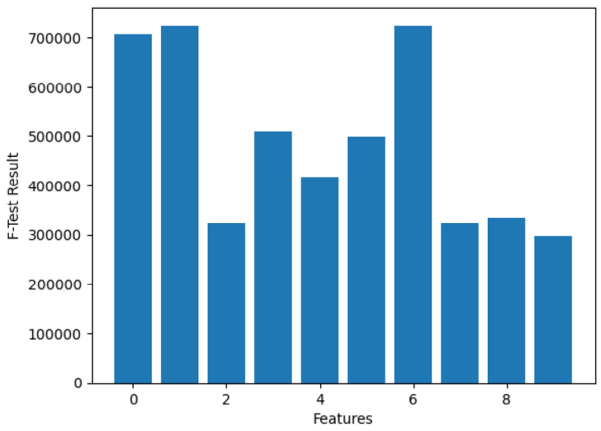


FIGURE 6 ANOVA F-test on top 10 features of overall (all packets inspection) dataset for SVM and TwinSVM.

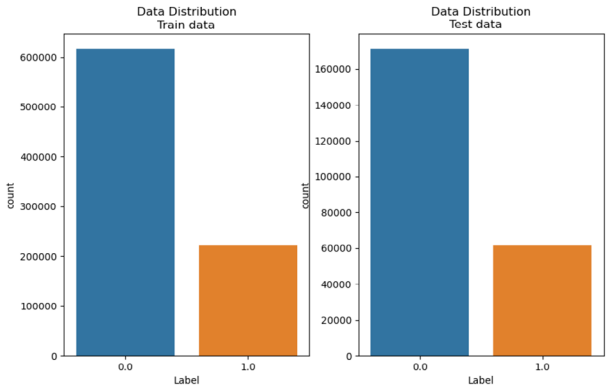


FIGURE 8 Data distribution of top 15 features in CIC-IDS-2017 dataset where Label 0 is Benign and Label 1 is Attacks.

value) of each DT. The proposed DT algorithm is given in Algorithm 1. The overall (all packets inspection) dataset and DoS/DDoS performance parameters are given in Tables 5 and 6, respectively.

3.4 | Decision Tree in eBPF

After getting the parameters of the DT model, we write an eBPF program to process the incoming malicious packets. In this eBPF program, we created eBPF maps to store the parameters of the DT model like left children, right children, threshold, features, value etc. The eBPF program code is loaded and

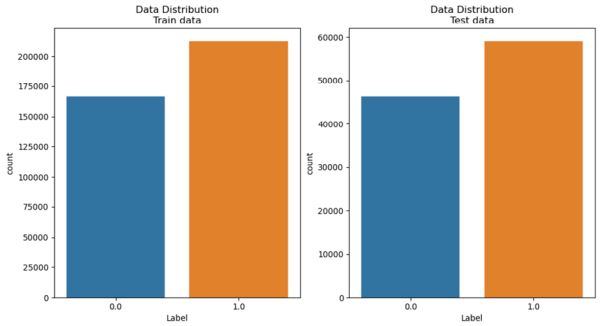


FIGURE 9 Data distribution of top 15 features for DoS/DDoS in CIC-IDS-2017 dataset where Label 0 is Benign and Label 1 is Attacks.

ALGORITHM 1 Proposed Decision Tree Algorithm

Initialize *children_left*, *children_right*, *threshold*, *feature* and *value* from model parameters
Initialize *real_feature_value* from packet data
Input: *current_node* ← 0
for *i* = 0 to *MAX_TREE_DEPTH* − 1 **do**
  current_left_child ← *children_left[current_node]*
  current_right_child ← *children_right[current_node]*
  current_feature ← *feature[current_node]*
  current_threshold ← *threshold[current_node]*
  if *current_left_child* = *TREE_LEAF* || *current_right_child* = *TREE_LEAF* **then**
    break
  else
    real_feature_value ← *[current_feature]*
    if *real_feature_value* ≤ *current_threshold* **then**
      current_node ← *current_left_child*
    else
      current_node ← *current_right_child*
    end if
  end if
end for
correct_value = value[current_node]
prediction ← 1 if *correct_value*=1 else 0
Output: 0 or 1 Prediction

combined with other necessary code, such as maps and tables. eBPF maps store left and right children, thresholds, features, and value parameters and can access data within the eBPF program. The eBPF program, along with its associated maps, is loaded into the kernel. The program is attached to a specific hook or event point within the kernel, socket. The socket where it sends and receives packets from network interfaces. Each captured packet is passed to the eBPF program, which is attached to the socket for packet filtering as shown in Figure 10. Time taken in user space Vs. Kernel space of overall (all packets inspection) and DoS/DDoS detection meantime for packet/s is shown in Tables 7 and 8.

TABLE 5 Experimental results of overall (all packets inspection) CIC-IDS-2017 dataset in userspace.

Performance parameters	Method	DT	RF	SVM	TwinSVM
Accuracy	Train	99.52	99.59	88.77	93.87
	Test	99.38	99.44	88.74	93.82
Precision	Train	99.71	99.74	78.97	98.58
	Test	99.46	99.51	78.97	98.49
Recall/sensitivity	Train	98.49	98.72	73.41	75.9
	Test	98.21	98.37	73.26	75.78
F1 score	Train	99.09	99.23	76.09	85.76
	Test	98.83	98.94	76.01	85.65
Specificity	Train	99.89	99.91	93.71	99.65
	Test	99.81	99.82	93.73	99.63

TABLE 6 Experimental results of DoS/DDoS of CIC-IDS-2017 dataset in userspace.

Performance parameters	Method	DT	RF	SVM	TwinSVM
Accuracy	Train	99.73	99.82	84.43	88.49
	Test	99.57	99.58	84.43	88.42
Precision	Train	99.87	99.88	84.28	98.53
	Test	99.71	99.65	84.31	98.49
Recall/sensitivity	Train	99.65	99.80	86.74	79.42
	Test	99.52	99.60	86.70	79.33
F1 score	Train	99.76	99.84	85.49	87.95
	Test	99.62	99.62	85.48	87.88
Specificity	Train	99.83	99.85	81.83	98.67
	Test	99.64	99.55	81.88	98.63

TABLE 7 Time taken in user space vs kernel space of overall (all packet inspection) CIC-IDS-2017 dataset.

Algorithms	User space (mean)	eBPF (mean)
Decision Tree packet/s	46,239	109,691
Random Forest packet/s	45,978	108,534
SVM packet/s	45,590	92,978
TwinSVM packet/s	38,430	109,865

3.5 | Random Forest model

We implemented the RF model on the CIC-IDS-2017 dataset, with the top 15 data analysis and prediction features. Data distribution with Label 0 is "Benign" and with Label 1 is "Attack" is shown in Figure 8 and Data distribution for DoS or DDoS with Label 0 is "Benign" and with Label 1 is "Attack" is shown in Figure 9. After training and testing with RF, we export model parameters: left children, right children, threshold, features, and value of each DT. The proposed RF algorithm is given in Algorithm 2. The overall (all packet inspection) dataset and

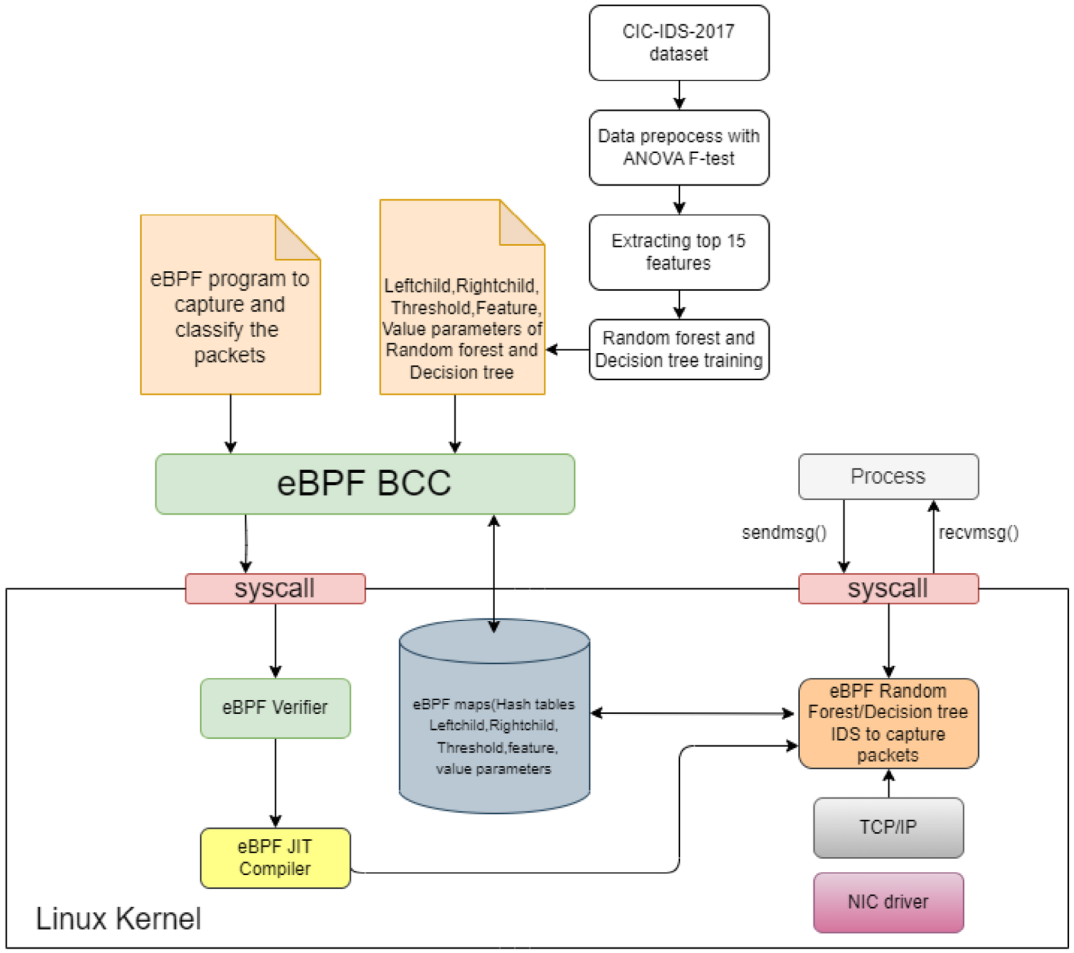


FIGURE 10 Proposed model using RF and DT algorithms.

TABLE 8 Time taken in user space vs kernel space of DoS/DDoS of CIC-IDS-2017 dataset.

Algorithms	Userspace (mean)	eBPF (mean)
Decision Tree packet/s	42,463	106,421
Random Forest packet/s	41,632	105,245
SVM packet/s	49,376	92,581
TwinSVM packet/s	42,487	117,536

DoS/DDoS detection performance parameters are shown in Tables 5 and 6.

3.6 | Random Forest algorithm in eBPF

After getting the parameters of the RF model, we write an eBPF program to process the incoming malicious packets. In this eBPF program, we created eBPF maps to store the parameters of the RF model, such as left children, right children, threshold, features, and value of each DT. The eBPF program code is loaded and combined with other necessary code, such as maps and tables. eBPF maps store left and right children, thresholds, features, and value parameters and can access data within the eBPF program. The eBPF program, along with its associated maps, is loaded into the kernel. The program is attached to a specific hook or event point within the kernel, such as a socket. The socket where it sends and receives packets from network interfaces. Each captured packet is passed to the eBPF program, which is attached to the socket for packet filtering, shown in Figure 10. Time taken in user space Vs. Kernel space of overall (all packets inspection) and DoS/DDoS detection mean time for packet/s is shown in Tables 7 and 8.

3.7 | SVM and TwinSVM

We trained machine learning algorithms, SVM, and TwinSVM in Python on the CIC-IDS-2017 dataset with the top 10 features. Data distribution with Label 0 is "Benign" and with Label 1 is "Attack" is shown in Figure 11 and Data distribution for DOS/DDOS with Label 0 is "Benign" and with Label 1 is "Attack" is shown in Figure 12. Then, we deploy the models in eBPF and user space to compare the performance regarding the number of packets processed in a timeframe. One of the main hurdles in deploying these algorithms was representing the weights, which are floating point numbers, in eBPF.

ALGORITHM 2 Proposed Random Forest Algorithm

Initialize *children_left*, *children_right*, *threshold*, *feature* and *value* from model parameters
Initialize *real_feature_value* from packet data
Initialize *tree_number* from model parameters
Input: *current_node* ← 0
Input: *True_count* ← 0
Input: *False_count* ← 0
for *tree_number* = 0 to *n_estimators* − 1 **do**
  for *i* = 0 to *MAX_TREE_DEPTH* − 1 **do**
    current_left_child ← *children_left[current_node,tree_number]*
    current_right_child ← *children_right[current_node,tree_number]*
    current_feature ← *feature[current_node,tree_number]*
    current_threshold ← *threshold[current_node,tree_number]*
    if *current_left_child* = *TREE_LEAF* || *current_right_child* = *TREE_LEAF* **then**
      break
    else
      real_feature_value ← *[current_feature,tree_number]*
      if *real_feature_value* ≤ *current_threshold* **then**
        current_node ← *current_left_child*
      else
        current_node ← *current_right_child*
      end if
    end if
  end for
end for
for *tree_number* = 0 to *n_estimators* − 1 **do**
  correct_value = value[current_node,tree_number]
  if ∗ correct_value **then**
    True_count ← True_count + 1
  else
    False_count ← False_count + 1
  end if
end for
if True_count > False_count **then**
  correct_value = 1
else
  correct_value = 0
end if
prediction ← 1 if *correct_value*=1 else 0
Output: 0 or 1 Prediction

eBPF does not allow floating-point numbers, so instead, we use a fixed-point representation of these floating-point numbers, which is nothing but multiplying all the weights with some large values (in our case, 216). This method has no problem for linear SVM and TwinSVM since they compare distances using these weights. However, Neural Networks (NN) and XGBoost

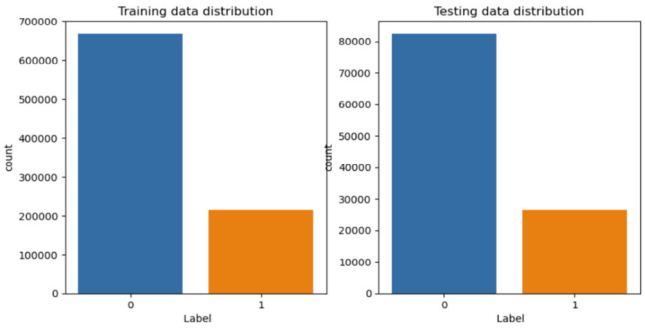


FIGURE 11 Data distribution of top 10 features in CIC-IDS-2017 dataset with Label 0 is Benign and Label 1 is Attacks.

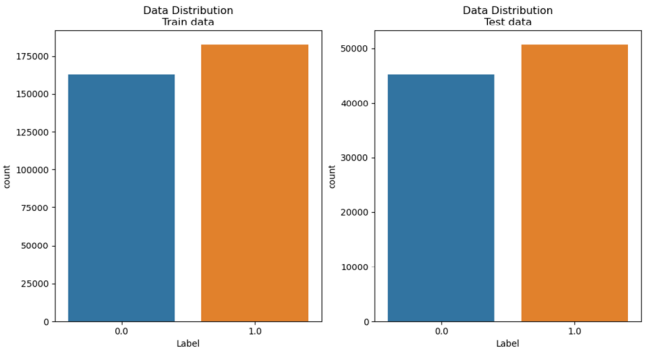


FIGURE 12 Data distribution of top 10 features for DoS/DDoS in CIC-IDS-2017 dataset with Label 0 is Benign and Label 1 is Attacks.

ALGORITHM 3 Proposed SVM Algorithm

Initialize *coefficients* from model parameters
Initialize *intercept* from model parameters
Initialize *features* from packet data
Input: *sum* ← 0
for *i* ← 0 to *MAX_FEATURES* − 1 **do**
  sum ← *sum* + *coefficients*[*i*] ∗ *features*[*i*]
end for
sum ← *sum* + *intercept*
prediction ← 1 if *sum* ≥ 0 else 0
Output: 0 or 1 Prediction

inherently operate in continuous values for outputs and later convert them into classes using some function like sigmoid or tanh. The final output is some function of the weights with some non-linear transformation, so we cannot use fixed point representation in NN and XGBoost. The proposed SVM and TwinSVM algorithms are given in Algorithm 3 and Algorithm 4. The overall (all packet inspection) dataset and DoS/DDoS detection performance parameters are shown in Tables 5 and 6. The Proposed models using SVM and TwinSVM algorithms are shown in Figure 13.

ALGORITHM 4 Proposed TwinSVM Algorithm

Input-1: $sum1 \leftarrow 0$
Input-2: $sum2 \leftarrow 0$
Initialize $coefficients1$ from model parameters
Initialize $coefficients2$ from model parameters
Initialize $intercept1$ from model parameters
Initialize $intercept2$ from model parameters
Initialize $w1mod$ from model parameters
Initialize $w2mod$ from model parameters
Initialize $features$ from packet data
for $i \leftarrow 0$ to $MAX\_FEATURES - 1$ **do**
  $sum1 \leftarrow sum1 + coefficients1[i] * features[i]$
  $sum2 \leftarrow sum2 + coefficients2[i] * features[i]$
end for
$sum1 \leftarrow sum1 + intercept1$
$sum2 \leftarrow sum2 + intercept2$
$prediction \leftarrow 1$ if $sum1 * w2mod\_val \geq sum2 * w1mod\_val$ else 0
Output: 0 or 1 Prediction

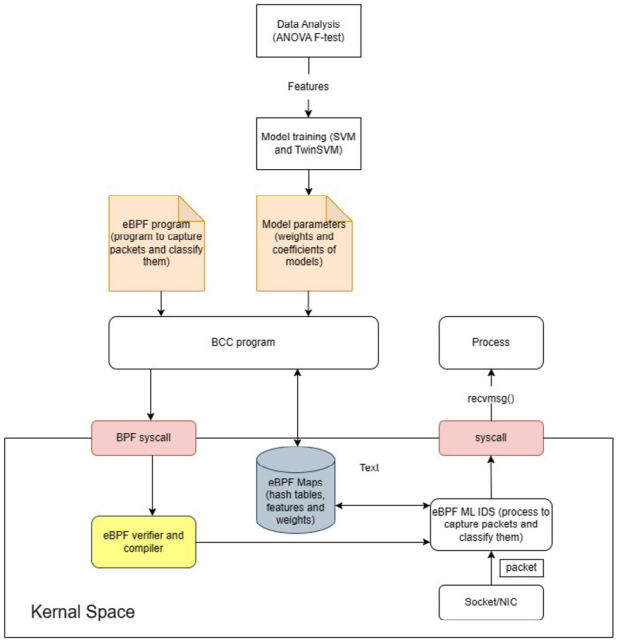


FIGURE 13 Proposed model using SVM and TwinSVM algorithms.

3.8 | SVM and TwinSVM in eBPF

We implemented SVM and TwinSVM with a different kind of training, wherein we split the dataset into batches of 100 and trained a separate model for each of the 100 data points. Then we pick the best models of these and take a mean of their weights. We assign these weights to a new model, and it outperforms the top models by a slight margin over the validation set. Time taken in user space Vs. Kernel space of overall (all packet inspection) and DoS/DDoS detection mean time for packet/s is shown in Tables 7 and 8.

4 | PERFORMANCE ANALYSIS

The performance parameters of ML algorithms are accuracy, precision, recall/sensitivity, F1-score, and specificity. Using the widely used CIC-IDS-2017 dataset with 12 features, Maximilian Bachl et al. [22, 23] train the DT using sci-kit-learn with a maximum depth of ten and a maximum number of leaves of one thousand using a train/test split of 2:1. This results in an accuracy of 99.0% on the testing dataset after training. We used the same CIC-IDS-2017 dataset and extracted the top 15 features using the ANOVA F-test method. We train the DT using sci-kit-learn with a maximum depth of 15, and a train/test split of 4:1. The comparison of accuracy in user space and kernel space is shown in Table 9. We can see in Table 9 that our experimental results show an accuracy of 99.38% when testing the dataset after training, which is better than the accuracy attained with [22, 23] (existing related work). Further, compared to DT and SVM models, RF is performing better, with an accuracy of 99.44%. Also, we can see in the table that we calculated DoS/DDoS detection, and the accuracy was found to be 99.57%.

RF is an ensemble learning method for classification and regression that can detect DoS/DDoS attacks by analyzing network traffic patterns and identifying unusual behavior that may indicate an attack. We trained RF using popular CIC-IDS-2017 with a test/train split of 1:4 with a max depth of 20. After training and testing, we export RF model parameters. This approach uses eBPF and Socket filters for DoS/DDoS mitigation. It is implemented using an RF classifier in eBPF, which helps classify incoming network packets in real time. The eBPF program attached to the socket hook can filter and record the necessary traffic features and then pass the data to user space. The algorithm then makes the prediction, and then eBPF can take the required actions, such as dropping the packets or redirecting them to a DDoS mitigation system. In this process, we achieve better RF performance than DT, SVM, and TwinSVM, 99.44 overall (all packets inspection) accuracy, and 99.58 DoS/DDoS detection accuracy.

DT is a supervised machine learning algorithm used for classification and regression purposes. It is a tree-based structure where each node represents a feature or attribute, each edge means a decision rule, and each leaf node represents prediction. We trained DT using the popular CIC-IDS-2017 with a test/train split of 1:4 with a max depth of 15. After training and testing, we export DT model parameters. In this method, we use DT in eBPF, which is attached to a specific event named Socket, which will filter network data and pass it to the user space. In this method, DT model parameters predict each packet, and then eBPF takes necessary action, such as dropping or redirecting packets. In this process, we achieve more performance compared to DT implemented by this author [22, 23] with 99.38 accuracies for the overall (all packet inspection) CIC-IDS-2017 dataset and 99.57 DoS/DDoS accuracy.

TABLE 9 Accuracy, user space, eBPF space comparison with related work.

Author	Algorithm	Accuracy	User space packet/s	eBPF packet/s
[22]	Decision Tree (DT)	97.6	–	–
[22]	Neural networks (NN)	99.1	–	–
[23]	Decision Tree (DT)	99	125,420	152,274
Proposed	Decision Tree (DT)	**99.38**	46,239	109,691
Proposed	Random Forest (RF)	**99.44**	45,978	108,534
Proposed	SVM	**88.74**	45,590	92,978
Proposed	TwinSVM	**93.82**	38,430	109,865

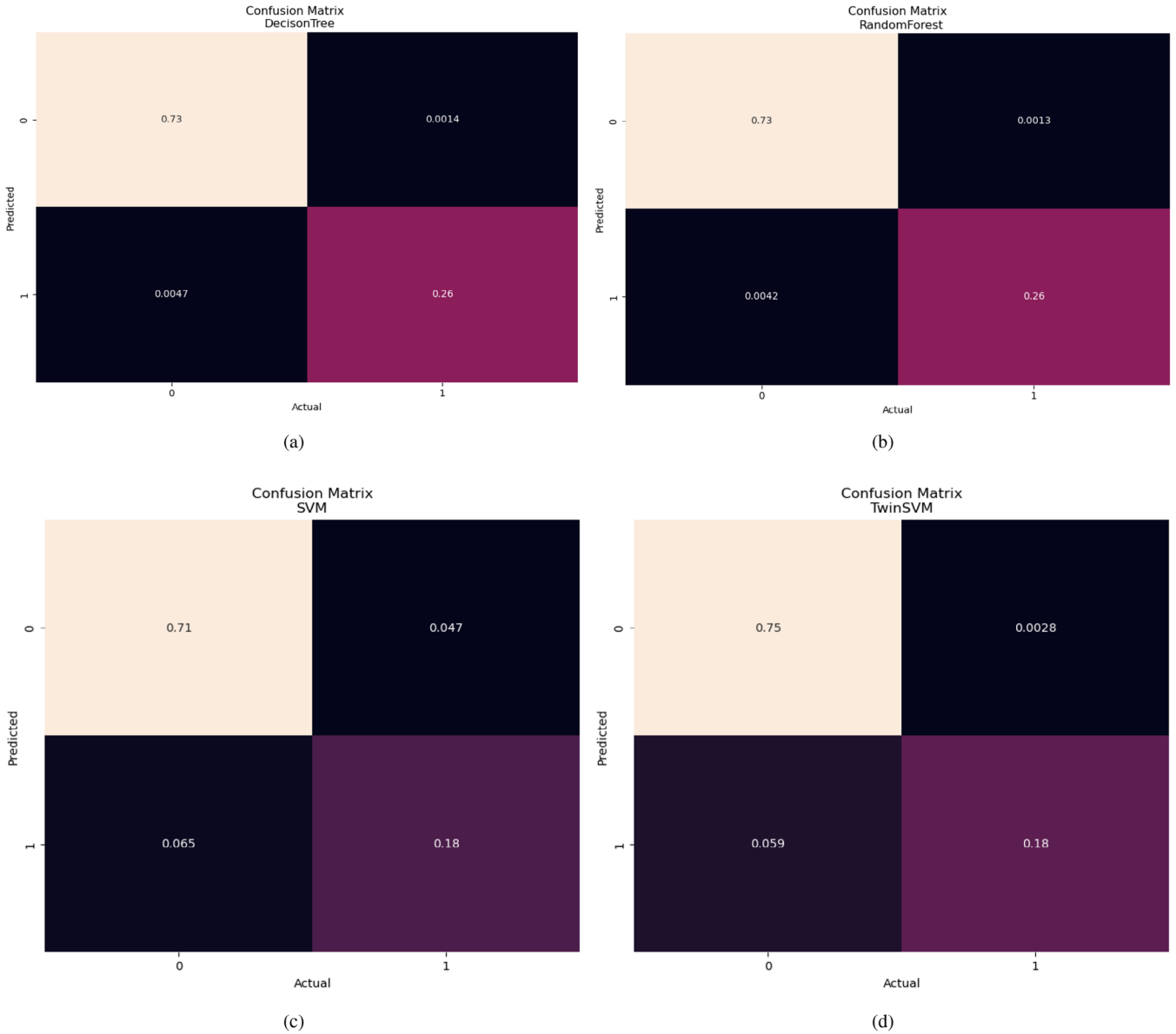


FIGURE 14 Confusion matrix for (a) Decision Tree, (b) Random Forest, (c) SVM, and (d) TwinSVM of overall (all packets inspection) CIC-IDS-2017 dataset.

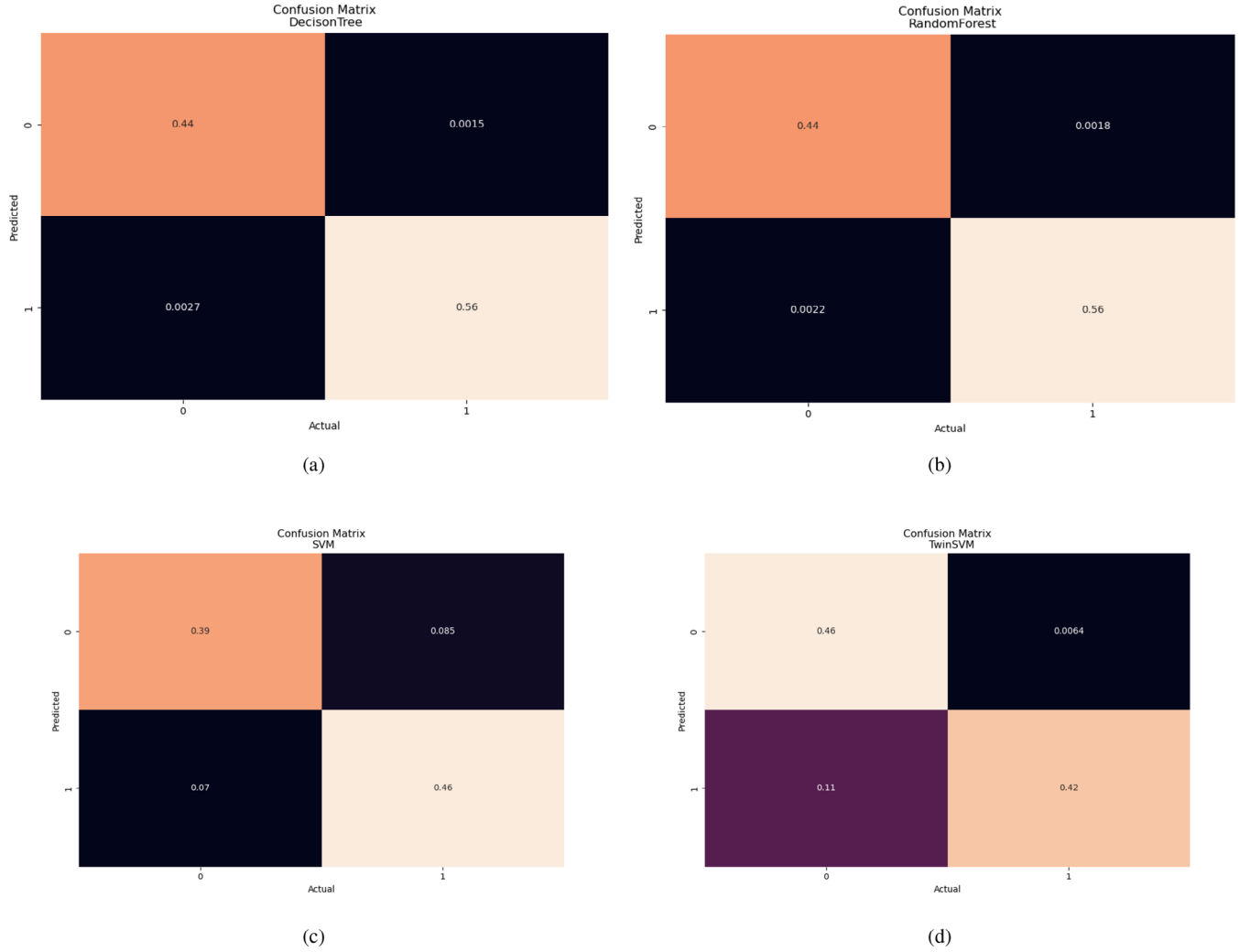


FIGURE 15 Confusion Matrix for (a) Decision Tree, (b) Random Forest, (c) SVM, and (d) TwinSVM of DoS/DDoS in CIC-IDS-2017 dataset.

SVM and TwinSVM algorithms have accuracy for the overall (all packet inspection) dataset of 88.74 and 93.82. Comprehensive dataset using different ML algorithms confusion matrix results are shown in Figure 14, and DoS/DDoS confusion matrix is shown in Figure 15. Both a traditional user space program version and eBPF version are implemented for the execution of the intrusion detection system (IDS). Both programs were executed independently for ten seconds. According to the data in Tables 7 and 8, the user space implementation examines fewer packets per second than the eBPF version.

5 | CONCLUSION

eBPF is a byte-code-based virtual machine that runs programs without modifying the kernel source code. The eBPF implementation is faster than the user space. used the CIC-IDS-2017 dataset and trained with DT, RF, SVM, and TwinSVM using scikit-learn with a test/train split of 1:4. Our experimental results indicate that the accuracy of our proposed machine learning algorithms—Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), and Twin Support Vector Machine (TwinSVM)—outperforms existing related work with accuracies of 99.38%, 99.44%, 88.73%, and 93.82%, respectively. For future research analysis, we can build an IDS that utilizes the eBPF and the XDP with ML algorithms such as DT, RF, SVM, and TwinSVM. XDP is a programmable data path interface in the Linux kernel that allows for efficient filtering and manipulation of network packets at the Network Interface Card (NIC) driver level. XDP programs are written in eBPF byte code, are attached to network devices, and can execute on the NIC before the packet reaches the kernel network stack and act on the packet directly on the NIC.

AUTHOR CONTRIBUTIONS

Anand Nemalikanti: Conceptualization; methodology. **Saifulla M A**: Conceptualization; methodology; supervision. **Pavan Kumar Aakula**: Data curation; methodology. **Raveendra Babu Ponnuru**: Data curation; supervision; writing—original draft; writing—review and editing. **Rizwan Patan**:

Supervision; writing—review and editing. **Rama Prakasha Reddy Chegireddy**: Supervision; writing—review and editing.

CONFLICT OF INTEREST STATEMENT

The authors declare no conflicts of interest.

DATA AVAILABILITY STATEMENT

Data openly available in a public repository that does not issue DOIs. The experimental code is available at the GitHub repository link: https://github.com/NemalikantiAnand/Project.

ORCID

Raveendra Babu Ponnuru https://orcid.org/0000-0001-5340-8526
Rizwan Patan https://orcid.org/0000-0003-4878-1988
Chegireddy Rama Prakasha Reddy https://orcid.org/0000-0002-6413-5225

REFERENCES

1. Babu, P.R., Palaniswamy, B., Reddy, A.G., Odelu, V., Kim, H.S.: A survey on security challenges and protocols of electric vehicle dynamic charging system. Secur. Privacy 5(3), e210 (2022)
2. Babu, P.R., Amin, R., Reddy, A.G., Das, A.K., Susilo, W., Park, Y.: Robust authentication protocol for dynamic charging system of electric vehicles. IEEE Trans. Veh. Technol. 70(11), 11338–11351 (2021)
3. Miano S., Bertrone M., Risso F., Tumolo M., Bernal M.V.: Creating complex network services with ebpf: Experience and lessons learned. In: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), pp. 1–8. IEEE, 2018.
4. Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Câmara Júnior, E.P.M., Vieira, L.F.M.: Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. ACM Comput. Surv. 53(1), 1–36 (2020)
5. Sharaf, H., Ahmad, I., Dimitriou, T.: Extended berkeley packet filter: An application perspective. IEEE Access 10, 126370–126393 (2022)
6. Anand, N., Saifulla, M.A.: A probabilistic method to identify http/1.1 slow rate dos attacks. In: International Conference on Communication and Intelligent Systems, pp. 17–28. Springer, Singapore (2022)
7. Anand, N., Saifulla, M.A., Ashok Reddy, G.R., Pavan, P.: Effective encrypted traffic analysis. In: International Conference on Advanced Computing and Intelligent Engineering, pp. 395–400. Springer, Singapore (2022)
8. Pavan, P., Saifulla, M.A., Anand, N.: Improvising encrypted traffic analysis using stacking ensemble model. In: International Conference on Advanced Computing and Intelligent Engineering, pp. 387–394. Springer, Singapore (2022)
9. Anand, N., Saifulla, M.A.: En-lakp: Lightweight authentication and key agreement protocol for emerging networks. IEEE Access 11, 28645–28657 (2023)
10. Anand, N., Saifulla, M.A.: An efficient ids for slow rate http/2.0 dos attacks using one class classification. In: 2023 IEEE 8th International Conference for Convergence in Technology (I2CT), pp. 1–9. IEEE, Piscataway (2023)
11. Høiland-Jørgensen, T., Brouer, J.D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D.: The express data path: Fast programmable packet processing in the operating system kernel. In: Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies, pp. 54–66. ACM, New York (2018)
12. Cvitić, I., Perakovic, D., Gupta, B.B., Choo, K.-K.R.: Boosting-based ddos detection in internet of things systems. IEEE Internet Things J. 9(3), 2109–2123 (2021)
13. Singh, A., Gupta, B.B.: Distributed denial-of-service (ddos) attacks and defense mechanisms in various web-enabled computing platforms: issues, challenges, and future research directions. Int. J. Semantic Web Inf. Syst. (IJSWIS) 18(1), 1–43 (2022)
14. Mishra, A., Gupta, N., Gupta, B.B.: Defense mechanisms against ddos attack based on entropy in sdn-cloud using pox controller. Telecommun. Syst. 77(1), 47–62 (2021)
15. Mishra, A., Joshi, B.K., Arya, V., Gupta, A.K., Chui, K.T.: Detection of distributed denial of service (ddos) attacks using computational intelligence and majority vote-based ensemble approach. Int. J. Softw. Sci. Comput. Intell. (IJSSCI) 14(1), 1–10 (2022)
16. Lattner, C., Adve, V.: The llvm compiler framework and infrastructure tutorial. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds) Languages and Compilers for High Performance Computing. Lecture Notes in Computer Science, vol. 3602, pp. 15–16. Springer, Berlin, Heidelberg (2005)
17. Dumazet, E.: A jit for packet filters (2011)
18. Rybczynska, M.: Bounded loops in bpf for the 5.3 kernel (2019)
19. IOVisor Project. Bcc (bpf compiler collection) (2022)
20. Miller, D.: Bpf verifier overview (2019)
21. Maguire, A.: Notes on bpf (1)–a tour of program types (2019)
22. Hara, T., Sasabe, M.: Practicality of in-kernel/user-space packet processing empowered by lightweight neural network and decision tree. Comp. Netw. 240, 110188 (2024)
23. Bachl, M., Fabini, J., Zseby, T.: A flow-based ids using machine learning in ebpf. arXiv preprint, arXiv:2102.09980 (2021)
24. Wang, S.-Y., Chang, J.-C.: Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. J. Netw. Comp. Appl. 198, 103283 (2022)
25. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al.: P4: Programming protocol-independent packet processors. ACM SIGCOMM Comp. Commun. Rev. 44(3), 87–95 (2014)
26. Roesch, M.: Snort users manual. http://www.snort.org (2002)
27. Leblond, É., Manev, P.: Introduction to ebpf and xdp support in suricata (2019)
28. Baidya, S., Chen, Y., Levorato, M.: ebpf-based content and computation-aware communication for real-time edge computing. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 865–870. IEEE, Piscataway (2018)
29. Findlay, W.: Extended berkeley packet filter for intrusion detection implementations. Ph.D. Thesis, Honours Thesis Proposal, Carleton University (2019)
30. Xhonneux, M., Duchene, F., Bonaventure, O.: Leveraging ebpf for programmable network functions with ipv6 segment routing. In: Proceedings of the 14th International Conference on emerging Networking Experiments and Technologies, pp. 67–72. ACM, New York (2018)
31. Viljoen, N., Kicinski, J.: Using ebpf as an abstraction for switching. URL http://vger.kernel.org/lpc_net2018_talks/eBPF_For_Switches.pdf (2018)
32. Miano, S., Bertrone, M., Risso, F., Bernal, M.V., Lu, Y., Pi, J.: Securing linux with a faster and scalable iptables. ACM SIGCOMM Comp. Commun. Rev. 49(3), 2–17 (2019)
33. Lazri, K., Blin, A., Sopena, J., Muller, G.: Toward an in-kernel high performance key-value store implementation. In: 2019 38th Symposium on Reliable Distributed Systems (SRDS), pp. 268–2680. IEEE, Piscataway (2019)
34. cilium: eBPF-based networking, observability, security (2022)
35. Miano, S., Doriguzzi-Corin, R., Risso, F., Siracusa, D., Sommese, R.: High-performance server-based ddos mitigation through programmable data planes.
36. Ahmed, Z., Alizai, M.H., Syed, A.A.: Inkev: In-kernel distributed network virtualization for dcn. ACM SIGCOMM Comp. Commun. Rev. 46(3), 1–6 (2018)
37. Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M.V.: Creating complex network services with ebpf: Experience and lessons learned. In: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), pp. 1–8. IEEE, Piscataway (2018)
38. Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., Carle, G.: Performance implications of packet filtering with linux ebpf. In: 2018 30th International Teletraffic Congress (ITC 30), vol. 1, pp. 209–217. IEEE, Piscataway (2018)

39. Parola, F., Risso, F., Miano, S.: Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway. In: 2021 IEEE 7th International Conference on Network Softwarization (NetSoft), pp. 221–225. IEEE, Piscataway (2021)
40. Hong, J., Jeong, S., Yoo, J.-H., Hong, J.W.-K.: Design and implementation of ebpf-based virtual tap for inter-vm traffic monitoring. In: 2018 14th International Conference on Network and Service Management (CNSM), pp. 402–407. IEEE, Piscataway (2018)
41. Miano, S., Risso, F., Bernal, M.V., Bertrone, M., Lu, Y.: A framework for ebpf-based network functions in an era of microservices. IEEE Trans. Netw. Serv. Manage. 18(1), 133–151 (2021)
42. Caviglione, L., Mazurczyk, W., Repetto, M., Schaffhauser, A., Zuppelli, M.: Kernel-level tracing for detecting stegomalware and covert channels in linux environments. Comp. Netw. 191, 108010 (2021)
43. Gallego-Madrid, J., Bru-Santa, I., Ruiz-Rodenas, A., Sanchez-Iborra, R., Skarmeta, A.: Machine learning-powered traffic processing in commodity hardware with ebpf. Comp. Netw. 243, 110295 (2024)
44. Hohlfeld, O., Krude, J., Reelfs, J.H., Rüth, J., Wehrle, K.: Demystifying the performance of xdp bpf. In: 2019 IEEE Conference on Network Softwarization (NetSoft), pp. 208–212. IEEE, Piscataway (2019)
45. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
46. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Comm. ACM 18(6), 333–340 (1975)
47. Ben-Yair, I., Rogovoy, P., Zaidenberg, N.: Ai & ebpf based performance anomaly detection system. In: Proceedings of the 12th ACM International Conference on Systems and Storage, pp. 180–180. ACM, New York (2019)
48. Demoulin, H.M., Pedisich, I., Vasilakis, N., Liu, V., Loo, B.T., Phan, L.T.X.: Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In: USENIX Annual Technical Conference, pp. 693–708. USENIX Association, Berkeley, CA (2019)
49. Wieren, H.D.: Signature-based ddos attack mitigation: Automated generating rules for extended berkeley packet filter and express data path. Master's Thesis, University of Twente (2019)
50. Choe, Y., Shin, J.-S., Lee, S., Kim, J.: ebpf/xdp based network traffic visualization and dos mitigation for intelligent service protection. In: Advances in Internet, Data and Web Technologies: The 8th International Conference on Emerging Internet, Data and Web Technologies (EIDWT-2020), pp. 458–468. Springer, Cham (2020)
51. Panigrahi, R., Borah, S.: A detailed analysis of cicids2017 dataset for designing intrusion detection systems. Int. J. Eng. Technol. 7(3.24), 479–482 (2018)

How to cite this article: Anand, N., M A, S., Aakula, P.K., Ponnuru, R.B., Patan, R., Reddy, C.R.P.: Enhancing intrusion detection against denial of service and distributed denial of service attacks: Leveraging extended Berkeley packet filter and machine learning algorithms. IET Commun. 19, e12879 (2025). https://doi.org/10.1049/cmu2.12879