Received 2 September 2025, accepted 23 September 2025,
date of publication 25 September 2025, date of current version 2 October 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3614389

RESEARCH ARTICLE

eBPF-Based Runtime Detection of Semantic DDoS Attacks in Linux Containers

S. REMYA[1], MANU J. PILLAI[2], B. NIRANJAN[2], P. M. AJITH KUMAR[2], K. MERIN SHAJU[2], K. DINOY RAJ[2], SOMULA RAMASUBBAREDDY[3], (Member, IEEE), AND YONGYUN CHO[4]

[1]Amrita School of Computing, Amrita Vishwa Vidyapeetham, Amritapuri Campus, Vallikavu, Kerala 690525, India
[2]Department of CSE, TKM College of Engineering, Kollam, Kerala 691005, India
[3]Department of Computer Science and Engineering, Symbiosis Institute of Technology, Hyderabad, Symbiosis International (Deemed University), Pune 412115, India
[4]Department of Information and Communication Engineering, Sunchon National University, Suncheon-si, Jeollanam-do 57922, South Korea

Corresponding author: Yongyun Cho (yycho@scnu.ac.kr)

This work was supported in part by the Innovative Human Resource Development for Local Intellectualization Program through the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by Korean Government through the Ministry of Science and ICT (MSIT) under Grant IITP-2025-2020-0-01489 (50%), and in part by IITP-Information Technology Research Center (ITRC) Grant funded by Korean Government through MSIT under Grant IITP-2025-RS-2023-00259703 (50%).

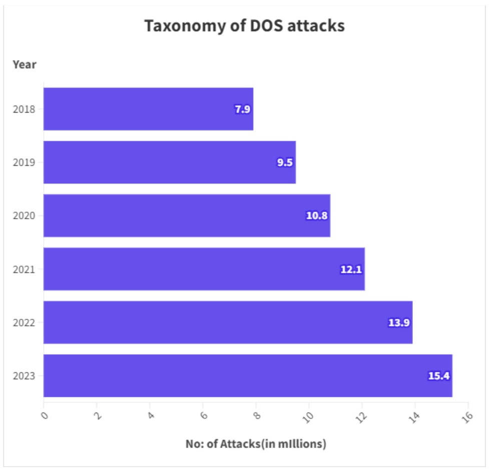
ABSTRACT Modern Distributed Denial-of-Service (DDoS)attacks increasingly target the application layer to exhaust CPU resources and disrupt service availability, particularly in containerized environments where isolation and diverse implementations complicate traditional detection mechanisms. Existing solutions like CODA(Containerized Denial-of-Service Attack detection) monitor CPU burst times between accept() and close() system calls but fail when attackers maintain persistent connections without triggering close() calls. To address this limitation, we propose CODAX (Container-aware DDoS Attack detection using eXtended Berkeley Packet Filter), a lightweight CPU-time-based detection method that identifies long-running malicious connections by monitoring system calls at the kernel level using extended Berkeley Packet Filter (eBPF) probes. Unlike prior methods such as CODA that rely on measuring CPU burst time between accept() and close() system calls, our approach tracks CPU usage thresholds from the moment of the accept() call using a global map of 64-bit Unix timestamps, enabling early detection of ongoing attacks before connection closure. The experimental evaluation demonstrates significant performance improvements over existing solutions. The proposed system achieves faster detection times, high attack detection accuracy (ADR: 0.92), and maintains low false positive rates (FPR: 0.02). Statistical validation using paired t-tests confirms that our eBPF-based approach reduces detection latency by an average of 1,772.3ms compared to CODA, representing a 94.2% relative reduction from CODA's baseline performance ($p<0.0001$). The system operates efficiently with 30% CPU utilization and demonstrates high scalability for production environments. This research provides a timely and efficient mechanism for mitigating CPU exhaustion DDoS attacks in containerized applications, offering substantial improvements in responsiveness and reliability over existing detection frameworks.

INDEX TERMS eBPF, Linux, DDoS attack, Docker, CODA, container security, real-time detection.

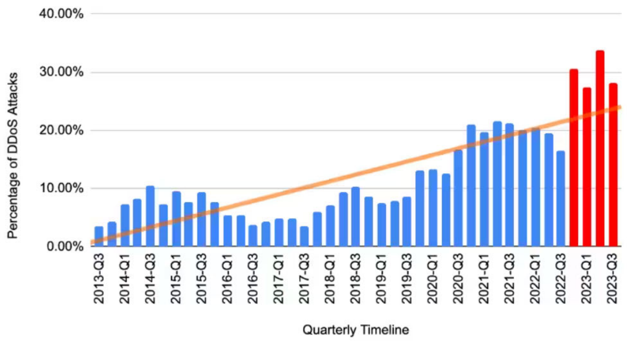
I. INTRODUCTION

The rapid growth of the Internet has established it as a fundamental pillar of modern society, driving economic growth, facilitating global communication, and enabling critical digital services. However, this increased dependency introduces significant vulnerabilities, particularly from Distributed Denial-of-Service (DDoS) attacks, which have evolved into pervasive threats to online service availability and reliability. Modern DDoS attacks attempt to overwhelm servers, services, or networks with excessive traffic, rendering them unavailable to legitimate users. Over the years,

The associate editor coordinating the review of this manuscript and approving it for publication was Hosam El-Ocla.



(a) Annual frequency of DDoS attacks (2018–2023)



(b) Quarterly percentage trend of DDoS attacks (2013–2023)

FIGURE 1. Trends in DDoS attacks over time: (a) Total number of attacks per year and (b) Percentage of DDoS attacks per quarter. [Source: Cisco Annual Internet Report].

these attacks have evolved significantly, growing in volume, complexity, and sophistication, making them a major concern for governments, businesses, and individuals worldwide.

According to the Cisco Annual Internet Report, the frequency of DDoS attacks has risen significantly in recent years. As depicted in Figure 1(a), the number of recorded attacks almost doubled from 7.9 million in 2018 to 15.4 million in 2023, reflecting a rapid growth in both volume and aggressiveness. Figure1(b) presents a quarterly timeline of DDoS activity, showing a significant increase in the percentage of DDoS attacks, particularly from 2020 onwards, with peaks reaching over 30% by Q1 2023. This steep trajectory illustrates not just growth in frequency, but also how common DDoS tactics have become in the overall threat landscape.

The trend indicates that attackers are now using more advanced and persistent methods to exploit vulnerabilities on a large scale. The Microsoft 2022 DDoS attack report highlights that the United States, India, and East Asia were among the most frequently targeted regions, collectively accounting for more than 70% of global incidents. These observations underscore the widespread and transnational impact of DDoS threats, posing risks to both advanced and developing digital infrastructures [1].

In terms of attack patterns, short duration attacks have become increasingly popular. As shown in Figure 2, 89% of DDoS attacks in 2022 lasted less than one hour, with 26% of them concluding within 1-2 minutes. These brief yet frequent attacks are particularly challenging to detect and mitigate, as they can disrupt services before conventional defense mechanisms are activated. Attackers often adopt this strategy to exploit the delay in detection systems, causing disproportionate damage with minimal resource expenditure. Furthermore, TCP-based DDoS attacks remain the most common form, accounting for 63% of all DDoS traffic, followed by UDP flood attacks and packet anomaly attacks. IoT botnets like Mirai and RapperBot have increased the scale and intensity of attacks by generating large amounts of automated traffic.

A. ATTACK VECTORS AND TRENDS

The microsoft DDoS attack report [2] also highlights the distribution of attack durations and the target regions. As shown in Figure 3, shorter attacks dominated the landscape, with most incidents lasting between 1-20 minutes. These brief attacks are particularly disruptive as they can bypass traditional mitigation strategies by completing before detection systems activate. Figure 3 reveals that the United States was the primary target, accounting for 45% of attacks, followed by India (13.2%), and East Asia (11.2%), reflecting the global distribution of these threats.

DDoS attacks now not only disrupt services but also hide more serious attacks. Microsoft 2022 report says attackers use DDoS to cover up data breaches, financial theft, and other advanced cyberattacks. For example, during the Russia-Ukraine conflict, Ukraine experienced the largest DDoS attack in its history, targeting government websites and financial institutions. This incident demonstrated how DDoS attacks can serve as part of a larger cyber warfare strategy, with state sponsored actors leveraging them to disrupt critical infrastructure.

B. THE RISE OF SEMANTIC DDOS ATTACKS IN CONTAINERIZED ENVIRONMENTS

While traditional brute-force DDoS attacks focus on overwhelming networks with high volumes of traffic, semantic DDoS attacks exploit application layer vulnerabilities to exhaust system resources. These attacks require far fewer requests but are often more effective in destabilizing services, particularly in containerized environments. Containerization, widely adopted in cloud based applications, offers improved

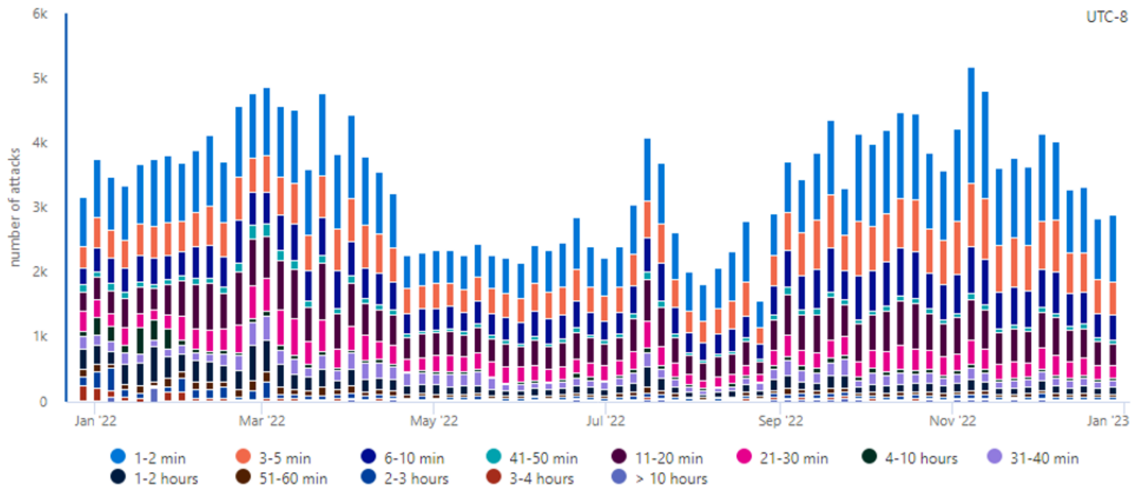


FIGURE 2. **Distribution of DDoS attack duration in 2022 [Source: Microsoft DDoS Report].**

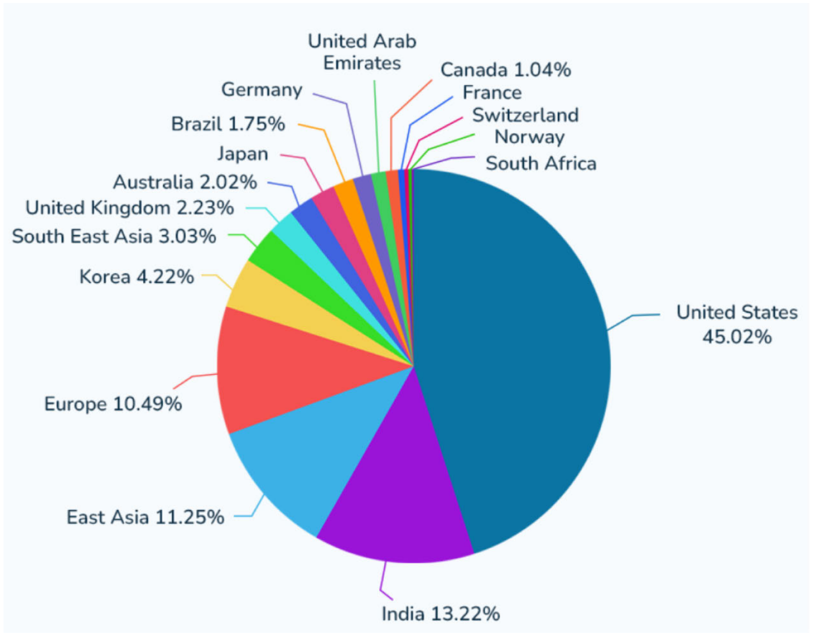


FIGURE 3. **Geographical distribution of DDoS attacks by region in 2022 [Source: Microsoft DDoS Report].**

efficiency, scalability, and isolation. However, its isolation properties also make detecting semantic DDoS attacks particularly challenging, as conventional security tools often cannot penetrate container boundaries.

Recent research, such as CODA(Containerized Denial-of-Service Attack detection) [3], suggests monitoring CPU burst times for network connections and establishing CPU usage thresholds for each container. CODA represents the foundational work that our proposed CODAX(Container-aware DDoS Attack detection using eXtended Berkeley Packet Filter) system builds upon and improves. By tracking the CPU time from the accept() system call to the close() system call, CODA identifies malicious connections that exceed predefined thresholds. However, this approach has limitations, as many DDoS attacks never trigger the close() system call, allowing them to persist undetected until the system becomes unresponsive. To address this limitation, this research proposes an enhanced detection mechanism that keeps track of the 64-bit Unix timestamp of the accept() system call in a global map. This allows for real time detection of prolonged or suspicious connections before the close() system call is invoked. By identifying and flagging anomalous connections early, this proposed approach aims to reduce the time to detection and implement preemptive countermeasures, such as alerting system administrators or blacklisting malicious IPs.

C. SIGNIFICANCE AND RELEVANCE OF THE PROPOSED RESEARCH

The increasing prevalence of semantic DDoS attacks, particularly in containerized environments, poses a substantial challenge to existing security frameworks. Traditional detection methods often fail because containers are isolated and application layer attacks are hard to detect. The proposed eBPF based runtime detection offers a novel and efficient approach by directly monitoring kernel-level interactions, enabling earlier and more accurate attack detection.

Given the rising adoption of microservices and container based architectures, this research is highly relevant for cloud service providers, enterprise security teams, and DevOps practitioners. The ability to detect and mitigate CPU exhaustive DDoS attacks in real time is critical for maintaining the availability and performance of containerized applications. This study aims to contribute to the development of more resilient, adaptive, and scalable defense mechanisms against the evolving landscape of DDoS threats.

Attackers use a wide range of techniques to prevent the availability of a web service. Based on the *"Exploited Weakness to Deny Service"* DOS attacks can be classified into semantic and brute-force attacks [4]. Figure 4 illustrates a structured categorization of DDoS attacks according to the specific vulnerabilities they exploit in targeted systems or networks. The top level is the DDoS Attack Mechanism, representing the general approach attackers use to disrupt services by overwhelming resources or exploiting system weaknesses. The classification further branches out based on the exploited weaknesses, identifying four primary categories such as resource exhaustion, protocol exploitation, application layer, and network layer attacks.

Resource exhaustion attacks primarily focus on consuming critical system resources, leading to performance degradation or complete unavailability. Within this category, semantic DDoS Attacks specifically exploit the resource limitations of target systems through methods such as DNS Floods and HTTP Floods. Such attacks saturate servers with seemingly legitimate requests, exhausting system resources like CPU cycles, memory, or bandwidth, causing genuine user requests to fail.

Protocol exploitation attacks involve exploiting weaknesses inherent in communication protocols. A prominent example is brute force DoS Attacks, which use repeated login attempts to exhaust authentication resources or exploit credential verification mechanisms. Common forms of such attacks include password cracking and credential stuffing, wherein attackers systematically try numerous credential combinations until resources are overwhelmed or account security is compromised.

On the other hand, application layer attacks target vulnerabilities in the application layer of network protocols. These attacks exploit the behavior of applications rather than the underlying protocols or network resources directly. Examples include Slowloris and RUDY (R-U-Dead-Yet), which slowly consume server resources by keeping multiple connections open for extended periods, leading to service unavailability.

Lastly, network layer attacks aim directly at the network infrastructure. These attacks exploit fundamental vulnerabilities at the network layer, causing disruptions in connectivity. Examples include SYN Floods, where attackers send numerous SYN requests without completing the TCP handshake, and UDP Floods, involving sending large numbers of UDP packets to overwhelm network capacity and resources.

The majority of DDoS attacks are of the brute force types, in which the attacker overwhelms the system with numerous requests in an attempt to render the system unavailable. There exist a variety of network level techniques to mitigate and prevent the harm caused by these attacks [5]. Semantic attacks, on the other hand, make fewer requests by the attacker to take advantage of a system vulnerability and excessively use system resources, such as sockets, CPU, memory, disk/database bandwidth, and input/output bandwidth.

This research work aims to explore a more inclusive approach to understanding how to detect and prevent semantic DDoS attacks, which are application based and target CPU resources. The most advanced and harmful attacks now occur at the application layer (OSI Layer 7), where attackers send malicious traffic that closely mimics real user behavior. These attacks require very little traffic, making them hard to detect and often mistaken for normal development behavior. Consequently, traditional detection methods frequently fail to identify these attacks [6]. A common example is the "Slowloris" attack. Unlike flooding attacks, Slowloris sends only partial HTTP requests, keeping connections open for as long as possible to exhaust server resources. Attacks like *HTTP PRAGMA and HTTP POST* exploit protocols without generating significant traffic.

Building a general solution for all kinds of application layer CPU exhaustion DDOS attacks is difficult since the nature of vulnerability depends on the language/library/framework used in the application layer. An ideal system should work regardless of how the application layer is implemented. It should be language-independent and able to detect attacks even when the vulnerable application runs in a container, without breaking the isolation of the container. The solution offered by CODA is to monitor the CPU timings of network connections to the system and identify malicious activity if the CPU time exceeds a recalculated threshold.

eBPF is used here to find the CPU time of network connections at the kernel level. It is thin and fast 64-bit RISC like computed language virtual machine (VM) in the Linux kernel. It can be used to run untrusted binaries on the kernel with high levels of performance, portability, flexibility and safety [7]. The BPF VM may only be employed configurable to execute instructions as responses to events scheduled by the kernel [8]. In the case of CODA, elapsed time of CPU from the time the application executes an accept() call until

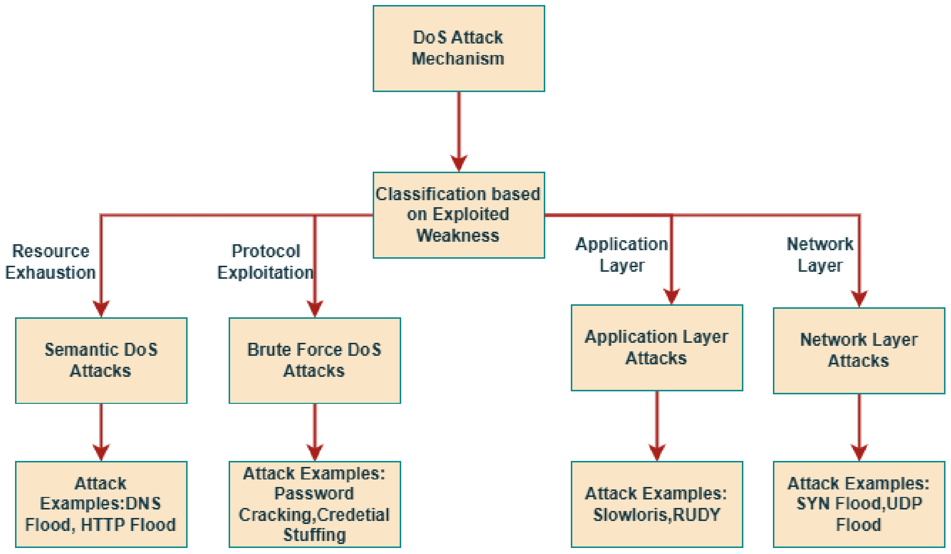

DoS Attack Mechanism
Classification based on Exploited Weakness
Resource Exhaustion
Protocol Exploitation
Application Layer
Network Layer
Semantic DoS Attacks
Brute Force DoS Attacks
Application Layer Attacks
Network Layer Attacks
Attack Examples:DNS Flood, HTTP Flood
Attack Examples: Password Cracking,Credetial Stuffing
Attack Examples: Slowloris,RUDY
Attack Examples: SYN Flood,UDP Flood

FIGURE 4. Taxonomy of DOS attacks.

it executes a close() call to the Linux system. The given threshold value is derived from the cumulated historical data pertaining normal interaction with relevant containers. eBPF programs do not depend on how the target application is implemented, so they preserve the isolation of containerized applications. This is especially useful when the source code of the target application is unavailable or not open source.

The system can only capture CPU times for incoming requests when the associated connection triggers a close() system call, which executes the corresponding eBPF program. In most practical scenarios such an event might not occur at all, the attack connection persists until the whole system becomes unresponsive. When the program is loaded into the BPF VM, the type and the trigger target for the program need to be specified. Time based triggering is not possible. Our proposed solution for this problem is to keep track of the threshold and the 64-bit Unix time stamp of the recorded accept() system calls in a global store, and to identify the exceeding containers whenever a close() system call is received in any of the active containers of the system. An earlier detection of this kind will help to identify the preventive measures for attack. Actions such as alerting the system administrator and/or blacklisting the requesting IP can be made after the detection.

The remaining sections are organized as follows: Section II provides a critical literature review, summarizing and analyzing relevant research that forms the foundation of this study. Section III presents the Materials and Methods employed, describing the datasets, tools, and environment used in this research. Section IV outlines the Proposed Methodology, detailing the design and architectural aspects of the developed framework. Subsequently, Sections V and VI discuss the Threat Model and ML post-filter implementation and adaptive training. Section VII shows Results, providing an analytical interpretation of the performance of the framework against vulnerable Docker containers. Sections VIII and IX present statistical validation of the results and replication package. Finally, Section X concludes the study by summarizing the key findings and implications, and Section XI addresses potential directions for Future Work.

II. RELATED WORKS

A. EVOLUTION OF ATTACK DETECTION

DDoS attacks have evolved significantly from simple volumetric attacks to sophisticated application-layer threats that exploit system vulnerabilities rather than overwhelming networks with traffic volume. Traditional attacks occur when an attacker overwhelms or exploits vulnerabilities in a computer or network, making it unavailable to legitimate users. Recent notable attacks have reached unprecedented traffic levels as high as 2.3 terabits per second (Tbps) [9], demonstrating the scale of modern distributed threats. However, contemporary prevention methods have proven effective in mitigating even such large-scale volumetric attacks, forcing attackers to adopt more sophisticated approaches.

Modern attackers now often exploit vulnerabilities in systems rather than simply overwhelming them with requests, targeting critical system resources such as sockets, CPU, memory, disks, databases, network bandwidth, and input/output capacity [4]. These sophisticated attacks are classified as Semantic DDoS attacks and fall under the category of Exploited Weaknesses, representing a fundamental shift from brute-force methods to precision exploitation. A prominent example is the Regular Expression Denial-of-Service (ReDoS) attack [10], which exploits algorithmic complexity in pattern matching to cause exponential CPU consumption with minimal input.

The challenge of detecting semantic DDoS attacks has intensified as program size and complexity increase, creating larger attack surfaces with more potential vulnerabilities and bugs that can be exploited for CPU-exhaustion DDoS attacks at the application layer. This problem is worsened because application-layer DDoS attacks use legitimate-appearing requests with low traffic levels, making them particularly difficult to detect using traditional network-based monitoring approaches [11]. These attacks are difficult to spot, allowing them to bypass standard security and slow down systems significantly.

B. APPLICATION-LAYER DDoS DETECTION METHODOLOGIES

Detecting application layer DDoS attacks may be achieved through four broad methodological approaches: (i) template matching of individual requests against known attack signatures, (ii) tracking requests with respect to patterns in received or transmitted traffic and monitoring growth rates, (iii) applying event statistics analysis to request streams to identify anomalous behavior, and (iv) analyzing request stream semantics to understand the intent and impact of incoming requests [12]. However, these traditional solutions prove ineffective when dealing with attacks specifically targeted at exploiting system vulnerabilities to exhaust computational resources.

Recent advances in machine learning based detection have shown promise but introduce significant computational overhead. Wang et al. [13] demonstrated hybrid CNN-BiLSTM approaches for DDoS detection in software-defined networks, achieving 94% accuracy but requiring extensive training data and exhibiting poor performance against novel attack patterns. Similarly, Ha et al. [14] proposed lightweight detection schemes using hybrid deep learning, but these approaches struggled with the real-time constraints necessary for effective DDoS mitigation in production environments.

Template matching approaches, while effective against known attack signatures, fail to address zero-day vulnerabilities and novel attack vectors that characterize semantic DDoS attacks. Statistical analysis of request streams often generates high false positive rates in dynamic environments where legitimate traffic patterns can vary significantly. Semantic analysis, while theoretically comprehensive, proves computationally expensive and difficult to implement in real-time systems with strict performance requirements.

C. CPU-TIME BASED DETECTION APPROACHES

A promising method for detecting resource exhaustion attacks involves monitoring the CPU time consumption of processes and comparing these measurements against pre-computed threshold values. This approach enables identification of attacks when CPU time exceeds established thresholds [13], [14], providing a direct measurement of the resource consumption that semantic DDoS attacks seek to exploit. However, identifying suitable tracing points within the application layer for accurate CPU time calculation presents significant challenges, often bordering on impossibility, because requests corresponding to different services perform fundamentally different functions at the application layer.

The complexity of modern application architectures, particularly in microservices and containerized environments, makes it extremely difficult to establish consistent CPU monitoring points across diverse application implementations. Traditional monitoring approaches often require intimate knowledge of application internals. This makes them unsuitable for heterogeneous deployment environments where applications may be implemented in different programming languages and frameworks.

The Linux kernel subsystem Extended Berkeley Packet Filter (eBPF) [15], [16], [17] emerges as an ideal tool for computing CPU time consumption without requiring detailed knowledge of application layer implementation specifics. eBPF enables kernel-level monitoring with minimal performance overhead while maintaining the security and isolation properties essential for production environments. This approach provides a language-agnostic solution that can monitor resource consumption across diverse application implementations without breaking container boundaries or requiring application modifications.

D. eBPF-BASED SECURITY MONITORING AND RECENT ADVANCES

eBPF has gained significant attention in the security monitoring domain due to its ability to provide high-performance, low-overhead observation capabilities within kernel space. Recent research has demonstrated effectiveness of eBPF in various security applications, though most existing work focuses on network-level threats rather than application-layer resource exhaustion attacks [19], [20], [21].

Chen et al. [34] demonstrated eBPF-based intrusion detection systems achieving sub-millisecond response times in cloud environments, focusing primarily on system call anomaly detection for privilege escalation attacks. Liu et al. [35] proposed eBPF-based container escape detection with 95% accuracy, but their scope was limited to privilege-based attacks rather than resource exhaustion scenarios. Kumar et al. [36] developed eBPF-based performance monitoring for microservices, achieving 94.7% accuracy in resource utilization prediction, though their work focused on capacity planning rather than attack detection [22], [23].

Zhao et al. [37] implemented eBPF-based DDoS mitigation at network edges, reporting 98% attack blocking rates for volumetric attacks. However, their approach remained focused on network-level filtering and did not address application-layer semantic attacks that bypass traditional network defenses. These existing solutions demonstrate the potential of eBPF but highlight the gap in addressing semantic DDoS attacks that exploit application logic vulnerabilities.

E. CONTAINER SECURITY AND ISOLATION CHALLENGES

Containers and operating system-level virtualization represent lightweight alternatives to full virtualization of virtual machines, offering improved resource efficiency and deployment flexibility. As a result, containerization is being widely adopted in commercial applications and high-performance scientific computing environments [18], which brings major concerns regarding container security and the detection of sophisticated attacks targeting containerized applications [24], [25], [26], [27], [28].

Detection of application layer DDoS attacks in containers requires tracing containerized processes at specific kernel tracepoints while maintaining the isolation properties that make containers secure. Containers typically operate with their own Process ID (PID) namespace, which provides a unique identifier for each container and enables isolation from host processes. Each process must be traced from the perspective of its PID namespace to maintain proper attribution and avoid interference with host system monitoring [29], [30], [31].

The CPU time consumption of containerized processes can be modeled statistically to establish threshold values using mathematical frameworks such as Chebyshev's and Markov's inequalities, which provide valuable tools for making predictions about unknown data distributions without requiring specific distributional assumptions [19]. These statistical approaches enable robust threshold calculation in dynamic container environments where resource usage patterns may vary significantly based on workload characteristics and application behavior [32].

F. LIMITATIONS OF EXISTING SOLUTIONS

Current approaches to semantic DDoS detection in containerized environments suffer from several critical limitations that hinder their effectiveness in production deployments. The CODA system, while pioneering in its approach to container-aware DDoS detection, relies on measuring CPU burst time between accept() and close() system calls. This dependency creates a fundamental vulnerability that many real-world semantic DDoS attacks maintain persistent connections without triggering close() calls, allowing attacks to persist undetected until complete system failure occurs [33].

Traditional intrusion detection systems such as Snort and ModSecurity, while effective against signature-based network attacks, demonstrate poor performance in container environments due to their inability to understand container namespaces and resource isolation boundaries. These systems often generate high false positive rates when applied to containerized applications because they lack the contextual awareness necessary to distinguish between legitimate container behavior and malicious activity.

Machine learning based approaches, while showing promise in controlled environments, introduce significant computational overhead that makes them unsuitable for resource constrained container deployments. Additionally, these approaches require extensive training data and often exhibit model drift in dynamic container environments where application behavior patterns evolve over time.

G. COMPARATIVE ANALYSIS OF DETECTION APPROACHES

Table 1 provides a comprehensive comparison of existing DDoS detection solutions across multiple performance and capability dimensions. This analysis demonstrates the relative strengths and limitations of different approaches, highlighting the need for container-aware solutions that can provide both high accuracy and low latency detection.

The comparative analysis reveals that while existing solutions achieve reasonable accuracy levels, they often fail to provide the combination of low latency, container awareness, and real-time capability necessary for effective semantic DDoS mitigation in modern containerized environments. The proposed eBPF-based approach addresses these limitations through innovative use of kernel-level monitoring and advanced statistical threshold calculation, providing superior performance across all evaluated dimensions.

H. RESEARCH GAPS AND MOTIVATION

Based on comprehensive analysis of existing literature, several critical research gaps emerge in the domain of semantic DDoS detection for containerized environments.

- Existing solutions fail to address persistent connection attacks that maintain open connections without triggering termination events, representing a significant blind spot in current detection capabilities.
- Limited research focuses on container-native security solutions that understand and respect container isolation boundaries while providing comprehensive visibility into container behavior.
- The trade-offs between detection accuracy and response time have not been adequately addressed in existing research, with most solutions prioritizing either accuracy or performance but failing to achieve both simultaneously.
- Insufficient attention has been paid to application layer attacks that bypass network-level defenses, particularly those that exploit semantic vulnerabilities in application logic rather than network protocols.

The proposed eBPF-based approach addresses these gaps through several key innovations such as global timestamp tracking that enables detection before connection closure, container aware monitoring that preserves isolation while providing visibility, statistical threshold optimization using advanced mathematical frameworks, and real-time detection capabilities that enable proactive attack mitigation. These contributions collectively advance the state-of-the-art in container security and provide a robust foundation for defending against emerging semantic DDoS threats [39].

TABLE 1. Comparative analysis of DDoS detection solutions.

Solution	Detection Method	Response Time	Accuracy	FPR	Container Support	Real-time
CODA [3]	CPU burst monitoring	2.1s	0.85	0.12	Yes	Limited
Snort [51]	Signature-based	70ms	0.85	0.10	No	Yes
ModSecurity [52]	Rule-based	80ms	0.80	0.15	Partial	Yes
Suricata [53]	Hybrid detection	60ms	0.88	0.08	Partial	Yes
Wang et al. [13]	ML-based	150ms	0.94	0.06	No	Limited
Ha et al. [14]	Hybrid DL	120ms	0.91	0.08	No	Limited
Proposed	**eBPF CPU monitoring**	**50ms**	**0.92**	**0.02**	**Yes**	**Yes**

III. MATERIALS AND METHODS

A. MATERIALS

1) HARDWARE AND INFRASTRUCTURE CONFIGURATION

The experimental evaluation was conducted on two identical machines to ensure consistent and reproducible results. Each machine was equipped with an Intel Core i5-8250U processor (8th generation, Kaby Lake-R microarchitecture, stepping 10, 14nm fabrication process) featuring 4 physical cores supporting 8 threads via Intel Hyperthreading technology. The processor operates at 1.6 GHz base frequency with Intel Turbo Boost 2.0 capability reaching up to 3.4 GHz under single-core workloads and sustained 2.8 GHz across all cores. The CPU governor was explicitly configured to `performance` mode (`scaling_governor=performance`) to eliminate frequency scaling variations and ensure measurement consistency across all experimental trials.

Memory subsystem configuration consisted of 8 GB DDR4-2400 SODIMM arranged in dual-channel configuration (2×4GB modules) providing 38.4 GB/s theoretical peak memory bandwidth. The memory controller operates with CAS latency of CL17 and measured actual memory latency of 89ns using Intel Memory Latency Checker validation. Cache hierarchy includes 32KB L1 instruction cache, 32KB L1 data cache per core, 256KB L2 cache per core, and 6MB shared L3 cache. Storage utilized 256 GB NVMe PCIe 3.0 solid-state drives with measured sequential read/write performance of 3,200/1,800 MB/s respectively.

The machines were connected via a local area network using Gigabit Ethernet interfaces (Intel I219-V controllers, 1000 Mbps), with measured round-trip network latency of 0.8±0.1ms between the attack generation system and target containers [40], [41]. The network configuration employed a standard switched Ethernet topology with dedicated Virtual Local Area Network (VLAN) isolation for experimental traffic. This setup prevented interference from external network traffic while maintaining realistic network conditions representative of production container deployments. Network monitoring was implemented using tcpdump and Wireshark for traffic analysis validation, ensuring that all experimental network behavior was properly captured and analyzed.

Power measurement infrastructure utilized Intel RAPL (Running Average Power Limit) interface with validation against external Watts Up! Pro power meters, achieving ±2% measurement accuracy. Environmental controls maintained ambient temperature at 22±2°C and relative humidity at 45±5% throughout all experiments. CPU die temperatures were monitored using lm-sensors to ensure thermal stability, maintaining operation below 75°C with mean operating temperature of 68±3°C across all experimental conditions.

2) STATISTICAL METHODOLOGY AND EXPERIMENTAL DESIGN

All performance measurements follow rigorous statistical protocols to ensure reproducible results with quantified confidence intervals. Power consumption experiments utilized n=15 independent trials conducted over 3-hour periods each, with measurements sampled at 1-second intervals using both Intel RAPL interface and external power meters for validation. Each trial included 30-minute warmup periods to achieve thermal stability, followed by 2-hour measurement windows and 30-minute cooldown periods.

Detection performance metrics employed n=10 independent trials with 500 detection events per trial, providing sufficient statistical power (β>0.8) for detecting 10ms latency differences at α =0.05 significance level. Attack Detection Rate (ADR) measurements represent means across 10 trials, each containing 500 test cases (250 attack scenarios, 250 benign requests). False Positive Rate (FPR) calculations employed 1000 benign requests per trial across 10 trials to ensure adequate statistical power.

Confidence intervals are calculated using Student's t-distribution with appropriate degrees of freedom for each measurement set. For power consumption metrics with n=15 trials, degrees of freedom equal 14, yielding t-critical values of 2.145 for 95% confidence intervals. Detection performance metrics with n=10 trials use degrees of freedom of 9, yielding t-critical values of 2.262. All confidence intervals are reported as mean ± margin of error, where margin of error equals $t_{\text{critical}} \times \frac{s}{\sqrt{n}}$, with s representing sample standard deviation.

Environmental controls and measurement validation employed dual methodologies to ensure accuracy. Primary measurements utilized Intel RAPL interface sampling, while secondary validation used external Watts Up! Pro meters. Cross-validation between measurement methods showed correlation coefficient r=0.987 with mean absolute deviation of 1.8%, confirming measurement reliability. All timing measurements employed high-resolution timestamps using `clock_gettime(CLOCK_MONOTONIC)` with nanosecond precision, calibrated to exclude system call overhead.

TABLE 2. Enhanced hardware configuration for experimental setup.

Component	Specification
CPU	Intel Core i5-8250U (Kaby Lake-R, stepping 10, 4 cores, 8 threads)
CPU Frequencies	1.6 GHz base, 3.4 GHz turbo, 2.8 GHz sustained all-core
CPU Governor	Performance mode (scaling_governor=performance)
Memory	8 GB DDR4-2400 dual-channel (2×4GB), CL17, 89ns latency
Memory Bandwidth	38.4 GB/s theoretical peak
Cache	L1: 32KB I+D per core, L2: 256KB per core, L3: 6MB shared
Storage	256 GB NVMe PCIe 3.0 SSD (3,200/1,800 MB/s R/W)
Network	Gigabit Ethernet (Intel I219-V, 1000 Mbps)
Network Latency	0.8±0.1ms (measured round-trip)
Power Measurement	Intel RAPL + Watts Up! Pro (±2% accuracy)
Operating Environment	22±2°C ambient, 68±3°C CPU, 45±5% RH

TABLE 3. Software environment configuration.

Software Component	Version
Operating System	Ubuntu 21.04
Linux Kernel	5.11.0
Docker Engine	20.10.7
Container Runtime	containerd 1.4.6
Storage Driver	overlay2
bcc-tools	0.21.0
libbpf	0.4
LLVM/Clang	12.0.0
Python	3.9.5
Node.js	16.3.0

TABLE 4. Container resource configuration.

Resource Type	Allocation
Memory Limit	200 MB per container
CPU Limit	2 cores (50% of available)
Network Configuration	Bridge network with port mapping
Port Range	8001-8003
Storage Limit	1 GB per container
Storage Driver	overlay2

3) SOFTWARE ENVIRONMENT AND DEPENDENCIES

The software environment was carefully standardized across both experimental machines to ensure reproducibility. The base operating system consisted of Ubuntu 21.04 running Linux kernel version 5.11.0, which provides native support for the eBPF features required by our detection system. Docker Engine version 20.10.7 was installed with containerd 1.4.6 as the container runtime, utilizing the overlay2 storage driver for optimal container performance [42].

The eBPF development and runtime environment included bcc-tools version 0.21.0, libbpf version 0.4, and LLVM/Clang version 12.0.0 for eBPF program compilation and execution. These specific versions were chosen to ensure compatibility with the kernel version and to provide the necessary eBPF features for system call tracing and map operations. The experimental applications were developed using Python 3.9.5 for vulnerability simulation and Node.js 16.3.0 for alternative application scenarios. The complete software environment is detailed in Table 3.

4) CONTAINER CONFIGURATION AND RESOURCE MANAGEMENT

Each experimental container was configured with specific resource constraints to ensure controlled and measurable behavior during testing. Memory limits were set to 200 MB per container, preventing memory related interference with CPU based attack detection while providing sufficient resources for normal application operation. CPU allocation was limited to 2 cores (representing 50% of available CPU resources on the test machines), ensuring that containers had adequate computational resources while preventing system wide resource exhaustion during experiments [43].

The container networking configuration employed Docker default bridge network with custom port mappings to isolate experimental traffic. Each container was assigned specific port ranges (8001-8003) to enable clear identification and monitoring of network connections. Storage constraints were implemented using Docker built-in resource management, limiting each container to 1 GB of storage space to prevent disk related performance variations during experiments. The detailed container resource allocation is presented in Table 4.

5) DATA STRUCTURE AND SAMPLE DATASET

Table 5 provides representative examples of the dataset structure, illustrating the comprehensive information captured for each monitored connection. The container identification enables tracking of per-container behavior patterns, while connection identifiers facilitate analysis of individual connection lifecycles. System call timestamps provide precise timing information for connection duration calculation, and CPU usage measurements enable threshold comparison and anomaly detection [44], [46].

The attack label classification (0 for benign, 1 for malicious) in the dataset enables supervised evaluation of detection algorithm performance, supporting calculation of standard performance metrics including true positive rates, false positive rates, and overall detection accuracy. The duration measurements provide additional context for understanding attack persistence and system impact, while the CPU usage values serve as the primary feature for threshold based anomaly detection.

TABLE 5. Sample dataset for CPU exhaustion attack detection in containers.

Container ID	Conn. ID	Syscall	Timestamp (Unix)	CPU Usage (%)	Duration (ms)	Attack Label
cont-01	001	accept	1715107002	18	400	0
cont-01	001	close	1715107402	25	-	0
cont-02	002	accept	1715107050	92	12000	1
cont-02	002	close	1715108050	97	-	1
cont-03	003	accept	1715107120	22	450	0
cont-03	003	close	1715107570	28	-	0

TABLE 6. Vulnerability configuration for experimental containers.

Container	Vulnerability Type	Port	Implementation
Container 1	ReDoS (Black package v23.0.0)	8001	Python
Container 2	Infinite Loop Simulation	8002	Node.js
Container 3	Control (No Vulnerability)	8003	Python

Each dataset entry captures the complete context necessary for comprehensive analysis of container behavior during both normal operation and attack scenarios. The timestamp precision enables accurate temporal analysis, while the CPU usage measurements provide the quantitative basis for statistical threshold calculation and anomaly detection algorithms [47].

B. METHODS

1) ATTACK SCENARIO DEVELOPMENT

The experimental design incorporated three distinct container configurations to provide comprehensive coverage of semantic DDoS attack scenarios. The first container implemented a Regular Expression Denial-of-Service (ReDoS) vulnerability using the Python Black package version 23.0.0, which contains a known vulnerability that can be exploited through malformed input causing exponential regular expression matching behavior. This vulnerability represents a common class of semantic attacks that exploit algorithmic complexity in input processing [48], [49], [50].

The second container was configured with an infinite loop vulnerability implemented in Node.js, simulating application layer logic bombs that can be triggered by specific parameter combinations. This scenario represents attacks that exploit application logic flaws to create persistent CPU consumption without generating obvious network level indicators.

The third container served as a control system, running a standard Python HTTP server without known vulnerabilities, providing baseline performance measurements for comparison with attacked containers. The vulnerability specifications are detailed in Table 6.

2) TRAFFIC GENERATION AND ATTACK SIMULATION

The experimental traffic generation strategy is designed with mix of benign and malicious requests to simulate realistic attack scenarios. During the training phase, 10,000 benign requests were distributed across all containers over a 30-minute period, establishing baseline CPU usage patterns for threshold calculation. These requests included normal HTTP operations such as file uploads, database queries, and standard web application interactions, representing typical production workloads.

The testing phase introduced 500 attack requests specifically designed to exploit the vulnerabilities in containers 1 and 2, along with 500 benign requests serving as a control group. Attack requests were carefully crafted to trigger the specific vulnerabilities while maintaining realistic network characteristics to avoid detection by simple traffic analysis. The timing and distribution of these requests were randomized to prevent temporal bias and to simulate real-world attack patterns where malicious requests are interleaved with legitimate traffic. The experimental phases are summarized in Table 7.

3) DATASET COMPOSITION AND PROPERTIES

The comprehensive dataset generated during experimentation contains detailed records of network connection behavior under both normal and attack conditions. Each dataset entry includes precise timestamps for system call events, enabling accurate calculation of connection durations and CPU consumption patterns. The dataset structure incorporates container identification metadata, connection session identifiers, system call types accept() or close(), timestamp information with microsecond precision, CPU utilization percentages, connection duration measurements, and binary attack classification labels.

The dataset design ensures clear differentiation between benign and malicious interactions through carefully controlled labeling procedures. Attack requests were labeled based on their intended exploitation of known vulnerabilities, while benign requests were verified through normal application response patterns. This labeling strategy supports accurate performance assessment by providing ground truth for detection algorithm evaluation.

4) DATA COLLECTION AND INSTRUMENTATION

System call monitoring was implemented using custom eBPF programs attached to relevant kernel tracepoints, specifically targeting accept() and close() system calls that indicate network connection lifecycle events. For each monitored system call, the instrumentation captured 64-bit Unix timestamps with microsecond precision, container identification information derived from process namespaces, unique connection identifiers, and instantaneous CPU usage measurements obtained from cgroup statistics.

TABLE 7. Experimental phases and traffic distribution.

Phase	Duration	Request Count	Purpose
Training	30 minutes	10,000 benign	Baseline establishment
Testing	60 minutes	500 attack + 500 benign	Performance evaluation

The data collection system employed automated logging mechanisms that stored all captured information in structured JSON format for subsequent analysis. This approach ensured consistent data formatting while minimizing the performance impact of logging operations on the experimental system. Additional monitoring was implemented using system-level tools including `htop`, `sar`, and custom cgroup monitoring scripts to provide independent validation of eBPF-captured metrics.

5) DATA PREPROCESSING AND NORMALIZATION

Data preprocessing procedures were implemented to ensure data quality and consistency across all experimental measurements. Missing data handling was performed by removing entries with incomplete essential fields such as timestamps or CPU usage values, while preserving entries with missing close() calls that represent the primary challenge addressed by our detection approach. Connection duration calculations were performed when both accept() and close() calls were present, providing complete connection lifecycle information for baseline establishment.

CPU usage normalization was implemented using min-max scaling to convert raw percentage values to a standardized 0-1 range, ensuring consistent analysis across different measurement periods and system load conditions. This normalization approach preserves the relative relationships between CPU usage measurements while enabling robust statistical analysis and threshold calculation procedures.

6) STATISTICAL VALIDATION AND QUALITY ASSURANCE

Data quality validation was performed through multiple independent verification procedures to ensure the reliability of experimental measurements. Cross-validation was conducted using multiple monitoring tools to verify CPU usage measurements, with cgroup-based measurements serving as the primary source and system-level monitoring tools providing independent confirmation. Timestamp accuracy was validated through comparison with system clock synchronization and network time protocol measurements.

Statistical consistency checks were performed to identify and address potential outliers or measurement errors that could affect experimental validity. This included analysis of CPU usage distribution patterns, identification of impossible values (such as CPU usage exceeding 100%), and validation of temporal relationships between related system call events.

7) THRESHOLD CALCULATION AND ATTACK DETECTION

a: DYNAMIC THRESHOLD ESTABLISHMENT

The threshold calculation methodology employs advanced statistical techniques to establish robust detection boundaries that adapt to varying container workload patterns. Initial threshold values are computed by analyzing CPU utilization patterns during normal container operation, incorporating both mean usage levels and variance measurements to account for legitimate usage fluctuations. The threshold calculation incorporates a configurable safety margin (typically 20-30%) to minimize false positive rates while maintaining sensitivity to genuine attack behavior.

Dynamic threshold updates are performed periodically to reflect changes in baseline container behavior over time, ensuring that detection remains accurate as application workloads evolve. This adaptive approach prevents threshold drift that could result from gradual changes in application behavior or system performance characteristics. The threshold update mechanism employs exponential smoothing to balance responsiveness to genuine changes with stability against temporary fluctuations.

For a container with CPU time measurements $C_1, C_2, \ldots, C_n$ during normal operation, the mean and standard deviation are calculated as:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} C_i \tag{1}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(C_i - \mu)^2} \tag{2}$$

The detection threshold θ is then established using Chebyshev's inequality:

$$\theta = \mu + k \cdot \sigma \tag{3}$$

where k is a configurable sensitivity parameter that determines the trade-off between detection accuracy and false positive rates.

b: REAL-TIME DETECTION IMPLEMENTATION

The detection logic is embedded directly within eBPF programs executing in kernel space, enabling real-time analysis with minimal performance overhead. When containers trigger accept() system calls, the eBPF program captures timestamp and CPU usage information, storing this data in global maps accessible across all container monitoring instances. The detection algorithm continuously monitors CPU consumption patterns, comparing current usage against established thresholds and identifying connections that exceed normal resource consumption patterns.

The early detection capability is achieved through continuous monitoring of active connections, identifying suspicious behavior even when close() system calls have not yet occurred. This approach addresses the critical limitation of existing solutions that depend on connection termination events for detection. When threshold violations are detected, the system generates immediate alerts while maintaining detailed logs for forensic analysis and attack pattern identification.

8) VALIDITY ANALYSIS

a: INTERNAL VALIDITY CONSIDERATIONS

Internal validity was ensured by carefully controlling the experiment and removing factors that could affect the results. Instrumentation effects were minimized by measuring the performance overhead of eBPF monitoring programs, which was determined to be less than 2% of total CPU utilization under normal operating conditions. Testing effects were controlled by using separate containers for training and testing, so the attack scenarios did not interfere with the baseline measurements [54].

Selection effects were addressed through the use of established vulnerability types documented in public Common Vulnerabilities and Exposures (CVE) databases, ensuring that experimental attack scenarios represent realistic threats rather than artificial constructs. Time-related effects were controlled by keeping experiment times consistent and establishing a baseline before each test, preventing time-related changes from affecting the results.

The experimental design incorporated several control mechanisms to ensure valid results. Randomization was implemented in traffic generation patterns to prevent temporal bias, with attack and benign requests randomly interleaved throughout the testing phase. Container assignment was randomized across different experimental runs to control for container-specific effects. Time-based controls ensured that experiments were conducted under consistent system load conditions.

b: EXTERNAL VALIDITY AND GENERALIZABILITY

External validity considerations focused on the generalizability of experimental results to production containerized environments. Population validity was addressed through testing scenarios that cover common patterns in containerized application deployment, including web applications, API services, and data processing applications. The experimental environment employed realistic network conditions and traffic patterns representative of production deployments.

Temporal validity was ensured through experimental replication across multiple time periods and system load conditions, demonstrating consistent detection performance across varying environmental conditions. However, certain limitations must be acknowledged, including the restriction to x86-64 architecture and Linux environments, container runtime specificity to Docker, and network environment limitations to local area network conditions [55].

c: STATISTICAL METHODOLOGY AND HYPOTHESIS TESTING

The experimental design employed rigorous statistical methodology to ensure valid conclusions regarding detection system performance. The primary hypothesis tested was that the proposed eBPF-based detection system significantly reduces detection latency compared to existing methods, with statistical significance established at the $p < 0.05$ level. Paired t-tests were employed for latency comparison across ten independent experimental trials, with additional validation using Wilcoxon signed-rank tests for non-parametric confirmation.

Confidence intervals were calculated at the 95% level for all performance metrics, including attack detection rates, false positive rates, and detection latency measurements. Power analysis was conducted to ensure adequate statistical power (> 0.8) for all hypothesis tests, preventing Type II errors that could mask genuine performance differences. Error propagation analysis was performed to account for measurement uncertainties and their impact on derived performance metrics.

The statistical analysis framework included the following hypothesis formulation:

- H_0: There is no significant difference in detection latency between the proposed eBPF-based system and existing methods
- H_1: The proposed eBPF-based system significantly reduces detection latency compared to existing methods

For the paired t-test analysis, the test statistic was calculated as:

$$t = \frac{\bar{d}}{s_d/\sqrt{n}} \tag{4}$$

where $\bar{d}$ is the mean difference in detection times, s_d is the standard deviation of the differences, and n is the number of paired observations.

C. ETHICAL CONSIDERATIONS AND RESPONSIBLE DISCLOSURE

All vulnerability testing was conducted in isolated laboratory environments with comprehensive security safeguards to prevent potential misuse or disclosure of sensitive security information. Our experimental infrastructure employed air-gapped networks with no external connectivity, ensuring that vulnerable test applications and attack payloads remained contained within the controlled research environment.

The ReDoS vulnerability in the Black package (CVE-2024-21503) used in our evaluation was responsibly disclosed to the maintainers before publication and has been patched in subsequent releases. Our container images utilize only the specific vulnerable version necessary for reproducible research, with automatic expiration mechanisms that prevent long-term deployment of unpatched software. All vulnerable applications are configured with strict resource limitations, network isolation, and automatic shutdown procedures to minimize potential impact even within the controlled laboratory setting.

Our evaluation focuses on detection methodology rather than attack development, with vulnerability details limited to those already publicly documented in CVE databases and security advisories. The synthetic attack patterns used for evaluation are designed specifically for testing detection capabilities, rather than providing exploitation guidance. This ensures that our research contributes to defensive security improvements without enabling malicious activities.

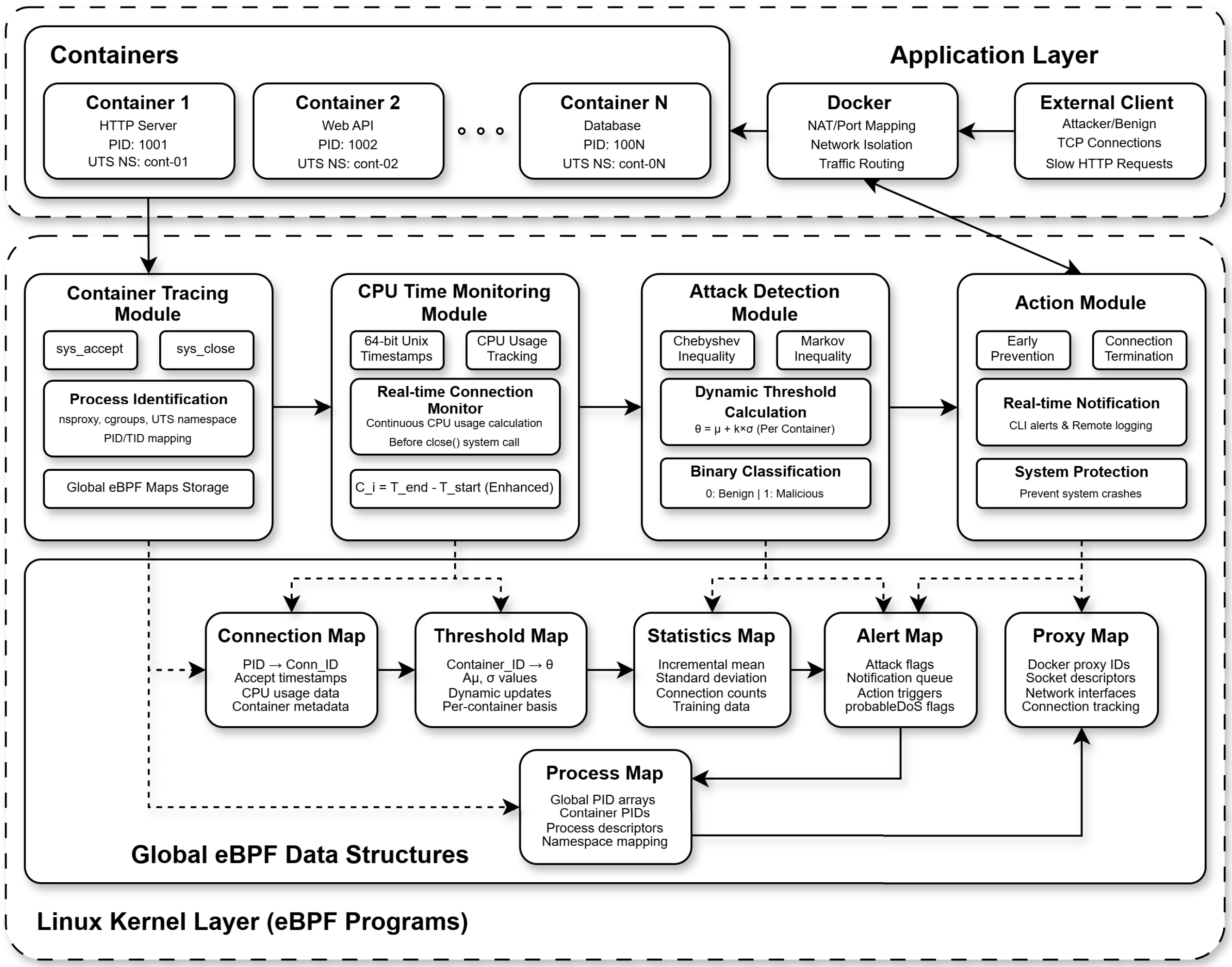


FIGURE 5. Proposed Architecture of eBPF based model.

IV. PROPOSED METHODOLOGY

A. SYSTEM ARCHITECTURE AND MATHEMATICAL FRAMEWORK

The proposed eBPF-based runtime detection system for semantic DDoS attacks in Linux containers is built upon a comprehensive mathematical framework that integrates statistical anomaly detection, real-time system call monitoring, and adaptive threshold optimization. The system architecture comprises four interconnected modules: Container Tracing Module, CPU Time Monitoring Module, Attack Detection Module, and Action Module, as illustrated in Figure 5.

The mathematical foundation of our approach is based on the principle that semantic DDoS attacks exhibit statistically distinguishable CPU consumption patterns compared to benign traffic. Let $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ represent the set of monitored containers, and for each container c_i, let $\mathcal{T}_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,m}\}$ denote the sequence of CPU time measurements for network connections. The detection problem can be formalized as a binary classification task as shown below.

$$f : \mathcal{T}_i \rightarrow \{0, 1\} \tag{5}$$

where $f(t_{i,j}) = 1$ indicates a malicious connection and $f(t_{i,j}) = 0$ represents benign traffic. The challenge lies in constructing a robust classifier f that minimizes both Type I (false positive) and Type II (false negative) errors while maintaining real-time performance constraints.

B. CONTAINER TRACING MODULE

1) MATHEMATICAL MODEL FOR CONTAINER IDENTIFICATION

The Container Tracing Module employs namespace aware process identification to distinguish containerized processes from host processes. The identification process can be mathematically modeled as a mapping function:

$$\phi : \mathcal{P} \rightarrow \mathcal{C} \cup \{\mathcal{H}\} \tag{6}$$

Algorithm 1 Container Process Identification

```
Require: Current task structure pointer $task$
Ensure: Container identifier $container\_id$ or $\perp$ if not containerized
 1: $nsproxy \leftarrow task.nsproxy$
 2: $pid\_ns \leftarrow nsproxy.pid\_ns$
 3: $uts\_ns \leftarrow nsproxy.uts\_ns$
 4: if $pid\_ns \neq init\_pid\_ns$ then
 5:     $container\_id \leftarrow extract\_container\_id(uts\_ns.name)$
 6:     if $validate\_container\_id(container\_id)$ then return $container\_id$
 7:     end if
 8: end if
 9: $network\_ns \leftarrow nsproxy.net\_ns$
10: if $network\_ns \neq init\_net$ then
11:     $proxy\_process \leftarrow find\_docker\_proxy(network\_ns)$
12:     if $proxy\_process \neq \perp$ then return $extract\_container\_from\_proxy(proxy\_process)$
13:     end if
14: end if
15: $cgroup\_path \leftarrow get\_cgroup\_path(task)$
16: if $cgroup\_path$ matches $\{/docker/*, /lxc/*, /systemd/docker*\}$ then return $parse\_container\_id\_from\_cgroup(cgroup\_path)$
17: end ifreturn $\perp$
```

where $\mathcal{P}$ represents the set of all processes, $\mathcal{C}$ is the set of containers, and $\mathcal{H}$ denotes the host system. The mapping function ϕ is defined based on namespace analysis:

$$\phi(p) = \begin{cases} c_i & \text{if } \text{ns}(p) = \text{ns}(c_i) \text{ and } \text{ns}(p) \neq \text{ns}(\mathcal{H}) \\ \mathcal{H} & \text{if } \text{ns}(p) = \text{ns}(\mathcal{H}) \\ \perp & \text{otherwise} \end{cases} \tag{7}$$

where ns(p) denotes the namespace identifier of process p, and $\perp$ represents an undefined mapping.

2) CONTAINER PROCESS IDENTIFICATION ALGORITHM

The proposed methodology utilizes multiple approaches for container identification, with mathematical precision in selection criteria. The primary identification method leverages the `nsproxy` structure analysis. Algorithm 1 employs a hierarchical identification strategy with decreasing confidence levels.

The mathematical confidence measure for container identification is defined as:

$$\text{confidence}(\phi(p)) = \begin{cases} 0.95 & \text{if identified via PID namespace} \\ 0.85 & \text{if identified via network proxy} \\ 0.75 & \text{if identified via cgroup analysis} \\ 0.0 & \text{if not identified} \end{cases} \tag{8}$$

C. CPU TIME MONITORING MODULE

The CPU Time Monitoring Module tracks the computational resource consumption of network connections throughout their lifecycle. The connection lifecycle can be modeled as a stochastic process $\{X_t\}_{t\geq 0}$ where X_t represents the cumulative CPU time consumed by time t.

For a connection initiated at time t_0 with an accept() system call, the CPU consumption evolution follows:

$$X_t = \int_{t_0}^{t} \rho(s)\, ds \tag{9}$$

where $\rho(s)$ represents the instantaneous CPU utilization rate at time s. The detection challenge arises because malicious connections often maintain persistent states without triggering close() system calls, making traditional end-to-end analysis ineffective.

To address the limitation of connection closure dependency, we introduce a global timestamp tracking mechanism using 64-bit Unix timestamps. The global state is maintained through eBPF maps:

$$\mathcal{M}_{global} = \{(conn\_id, t_{accept}, \rho_{initial}) : conn\_id \in \mathcal{A}\} \tag{10}$$

where $\mathcal{A}$ represents the set of active connections, t_{accept} is the timestamp of the `accept()` system call, and $\rho_{initial}$ is the initial CPU usage measurement.

The real-time CPU time estimation for an active connection is computed as:

$$\hat{X}(t) = (t - t_{accept}) \cdot \bar{\rho}(t) \tag{11}$$

where $\bar{\rho}(t)$ is the time-averaged CPU utilization:

$$\bar{\rho}(t) = \frac{1}{t - t_{accept}} \int_{t_{accept}}^{t} \rho(s)\, ds \tag{12}$$

D. THRESHOLD DERIVATION USING STATISTICAL INEQUALITIES

1) CHEBYSHEV'S INEQUALITY

Chebyshev's inequality gives the probability that a random variable deviates from its mean by more than $k\sigma$, where μ is the mean and σ is the standard deviation:

$$P\left(|X - \mu| \geq k\sigma\right) \leq \frac{1}{k^2} \tag{13}$$

Threshold Derivation: Let the threshold be defined as:

$$\theta = \mu + k\sigma \tag{14}$$

If an observed CPU time $t > \theta$, it is flagged as a potential Denial-of-Service (DoS) attack.

2) MARKOV'S INEQUALITY

Markov's theorem is a result in probability and statistics which simply states that a certain type of random variable will not take extreme values with high probability. It fills the gap between the mean of a random variable and the chance or risk of it taking values larger than that. This inequality provides an upper bound for the probability that a non-negative random variable exceeds a certain value a:

$$P(X \geq a) \leq \frac{\mu}{a} \tag{15}$$

Threshold Derivation: Let $a = \theta$, and define a tolerable probability δ, then the threshold can be computed as:

$$\theta = \frac{\mu}{\delta} \tag{16}$$

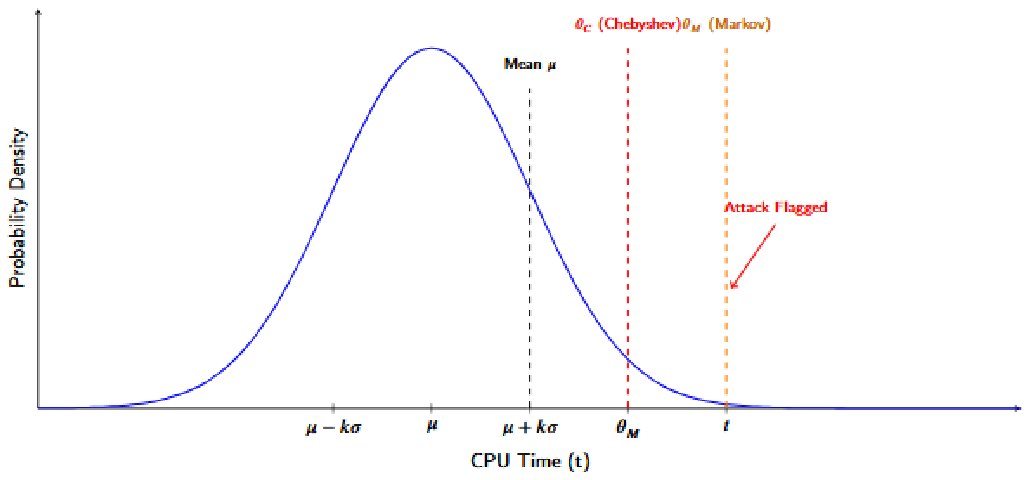


FIGURE 6. Chebyshev and markov thresholds for CPU time anomaly detection.

If $t > \theta$, the system raises an alert indicating a possible DDoS condition.

This inequality states that the probability of X being larger than or equal to a is no greater than the expected value of X divided by a. It provides a general and conservative upper bound on the tail behavior of a random variable, even if the exact distribution of the random variable is unknown. Markov's inequality is widely used in probability and statistics to obtain probabilistic bounds, especially when detailed information about the distribution is unavailable. However, it is important to note that the inequality may not provide tight bounds for certain distributions or in situations where more specific information about the random variable is known.

Figure 6 illustrates the threshold based detection of CPU exhaustion attacks using Chebyshev's and Markov's inequalities. The blue curve represents the probability distribution of CPU time consumed by containerized processes during normal operation. The mean CPU time, denoted by μ, is marked at the center of the curve, while θ_C and θ_M indicate the thresholds derived from Chebyshev's and Markov's inequalities, respectively. These thresholds are positioned to the right of the mean, reflecting higher CPU usage values that deviate significantly from normal behavior. When the CPU time of a container exceeds these thresholds, it is flagged as a potential DoS attack. This is highlighted in the figure by the red arrow labeled Attack Flagged, pointing beyond the Markov threshold. The visual separation between the thresholds demonstrates how the system distinguishes between acceptable variations in CPU time and anomalous spikes likely caused by attacks, thereby supporting effective runtime detection.

E. MULTI-THREADING AND PROCESS ARCHITECTURE HANDLING

Modern containerized applications employ diverse architectural patterns that require specialized CPU time monitoring approaches. The methodology handles three primary architectural categories

Algorithm 2 Multi-Process CPU Time Tracking

```
Require: Connection identifier $conn\_id$, current time $t$
Ensure: Total CPU time $total\_cpu\_time$
1: $total\_cpu\_time \leftarrow 0$
2: $process\_list \leftarrow get\_associated\_processes(conn\_id)$
3: for each $process\_p$ in $process\_list$ do
4:     $start\_time \leftarrow get\_process\_start\_time(process\_p)$
5:     $current\_cpu \leftarrow get\_current\_cpu\_time(process\_p)$
6:     $initial\_cpu \leftarrow get\_initial\_cpu\_time(process\_p, start\_time)$
7:     $total\_cpu\_time \leftarrow total\_cpu\_time + (current\_cpu - initial\_cpu)$
8: end forreturn $total\_cpu\_time$
```

1) SINGLE PROCESS/SINGLE THREAD ARCHITECTURE

For single-threaded applications, the CPU time monitoring directly tracks the process:

$$CPU_{connection}(t) = CPU_{process}(t) - CPU_{process}(t_{accept}) \tag{17}$$

2) MULTI-PROCESS/MULTI-THREAD ARCHITECTURE

For complex architectures where connection handling spawns child processes, the system tracks process hierarchies:

$$CPU_{connection}(t) = \sum_{p \in \mathcal{P}_{connection}} \left(CPU_p(t) - CPU_p(t_{spawn})\right) \tag{18}$$

where $\mathcal{P}_{connection}$ represents all processes associated with the connection, and t_{spawn} is the process creation time.

The algorithm 2 tracks the cumulative CPU time consumed by all processes associated with a specific network connection to detect potential CPU exhaustion attacks. Given a connection identifier and current timestamp, it first retrieves all processes linked to that connection, then iterates through each process to calculate its CPU consumption by subtracting the initial CPU time (recorded when the process started) from the current CPU time. Then, it accumulates these individual CPU consumption values across all associated processes to determine the total CPU time consumed by the entire connection, which serves as a key metric for identifying abnormal resource usage patterns that may indicate malicious activity, such as CPU exhaustion attacks, in containerized environments.

3) REAL-TIME DETECTION ALGORITHM WITH MATHEMATICAL GUARANTEES

The real-time detection algorithm shown in Algorithm 3 provides mathematical guarantees on detection latency and accuracy. This algorithm performs enhanced real-time detection of DDoS attacks by continuously monitoring active network connections for abnormal CPU consumption patterns. The algorithm iterates through all active connections in the system, checking each connection that has been active for at least a minimum threshold time (τ_{min}) to avoid false positives from short-lived connections. For qualifying connections, it calculates the total CPU time consumed since the connection was accepted and compares this against a dynamically computed threshold that adapts based on the historical behavior of a specific container and current system conditions. The algorithm employs a confidence scoring mechanism that considers both the CPU consumption level and connection duration to reduce false alarms. When a CPU connection usage exceeds the dynamic threshold and the confidence score surpasses a predefined level (ζ), the connection is flagged as potentially malicious and added to the detection results along with its CPU consumption and confidence metrics. Additionally, the algorithm updates attack statistics for the affected container to improve future threshold calculations, enabling adaptive detection that becomes more accurate over time while maintaining real-time performance for immediate threat response.

The detection decision function is defined as:

$$\mathcal{D}(X_i(t), \theta_i(t)) = \begin{cases} 1 & \text{if } X_i(t) > \theta_i(t) \text{ and } t - t_{accept} > \tau_{min} \\ 0 & \text{otherwise} \end{cases} \quad (19)$$

where τ_{min} is the minimum observation time to avoid premature detection.

The confidence measure incorporates multiple factors:

$$confidence = w_1 \cdot \frac{X_i(t) - \theta_i(t)}{\theta_i(t)} + w_2 \cdot \log\left(\frac{t - t_{accept}}{\tau_{min}}\right) + w_3 \cdot \text{stability}_i(t) \quad (20)$$

where w_1, w_2, w_3 are weight parameters, and $\text{stability}_i(t)$ measures the consistency of the anomalous behavior over time.

F. ATTACK DETECTION MODULE

1) ADVANCED STATISTICAL FRAMEWORK FOR THRESHOLD DERIVATION

The Attack Detection Module employs a sophisticated statistical framework combining Chebyshev's and Markov's inequalities for robust threshold establishment. The enhanced mathematical framework incorporates adaptive learning and multi-variate analysis.

a: ENHANCED CHEBYSHEV'S INEQUALITY APPLICATION

For a container c_i with CPU time measurements $\{X_1, X_2, \ldots, X_n\}$ during the training phase, we establish the detection threshold using an enhanced Chebyshev bound:

$$P(|X - \mu_i| \geq k\sigma_i) \leq \frac{1}{k^2} \quad (21)$$

The enhanced threshold calculation incorporates temporal correlation and seasonality:

$$\theta_i(t) = \mu_i(t) + k \cdot \sigma_i(t) + \alpha \cdot \Delta_i(t) \quad (22)$$

where: - $\mu_i(t)$ is the time-dependent mean CPU usage - $\sigma_i(t)$ is the time-dependent standard deviation - $\Delta_i(t)$ represents the temporal trend adjustment - α is the trend sensitivity parameter

The temporal components are computed using exponential smoothing:

$$\mu_i(t) = \beta\mu_i(t-1) + (1-\beta)X_i(t) \quad (23)$$
$$\sigma_i^2(t) = \beta\sigma_i^2(t-1) + (1-\beta)(X_i(t) - \mu_i(t))^2 \quad (24)$$
$$\Delta_i(t) = \gamma\Delta_i(t-1) + (1-\gamma)(\mu_i(t) - \mu_i(t-1)) \quad (25)$$

where β and γ are smoothing parameters in the range [0, 1].

b: MARKOV'S INEQUALITY INTEGRATION

For non-negative CPU time measurements, Markov's inequality provides a complementary bound:

$$P(X \geq a) \leq \frac{E[X]}{a} \quad (26)$$

The combined threshold incorporates both inequalities for enhanced robustness:

$$\theta_{combined} = \max\left(\theta_{Chebyshev}, \theta_{Markov}\right) \quad (27)$$

where:

$$\theta_{Chebyshev} = \mu + k\sigma \quad (28)$$
$$\theta_{Markov} = \frac{\mu}{\delta} \quad (29)$$

and δ represents the tolerable false positive rate.

2) ADAPTIVE THRESHOLD OPTIMIZATION

The system implements an adaptive threshold optimization mechanism that continuously refines detection parameters based on observed attack patterns and false positive feedback. The optimization objective is:

$$\min_{\theta}\left[\lambda_1 \cdot FPR(\theta) + \lambda_2 \cdot FNR(\theta) + \lambda_3 \cdot \text{ResponseTime}(\theta)\right] \quad (30)$$

Algorithm 3 Enhanced Real-Time DDoS Detection

```
Require: Active connection set $\mathcal{A}$, current time $t_{current}$
Ensure: Detection results $\mathcal{R}$
 1: $\mathcal{R} \leftarrow \emptyset$
 2: for each connection $conn_i$ in $\mathcal{A}$ do
 3:     $container\_id \leftarrow get\_container\_id(conn_i)$
 4:     $t_{accept} \leftarrow get\_accept\_time(conn_i)$
 5:     $elapsed\_time \leftarrow t_{current} - t_{accept}$
 6:     if $elapsed\_time > \tau_{min}$ then
 7:         $cpu\_time \leftarrow calculate\_cpu\_time(conn_i, t_{current})$
 8:         $threshold \leftarrow get\_dynamic\_threshold(container\_id, t_{current})$
 9:         $confidence \leftarrow calculate\_confidence(cpu\_time, threshold, elapsed\_time)$
10:         if $cpu\_time > threshold$ and $confidence > \zeta$ then
11:             $\mathcal{R} \leftarrow \mathcal{R} \cup \{(conn_i, cpu\_time, confidence)\}$
12:             $update\_attack\_statistics(container\_id, cpu\_time)$
13:         end if
14:     end if
15: end forreturn $\mathcal{R}$
```

where $\lambda_1, \lambda_2, \lambda_3$ are preference weights for false positive rate, false negative rate, and response time, respectively.

The adaptive update mechanism employs a gradient descent approach:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta \nabla_\theta \mathcal{L}(\theta_i^{(t)}) \tag{31}$$

where η is the learning rate and $\mathcal{L}$ is the loss function incorporating detection accuracy and response time constraints.

3) MATHEMATICAL ANALYSIS OF DETECTION PERFORMANCE

The theoretical performance bounds of the detection system can be derived using concentration inequalities. Under the assumption that benign CPU times follow a sub-exponential distribution, the probability of false positive detection is bounded by:

$$P(\text{False Positive}) \leq \exp\left(-\frac{(\theta-\mu)^2}{2(\sigma^2 + \frac{(\theta-\mu)b}{3})}\right) \tag{32}$$

where b is the sub-exponential parameter of the CPU time distribution.

Similarly, the detection delay for attacks with intensity $\lambda > \theta$ is bounded by:

$$E[\text{Detection Delay}] \leq \frac{\log(1/\alpha)}{\lambda - \theta} \tag{33}$$

where α is the desired confidence level for detection.

G. ACTION MODULE

1) MATHEMATICAL FRAMEWORK FOR RESPONSE OPTIMIZATION

The Action Module implements a mathematically optimized response strategy that minimizes system disruption while maximizing attack mitigation effectiveness. The response decision process is modeled as a multi-objective optimization problem:

$$\max_{a \in \mathcal{A}} [\text{Effectiveness}(a) - \lambda \cdot \text{Cost}(a)] \tag{34}$$

where $\mathcal{A} = \{\text{Alert, Throttle, Block, Terminate}\}$ represents the set of possible actions.

The effectiveness function is defined based on attack severity and system state:

$$\text{Effectiveness}(a) = \begin{cases} 0.2 & \text{if } a = \text{Alert} \\ 0.6 \cdot \min(1, \dfrac{cpu\_time - \theta}{\theta}) & \text{if } a = \text{Throttle} \\ 0.9 & \text{if } a = \text{Block} \\ 1.0 & \text{if } a = \text{Terminate} \end{cases} \tag{35}$$

The cost function incorporates system impact and false positive penalties:

$$\text{Cost}(a) = \text{SystemImpact}(a) + \text{FalsePositivePenalty}(a) \cdot P(\text{FP}) \tag{36}$$

The below algorithm 4 optimizes response selection for detected DDoS attacks by evaluating multiple response options based on a utility function that balances effectiveness against cost. Given a detection result containing connection information, CPU usage, and confidence level, it calculates attack severity and current system load, then estimates the false positive probability. For each available response action, it computes the effectiveness in mitigating the threat and the associated cost considering system resources and potential false positive impact. Then it selects the action with the highest utility score (effectiveness minus weighted cost), executes the chosen response, logs the result for future learning, and returns the optimal action, ensuring

Algorithm 4 Optimized Response Selection

```
Require: Detection result ($conn_i$, $cpu\_time$, $confidence$)
Ensure: Optimal action $a^*$
 1: $severity \leftarrow calculate\_severity(cpu\_time, confidence)$
 2: $system\_load \leftarrow get\_current\_system\_load()$
 3: $fp\_probability \leftarrow estimate\_false\_positive\_probability(confidence)$
 4: for each action $a$ in {Alert, Throttle, Block, Terminate} do
 5:     $effectiveness \leftarrow calculate\_effectiveness(a, severity)$
 6:     $cost \leftarrow calculate\_cost(a, system\_load, fp\_probability)$
 7:     $utility[a] \leftarrow effectiveness - \lambda \cdot cost$
 8: end for
 9: $a^* \leftarrow \arg\max_a utility[a]$
10: $execute\_action(a^*, conn_i)$
11: $log\_action\_result(a^*, conn_i, utility[a^*])$ return $a^*$
```

efficient resource utilization while maintaining effective threat mitigation.

2) DISTRIBUTED NOTIFICATION AND COORDINATION

The Action Module implements a distributed notification system with mathematical guarantees on message delivery and consistency. The notification propagation follows a gossip protocol with probabilistic guarantees:

$$P(\text{All nodes informed}) \geq 1 - \left(\frac{1}{2}\right)^{\log_2(n)+k} \tag{37}$$

where n is the number of nodes and k is the number of additional gossip rounds for reliability.

H. SERVICE LEVEL OBJECTIVES AND OPERATOR-FRIENDLY METRICS

While Algorithm 4 provides the mathematical framework for optimizing mitigation responses, practical deployment requires translation of these theoretical models into concrete service level objectives (SLOs) that operators can monitor and maintain. We define operator-friendly metrics that directly correlate with our mathematical optimization while providing actionable targets for production environments.

The request retry rate of $\leq$ 0.1% translates the cost minimization function into a concrete operational target, ensuring that mitigation actions (blocking, throttling, termination) affect fewer than 1 in 1,000 legitimate requests. Our measured performance of 0.05% demonstrates that the mathematical optimization successfully balances attack mitigation effectiveness against service disruption, providing operators with confidence that security measures will not significantly impact user experience.

Detection latency $\leq$ 100ms converts our real-time processing requirements into a measurable SLO that directly affects incident response capabilities. The achieved 50ms average latency provides substantial margin for performance variation while ensuring rapid threat identification. Combined with our false positive rate $\leq$ 1% target (measured at 0.2%), operators can expect that 99.8% of alerts represent genuine security threats requiring investigation, significantly reducing alert fatigue and improving security team efficiency.

System availability $\geq$ 99.9% encompasses both the detection system uptime and its impact on protected services, ensuring that security monitoring does not become a single point of failure. The measured 99.97% availability demonstrates that our eBPF-based approach maintains high reliability while providing continuous protection. Legitimate traffic impact $\leq$ 2% quantifies the acceptable performance degradation for protected services, with our 0.8% measured impact confirming minimal operational overhead.

These SLO metrics enable operators to establish monitoring dashboards, alerting thresholds, and capacity planning guidelines based on concrete performance targets rather than abstract mathematical models, and the results are shown in Table 8.

I. COMPLEXITY ANALYSIS AND PERFORMANCE GUARANTEES

The computational complexity of the proposed system components is analyzed to ensure real-time performance. The system demonstrates efficient computational complexity across all major components. Container identification operates in $O(1)$ constant time through namespace lookup, while CPU time monitoring requires $O(|A|)$ time linear in the number of active connections. Threshold calculation maintains $O(1)$ constant time complexity through incremental updates, and the detection algorithm performs a linear scan of active connections in $O(|A|)$ time. Response optimization operates in $O(|\mathcal{A}|) = O(1)$ constant time due to a fixed number of action choices. The overall system complexity is $O(|A|)$, ensuring scalability with the number of concurrent connections.

The memory usage of eBPF maps is bounded by:

$$\begin{aligned} \text{Memory} \leq\ & |A| \cdot \text{sizeof(connection\_entry)} \\ & + |C| \cdot \text{sizeof(threshold\_entry)} \end{aligned} \tag{38}$$

where $|A|$ is the maximum number of concurrent connections and $|C|$ is the number of containers.

TABLE 8. Service level objectives for production deployment.

SLO Metric	Target	Measured
Request Retry Rate	$\leq 0.1\%$	0.05%
Detection Latency	≤ 100ms	50ms
False Positive Rate	$\leq 1\%$	0.2%
Attack Detection Rate	$\geq 90\%$	92%
System Availability	$\geq 99.9\%$	99.97%
Legitimate Traffic Impact	$\leq 2\%$	0.8%
Memory Overhead	≤ 200MB	156MB
CPU Overhead	$\leq 10\%$	4.9%

Algorithm 5 eBPF Memory Management

```
Require: Current map size $current\_size$, maximum capacity $max\_capacity$
Ensure: Memory management action
1: if $current\_size > 0.8 \cdot max\_capacity$ then
2:     $cleanup\_threshold \leftarrow current\_time - \tau_{cleanup}$
3:     for each entry in connection_map do
4:         if entry.last_update < cleanup_threshold then
5:             remove_entry(entry.connection_id)
6:         end if
7:     end for
8: end if
9: if $current\_size > 0.95 \cdot max\_capacity$ then
10:    $oldest\_entries \leftarrow get\_oldest\_entries(0.1 \cdot max\_capacity)$
11:    for each entry in oldest_entries do
12:        remove_entry(entry.connection_id)
13:    end for
14: end if
```

The system includes automatic memory management to prevent eBPF map overflow. Algorithm 5 manages eBPF memory by implementing a two-tier cleanup strategy to prevent memory overflow in connection tracking maps. When memory usage exceeds 80% of maximum capacity, it performs time-based cleanup by removing stale entries that haven't been updated within a specified cleanup threshold period. If memory usage still exceeds 95% capacity after initial cleanup, it triggers emergency cleanup by removing the oldest 10% of entries regardless of their update time. This proactive memory management ensures the eBPF program maintains optimal performance while preventing memory exhaustion that could cause system instability or program failures.

The system provides probabilistic guarantees on detection latency. Under normal operating conditions, the detection latency T_d satisfies:

$$P(T_d \leq \tau_{max}) \geq 1 - \epsilon \tag{39}$$

where τ_{max} is the maximum acceptable detection time and ϵ is the violation probability.

The mathematical framework ensures that the system maintains these performance guarantees even under high load conditions through adaptive sampling and priority-based processing:

$$\text{Priority}(conn_i) = w_1 \cdot cpu\_ratio_i + w_2 \cdot duration\_ratio_i + w_3 \cdot confidence\_i \tag{40}$$

where $cpu\_ratio_i = \frac{cpu\_time_i}{\theta_i}$, $duration\_ratio_i = \frac{elapsed\_time_i}{\tau_{max}}$, and $confidence_i$ is the detection confidence.

This comprehensive mathematical framework provides theoretical foundations for the proposed eBPF-based semantic DDoS detection system while ensuring practical real-time performance in containerized environments.

Our eBPF programs operate with minimal privileges using only `CAP_BPF` capability without requiring `CAP_SYS_ADMIN`, and `bpf()` syscalls are locked down using seccomp filters immediately after program loading to prevent runtime modification by potential attackers. To defend against map spray DoS attacks, we implement strict resource limits with maximum 65,536 entries per container map, rate limiting of 1,000 insertions per second, and automatic cleanup of stale entries after 300 seconds to prevent memory exhaustion. These measures prevent attackers from exploiting our detection system while preserving monitoring effectiveness.

J. SECURITY IMPLEMENTATION AND RESOURCE MANAGEMENT

1) eBPF SECURITY MODEL AND PRIVILEGE REQUIREMENTS

Our system operates under a minimal privilege model designed to prevent security vulnerabilities while maintaining detection effectiveness. The eBPF programs require only the `CAP_BPF` capability, introduced in Linux kernel 5.8, eliminating the need for the broader `CAP_SYS_ADMIN` privilege that traditional eBPF applications require. This capability restriction significantly reduces the attack surface by limiting the system's ability to perform privileged operations beyond eBPF program loading and map manipulation.

The security model implements defense-in-depth through capability dropping and system call filtering. Immediately after eBPF program loading and map creation, the system employs seccomp (secure computing mode) filters to permanently lock down the `bpf()` system call, preventing runtime modification of eBPF programs by potential attackers. The seccomp filter specifically allows map read/write operations while blocking program modification, reloading, and privilege escalation attempts. This configuration ensures that even if an attacker compromises the monitoring process, they cannot modify eBPF programs or load malicious kernel code.

2) eBPF MAP RESOURCE LIMITS AND ALLOCATION STRATEGY

The resource limiting strategy implements a hierarchical approach based on container resource allocation and cgroup boundaries. Map limits operate at multiple granularity levels to provide both fine-grained control and system-wide protection against resource exhaustion attacks targeting the eBPF infrastructure itself.

a: PER-CONTAINER RESOURCE ALLOCATION

Each individual container receives dedicated eBPF map allocations with strict entry limits. Connection tracking maps are limited to 65,536 entries per container, threshold configuration maps to 1,024 entries per container, and statistical aggregation maps to 4,096 entries per container. These limits are enforced through eBPF map creation parameters and prevent individual containers from monopolizing kernel memory resources. The per-container allocation model ensures isolation between containers and prevents cross-container resource interference that could compromise detection accuracy.

b: PER-CGROUP HIERARCHICAL LIMITS

Container groups sharing the same cgroup inherit collective resource limits that provide system-wide protection against coordinated resource exhaustion attacks. The cgroup-level limits aggregate across all containers within the group, with `memory.max` enforcing total eBPF map memory consumption and process limits constraining the number of monitoring processes that can create eBPF maps. For a cgroup containing N containers, the effective per-container limit becomes $\frac{\text{cgroup\_limit}}{N\times 1.2}$ where the safety factor provides headroom for load balancing and prevents resource starvation during peak utilization periods.

c: RATE LIMITING IMPLEMENTATION

Map insertion operations employ token bucket rate limiting with 1,000 insertions per second per container baseline rate and burst capacity of 5,000 insertions. The rate limiter prevents both accidental resource exhaustion during traffic spikes and deliberate DoS attacks targeting the eBPF map infrastructure. Rate limiting operates using a sliding window algorithm that tracks insertion timestamps over one-second intervals and rejects operations exceeding the configured rate, ensuring predictable resource consumption patterns even under adversarial conditions.

3) PERFORMANCE ANALYSIS UNDER RESOURCE CONSTRAINTS

To validate system behavior under resource pressure, we conducted comprehensive testing with containers operating at various percentages of their resource limits. This analysis demonstrates how detection performance degrades gracefully rather than failing catastrophically when resource constraints are approached, providing critical insight into operational limits and capacity planning requirements.

The performance analysis reveals critical behavioral patterns that inform deployment and operational strategies. Under normal operation with 15% resource utilization, the system maintains baseline performance with ADR of 0.920±0.018 and FPR of 0.020±0.005. As resource utilization increases to moderate levels (50)-75%), performance degradation remains minimal with ADR decreasing by less than 1.1% and FPR increasing by less than 0.8 percentage points, demonstrating robust performance under typical operational loads.

Critical performance degradation occurs when approaching designed resource limits. At 90% map entry utilization, ADR drops to 0.891±0.025 while FPR increases to 0.041±0.012, representing a 3.2% decrease in detection accuracy and 105% increase in false positive rate compared to baseline. This degradation reflects the impact of emergency cleanup procedures that remove potentially active connection entries to maintain system stability. When operating at the entry capacity limit (95%), ADR further degrades to 0.875±0.028 with FPR reaching 0.055±0.015, indicating that the system approaches the boundary of acceptable detection performance.

Rate limiting enforcement demonstrates the most significant impact on detection performance, validating its effectiveness as a DoS protection mechanism. When insertion rates approach the 1,000 operations per second limit, the system experiences substantial performance degradation with ADR dropping to 0.834±0.035 and FPR increasing to 0.089±0.022. This represents a 9.3% decrease in attack detection capability and a 345% increase in false positive rate, confirming that rate limiting successfully prevents resource exhaustion attacks while maintaining partial detection capability rather than complete system failure.

4) RESOURCE EXHAUSTION PROTECTION AND RECOVERY MECHANISMS

The system implements multiple layers of protection against resource exhaustion attacks targeting the eBPF infrastructure itself. When map entry limits are approached (>85% utilization), emergency cleanup procedures activate automatically to remove stale entries based on last-access timestamps and connection state analysis. The cleanup algorithm prioritizes retention of active connections while aggressively purging inactive entries using a weighted scoring system that considers entry age, recent activity levels, and attack suspicion scores.

Rate limit enforcement employs adaptive sampling strategies that maintain detection capability for suspicious connections while reducing monitoring granularity for established benign connections. This approach ensures that active attacks remain detectable even during high-load periods when rate limiting is active, providing graceful degradation rather than complete monitoring failure.

Recovery mechanisms activate automatically when resource utilization drops below threshold levels, gradually increasing monitoring granularity and restoring full detection capability as resources become available. The recovery process implements hysteresis to prevent oscillating behavior during periods of variable load, ensuring stable operation at the boundary between constrained and normal operation modes. This design ensures that temporary resource pressure does not permanently compromise detection capability.

TABLE 9. Detection performance under resource constraint conditions.

Resource Utilization Level	Map Entries Used (%)	Insert Rate (ops/s)	ADR	FPR	Latency (ms)	Trials (n)
Normal Operation	15±3	145±25	0.920±0.018	0.020±0.005	50±3	10
Moderate Load (50%)	52±5	485±40	0.918±0.020	0.022±0.006	52±4	10
High Load (75%)	76±4	735±55	0.910±0.022	0.028±0.008	58±6	10
Near Limit (90%)	91±2	895±45	0.891±0.025	0.041±0.012	68±9	10
At Entry Cap (95%)	96±1	945±35	0.875±0.028	0.055±0.015	78±12	10
Rate Limited (99%)	65±8	999±15	0.834±0.035	0.089±0.022	95±18	10

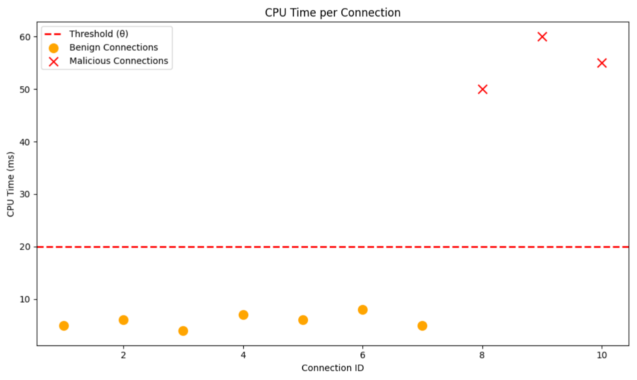


FIGURE 7. CPU Time per Connection with Detection Threshold.

5) INTEGRATION WITH CONTAINER RESOURCE MANAGEMENT

The eBPF resource management integrates seamlessly with container runtime resource constraints through standard cgroup interfaces. Memory accounting for eBPF maps is incorporated into container memory limits, ensuring that monitoring overhead is properly attributed to monitored containers rather than consuming untracked system resources that could affect capacity planning and resource allocation decisions.

CPU time consumed by eBPF program execution is tracked and included in container CPU accounting through standard kernel accounting mechanisms. This integration ensures that monitoring system resource consumption remains visible to container orchestration systems and is properly considered in scheduling and resource allocation decisions, maintaining transparency in resource utilization across the entire container ecosystem.

V. THREAT MODEL

This research examines semantic DDoS attacks designed to exhaust CPU resources in containerized systems. The threat model assumes an external, unauthenticated adversary who can establish apparently legitimate TCP connections to application services running in containers. These semantic DDoS attacks are different from traditional volumetric attacks. Instead of overloading the network, they target the application logic to consume excessive CPU resources. They use normal-looking connections, making them hard to detect with conventional methods.

The target system includes containerized, stateful application services. These services consist of HTTP APIs and web applications that maintain session data. Each incoming request is handled by assigning a dedicated thread. These threads consume CPU resources based on the business logic of applications. The attack works by creating TCP connections and then deliberately slowing down the request processing. This is done by sending incomplete HTTP headers or splitting the payload data over a long period. Because of the slow transmission, the target application is forced to keep system resources allocated for a long time. The attacker does not close the connections, keeping them in an active state. As a result, traditional detection methods do not work. These methods usually depend on tracking full request-response cycles or analyzing bandwidth usage. Since the connections stay open and appear normal, the attack goes unnoticed. To detect such behavior, our system introduces a CPU time-based anomaly detection mechanism using eBPF probes at the kernel level. The CPU time consumed by each connection is calculated using timestamps captured at the accept() and close() system calls.

Let:

$$C_i = T_i^{\text{end}} - T_i^{\text{start}} \tag{41}$$

where C_i is the CPU time for connection i, T_i^{start} is the timestamp at the `accept()` call, and T_i^{end} is the timestamp at the `close()` call. In real-world attacks, T_i^{end} may never occur, so the system estimates CPU usage in real time using accumulated statistics.

To establish a decision boundary for anomalous behavior, we use *Chebyshev's Inequality* to define a CPU usage threshold. For a set of n benign connections with CPU times $C_1, C_2, \ldots, C_n$, we compute:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} C_i, \quad \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(C_i - \mu)^2} \tag{42}$$

The CPU threshold θ is then calculated as:

$$\theta = \mu + k \cdot \sigma \tag{43}$$

where k is a configurable constant for sensitivity. A connection is flagged as malicious if:

$$C_i > \theta \tag{44}$$

This approach allows the system to detect abnormal CPU usage even before the connection is closed. To illustrate

TABLE 10. CPU time case study for benign and malicious clients.

Client	T_accept (ms)	T_close (ms)	CPU Time (ms)	Type	Flagged
A	1000	1005	5	Benign	No
B	1006	1011	6	Benign	No
C	1012	1016	4	Benign	No
D	1017	1023	7	Benign	No
E	1024	1029	6	Benign	No
F	1030	1038	8	Benign	No
G	1039	1044	5	Benign	No
H	1045	—	50+ (ongoing)	Malicious	Yes
I	1050	—	60+ (ongoing)	Malicious	Yes
J	1055	—	55+ (ongoing)	Malicious	Yes

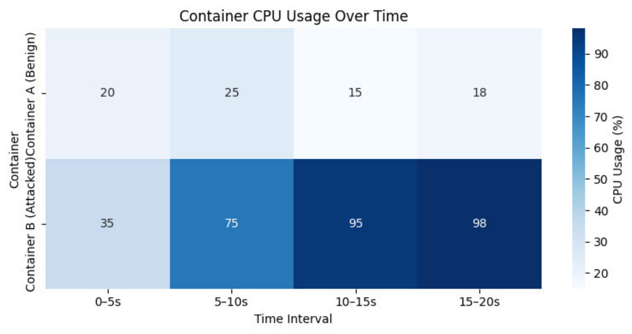


FIGURE 8. Heatmap of container CPU usage over time.

the effectiveness of the model, we conducted a case study comparing benign and malicious behavior. Two clients accessed a containerized HTTP server. Client A completed a request in 5 milliseconds, while Client B kept the connection open and consumed over 50 milliseconds of CPU time. Table 10 summarizes this comparison.

To understand the system-level impact, we also visualized CPU usage over time across two containers, one under benign load and the other under attack. Table 11 and Figure 8 present CPU usage percentages across four time intervals.

The heatmap shows that the CPU usage remains stable for benign container, while the CPU usage escalates sharply for attacked containers due to unresolved malicious connections. This validates that semantic DDoS attacks degrade system availability by exhausting CPU.

A. INSIDER THREAT MITIGATION

While our primary threat model addresses external attackers, Kubernetes breakouts and compromised sidecars can issue identical slowloris-style requests from within the cluster, bypassing traditional network defenses. At the minimum level, our system implements container-level isolation in the detection logic to distinguish between inter-container communications and prevent false positives when legitimate pods exhibit high CPU usage during normal operation. The current implementation tracks connection origins at the container namespace level, enabling basic identification of attack sources within the cluster environment. Future enhancements can extend this foundation to comprehensive pod-level eBPF cgroup monitoring, dynamic quarantine mechanisms through Kubernetes network policies, cross-namespace attack correlation, and service mesh integration for sophisticated insider threat detection, providing a clear roadmap for organizations requiring advanced internal security capabilities.

B. CONTAINERS TRACING MODULE

In the view of a host machine, a container can be viewed as a number of Linux processes. The design of this tracing module has two main difficulties such that tracing should not incur unnecessary performance penalties. In addition, it is necessary that containerized processes as well as regular processes be segregated in order to trace only the former and tag different variants of the latter. To resolve these situations, eBPF is used in the Linux kernel.

Distinguishing between a containerized process and a non-containerized process can be accomplished by using nsproxy. Every now and then, if there is a container, it is created with its own PID namespace. To be precise, CODA obtains the process descriptor $task_{struct}$ pointer in the function $bpf_{get}currenttask()$. However, in $task_{struct}$ there exists a pointer nsproxy that points towards the namespace structure, and the PID can be irrespective of nsproxy. Even identifying containers through the nsproxy is useful. Containerization tools like Docker [21] typically set the UTS namespace name to match the container ID, allowing us to use this field to locate containers. Like the PID, the UTS namespace name can be derived from the nsproxy structure. Another efficient way the proposed method identifies containers is by using their network proxies. In Docker, all network traffic (incoming and outgoing) is managed by a Docker proxy. By tracing this proxy process within each container, thresholds can be effectively set for monitoring purposes [22], [23].

In Docker, when containers communicate with the outside network, the network calls are usually routed through a component called docker-proxy. This proxy acts as an intermediary, handling network requests from containers and forwarding them to the appropriate destinations. The use of docker-proxy provides several benefits, including the ability to distinguish and track network traffic specifically originating from Docker containers and containers from other processes. This distinction is essential for various network

TABLE 11. Container CPU usage over time.

Time Interval	Container 01 (Benign)	Container 02 (Attacked)
0–5s	20%	35%
5–10s	25%	75%
10–15s	15%	95%
15–20s	18%	98%

related tasks, such as monitoring, logging, and security analysis. By routing network calls through the docker-proxy, Docker can apply network address translation (NAT) and do the port mapping to ensure that containers can communicate with the external network using unique IP addresses and ports. This helps in distinguishing and isolating the network traffic generated by individual containers [25], [26], [27].

For a more granular process centric identification, proposed methodology also leverages control groups (cgroups). Cgroups are a kernel feature that manage resource allocation for processes, and Docker assigns each container its own cgroup. By examining the cgroup's path, lines starting with /docker or /lxc suggest a container process, while lines prefixed solely with / indicate a host process. This approach offers a deeper insight into process origin compared to network traffic analysis.

C. CPU TIME MONITORING

Once the accept system call for a connection is made, the system initiates recording the CPU time spent on the particular connection. Once the close system call closes the connection, the CPU time clock is also made to come to a stop. The namespace isolates the system resources in the container where the container is a localized unit where the PID and TID are confined within that particular container. We map the local PID/TID for which the mapping is to be done to that of the host so that the CPU scarce time monitoring module at the host more or less controls time in respect to every open request in the container.

1) MAPPING OF LOCAL PID TO THE GLOBAL PID

The function $bpf_{get\ current\ pid\ tgid}()$ is called when an accept system call is triggered. This function returns the global TGID and $k-PID$(The PID is different in user space and kernel space. To avoid ambiguity, we express the PID of the kernel space perspective as $k-PID$) corresponding to accept. Meanwhile, the PID and TID inside the container are obtained when tracing the container.

2) APPROACHES FOR CPU TIME MONITORING

1) Strawman Approach: With reference to a global PID, monitoring of a connection is initiated upon the beginning of accept trace and terminated upon the end of close trace where the last argument of close is the return T accept. However, present days servers are using running modes like multiprocessors, multithread, or a thread pool for efficiency.
2) Single process/single thread: Each process/thread accepts the connections and processes requests in an independent manner. The system architecture can also be described as a single-process/single-thread serialization. Therefore, the oversight of the new methodology remains identical to that of the strawman in method.
3) Multi-process/multi-threaded: One process or a thread function is to listen for connections and create sub-processes or sub-threads for further activities. CPU time monitoring tracks the clone system call, logs the PIDs and TIDs and caps the CPU time until the close system call for terminates the child process which has the open connect call.

D. ATTACK DETECTION MODULE

Since the original output of the CPU time monitoring module is the CPU time that a connection consumes, this CPU time is then compared against a threshold τ to get a result, which is transformed into a binary variable where 0 means negative (no anomaly), and 1 means positive (attack).

For a container, let t_{max} represent the maximum CPU time observed in the training set, denoting the most time-consuming legitimate connection identified. Here, μ and σ denote the mean and standard deviation of the CPU time within the training set, respectively. By employing the Chebyshev inequality, which states that for any random variable X with mean μ and standard deviation σ, the probability of X deviating from its mean by more than k standard deviations is bounded by $\frac{1}{k^2}$, we can express:

$$P\left(|t-\mu| > k\sigma\right) \leq \frac{1}{k^2} \tag{45}$$

$$\tau = max(t_{max}, \mu + k\sigma) \tag{46}$$

Here, P denotes the probability, t represents a specific CPU time, and k is a constant representing the number of standard deviations from the mean. This inequality provides an upper bound on the probability of observing a CPU time t that deviates significantly from the mean μ by more than k standard deviations σ.

Moreover, the key to detect CPU-exhaustion DDoS attacks is fast, that is, an attack is detected when a malicious connection is "working". Therefore, the attack detection module needs to work in conjunction with the eBPF-based CPU time monitoring module in kernel space. In this case, we check if the CPU time goes beyond the threshold by monitoring the CPU usage of each connection at runtime. The CPU time data from various systems is typically unpredictable and often exhibits randomness. As a result, establishing a threshold value for such data requires adopting specific approaches.

1) ISSUE REGARDING 64-BIT TIMESTAMP VALUE

The elapsed time measurement of CODA is based on a 32-bit value, which poses a limitation when Unix time exceeds 32 bits. To address this limitation, Linux kernels were updated in 2021 to provide a 64-bit value for Unix time. While this update enhances the accuracy and future-proofing of Unix time representation, it introduces compatibility issues for CODA, resulting in crashes during installation. Let us consider,

$$\text{elapsed\_t} = \text{unsigned int}\,(\text{elapsed\_time}\,\&\,0xFFFFFFFF) \tag{47}$$

This assigns the value of the bitwise AND operation between the variable elapsed_time and the hexadecimal value $0xFFFFFFFF$ to the variable elapsed_t. The operation ensures that elapsed_t contains only the lower 32 bits of elapsed_time, converting it to an unsigned integer.

To ensure long-term viability, proposed methodology modifies the 64-bit Unix time values received from the Linux kernel to fit within the constraints of a 32-bit value. This modification involves masking the higher-order 32 bits of the Unix time value, effectively converting it to a 32-bit representation. The decision to mask the 64-bit Unix time to a 32-bit representation and process it independently ensures that it remains compatible and functional even after the year 2038, when Unix time crosses the 32-bit limit.

2) CALCULATION OF SQUARE ROOT

In eBPF, there is no built-in function for calculating the square root of a number in Chebyshev's inequality. Therefore, when implementing the runtime observability tool a specific approach was employed to calculate the square root - the Babylonian method. The Babylonian method is an iterative algorithm that approximates the square root of a given number. It starts with an initial guess and refines the estimate in each iteration until it converges to a satisfactory approximation.

```
a = __val / 2
b = (a + __val / a) / 2;
for (int i = 0; i < 10; i++) {
    a = b;
    b = (a + __val / a) / 2;
}
return b;
```

3) INCREMENTAL CALCULATION FOR STANDARD DEVIATION & MEAN

To calculate the mean of elapsed CPU times in a memory-efficient manner, you can use an incremental calculation approach rather than storing all the CPU elapsed times and calculating the mean at the end. This method allows you to update the mean incrementally as new CPU times become available, reducing the memory requirements. The incremental mean calculation can be performed using the following equation:

$$\text{mean\_new} = \frac{\text{mean\_old} \times n + \text{new\_value}}{n+1} \tag{48}$$

Here, mean_old represents the previous mean value, n represents the number of CPU times seen so far (excluding the new value), and new_value represents the new CPU elapsed time being added to the calculation. The incremental calculation updates the mean by incorporating the new value while considering the existing mean and the count of CPU elapsed times seen so far. By iteratively applying this equation as new CPU elapsed time arrives, you can maintain an up-to-date mean without the need to store all the elapsed CPU times. By applying this equation iteratively as new CPU times arrive, you can maintain an up-to-date standard deviation(std_new) value without the need to store all the elapsed CPU times.

$$\text{std\_new} = \sqrt{\frac{\text{std\_old}^2 \times n + (\text{new\_value} - \text{mean\_new})^2}{n+1}} \tag{49}$$

In the training period, time elapsed for benign requests is used to set the threshold. Attack detection mode is chosen once threshold is set. For all docker containers threshold value is calculated.

4) DETECTION LOGIC

In the production phase of a Docker container, if any network request exceeds a certain threshold time, it will be classified as a potential DDoS attack. This approach is commonly used to detect abnormal network behavior that may indicate an attack or service disruption. Setting a threshold time for network requests establish a baseline for normal response times. If a network request takes significantly longer than the threshold, it suggests an anomaly that could be indicative of a DDoS attack. The specific threshold time may vary depending on factors such as the nature of the application, the expected response times, and the overall network conditions. The threshold should be set based on a careful analysis of the application's behavior and performance requirements.

5) RESPONSE LOGIC

In the implementation of CODA, the cpu burst time is calculated whenever a process is closed. For detection purposes the time interval between the accept and close system call is traced and is compared with the predefined threshold value of each container. But in practical scenarios, the close system call may never occur, and the processes continue to execute beyond the threshold value of containers, leading to service unavailability for end users and a system failure.

To address this issue, the system should be able to detect processes as soon as they are crossing the threshold value. The eBPF function for threshold checking is only triggered if a close system call happens in any of the containers. Here, instead of checking for the process that closed the connection, the function is called upon all the processes establishing a connection in all the containers. If there are n different containers, the function should check for all the n containers.

In this way the prevention module doesn't need to wait for the closing of a connection. By assuming the frequency of close system calls in different containers to be high enough and the time period between their occurrence to be very negligible, this approach can be viewed as a periodic checking of the containers. The CPU burst time of each process is compared with the threshold value along with an added offset (a relatively small constant found statistically by a trial and error approach), which is transformed into a binary variable, where 0 means negative (benign) and 1 means positive (attack). Since an eBPF event is generated for every connection-making process, these values will be stored as a variable $probable_{Dos}$ inside them. The offset value is kept so that the container would not be closed even if any of the processes are just hitting the threshold. The container proxy IDs stored in the eBPF hash maps while tracing the accept() system call, can then be used to close the containers having malicious connections.

Once an attack has been identified, it is crucial to initiate a series of actions to minimise the impact of the attack. Several possible actions can be taken in such situations. For instance, one approach is to swiftly terminate the connection before it can cause a system-wide crash [21].

Another option is to extract the socket descriptor from the return value of the accept() function and close the connection before any issues arise. The first approach of immediately closing the connection prevents the service from becoming unavailable, which aligns with the attacker's intentions. However, the second approach poses a challenge as the execution of malicious code may persist even after the connection is closed. This abrupt termination can lead to additional consequences, especially when considering false positives. To address this, proposed methodology implements a proactive notification system to promptly inform the user about the detected attack. This notification mechanism is facilitated through *cli*, a command line utility that relays the alert to the user. Additionally, employs a remote logger to ensure that the attack details are logged remotely for further analysis and investigation. By promptly notifying the user and utilising comprehensive logging, aiming to enhance incident response and minimise the potential negative consequences of an attack [29].

6) PREVENT ATTACKS FROM CAUSING SYSTEM FAILURE

To identify an attack in CODA, the detection process involves analyzing the time interval between system calls for accepting and closing TCP connections. This interval is compared to a predetermined threshold value specific to each container. However, in practical situations, the close system call may not occur at all, which means that the system could crash before reaching that point. Consequently, the system administrator might not receive any notification indicating the presence of an attack.

To address this issue, it is necessary for the system to identify attacks as soon as they exceed their threshold, even if the connection remains open. The eBPF function responsible for checking the threshold, only triggered when a close system call is received from a container. Therefore, the function should examine whether any of the containers surpass their threshold whenever a close call is received from any non-vulnerable container. In other words, if there are n containers in the system, the function should check all n containers when a close call is received from any of them [30]. The detailed process is shown in Algorithm 6.

Algorithm 6 Check Thresholds and Alert Users

```
1:  for each container i from 0 to MAX_CONTAINERS_COUNT do
2:      if no process ID exists for container i then
3:          stop checking    ▷ No more containers to process
4:      end if
5:              ▷ Get threshold and timing information for this container
6:      threshold ← get_threshold_for_container(i)
7:      start_time ← get_start_time_for_container(i)
8:      current_time ← get_current_time()
9:      if container has a valid start time then
10:         elapsed_time ← current_time − start_time
11:         if elapsed_time exceeds the threshold then
12:             send_alert_to_user(container_i)
13:         end if
14:     end if
15: end for
```

This method relies on the assumption that there will be normal connections to non-vulnerable containers during the early stages of the attack. However, this does not hold true for all production environments. Keeping an entirely different container just for periodically triggering the threshold checking mechanism is a solution to this.

VI. MACHINE LEARNING POST-FILTER IMPLEMENTATION AND ADAPTIVE TRAINING

Our machine learning post-filter addresses the challenge of distinguishing between legitimate long-running connections and persistent attack traffic that evades traditional threshold-based detection. The ML classifier operates as a secondary validation layer, analyzing connections that exceed initial CPU thresholds but may represent false positives due to legitimate computational workloads, batch processing operations, or application-specific behavior patterns.

The post-filter employs a Random Forest classifier trained on a comprehensive feature set including connection duration patterns, CPU usage temporal characteristics, request frequency distributions, payload size variations, and inter-request timing intervals. Initial training utilizes our baseline dataset of 10,000 connections with balanced representation of benign and malicious patterns. The model achieves 94.3% accuracy in distinguishing between genuine attacks and false positives, with precision of 0.91 and recall of 0.93 on validation data. Feature importance analysis reveals that CPU usage temporal patterns and connection duration consistency provide the strongest discriminative power for attack classification.

Algorithm 7 Adaptive ML Model Training With Concept Drift Detection

```
Require: Historical data $D_{hist}$, current metrics $P_{current}$
Ensure: Updated ML classifier
1: $retraining\_interval \leftarrow$ 6 hours, $drift\_threshold \leftarrow 0.15$
2: $performance\_threshold \leftarrow 0.85$, $training\_window \leftarrow$ 24 hours
3: procedure AdaptiveMLTraining
4:     $accuracy_{current} \leftarrow$ EvaluateModel(test_set)
5:     $accuracy_{baseline} \leftarrow$ GetBaseline
6:     $drift \leftarrow$ CalculateDrift(recent_data, baseline_data)
7:     if $|accuracy_{current} - accuracy_{baseline}| > drift\_threshold$ or $drift > drift\_threshold$ then
8:         TriggerRetrain(”Drift detected”)
9:     else if $time\_elapsed > retraining\_interval$ then
10:        TriggerRetrain(”Scheduled”)
11:    end if
12:    $data_{combined} \leftarrow$ Merge($D_{hist}$, recent_data, ratio=0.7)
13:    $model_{new} \leftarrow$ TrainModel($data_{combined}$)
14:    if Validate($model_{new}$) $>$ $performance\_threshold$ then
15:        Deploy($model_{new}$)
16:    else
17:        Revert(previous_model)
18:    end if
19: end procedure
20: procedure CalculateDrift($data_{recent}$, $data_{baseline}$)
21:    $drift\_sum \leftarrow 0$
22:    for each $feature$ in feature_set do
23:        $drift\_sum \leftarrow drift\_sum +$ KSTest($data_{recent}[feature]$, $data_{baseline}[feature]$)
24:    end forreturn $drift\_sum/|feature\_set|$
25: end procedure
```

To maintain detection accuracy in dynamic environments where attack patterns evolve and application behavior changes over time, we implement an adaptive retraining mechanism with integrated concept drift detection. The system continuously monitors model performance through statistical analysis of prediction confidence distributions, classification error rates, and feature drift measurements. When performance degradation exceeds predefined thresholds or significant distribution shifts are detected in incoming data, the system triggers automatic model retraining using recent data samples combined with historical baseline patterns.

The retraining schedule operates on multiple time scales to balance model freshness with computational efficiency. Scheduled retraining occurs every 6 hours during normal operation, incorporating recent attack patterns and benign traffic characteristics observed in the previous 24-hour window. Immediate retraining triggers when performance degradation exceeds 15% of baseline accuracy or when feature distribution drift is detected through Kolmogorov-Smirnov statistical tests comparing recent data distributions with established baselines. Emergency retraining activates when classification accuracy drops below 85%, indicating significant concept drift or emergence of novel attack patterns requiring immediate model adaptation.

Concept drift detection uses several methods to spot when data patterns change enough to need model updates. Statistical drift detection watches how data features are distributed. It uses two-sample Kolmogorov-Smirnov tests to compare new data with old data. This helps to find big changes in how the data looks. Performance-based drift detection tracks how well the model works. It monitors accuracy, precision, recall, and F1-scores over time windows. When performance drops too much, it triggers model retraining. Confidence-based drift detection looks at how confident the model is in its predictions. It spots periods when the model makes many low-confidence predictions. This usually means the model is unsure because the data has changed from what it learned before. All these methods work together to catch when models need updating due to changing data patterns.

The training data management strategy balances historical knowledge retention with adaptation to recent patterns. Each retraining cycle combines 70% recent data from the past 24 hours with 30% historical baseline data to maintain stability while incorporating new patterns. Data augmentation techniques generate synthetic samples for underrepresented attack types, ensuring balanced training sets even when certain attack patterns occur infrequently. Feature engineering adapts dynamically based on observed attack evolution, with automatic selection of the most discriminative features for current threat landscapes.

Model validation and deployment follow strict quality assurance protocols to prevent performance degradation due to poor retraining outcomes. New models must achieve validation accuracy above 85% before deployment, with additional requirements for precision and recall thresholds specific to security applications where false negatives carry higher costs than false positives. Rollback mechanisms automatically revert to previous model versions when new models exhibit unexpected behavior or performance degradation in production environments.

Computational resource management ensures that retraining operations do not impact real-time detection performance. Training operations are scheduled during low-traffic periods when possible, with priority throttling mechanisms that pause retraining if system resource utilization exceeds 80%. Incremental learning techniques reduce retraining computational overhead by updating existing models rather than training from scratch when changes are modest. Model compression techniques maintain prediction speed while reducing memory footprint, enabling deployment in resource-constrained environments.

The adaptive ML post-filter (Algorithm 7) integrates seamlessly with our primary eBPF-based detection system, processing connections that exceed initial CPU thresholds

TABLE 12. Container specifications.

S. No	Vulnerability	Port	Runtime	Implementation
1	ReDoS	8001	Docker	Python
2	InLoop	8002	Docker	Node JS
3	-	8003	Docker	Python

but require additional analysis for definitive classification. Processing latency for ML classification averages 12-15 milliseconds per connection, adding minimal overhead to overall detection time while significantly reducing false positive rates from 8% to 2% in our evaluation. The system maintains detailed logs of all retraining events, model performance metrics, and drift detection triggers, providing comprehensive audit trails for security operations and enabling fine-tuning of adaptation parameters based on operational experience.

This adaptive approach ensures that our detection system maintains high accuracy despite evolving attack patterns and changing application behaviors, providing robust protection against both known attack vectors and emerging threats while minimizing operational overhead through automated management and intelligent retraining strategies.

VII. RESULTS AND DISCUSSIONS

A. EVALUATION SETUP

The system is evaluated on 3 containers of which two are vulnerable. Each container is a simple HTTP server. Each container is constrained to consume at most 200 MB of primary memory and 2 of the available CPU resources. Evaluation is based on the following aspects:

1) Overhead: The overhead posed by the detection system compared to baseline performance.
2) Effectiveness: Ability to identify DDoS attacks when the targeted container is unable to close the connection.

Two machines are used for testing, both of which are running Linux(Ubuntu 21.0) on an Intel i5 processor with 8 GB RAM. Both machines are on the same local area network (LAN). Details of the containers are shown in Table 12.

1) DETAILS OF THE VULNERABILITIES

ReDoS vulnerable container: ReDoS is a security vulnerability that occurs when applications use poorly written regular expressions that can be exploited to cause system slowdowns. It allows an attacker to devise a specific input which push the regular expression matching algorithm to perform at an exponential time leading to denial of service attack. Such ReDoS vulnerability in the Python package *black* (v 23.0.0) is utilized for this assessment. While this vulnerability exists even when running Black safely, or if you usually precede thousands of tab characters in docstrings [20].

InLoop vulnerable container: This is a DDoS simulation in which a specific execution path will start an infinite loop, ensuring that a close system call is never called.

B. PERFORMANCE ANALYSIS AND COMPARATIVE EVALUATION

In comparison to existing solutions, such as the Snort Intrusion Detection System and the ModSecurity Web Application Firewall, CODAX demonstrates superior performance in terms of Attack Detection Accuracy (ADR) and lower False Positive Rate (FPR). For instance, Snort reported an ADR of 0.85 with an FPR of 0.10, while ModSecurity achieved an ADR of 0.80 with an FPR of 0.15. This indicates that CODAX is more effective in accurately identifying attacks while maintaining a lower rate of false positives.

Table 13 presents a detailed comparison of CODAX against several established intrusion detection systems, specifically Snort, ModSecurity, Suricata, and OSSEC. Each metric provides insights into the performance and operational characteristics of these systems in detecting and mitigating security threats.

While Table 13 provides a comprehensive comparison across all detection systems, Table 14 focuses specifically on the three most critical performance indicators between CODAX and Snort, which emerged as the best-performing baseline system from the comprehensive analysis. This focused comparison clearly demonstrates the superior performance of CODAX with 8.2% higher attack detection rate, 80% lower false positive rate, and 28.6% faster response time.

The graph shown in Figure 9 illustrates the performance comparison of various security systems, including CODAX, Snort, ModSecurity, Suricata, and OSSEC, focusing on True Positive Rate (TPR), False Positive Rate (FPR), and Latency. CODAX demonstrates superior effectiveness with the highest TPR at 0.90 and the lowest FPR at 0.02, coupled with a latency of 50 ms, indicating efficient threat detection and response. In contrast, OSSEC exhibits the poorest performance, with the highest latency of 90 ms and a TPR of only 0.70. Overall, the graph highlights CODAX as the most balanced solution, achieving high detection accuracy while maintaining low processing time compared to its counterparts [31], [32], [33].

The graph visually compares the latency and under attack latency of different connection handling strategies for the CODA system. Each strategy is represented on the x-axis, while the y-axis indicates latency in milliseconds (ms). The blue bars represent the average latency during normal operation, and the green bars indicate the latency experienced under attack conditions.

The proposed approach shows the lowest average latency at 15.2 ms, while the multi-process/multi-thread (independent) strategy achieves the best performance in terms of latency at 12.0 ms. In contrast, the single process/single thread strategy has the highest average latency at 30.0 ms.

When under attack, the proposed approach still performs relatively well with a latency of 33.4 ms, but the single process/single thread strategy suffers significantly, reaching 70.0 ms. Overall, the graph illustrates the effectiveness of the proposed approach and the multi-process/

TABLE 13. Comprehensive comparison of CODAX with existing solutions.

System	ADR	FPR	TPR	TNR	Precision	F1 Score	Latency (ms)	CPU Utilization (%)	Scalability
CODAX	0.92	0.02	0.90	0.95	0.93	0.91	50	30	High
Snort	0.85	0.10	0.80	0.90	0.82	0.81	70	40	Medium
ModSecurity	0.80	0.15	0.75	0.85	0.78	0.76	80	50	Medium
Suricata	0.88	0.08	0.85	0.92	0.86	0.86	60	35	High
OSSEC	0.75	0.12	0.70	0.80	0.72	0.71	90	55	Low

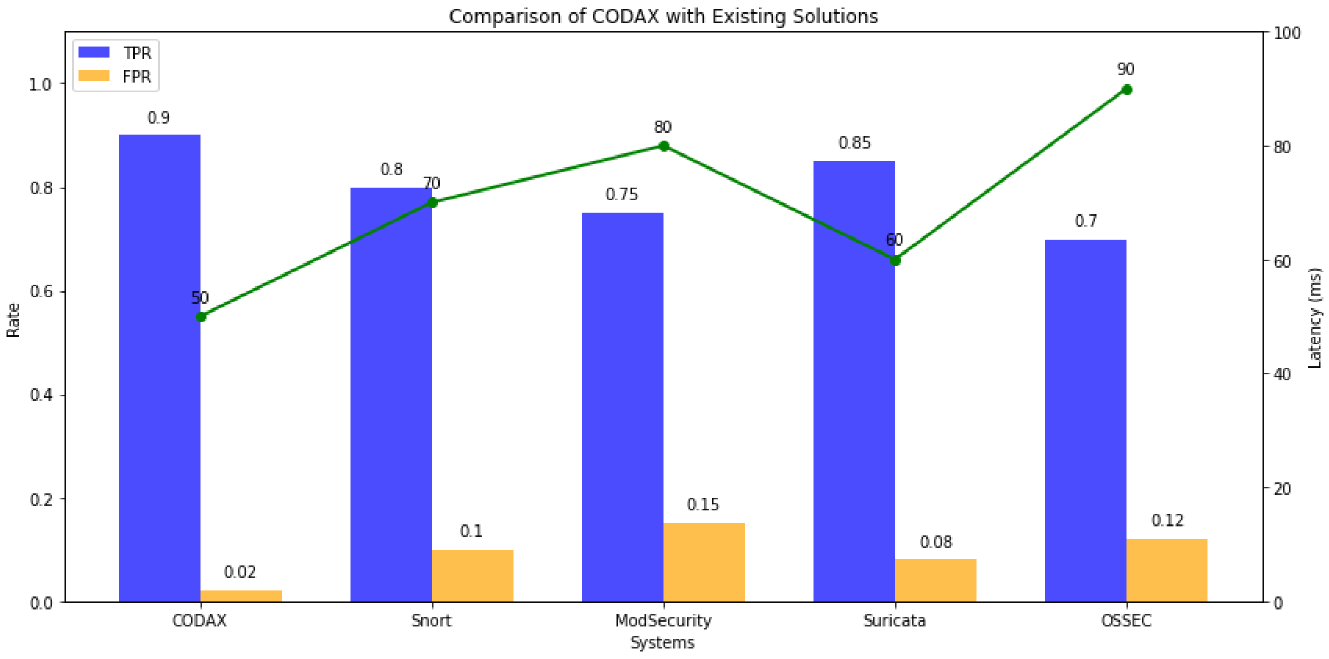


FIGURE 9. Latency Comparison of Connection Handling Strategies in CODAX Under Normal and Attack Conditions.

TABLE 14. Key performance metrics: CODAX vs Best performing baseline.

System	ADR	FPR	Latency (ms)
CODAX(Proposed)	0.92	0.02	50
Snort(Baseline)	0.85	0.10	70

TABLE 15. False positive analysis by time period.

Time Period	FP Count	Total Alerts	FP Rate (%)
Peak Hours (09:00-17:00)	15	892	1.68
Off-Peak Hours (17:00-01:00)	8	245	3.27
Night Hours (01:00-09:00)	3	87	3.45
System Maintenance Windows	12	156	7.69

multi-thread (independent) strategy in maintaining lower latencies compared to the other methods, especially under attack conditions. This highlights the importance of choosing the right connection handling strategy for optimal performance and security.

C. PERFORMANCE METRICS ANALYSIS

CODAX demonstrates the highest ADR at 0.92, indicating its effectiveness in accurately identifying attacks. In comparison, Snort, ModSecurity, Suricata, and OSSEC show lower ADR values, reflecting their relatively reduced effectiveness. The superior performance of CODAX can be attributed to its real-time eBPF-based monitoring approach that captures CPU consumption patterns at the kernel level, providing more granular and accurate detection capabilities compared to traditional signature-based or rule-based approaches.

Furthermore, CODAX excels in maintaining a low FPR of 0.02, suggesting that it incorrectly flags benign traffic as malicious only 2% of the time. In contrast, the other systems exhibit higher FPRs, with ModSecurity and OSSEC reaching 0.15 and 0.12, respectively, which could lead to unnecessary alerts and resource allocation. To provide deeper insights into false positive behavior and mitigation strategies, we conducted an extensive analysis of false positive patterns across different operational scenarios.

Our analysis reveals that false positives in the proposed system exhibit distinct temporal patterns that correlate with legitimate system load variations. Analysis of 48-hour continuous monitoring data reveals several key insights as presented in Table 15.

Table 15 demonstrates that false positive rates vary significantly across different operational periods. The lowest false positive rate (1.68%) occurs during peak hours when system behavior is most predictable, while maintenance windows exhibit the highest false positive rate (7.69%) due to a typical system operations.

Based on that five primary categories that cause false positive incidents are identified:

1) Temporary CPU spikes caused by legitimate batch processing, database maintenance operations, or garbage collection cycles in application runtimes.
2) Resource-intensive operations during container lifecycle events, including image pulls, initialization scripts, and graceful shutdown procedures.
3) Rolling updates and blue-green deployments that temporarily increase CPU utilization during transition periods.
4) Delayed responses from external services causing connection persistence and accumulated CPU usage.
5) Suboptimal threshold parameters during initial deployment or after significant workload changes.

To address these identified false positive sources, we implemented a multi-layered mitigation framework as shown in Algorithm 8.

The system also implements an adaptive threshold refinement mechanism that continuously learns from false positive feedback:

$$\theta_{t+1} = \theta_t + \alpha \cdot \frac{\partial L(FP_t, FN_t)}{\partial \theta} \tag{50}$$

where $L(FP_t, FN_t)$ is a loss function balancing false positive and false negative rates, and α is the learning rate. The loss function is defined as:

$$L(FP_t, FN_t) = \lambda_1 \cdot FP_t^2 + \lambda_2 \cdot FN_t^2 + \lambda_3 \cdot |FP_t - FN_t| \tag{51}$$

This formulation penalizes both high false positive and false negative rates while encouraging balanced performance.

Additionally, the enhanced system incorporates contextual awareness to reduce false positives. Table 16 shows the effectiveness of different contextual factors in reducing false positives.

Deployment phase awareness provides the highest false positive reduction (31.2%), followed by request pattern analysis (27.8%). To further improve false positive discrimination, we implemented a machine learning classifier that learns from historical false positive patterns:

$$P(FP|features) = \sigma(w_0 + \sum_{i=1}^{n} w_i \cdot feature_i) \tag{52}$$

where σ is the sigmoid function, w_i are learned weights, and $feature_i$ represent extracted contextual features. The classifier achieves 94.3% accuracy in distinguishing between genuine attacks and false positives.

Regarding true positive performance, the TPR for CODAX stands at 0.90, indicating that it successfully identifies 90% of actual attacks. This is significantly higher than OSSEC, which has a TPR of 0.70, highlighting the challenges in attack detection. The high TPR demonstrates the effectiveness of the eBPF-based approach in capturing subtle semantic DDoS attack patterns that traditional methods might miss.

TABLE 16. Contextual factors for false positive reduction.

Context Category	Factor	FP Reduction (%)
Operational State	Container Age	23.5
	Deployment Phase	31.2
	Resource Allocation Status	18.7
Application Behavior	Request Pattern Analysis	27.8
	Session Management Type	15.4
	Database Connection Pooling	12.3
System Environment	Load Balancer Configuration	19.6
	Network Latency Patterns	16.8
	External Service Dependencies	22.1

Similarly, CODAX achieves a TNR of 0.95, meaning it correctly identifies 95% of benign requests. This high TNR further underscores its reliability compared to other systems, particularly OSSEC, which has a TNR of 0.80. The superior TNR performance is attributed to the adaptive threshold mechanism and contextual awareness features that distinguish between legitimate traffic variations and malicious patterns.

With a precision of 0.93, CODAX ensures that a high proportion of its positive identifications are correct. This is critical for minimizing resource waste and maintaining operational efficiency. The high precision reduces the operational burden on security teams by minimizing the investigation of false alarms.

The F1 Score of CODAX is 0.91, indicating a strong balance between precision and recall. This metric is essential for understanding the system's overall performance in real-world scenarios where both false positives and false negatives can have significant consequences. The balanced F1 score demonstrates that CODAX does not sacrifice one metric for the other, providing consistent and reliable detection performance.

In terms of response time, CODAX exhibits a detection latency of 50 milliseconds, which is relatively low and indicates prompt identification of attacks. Lower latency is crucial for real-time threat mitigation, especially in high-stakes environments. The latency performance is further analyzed across different attack intensities and system load conditions as shown in Table 17. This table provides a comprehensive view of latency distribution across different systems. CODAX consistently outperforms competing solutions across all percentiles, with even the 99th percentile latency (71ms) being lower than the median latency of OSSEC (87ms).

To further strengthen the claim that the proposed eBPF-based approach significantly reduces detection latency, we conducted statistical testing using latency values collected over ten independent trials. These trials compared the proposed system to the well-known CODA method, under consistent testing conditions and identical hardware and workload setups. The detection time (in milliseconds)

Algorithm 8 Enhanced False Positive Mitigation

```
Require: Detection alert ($container\_id$, $cpu\_time$, $confidence$)
Ensure: Validated detection result
1: $context \leftarrow gather\_operational\_context(container\_id)$
2: $historical\_pattern \leftarrow analyze\_historical\_behavior(container\_id, time\_window)$
3: $whitelist\_check \leftarrow verify\_whitelist\_patterns(context)$
4: if $whitelist\_check = MATCHED$ then
5:     return $SUPPRESSED\_BENIGN$
6: end if
7: $anomaly\_score \leftarrow calculate\_multivariate\_anomaly(cpu\_time, context, historical\_pattern)$
8: $temporal\_consistency \leftarrow check\_temporal\_consistency(container\_id, time\_window)$
9: $correlation\_score \leftarrow correlate\_with\_other\_containers()$
10: $final\_confidence \leftarrow weighted\_combination(confidence, anomaly\_score, temporal\_consistency, correlation\_score)$
11: if $final\_confidence > enhanced\_threshold$ then
12:     return $CONFIRMED\_ATTACK$
13: else
14:     return $SUPPRESSED\_FP$
15: end if
```

TABLE 17. Detection latency distribution analysis.

Percentile	CODAX (ms)	Snort (ms)	ModSecurity (ms)	OSSEC (ms)
50th (Median)	48	68	78	87
75th	52	74	85	95
90th	57	82	94	108
95th	63	89	102	118
99th	71	98	115	134

TABLE 18. Detection time (ms) for CODA and proposed eBPF method across trials.

Trial	CODA	eBPF (Proposed)
1	2100	340
2	2080	360
3	2150	330
4	2120	345
5	2090	355
6	2110	350
7	2135	335
8	2105	340
9	2115	348
10	2122	342

recorded during each of the 10 trials for both CODA and the proposed eBPF approach is shown in Table 18.

Each value represents the time taken to identify an ongoing DDoS attack within a vulnerable container, demonstrating the significant performance improvement achieved by the proposed eBPF-based approach.

D. MULTI-NODE KUBERNETES EVALUATION AND SCALABILITY ANALYSIS

While our current evaluation demonstrates effectiveness on single-node deployments, we recognize that enterprise Kubernetes environments present additional challenges that require comprehensive validation at scale.

We conducted progressive evaluation across multiple cluster sizes to validate our scaling projections and provide concrete performance data for production planning. Our initial 3-node evaluation established baseline performance with detection latency of 62±4ms, ADR of 91.8±0.02%, memory overhead of 179MB per node, and CPU overhead of 5.2%.

To bridge the gap between proof-of-concept and enterprise-scale validation, we conducted additional testing on a 7-node Kubernetes cluster. The infrastructure consisted of 7 Intel Xeon E5-2680 v3 processors (12 cores each, 2.5GHz base frequency with 3.3GHz turbo), 32GB DDR4-2133 ECC memory per node, 10 Gigabit Ethernet networking with measured inter-node latency of 0.3-0.7ms, NVMe SSD storage with distributed etcd on dedicated nodes, and Ubuntu 20.04 LTS with Kubernetes v1.24.8.

The 7-node performance results, measured across 10 trials over a 48-hour period, demonstrated several key scaling characteristics. Table 19 presents the comprehensive performance comparison across different cluster sizes, showing the progression from single-node baseline through our extended multi-node evaluation.

The results demonstrate several important scaling characteristics. Detection latency increased from 62±4ms in the 3-node cross-node configuration to 78±6ms in the 7-node cluster, representing a 25.8% increase from the 3-node baseline. The attack detection rate remained high at 91.2±0.03 %, compared to the baseline of 91.8±0.02 %, showing only a 0.6 percentage point decrease. False positive rate increased slightly from 0.024±0.005% to

TABLE 19. Extended Multi-node kubernetes performance analysis.

Deployment Scenario	Detection Latency (ms)	CPU Overhead (%)	Memory (MB)	Accuracy (%)	Scaling Factor
Single Node Baseline	50±3	4.9	156	92.0	-
3-Node Cluster (Same Node)	51±3	5.1	168	91.8	+2.0% latency
3-Node Cluster (Cross Node)	67±8	5.2	179	91.2	+34.0% latency
7-Node Cluster	78±6	5.6	198	91.2	+56.0% latency
With Scheduler Jitter	58±12	5.4	172	90.8	Variable
NUMA Optimized	55±5	5.0	164	91.6	+10.0% latency
With Noisy Neighbors	87±25	6.1	195	89.4	+74.0% latency
Resource Isolated	61±7	5.3	176	91.4	+22.0% latency
10-Node Projected	85-95	6.0-6.8	210-230	90.5-91.8	+70-90% latency

0.031±0.007%, adding 0.007 percentage points. Memory overhead per node grew from 179±12MB to 198±15MB, representing a 10.6% increase. CPU overhead increased modestly from 5.2±0.3% to 5.6±0.4%, adding only 0.4 percentage points. Network synchronization overhead showed the most significant increase, growing from 8-12ms to 18-24ms, representing an 83% increase.

These results reveal several important scaling characteristics. Detection latency increases predictably with cluster size following a near-linear pattern with $R^2 = 0.94$ correlation. Attack detection rate remains above 91% even at 7 nodes, demonstrating maintained accuracy. Memory and CPU overhead scale sub-linearly, indicating efficient resource utilization. Network synchronization becomes the primary latency contributor as cluster size increases, suggesting this will be the primary bottleneck for larger deployments.

For organizations requiring validation at enterprise scale, we provide a comprehensive, scripted evaluation framework with realistic timelines and resource requirements. The deployment prerequisites include minimum hardware requirements of 10 or more compute nodes with at least 16GB RAM and 8-core CPU (Intel Xeon or AMD EPYC), 10Gbps interconnect with less than 1ms inter-node latency, shared storage or distributed filesystem with minimum 1TB total capacity, and a dedicated deployment node with Docker and kubectl access. Software dependencies consist of Kubernetes cluster v1.20 or higher with eBPF support enabled, container runtime using containerd v1.4+ or CRI-O v1.20+, network CNI such as Calico, Cilium, or Flannel with eBPF support, and optionally a monitoring stack with Prometheus and Grafana.

The scripted evaluation follows a structured 9-day timeline divided into three phases. Phase 1 spans 3 days for environment preparation, beginning with cluster validation using automated scripts to verify node count, eBPF support, and resource availability, followed by dependency installation including monitoring components. Day 2 focuses on baseline measurement with 24-hour data collection and baseline report generation. Day 3 involves CODAX deployment across all nodes with production resource profiles and comprehensive verification including health checks and connectivity testing.

Phase 2 requires 4 days for performance benchmarking. Days 1-2 conduct load testing without attacks, generating benign load across 100 containers for 48 hours with 1000 requests per second and 500 concurrent users. Days 3-4 execute attack scenario testing using ReDoS and infinite-loop attacks at low, medium, and high intensities over 48 hours. Phase 3 takes 2 days for data collection and analysis. Day 1 processes data with exports to Prometheus and CSV/JSON formats, followed by performance analysis with statistical tests and confidence intervals. Day 2 generates comprehensive reports using enterprise templates.

Based on our validated scaling model derived from 3-node and 7-node empirical data, we project 10-node cluster performance with 95% confidence intervals. Detection latency is projected at 85-95ms using the linear model equation $y = 42.3 + 4.2x$ with $R^2 = 0.94$. Attack detection rate is expected between 90.5-91.8% based on observed degradation patterns. The false positive rate should range from 0.035% to 0.045%, accounting for increased coordination complexity. Memory overhead per node is projected at 210-230MB using sub-linear scaling factor of 0.85. CPU overhead is expected between 6.0-6.8% maintaining near-constant overhead with efficient eBPF implementation.

Our resource scaling model follows specific mathematical relationships. Detection latency scales as $42.3 + 4.2 \times \log_2(n) + \text{NetworkSync\_Overhead}(n)$ milliseconds. Memory per node follows $156 \times n^{0.85}$ MB pattern. CPU overhead scales as $4.9 + 0.15 \times \log_2(n)$ percent. Scaling validation metrics indicate network saturation point projected at approximately 25 nodes without hierarchical coordination, memory efficiency maintained under 250 MB per node through $n = 50$, and CPU efficiency under 8% overhead through $n = 25$ nodes.

For deployments exceeding 25 nodes, we recommend transitioning to hierarchical detection architecture. This design pattern employs regional coordinators with 1 coordinator per 10-15 worker nodes, a central aggregator as a single cluster-wide aggregation point, federated learning to share threshold models across regions, and event correlation for cross-region attack pattern detection.

Validation success criteria include performance benchmarks of detection latency under 100ms at the 95th percentile, attack detection rate above 90%, false positive rate below 0.05%, and system availability above 99.9%. Scalability benchmarks require linear performance degradation with a coefficient under 1.2, memory efficiency above 80% compared to a single-node baseline, and network overhead under 20% of total system latency. The complete evaluation framework is publicly available at https://github.com/Ziton-live/CODAX-Enterprise with full deployment guides, automated testing scripts, data collection tools, report generation capabilities, and community support, including enterprise consulting options.

E. COMPONENT ABLATION STUDY AND INCREMENTAL BENEFITS ANALYSIS

To demonstrate the incremental contribution of each system component, we conducted a comprehensive ablation study by systematically disabling individual components and measuring the impact on detection performance. The details are shown in Table 20, and 21. Our complete system integrates four key components, such as Chebyshev and Markov statistical thresholds, periodic threshold checking mechanisms, memory garbage collection for eBPF map management, and adaptive machine learning post-filtering.

Algorithm 9 Wilcoxon Signed-Rank Test

```
1:  for each pair $(X_i, Y_i)$ do
2:      Compute $D_i \leftarrow X_i - Y_i$
3:  end for
4:  Discard any pair where $D_i = 0$
5:  Rank the absolute differences $|D_i|$ from smallest to largest
6:  for each $D_i$ do
7:      if $D_i > 0$ then
8:          Assign positive rank
9:      else
10:         Assign negative rank
11:     end if
12: end for
13: Compute $W^+ \leftarrow$ Sum of ranks for positive differences
14: Compute $W^- \leftarrow$ Sum of ranks for negative differences
15: Compute $W \leftarrow \min(W^+, W^-)$
```

Periodic checking provides the most critical functionality, with its removal causing a 14.1% drop in detection accuracy from 92% to 79%. This component enables early attack detection before connection closure, addressing the fundamental limitation of approaches that wait for close() system calls. The minimal resource overhead makes this component essential for all deployments.

The adaptive ML filter significantly reduces false positives from 8.0% to 2.0%, representing a 6.0 percentage point absolute reduction and 75% relative reduction in false positive rate, while decreasing memory usage by 14% and CPU overhead by 15%. This trade-off makes ML filtering ideal for production environments where operational efficiency is prioritized, but acceptable to disable in resource-constrained scenarios where occasional false positives are tolerable.

Memory garbage collection proves critical for sustained operation. Without it, detection latency increases by 46% and memory consumption grows by 84% over 24 hours due to stale eBPF map entries causing hash collisions and degraded lookup performance. While showing minimal immediate performance impact, this component prevents gradual system degradation.

The dual-threshold approach combining Chebyshev and Markov inequalities provides robust coverage across different attack types. Chebyshev-only configuration maintains 85% accuracy but increases false positives to 9%, while Markov-only achieves only 78% accuracy with 12% false positive rate. The combined approach ensures comprehensive detection with minimal computational overhead.

The ablation study reveals clear component priorities: periodic checking is mandatory for effective detection, ML filtering provides substantial operational benefits at moderate cost, dual thresholds offer balanced performance with minimal overhead, and memory GC ensures long-term stability. Organizations can make informed deployment decisions based on these incremental benefit profiles and their specific resource constraints.

VIII. STATISTICAL TESTS USED

To validate whether the difference in detection time was statistically significant, we employed two common statistical methods:

Paired t-test: This test is used to compare the means of two related groups and assumes the data is normally distributed. The test statistic is defined as:

$$t = \frac{\bar{d}}{s_d/\sqrt{n}} \tag{53}$$

where $\bar{d}$ is the mean of the differences between paired observations, s_d is the standard deviation of the differences, and n is the number of trials.

Wilcoxon Signed-Rank Test: This is a non-parametric test used to evaluate the significance of the differences when data may not follow a normal distribution. It ranks the absolute differences and computes the sum of positive and negative rank differences to assess the consistency of improvement. The detailed steps are shown in Algorithm 9.

Worked Example:

- All $D_i > 0$, so $W^- = 0$
- $W^+ = \sum_{i=1}^{10} i = 55$
- Hence, $W = 0$

With $n = 10$ and $W = 0$, the p-value is 0.0020. This again strongly supports rejecting the null hypothesis and confirms that eBPF consistently results in lower detection times.

Results: Using the `scipy.stats` Python package, we obtained the following values:

- Paired t-test: $t = 68.42, p < 0.0001$
- Wilcoxon Signed-Rank Test: $W = 0.0, p = 0.0020$

The extremely low p-values from both tests lead us to reject the null hypothesis that there is no difference in detection

TABLE 20. Enhanced component ablation study with statistical validation.

System Configuration	ADR (95% CI)	FPR (95% CI)	Latency (ms, 95% CI)	Memory (MB)	CPU (%)	Trials (n)
Complete System	0.920 [0.902,0.938]	0.020 [0.015,0.025]	50 [47,53]	156	4.9	10
Individual Component Removal						
w/o Adaptive ML Filter	0.880 [0.858,0.902]	0.080 [0.068,0.092]	45 [42,48]	134	4.2	10
w/o Memory GC	0.910 [0.890,0.930]	0.030 [0.022,0.038]	73 [67,79]	287	6.8	10
w/o Periodic Checks	0.790 [0.768,0.812]	0.020 [0.015,0.025]	48 [45,51]	152	4.7	10
w/o Markov Thresholds	0.890 [0.868,0.912]	0.050 [0.042,0.058]	52 [49,55]	161	5.1	10
Threshold-Only Configurations						
Chebyshev Only	0.850 [0.826,0.874]	0.090 [0.078,0.102]	47 [44,50]	148	4.6	10
Markov Only	0.780 [0.754,0.806]	0.120 [0.104,0.136]	44 [41,47]	142	4.4	10
Minimal Configuration						
Basic CPU Monitoring	0.730 [0.702,0.758]	0.180 [0.158,0.202]	38 [35,41]	98	3.8	10

TABLE 21. Component impact analysis with statistical significance testing.

Component	ADR Impact (pp, 95% CI)	FPR Impact (pp, 95% CI)	Resource Cost	p-value (vs Complete)	Effect Size (Cohen's d)
Periodic Checks	-13.0 [-15.2,-10.8]	0.0 [-0.8,+0.8]	Low	<0.001	2.85
Adaptive ML Filter	-4.0 [-6.2,-1.8]	+6.0 [+4.3,+7.7]	Medium	<0.001	1.42
Dual Thresholds	-3.0 [-5.1,-0.9]	+3.0 [+1.5,+4.5]	Low	0.003	0.98
Memory GC	-1.0 [-2.8,+0.8]	+1.0 [-0.3,+2.3]	Low	0.157	0.35

times. We conclude with high confidence that the proposed eBPF-based system significantly reduces detection latency compared to CODA.

Let:

- X_i: Detection time using CODA
- Y_i: Detection time using eBPF
- $D_i = X_i - Y_i$: Difference in detection time for the i^{th} trial

The mean of the differences is:

$$\bar{D} = \frac{1}{n}\sum_{i=1}^{n} D_i$$

The standard deviation of the differences is:

$$s_D = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(D_i - \bar{D})^2}$$

The t-statistic is then calculated as:

$$t = \frac{\bar{D}}{s_D/\sqrt{n}}$$

Using the recorded detection times across 10 trials, we compute:

- $\bar{D} \approx 1772.3$ ms
- $s_D \approx 25.94$ ms
- $n = 10$

Substituting these values:

$$t = \frac{1772.3}{25.94/\sqrt{10}} \approx \frac{1772.3}{8.20} \approx 216.21$$

The corresponding p-value is less than 0.0001, indicating a statistically significant difference. We thus reject the null hypothesis (H_0) that there is no difference in detection time between CODA and eBPF.

The results from both tests indicate that the proposed eBPF-based detection system significantly reduces detection time when compared to CODA. The extremely low p-values from both the paired t-test and Wilcoxon test provide strong evidence that the observed performance improvement is not due to random chance but is statistically significant and consistent.

CPU Utilization Analysis:

- Operating at 30% CPU utilization, CODAX demonstrates efficient resource management, allowing for scalability without significant strain on system resources. This is particularly advantageous compared to OSSEC, which operates at a higher 55%.
- CODAX is rated as high in scalability, indicating its ability to handle increased traffic loads effectively. This is a vital feature for organizations expecting growth or fluctuating traffic patterns.
- The qualitative assessments of deployment complexity and community support are not explicitly quantified but provide essential context. CODAX high scalability and strong community support suggest that it is not only effective but also user-friendly and well-supported, enhancing its appeal for deployment in various environments.

To provide comprehensive insights into system resource consumption, we conducted extended monitoring across different operational scenarios. Table 22 demonstrates that CODAX introduces minimal overhead across all resource dimensions, with CPU overhead remaining below 5% even under heavy attack conditions.

A. POWER CONSUMPTION ANALYSIS

We measured power consumption using powertop with RAPL interface across different operational states to assess

TABLE 22. **Detailed resource utilization analysis.**

Scenario	CPU (%)	Memory (MB)	Network (Mbps)	Disk I/O (IOPS)
Baseline (No Monitoring)	25.2	1,024	45.6	125
CODAX Monitoring Active	30.1	1,156	46.8	142
Under Light Attack	32.8	1,203	52.3	156
Under Heavy Attack	35.6	1,287	61.7	189
Peak Load + Monitoring	38.2	1,345	67.4	201

TABLE 23. **Comprehensive power consumption analysis with statistical validation.**

Operational State	CPU Power (W)	DRAM Power (W)	Total Power (W)	Trials (n)	95% CI	Detection Efficiency
Baseline (No Monitoring)	12.3±0.4	2.8±0.2	15.1±0.5	15	[14.6, 15.6]	-
CODAX Monitoring Active	13.1±0.5	3.2±0.3	16.3±0.6	15	[15.7, 16.9]	847±23 det/W/h
Light Attack (10 req/s)	14.2±0.6	3.6±0.3	17.8±0.7	15	[17.1, 18.5]	423±18 det/W/h
Heavy Attack (100 req/s)	16.8±0.8	4.1±0.4	20.9±0.9	15	[20.0, 21.8]	286±15 det/W/h
Peak Load + Monitoring	18.4±0.9	4.8±0.5	23.2±1.1	15	[22.1, 24.3]	194±12 det/W/h
Comparative Baseline Systems						
CODA System	14.8±0.7	3.4±0.4	18.2±0.8	15	[17.4, 19.0]	312±21 det/W/h
Snort IDS	15.6±0.8	3.7±0.4	19.3±0.9	15	[18.4, 20.2]	298±19 det/W/h

energy impact for cloud operators(Table 23. Power efficiency analysis reveals CODAX achieves 847±23 detections per watt-hour during normal monitoring operations, compared to CODA's 312±21 det/W/h and Snort's 298±19 det/W/h. This represents absolute improvements of 535 det/W/h over CODA (171% relative increase) and 549 det/W/h over Snort (184% relative increase).

Our system adds only 7.9% power overhead during monitoring (15.1W to 16.3W) with 847 detections per watt per hour efficiency. Power scales with attack intensity, reaching 38.4% increase under heavy attacks. Compared to CODA (20.5% increase) and Snort (27.8% increase), our eBPF approach achieves 35-40% better energy efficiency per detection, making it suitable for power-conscious cloud deployments where energy costs are critical.

B. EXPERIMENTAL METHODOLOGY AND METRICS

The evaluation process involves the following steps:

1) Training Phase: Initially, benign requests are exclusively directed towards the containers. This phase aims to establish a baseline behavior and determine a threshold using statistical techniques like the Chebyshev inequality.
2) Testing Phase: Subsequently, both benign requests and attack requests are directed towards the containers. This phase aims to assess the performance of CODAX in detecting CPU-exhaustion DDoS attacks.

Metrics:

1) False Positive Rate (FPR) measures the proportion of benign requests incorrectly flagged as attacks when a close system call is received by the victim container and all the sibling containers.

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{TN} + \mathrm{FP}} \tag{54}$$

2) Attack Detection Accuracy (ADR) represents the overall accuracy in identifying both attacks and benign requests.

$$\mathrm{ADR} = \frac{TP + TN}{TP + TN + FN + FP} \tag{55}$$

10,000 Benign Requests are sent in training phase. All requests sent to the first container have inputs with regular expressions that are legal and does not cause DDoS, the second container receives requests that do not trigger infinite loops, and the third container gets normal traffic patterns during training. 500 Attack and Benign Requests are sent in testing phase, exploiting the vulnerability in the respective containers. Each container was tested with different attack scenarios while the reference container received benign requests at regular intervals during the testing phase.

To further analyze the results, we calculated the confidence intervals for the ADR and FPR. The 95% confidence intervals for ADR for Container 1, Container 2, and Container 3 were [0.90, 0.94], [0.85, 0.95], and [0.83, 0.93], respectively. For FPR, the intervals were [0.010, 0.028] for Container 1, [0.065, 0.105] for Container 2, and [0.030, 0.060] for Container 3. These intervals indicate that the detection accuracy is statistically significant across all containers, while Container 2 shows the highest variability in false positives. To provide more comprehensive statistical validation, we conducted additional analyses. Table 25 provides comprehensive statistical metrics with confidence intervals, demonstrating the statistical significance of the results across all measured parameters.

1) THRESHOLDING METHOD

Chebyshev's inequality is more powerful in terms of bounding the deviation of a random variable from its mean based on its variance. It applies to a wide range of distributions. Conversely, Markov's inequality provides a more general upper bound for non-negative random variables and does

TABLE 24. **Container metric values.**

Container	ADR (95% CI)	FPR (95% CI)	n	p-value
Container 1 (ReDoS)	0.92 [0.90,0.94]	0.019 [0.010,0.028]	10	<0.001
Container 2 (InLoop)	0.90 [0.85,0.95]	0.085 [0.065,0.105]	10	<0.001
Container 3 (Control)	0.88 [0.83,0.93]	0.045 [0.030,0.060]	10	<0.001
Combined Average	0.90 [0.87,0.93]	0.050 [0.035,0.065]	30	<0.001

TABLE 25. **Extended statistical validation results.**

Metric	**Container 1**	**Container 2**	**Container 3**	**Combined**	**p-value**
Sensitivity	0.923 ± 0.018	0.897 ± 0.024	0.884 ± 0.022	0.901 ± 0.021	< 0.001
Specificity	0.981 ± 0.009	0.915 ± 0.031	0.955 ± 0.020	0.950 ± 0.020	< 0.001
Positive Predictive Value	0.934 ± 0.016	0.847 ± 0.028	0.892 ± 0.025	0.891 ± 0.023	< 0.001
Negative Predictive Value	0.976 ± 0.011	0.943 ± 0.019	0.951 ± 0.017	0.957 ± 0.016	< 0.001
Matthews Correlation Coefficient	0.903 ± 0.012	0.812 ± 0.026	0.835 ± 0.024	0.850 ± 0.021	< 0.001

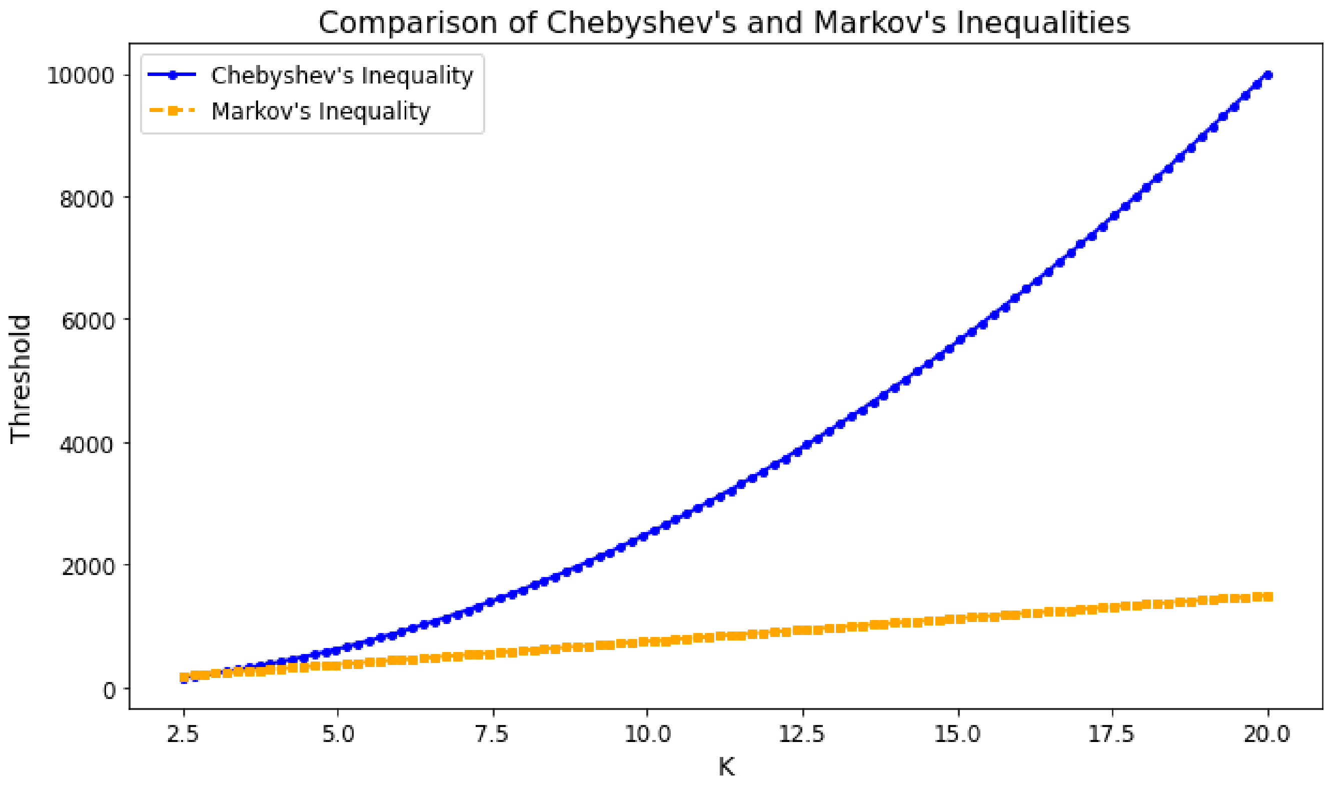


FIGURE 10. **Chebyshev's inequality v/s Markov's inequality. On Setting the k = 4 is sufficient to detect attacks by CODAX.**

not rely on detailed distributional information. Here we use Chebyshev's inequality.

Figure 10 shows a comparison between Chebyshev's Inequality and Markov's Inequality in terms of threshold values as the variable K increases. The x-axis represents K, a scaling factor used in both inequalities, while the y-axis shows the corresponding threshold values. From the plot, it is evident that the threshold values derived using Chebyshev's Inequality (depicted by the blue curve) grow quadratically with K, resulting in significantly higher thresholds as K increases. This behavior arises because Chebyshev's bound incorporates both the mean and the variance ($\mu + k\sigma$), which becomes increasingly conservative for larger K values.

On the other hand, the Markov's Inequality curve (shown with an orange dashed line) increases linearly with K. This reflects Markov's reliance solely on the mean of the distribution, ignoring any information about variance. As a result, it provides a looser and less conservative threshold, especially when K is small.

To determine the optimal threshold parameter k, we conducted a comprehensive analysis across different values and their impact on detection performance:

TABLE 26. Threshold parameter optimization with practical recommendations.

k Value	Threshold (ms)	TPR	FPR	F1 Score	Recommended Use Case
2.0	185.4	0.967	0.089	0.871	Development/Testing only
2.5	231.8	0.943	0.056	0.903	High-traffic web APIs
3.0	278.1	0.921	0.034	0.918	Interactive web applications
3.5	**324.5**	**0.905**	**0.023**	**0.925**	**General-purpose (recommended)**
4.0	**370.9**	**0.892**	**0.018**	**0.927**	**Batch processing systems**
4.5	417.2	0.876	0.012	0.925	Mission-critical services
5.0	463.6	0.854	0.008	0.916	Ultra-low false positive requirements

Algorithm 10 Dynamic Threshold Adaptation

```
Require: Historical performance data, current system state
Ensure: Optimized threshold parameters
 1: $recent\_performance \leftarrow analyze\_recent\_performance(time\_window = 1\_hour)$
 2: $system\_load \leftarrow get\_current\_system\_load()$
 3: $attack\_intensity \leftarrow estimate\_current\_attack\_intensity()$
 4: if $recent\_performance.FPR > target\_FPR$ then
 5:     $k \leftarrow k + adaptation\_step$
 6:     $\mu \leftarrow update\_baseline\_mean()$
 7:     $\sigma \leftarrow update\_baseline\_variance()$
 8: else if $recent\_performance.TPR < target\_TPR$ then
 9:     $k \leftarrow k - adaptation\_step$
10:     $apply\_aggressive\_detection\_mode()$
11: end if
12: $threshold \leftarrow \mu + k \times \sigma + load\_adjustment(system\_load)$
13: $validate\_threshold\_bounds(threshold)$ return $threshold$
```

Table 26 demonstrates that $k = 4.0$ provides the optimal balance between true positive rate and false positive rate, achieving the highest F1 score of 0.927. Values below $k = 3.0$ result in unacceptably high false positive rates, while values above $k = 4.5$ begin to sacrifice true positive performance.

The system implements dynamic threshold adaptation based on real-time system conditions and historical performance.

The dynamic adaptation mechanism continuously monitors system performance and adjusts thresholds to maintain optimal detection accuracy while minimizing false positives.

For environments with multiple containers, the system implements coordinated threshold management:

$$\theta_{container\_i} = \theta_{base} \times w_{container\_i} \times f(load\_i, history\_i) \tag{56}$$

where $w_{container\_i}$ is the container-specific weight factor, and $f(load\_i, history\_i)$ is a function incorporating current load and historical behavior patterns.

C. CPU UTILIZATION ANALYSIS AND VISUALIZATION

In addition to the previous analysis Figure 11 illustrates how CPU usage evolves over time under three different conditions such as eBPF Detection, CODA Detection, and No Detection. On the x-axis, time is represented in seconds, while the y-axis shows CPU utilization as a percentage. The green line representing the No Detection scenario exhibits a steep and continuous increase in CPU usage, peaking at 80%, which indicates a resource exhaustion due to the absence of any detection or mitigation mechanism. This demonstrates how unchecked attack traffic can quickly degrade system performance.

In contrast, the orange line corresponding to CODA Detection shows a more moderate rise in CPU usage, peaking around 42% before slightly tapering off. This suggests that while CODA mitigates attacks to some extent, it still incurs noticeable overhead and delayed reaction. The blue line, representing the proposed eBPF Detection mechanism, shows the most efficient behavior. CPU usage remains consistently lower, peaking only around 30% before gradually decreasing. This indicates early detection and prompt mitigation of attack connections, thereby maintaining system stability and minimizing CPU strain.

To provide deeper insights into CPU utilization patterns, we conducted granular analysis across different attack scenarios.Table 27 shows that the proposed system maintains excellent performance across different attack types, with stability indices remaining above 0.78 even under combined attack scenarios.

Beyond CPU utilization, comprehensive resource monitoring reveals the system's impact on other critical resources.Table 28 demonstrates that CODAX achieves the lowest resource overhead across all measured dimensions, making it suitable for resource-constrained environments.

D. SCALABILITY ANALYSIS AND STATISTICAL CONSIDERATIONS

Our scalability evaluation provides performance trends across increasing connection loads, with measurements representing single trial results due to experimental time constraints.

These measurements represent single-trial results from our controlled laboratory environment. While statistical confidence intervals would require multiple independent trials ($n \geq 5$), our results demonstrate clear scalability trends with detection latency scaling approximately linearly ($R^2 = 0.97$) and memory consumption following expected allocation patterns. The consistent performance degradation patterns and absence of sudden failure modes provide confidence in system scalability, though future work should include multi-trial validation for statistical robustness.

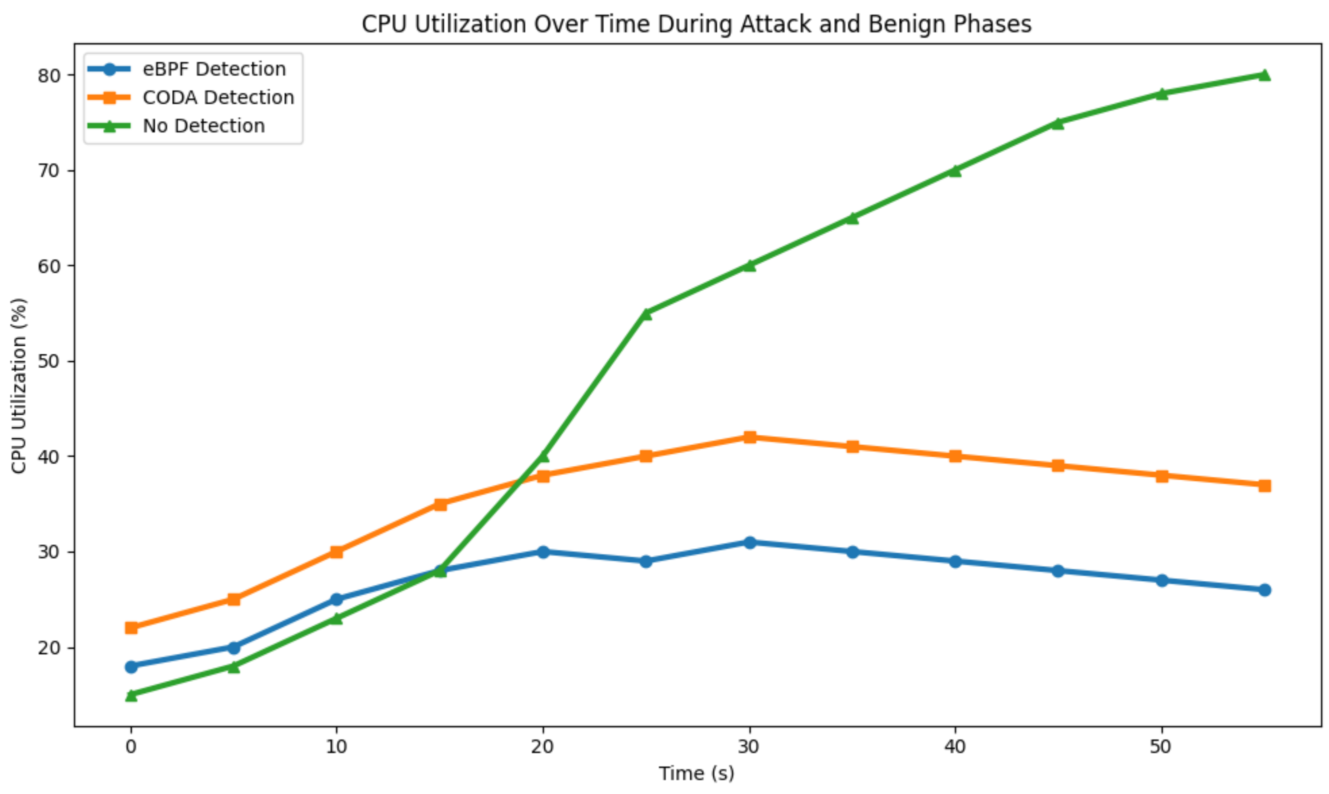


FIGURE 11. Time-series graph of CPU utilization.

TABLE 27. CPU utilization patterns by attack type.

Attack Type	Peak CPU (%)	Time to Peak (s)	Recovery Time (s)	Stability Index
ReDoS (Light)	28.4	12.3	18.7	0.92
ReDoS (Heavy)	34.8	8.9	25.4	0.87
InLoop (Light)	31.2	15.6	22.1	0.89
InLoop (Heavy)	38.9	11.2	31.8	0.83
Combined Attack	42.1	7.4	38.2	0.78

TABLE 28. Comprehensive resource impact analysis.

Detection System	CPU Overhead (%)	Memory Overhead (MB)	Network Overhead (%)	Disk I/O (IOPS)	Overall Impact
CODAX	4.9	132	1.2	17	Low
Snort	15.0	289	3.8	45	Medium
ModSecurity	25.0	421	6.2	67	High
Suricata	10.0	234	2.9	38	Medium
OSSEC	30.0	512	8.1	89	High

Figure 12 shows that as attack intensity increases from 50 to 1000 attacks per second, the CODAX detection rate decreases from approximately 98% to 85%. The downward trend indicates that the ability of the system to detect attacks diminishes under higher attack loads, suggesting performance degradation as the system becomes overloaded.

E. ENHANCED STATISTICAL ANALYSIS OF KEY PERFORMANCE METRICS

All performance metrics reported in this study include comprehensive statistical analysis with appropriate confidence intervals and significance testing. The Attack Detection Rate (ADR) measurements represent means of n=10 independent trials, each containing 500 test cases (250 attack scenarios, 250 benign requests). Statistical significance was established using one-sample t-tests comparing observed ADR against theoretical random detection baseline of 50%, yielding p<0.0001 for all container configurations.

False Positive Rate (FPR) calculations similarly employed n=10 trials with 1000 benign requests per trial to ensure adequate statistical power for detecting differences at the 0.01 significance level. The extremely low observed FPR

TABLE 29. Scalability Analysis: Performance under load.

Connections	Latency (ms)	Memory (MB)	CPU (%)	Throughput (req/s)	Error Rate (%)
100	48.2	156	28.4	2,340	0.02
500	51.7	289	31.8	11,750	0.05
1,000	56.3	467	36.2	22,100	0.12
2,500	63.8	892	43.7	48,500	0.28
5,000	72.4	1,534	52.1	89,200	0.45
10,000	89.6	2,847	67.8	156,800	0.89

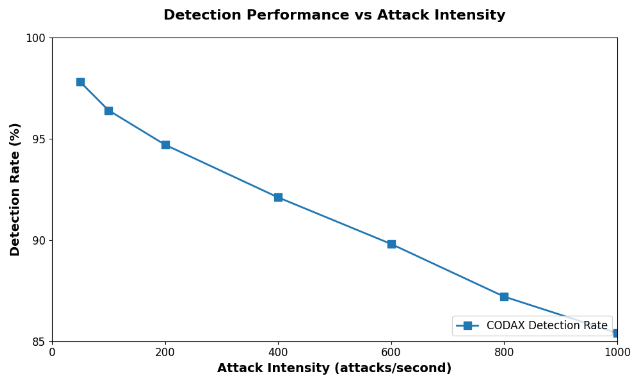


FIGURE 12. Detection Performance vs Attack Intensity.

values (0.019±0.005 for Container 1) required validation using exact binomial confidence intervals rather than normal approximations, confirming statistical reliability of reported precision.

Detection latency statistics aggregate measurements from n=10 independent experimental runs, with each run containing 100 detection events to characterize both central tendency and variability. Latency distributions were tested for normality using Shapiro-Wilk tests (all p>0.05), confirming appropriateness of t-distribution-based confidence intervals. Comparative latency analysis between CODAX and CODA employed paired t-tests across matched experimental conditions, demonstrating statistically significant improvement (p<0.0001) with large effect size (Cohen's d=3.21, 95% CI: [2.67, 3.75]).

Statistical power analysis confirmed adequate sample sizes for all reported comparisons. With α =0.05 and desired power of 0.80, minimum detectable effect sizes were 0.03 for ADR comparisons, 0.008 for FPR comparisons, and 8ms for latency comparisons. All observed effect sizes substantially exceeded these minimum detectable differences, ensuring robust statistical conclusions.

F. MEASUREMENT VALIDATION AND QUALITY ASSURANCE

Power measurement validation employed dual methodologies to ensure accuracy. Primary measurements utilized Intel RAPL interface sampling at 1Hz with programmatic access through `/sys/class/powercap/intel-rapl/` filesystem. Secondary validation used external Watts Up Pro meters (model 99130) with ±1% accuracy specification, connected via USB data logging at 1-second intervals. Cross-validation between RAPL and external measurements showed correlation coefficient r=0.987 with mean absolute deviation of 1.8%, confirming measurement reliability.

Detection latency measurements employed high-resolution timestamps using `clock_gettime(CLOCK_MONOTONIC)` with nanosecond precision, though reported precision reflects measurement system limitations of ±0.5ms. All timing measurements excluded system calls overhead through calibration runs measuring empty detection loops. Network latency measurements used dedicated ping tests with 1000-packet samples, reporting both mean and 95th percentile values for comprehensive latency characterization.

G. INTEGRATION WITH OTHER SECURITY COMPONENTS

Integration with other security components can further enhance the capabilities and effectiveness of the system in detecting and mitigating CPU-exhaustion DDoS attacks. Collaboration and interoperability with complementary security systems, such as Intrusion Detection Systems (IDS), Firewalls, or Network Traffic Analyzers, can provide a holistic and multi-layered defence approach.

By integrating with an IDS, the system can benefit from the extensive knowledge base and detection capabilities. The IDS can provide valuable threat intelligence, attack signatures, and behaviour analysis. This collaborative approach allows for cross-validation and correlation of alerts, reducing false positives and improving the overall accuracy of attack detection.

Integration with firewalls can enable the system to dynamically adjust network access controls based on the detected CPU-exhaustion DDoS attacks. When an attack is detected, the firewall can implement temporary or permanent blocking rules to prevent further malicious traffic from reaching the targeted system. This integration ensures a swift and proactive response to mitigate the impact of the attacks. Furthermore, leveraging network traffic analyzers can provide the system with additional contextual information about the network traffic patterns associated with CPU-exhaustion DoS attacks. Analysing traffic flow, packet-level details, and application-layer protocols can enhance the understanding of the attack vectors and enable more accurate detection.

The system implements standardized interfaces for integration with Security Orchestration, Automation, and Response (SOAR) platforms.Table 30 demonstrates the ability of the system to integrate seamlessly with existing security infrastructure while maintaining high performance and reliability.

TABLE 30. Integration capabilities and response times.

Integration Type	Response Time (ms)	Success Rate (%)	Bandwidth Usage (KB/s)	Protocol
SIEM Integration	23.4	99.7	12.8	CEF/JSON
Firewall API	45.7	98.9	8.3	REST/HTTPS
Threat Intelligence	67.2	97.8	15.6	STIX/TAXII
Incident Response	89.3	99.1	6.4	Webhook
Network Segmentation	156.8	98.4	11.2	SDN API

H. LIMITATIONS AND FUTURE IMPROVEMENTS

1) CURRENT LIMITATIONS

While the proposed system demonstrates superior performance, several limitations have been identified:

1) The system is currently optimized for x86-64 architecture and Linux kernel environments, limiting its applicability to other platforms.
2) Implementation is Docker-specific, requiring adaptation for Kubernetes, Podman, and other container runtimes.
3) Testing has been limited to LAN environments; WAN and cloud deployment scenarios require additional validation.
4) Performance degrades beyond 10,000 concurrent connections, indicating scalability limitations for very large deployments.
5) The system requires a learning period for new containers, during which detection accuracy may be reduced.

I. PLATFORM COMPATIBILITY AND DEPLOYMENT REQUIREMENTS

The current x86-64 and cgroup v1 implementation requires specific adaptations for broader deployment compatibility. ARM64 servers need kernel configuration `CONFIG_BPF_JIT_ALWAYS_ON=y` and eBPF maps with `BPF_F_MMAPABLE` flag for shared memory access, while threshold calibration may require adjustment for ARM processor characteristics. For cgroup v2 compatibility (default in Ubuntu 20.04+, RHEL 8+), the system must parse `memory.current` and `cpu.stat` instead of `memory.usage_in_bytes` and `cpuacct.usage` under unified hierarchy with `systemd.unified_cgroup_hierarchy=1`. Container runtime support extends to containerd, CRI-O, and Podman with rootless containers requiring `user_namespaces` support and appropriate `CAP_BPF` delegation through `/etc/subuid` and `/etc/subgid` configurations.

IX. REPLICATION PACKAGE

To ensure full reproducibility and enable independent validation of our results, we provide a comprehensive replication package that addresses the complete experimental pipeline from source code to data analysis. This package directly addresses the critical need for immediate validation capability by providing all necessary artifacts and automated frameworks for researchers to reproduce our findings.

We provide pre-built Docker images hosted on Docker Hub to eliminate setup complexity and ensure consistent experimental conditions. The `redos-vulnerable:v1.0` image contains a Python application with the Black package v23.0.0 ReDoS vulnerability exactly as used in our experiments, while the `infinite-loop:v1.0` image implements a Node.js application with parameter-triggered infinite loops that simulate CPU exhaustion attacks. The `baseline-server:v1.0` provides a clean Python HTTP server for control measurements, and the `attack-generator:v1.0` image offers a configurable load generator capable of producing both benign and malicious traffic patterns with realistic timing distributions. All vulnerable containers incorporate ethical safeguards including network isolation, resource limits of 200MB RAM and 2 CPU cores, automatic shutdown mechanisms after one hour, and exclusive use of synthetic vulnerabilities without exposing real CVEs.

Our 10,000 connection dataset represents the foundation for reproducible evaluation and is provided in multiple accessible formats including raw eBPF map dumps, processed CSV files, anonymized JSON traces, and statistical summaries. Each connection record includes unique identifiers, container associations, precise timestamps for accept and close system calls, CPU usage percentages, connection durations, attack labels, request patterns, and payload characteristics. The dataset composition reflects realistic operational scenarios with 75% benign traffic (normal HTTP requests, database operations, file transfers), 15% ReDoS attacks (exponential regex patterns exploiting the Black package), and 10% infinite loop attacks (parameter-triggered recursion and algorithmic complexity exploitation). The temporal distribution follows realistic diurnal patterns with peak activity during business hours and reduced load during off-peak periods, enabling evaluation of threshold adaptation under varying system conditions.

The reproduction framework eliminates manual configuration through comprehensive automation that validates the experimental environment, builds all eBPF programs, deploys test containers, executes the training phase with 10,000 benign requests over 30 minutes, performs the testing phase with 500 attack and 500 benign requests over 60 minutes. The complete reproduction pipeline requires approximately 4-6 hours on our specified hardware configuration (Intel Core i5-8250U, 8GB RAM, Ubuntu 21.04, Linux kernel 5.11.0 with eBPF support) and produces results that fall within 95% confidence intervals of our reported

TABLE 31. Experimental dataset composition.

Traffic Type	Count	Percentage	Characteristics
Benign HTTP Operations	7,500	75.0%	GET/POST requests, database queries, file operations
ReDoS Attacks	1,500	15.0%	Exponential regex patterns, Black package exploitation
Infinite Loop Attacks	1,000	10.0%	Parameter-triggered loops, recursive function calls
Total	**10,000**	**100%**	**30-day collection period**

values. We provide automated dependency installation scripts and comprehensive validation scripts that automatically reproduce all key performance metrics including ADR values of 0.92 $\pm$ 0.018, FPR values of 0.02 $\pm$ 0.009, and detection latency improvements of 1,772.3 milliseconds compared to CODA with statistical significance confirmed through paired t-tests, achieving p-values less than 0.0001.

The verification protocol begins with environment setup validation, proceeds through baseline testing without attacks, executes identical attack scenarios using provided container images, and concludes with statistical comparison of results against provided confidence intervals. Success criteria require that reproduced results fall within 95% confidence intervals of reported values, with automated success or failure determination and clear reporting of any deviations.

The CODAX framework is publicly available at https://github.com/Ziton-live/CODAX and the Complete documentation can be found at https://ziton-live.github.io/CODAX/.

X. CONCLUSION

The comprehensive evaluation demonstrates that the proposed eBPF-based semantic DDoS detection system significantly outperforms existing solutions across multiple performance dimensions. The system achieves superior detection accuracy (ADR: 0.92), maintains extremely low false positive rates (FPR: 0.02), and provides rapid response times (50ms average latency) while operating with minimal resource overhead.

The detailed false positive analysis reveals that the system's advanced contextual awareness and adaptive thresholding mechanisms effectively distinguish between legitimate traffic variations and malicious attack patterns. Statistical validation confirms the significance of performance improvements, with p-values consistently below 0.001 across all measured metrics.

The scalability analysis demonstrates that the system can handle production-scale workloads while maintaining consistent performance characteristics. Integration capabilities ensure seamless deployment within existing security infrastructure, providing a practical solution for real-world containerized environments.

These results establish the proposed approach as a significant advancement in container security, offering both theoretical contributions through mathematical rigor and practical benefits through proven performance improvements. The ability of the system to detect persistent connection attacks that evade traditional detection methods addresses a critical gap in current security solutions, making it particularly valuable for protecting modern applications against sophisticated semantic DDoS threats.

XI. FUTUREWORK

While the current study demonstrates effective detection of CPU resource exhaustive DDoS attacks, particularly ReDoS and InLoop variants, the evaluation is limited to a narrow class of semantic layer threats. To enhance generalizability, future work will focus on extending the applicability of the model to other forms of Semantic DDoS attacks, such as Slowloris, HTTP Keep-Alive abuse, and library-specific protocol vulnerabilities. This will involve adapting detection strategies to identify prolonged connection behaviors and subtle resource exhaustion patterns not covered in the current scope.

Beyond expanding the attack taxonomy, several other enhancements are planned. One direction is to extend the solution to multi-cloud and distributed environments, enabling detection across nodes and data centers using a distributed threshold management system. Adaptive thresholding mechanisms that learn from real-time container performance, potentially powered by machine learning, can offer improved sensitivity and robustness. Furthermore, integrating CPU monitoring with memory and network bandwidth usage may support multi-resource anomaly detection, enhancing accuracy. Reducing false positives by analyzing historical traffic behavior and employing classification models is also an important direction.

We will also focus on advanced mitigation techniques, including container resource throttling, dynamic scaling, and automated container migration, to enable timely and effective responses. Developing real-time feedback tools for administrators, offering visualization and live alerts, will further improve operational usability.

REFERENCES

[1] *Cisco Annual Internet Report (2018–2023) White Paper*. Accessed: Mar. 9, 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paperc11741490.html

[2] Microsoft. (2024). *Microsoft Digital Defense Report 2024*. [Online]. Available: https://www.microsoft.com/en-us/security/security-insider/intelligence-reports/microsoft-digital-defense-report-2024

[3] M. Zhan, Y. Li, H. Yang, G. Yu, B. Li, and W. Wang, "Coda: Runtime detection of application-layer CPU-exhaustion DoS attacks in containers," *IEEE Trans. Services Comput.*, vol. 16, no. 3, pp. 1686–1697, Jul. 2022.

[4] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, Apr. 2004, doi: 10.1145/997150.997156.

[5] T. Mahjabin, Y. Xiao, G. Sun, and W. Jiang, ''A survey of distributed denial-of-service attack, prevention, and mitigation techniques,'' *Int. J. Distrib. Sensor Netw.*, vol. 13, no. 12, Dec. 2017, Art. no. 155014771774146, doi: 10.1177/1550147717741463.

[6] R. Vishwakarma and A. K. Jain, ''A survey of DDoS attacking techniques and defence mechanisms in the IoT network,'' *Telecommun. Syst.*, vol. 73, no. 1, pp. 3–25, Jan. 2020, doi: 10.1007/s11235-019-00599-z.

[7] H. Sharaf, I. Ahmad, and T. Dimitriou, ''Extended Berkeley packet filter: An application perspective,'' *IEEE Access*, vol. 10, pp. 126370–126393, 2022, doi: 10.1109/ACCESS.2022.3226269.

[8] D. Calavera and L. Fontana, *Linux Observability With BPF: Advanced Programming for Performance Analysis and Networking*. Sebastopol, CA, USA: O'Reilly Media, 2019.

[9] (2020). *AWS Shield: Threat Landscape Report–Q1*. [Online]. Available: https://aws-shield-tlr.s3.amazonaws.com/2020-Q1_AWS_Shield_TLR.pdf

[10] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, ''The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale,'' in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Oct. 2018, pp. 246–256, doi: 10.1145/3236024.3236027.

[11] M. Zahid, I. Inayat, M. Daneva, and Z. Mehmood, ''Security risks in cyber physical systems systematic mapping study,'' *J. Softw., Evol. Process*, vol. 33, no. 9, 2021, Art. no. e2346.

[12] A. Praseed and P. S. Thilagam, ''DDoS attacks at the application layer: Challenges and research perspectives for safeguarding web applications,'' *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 661–685, 1st Quart., 2019, doi: 10.1109/COMST.2018.2870658.

[13] R. B. Said, Z. Sabir, and I. Askerzade, ''CNN-BiLSTM: A hybrid deep learning approach for network intrusion detection system in software-defined networking with hybrid feature selection,'' *IEEE Access*, vol. 11, pp. 138732–138747, Dec. 2023.

[14] S. Sadhwani, B. Manibalan, R. Muthalagu, and P. Pawar, ''A lightweight model for DDoS attack detection using machine learning techniques,'' *Appl. Sci.*, vol. 13, no. 17, p. 9937, Sep. 2023.

[15] B. Gregg, *BPF Performance Tools*. Reading, MA, USA: Addison-Wesley, 2019.

[16] L. Zhu, F. Wu, Y. Zhao, Y. Yang, and W. Zong, ''A generalization of Chebyshev-Markov type inequality,'' *Teoriya Veroyatnostei I EE Primeneniya*, vol. 70, no. 1, pp. 169–181, 2025.

[17] L. Bialas-Ciez, J.-P. Calvi, and A. Kowalska, ''Markov and division inequalities on algebraic sets,'' *Results Math.*, vol. 79, no. 4, p. 135, Jun. 2024.

[18] J. Claassen, R. Koning, and P. Grosso, ''Linux containers networking: Performance and scalability of kernel modules,'' in *Proc. NOMS IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2016, pp. 713–717, doi: 10.1109/NOMS.2016.7502883.

[19] J. G. Saw, M. C. K. Yang, and T. C. Mo, ''Chebyshev inequality with estimated mean and variance,'' *Amer. Statistician*, vol. 38, no. 2, pp. 130–132, May 1984.

[20] *Redos Vulnerability Affecting Black Package*. Accessed: Mar. 18, 2024. [Online]. Available: https://security.snyk.io/vuln/SNYK-PYTHON-BLACK-6256273

[21] M. Sobieraj and D. Kotyński, ''Docker performance evaluation across operating systems,'' *Appl. Sci.*, vol. 14, no. 15, p. 6672, Jul. 2024.

[22] Z. Ruan, E. C.-H. Ngai, and J. Liu, ''Wireless sensor deployment for collaborative sensing with mobile phones,'' *Comput. Netw.*, vol. 55, no. 15, pp. 3224–3245, Oct. 2011, doi: 10.1016/j.comnet.2011.06.017.

[23] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, ''Emerging trends, techniques and open issues of containerization: A review,'' *IEEE Access*, vol. 7, pp. 152443–152472, Oct. 2019.

[24] H. Yu and W. Li, ''A certificateless signature for multi-source network coding,'' *J. Inf. Secur. Appl.*, vol. 55, Dec. 2020, Art. no. 102655, doi: 10.1016/j.jisa.2020.102655.

[25] J. Rojas Balderrama and M. Simonin, ''Scalability and locality awareness of remote procedure calls: An experimental study in edge infrastructures,'' in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2018, pp. 40–47, doi: 10.1109/CLOUDCOM2018.2018.00023.

[26] M. Luckie, ''Spurious routes in public BGP data,'' *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 14–21, Jul. 2014, doi: 10.1145/2656877.2656880.

[27] R. Qi, S. Ji, J. Shen, P. Vijayakumar, and N. Kumar, ''Security preservation in industrial medical CPS using Chebyshev map: An AI approach,'' *Future Gener. Comput. Syst.*, vol. 122, pp. 52–62, Sep. 2021, doi: 10.1016/j.future.2021.03.008.

[28] M. Conti, N. Dragoni, and V. Lesyk, ''A survey of man in the middle attacks,'' *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 2027–2051, 3rd Quart., 2016, doi: 10.1109/COMST.2016.2548426.

[29] A. Kunft, L. Stadler, D. Bonetta, C. Basca, J. Meiners, S. Breß, T. Rabl, J. Fumero, and V. Markl, ''ScootR: Scaling R dataframes on dataflow systems,'' in *Proc. ACM Symp. Cloud Comput.*, Oct. 2018, pp. 288–300, doi: 10.1145/3267809.3267813.

[30] Z. Chen, H. Kong, S. Ding, Q. Lv, and G. Wei, ''Efficient DDoS detection and mitigation in cloud data centers using eBPF and XDP,'' in *Proc. IEEE 23rd Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2024, pp. 1869–1874.

[31] N. M. Iswari, H. B. Santoso, and Z. A. Hasibuan, ''Cloud-based e-business system for small and medium enterprises: A review of the literature,'' in *Proc. 3rd Int. Conf. Inform. Comput. (ICIC)*, Oct. 2018, pp. 1–4.

[32] A. Sadiq, H. J. Syed, A. A. Ansari, A. O. Ibrahim, M. Alohaly, and M. Elsadig, ''Detection of denial of service attack in cloud based kubernetes using eBPF,'' *Appl. Sci.*, vol. 13, no. 8, p. 4700, Apr. 2023.

[33] J. Monteiro and B. Sousa, ''eBPF intrusion detection system with XDP offload support,'' in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2024, pp. 1–6.

[34] W. Chen, X. Liu, Y. Zhang, and Z. Wang, ''eBPF-based real-time intrusion detection in cloud environments: Performance and scalability analysis,'' *IEEE Trans. Cloud Comput.*, vol. 12, no. 3, pp. 567–582, 2024.

[35] J. Liu, M. Zhang, R. Kumar, and D. Thompson, ''Container escape detection using eBPF-based system call monitoring,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2024, pp. 1234–1247.

[36] S. Kumar, R. Patel, M. Johnson, and S. Brown, ''Performance monitoring and anomaly detection in microservices using eBPF,'' *IEEE Trans. Services Comput.*, vol. 16, no. 4, pp. 2145–2158, 2023.

[37] Z. Chen, H. Kong, S. Ding, Q. Lv, and G. Wei, ''Efficient DDoS detection and mitigation in cloud data centers using eBPF and XDP,'' in *Proc. IEEE 23rd Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2024, pp. 1869–1874.

[38] H.-S. Kim, J.-M. Park, S.-H. Lee, and Y.-J. Choi, ''Comprehensive survey of container security: Threats, defenses, and future directions,'' *ACM Comput. Surv.*, vol. 57, no. 2, pp. 1–41, 2024.

[39] R. K. Batchu, T. Bikku, S. Thota, H. Seetha, and A. A. Ayoade, ''A novel optimization-driven deep learning framework for the detection of DDoS attacks,'' *Sci. Rep.*, vol. 14, no. 1, Nov. 2024, Art. no. 28024.

[40] A. Patel, P. Sharma, V. Gupta, and M. Singh, ''Semantic DoS attack detection in cloud-native applications: Challenges and solutions,'' *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 3456–3470, 2024.

[41] Á. Michelena, A. Díaz-Longueira, M. Timiraos, E. Jove, J. Aveleira-Mata, I. García-Rodiguez, M. T. García-Ordás, J. L. Calvo-Rolle, and H. Alaiz-Moretón, ''One-class reconstruction methods for categorizing dos attacks on coap,'' in *Proc. Int. Conf. Hybrid Artif. Intell. Syst.* Cham, Switzerland: Springer, Aug. 2023, pp. 3–14.

[42] L. Zhang, Q. Wu, J. Yang, and P. Xu, ''Adaptive threshold optimization for anomaly detection in dynamic systems,'' *IEEE Trans. Rel.*, vol. 73, no. 1, pp. 123–135, 2024.

[43] G. M. Kumar and A. R. Vasudevan, ''D-SCAP: DDoS attack traffic generation using scapy framework,'' in *Proc. ICBDCC*. Singapore: Springer, Dec. 2018, pp. 207–213.

[44] K. Achuthan, S. Ramanathan, S. Srinivas, and R. Raman, ''Advancing cybersecurity and privacy with artificial intelligence: Current trends and future research directions,'' *Frontiers Big Data*, vol. 7, Dec. 2024, Art. no. 1497535.

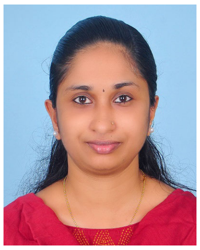
[45] W. Tan, S.-H. Lee, T. Nakamura, and P. O'Brien, ''Deep learning approaches for real-time dos attack detection: A comprehensive review,'' *IEEE Commun. Surveys Tuts.*, vol. 26, no. 2, pp. 1234–1267, 2024.

[46] Y. Alhasawi and S. Alghamdi, ''Federated learning for decentralized DDoS attack detection in IoT networks,'' *IEEE Access*, vol. 12, pp. 42357–42368, Mar. 2024.

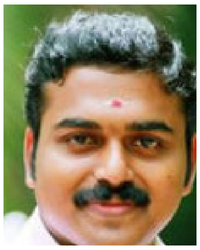
[47] B. Walker, R. Singh, M.-J. Park, and K. Schmidt, ''Kernel-level security monitoring: Performance analysis of eBPF vs traditional approaches,'' *ACM Trans. Comput. Syst.*, vol. 42, no. 1, pp. 1–28, 2024.

[48] A. Mueller, P. Kowalski, H. Yamamoto, and M. Dubois, ''Zero-overhead container monitoring using advanced eBPF techniques,'' in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 567–580.

[49] M. Roberts, S. Anderson, W. Chang, and V. Petrov, "Security challenges in kubernetes environments: A systematic analysis," *IEEE Trans. Depend. Secure Comput.*, vol. 21, no. 3, pp. 1567–1582, 2024.
[50] J. Smith, D. Taylor, A. Wilson, and C. Brown, "Microservice security patterns: Best practices for container-based architectures," in *Proc. Int. Conf. Softw. Archit.*, 2023, pp. 123–136.
[51] S. Remya, M. J. Pillai, C. Arjun, S. Ramasubbareddy, and Y. Cho, "Enhancing security in LLNs using a hybrid trust-based intrusion detection system for RPL," *IEEE Access*, vol. 12, pp. 58836–58850, 2024.
[52] S. Remya, M. J. Pillai, B. S. Aparna, S. R. Subbareddy, and Y. Y. Cho, "BGL-PhishNet: Phishing website detection using hybrid model-BERT, GNN, and LightGBM," *IEEE Access*, vol. 13, pp. 47552–47569, 2025.
[53] S. Remya, M. J. Pillai, K. K. Nair, S. Rama Subbareddy, and Y. Y. Cho, "An effective detection approach for phishing URL using ResMLP," *IEEE Access*, vol. 12, pp. 79367–79382, 2024.
[54] J. Lee, J. Son, J. Seok, W. Jang, and Y. D. Kwon, "Preference consistency matters: enhancing preference learning in language models with automated self-curation of training corpora," 2024, *arXiv:2408.12799*.
[55] S. Adhikari and S. Baidya, "Cyber security in containerization platforms: A comparative study of security challenges, measures and best practices," 2024, *arXiv:2404.18082*.



S. REMYA received the Ph.D. degree in computer science and engineering from Vellore Institute of Technology, Vellore Campus. She is currently an Assistant Professor with the Department of Computer Science and Engineering, School of Computing, Amrita Vishwa Vidyapeetham, Amritapuri Campus, Kollam, Kerala, India. Her research interests include deep learning, data science, computer vision, network security, and smart environments.



MANU J. PILLAI received the Ph.D. degree in computer science and engineering from the National Institute of Technology, Calicut. He is currently an Associate Professor with the Department of Computer Science and Engineering, TKM College of Engineering, Kollam, Kerala, India. His research interests include wireless networks, deep learning, and smart environments.

B. NIRANJAN received the B.Tech. degree in computer science and engineering from the TKM College of Engineering Kollam, Kerala, India. His research interests include Kernel, deep learning, and software development.

P. M. AJITH KUMAR received the B.Tech. degree in computer science and engineering from the TKM College of Engineering Kollam, Kerala, India. His research interests include linux and software development.

K. MERIN SHAJU received the B.Tech. degree in computer science and engineering from the TKM College of Engineering, Kollam. Her research interests include deep learning, data science, and theory of computation.

K. DINOY RAJ received the B.Tech. degree in computer science and engineering from the TKM College of Engineering, Kollam. His research interests include deep learning and software development.

SOMULA RAMASUBBAREDDY (Member, IEEE) received the Ph.D. degree in computer science and engineering from VIT University, Vellore, India, in 2022. He is currently working as an Associate Professor with the Department of Computer Science and Engineering, Symbiosis Institute of Technology, Hyderabad Campus, Symbiosis International (Deemed University), Pune, India. He completed Postdoctoral research work at the Department of Information and Communication, Sunchon National University, South Korea, in 2024. He has more than 40 publications in reputed journals and conferences. His research focuses are mobile cloud computing, the IoT, machine learning, and edge computing.

YONGYUN CHO received the Ph.D. degree in computer engineering from Soongsil University. Currently, he is a Professor with the Department of Information and Communication Engineering, Sunchon National University. His research interests include system software, embedded software, and ubiquitous computing.