

Resource-Aware eBPF/XDP-Assisted DDoS Mitigation for IoT Edge Systems: Design and Management Trade-Offs

Abdurrahman Tolay

`abdurrahman.tolay@outlook.com`

Istinye University

Research Article

Keywords: eBPF, XDP, DDoS mitigation, IoT edge computing, network and systems management, resource-aware security

Posted Date: September 16th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10908211/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Resource-Aware eBPF/XDP-Assisted DDoS Mitigation for IoT Edge Systems: Design and Management Trade-Offs

Abdurrahman Tolay¹

¹Istinye University, Istanbul, Türkiye

Corresponding author: abdurrahman.tolay@outlook.com
ORCID: 0009-0000-8855-6799

Abstract

Resource-constrained Internet of Things (IoT) edge nodes increasingly combine service delivery with local security enforcement, creating a systems-management problem in which mitigation effectiveness must be balanced against resource availability and service continuity. This paper evaluates a resource-aware architecture based on the extended Berkeley Packet Filter (eBPF) and eXpress Data Path (XDP) for Linux-based IoT edge systems. Kernel-level packet monitoring and per-source rate detection generate events for a user-space controller that manages source blocking, logging, and administrator notification. The architecture is evaluated in a controlled Docker environment and on a Raspberry Pi 4 under Distributed Denial of Service (DDoS) UDP flood traffic using malicious traffic mitigation, legitimate throughput, CPU and memory utilization, latency, and responsiveness as joint security and management metrics. In Docker, approximately 99% of malicious traffic was mitigated while the protected service remained responsive. On the Raspberry Pi, approximately 94 packets per second while more than 90 Raspberry Pi remained responsive, but filtering consumed approximately 85% and added approximately 3–4 ms of latency. The results show that local eBPF/XDP-assisted security management can preserve edge-service continuity under the evaluated workload, but only with platform-specific resource budgeting and threshold configuration.

Keywords: eBPF; XDP; DDoS mitigation; IoT edge computing; network and systems management; resource-aware security

1 Introduction

Internet of Things (IoT) systems increasingly support monitoring, control, and service-delivery functions at the network edge. Volumetric Distributed Denial of Service (DDoS) attacks can disrupt these functions by consuming network and packet-processing resources. The risk is acute on edge platforms with limited CPU capacity, memory, and thermal headroom [1, 2]. Compromised IoT devices can also be assembled into botnets that generate large-scale DDoS traffic [3].

From a network and systems management perspective, local DDoS protection is not only a packet-filtering problem. It requires continuous observation, triggering, policy actuation, resource monitoring, and preservation of service continuity under stress. These functions align with the broader management research space that covers security management, edge/fog computing, service assurance, and programmable network functions [4]. The management challenge is therefore to suppress abusive traffic without allowing the defensive mechanism itself to exhaust the constrained platform.

Upstream infrastructure and dedicated appliances remain important, but local protection is still required when filtering is unavailable or traffic has already reached an edge endpoint. Such protection must balance mitigation against CPU, memory, latency, and legitimate throughput; otherwise, defensive processing can itself reduce availability [2]. Gateway-level IoT filtering has also combined MUD policies with eBPF enforcement [5].

The extended Berkeley Packet Filter (eBPF) provides a mechanism for executing verified programs in the Linux kernel, while the eXpress Data Path (XDP) provides an early packet-processing hook in the receive path [6, 7]. These facilities can support lightweight telemetry and compact per-source state before much of the network stack. Management and policy functions can then remain in user space, where they can coordinate blocking, logging, notification, and operational control.

Prior work has evaluated XDP under high-rate packet-processing workloads [8], used eBPF to construct programmable network functions [9], implemented IoT gateway filters [5], and developed kernel-level intrusion detection and prevention mechanisms [10]. For constrained edge deployment, however, security effectiveness must be assessed together with management cost: CPU, memory, latency, legitimate traffic, and service responsiveness. Container experiments alone do not expose the same resource limits as a physical edge device.

This paper therefore evaluates a resource-aware, management-oriented eBPF/XDP-assisted DDoS mitigation architecture for Linux-based IoT edge systems. eBPF/XDP performs packet observation and per-source rate detection, then reports events to a Python controller that manages source-blocking iptables rules, local logging, and administrator notification. The Raspberry Pi workflow is intentionally described as a hybrid monitoring-and-actuation loop rather than a pure XDP-drop design.

The evaluation uses controlled Docker and Raspberry Pi 4 environments under UDP flood traffic. Docker reports approximately 99% malicious-traffic mitigation, whereas the Raspberry Pi reports approximately 94% at representative rates up to 30,000 packets per second while preserving more than 90% of legitimate throughput. Mitigated systems remained operational, whereas the corresponding unmitigated conditions were reported as unresponsive. On the Raspberry Pi, this continuity came with substantial resource cost, including approximately 85% of one core and approximately 3–4 ms of added latency.

The paper makes the following four contributions:

1. It specifies a management-oriented hybrid architecture that separates kernel-level monitoring from user-space policy actuation, source blocking, logging, and notification.
2. It evaluates the architecture on Raspberry Pi 4 edge-class hardware under controlled UDP flooding, complementing a Docker validation stage.
3. It assesses security and management outcomes jointly through malicious traffic mitigation, legitimate throughput, CPU, memory, latency, and service responsiveness.
4. It identifies deployment-relevant trade-offs for operating local security control loops on resource-constrained edge systems.

The remainder of the paper is organized as follows. Section 2 reviews related work on IoT and edge DDoS mitigation, eBPF/XDP-based network security, and the management positioning of the study. Section 3 describes the management-oriented architecture, threat model, monitoring path, and user-space actuation workflow. Section 4 presents the Docker and Raspberry Pi experimental methodology and management metrics, and Section 5 reports the corresponding results. Section 6 discusses the findings as security–performance and resource-management trade-offs. Section 7 identifies limitations and threats to validity. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Related Work

2.1 DDoS Mitigation in IoT and Edge Environments

IoT environments present two related DDoS concerns. First, devices that provide local sensing, control, or gateway services may themselves be targets of traffic floods. Their limited processing capacity, memory, energy budget, and network bandwidth can make service availability sensitive to sustained packet-processing load [1, 2]. Second, inadequately secured devices may be compromised and recruited into botnets that generate DDoS traffic against other systems. The empirical study of Mirai illustrates how large populations of Internet-accessible IoT devices can be discovered, compromised, and coordinated for this purpose [3, 11]. These two roles—IoT device as target and as traffic source—mean that mitigation must consider both protection of local services and control of traffic associated with compromised devices.

Common mitigation mechanisms include firewall rules, source-address blocking, rate limiting, intrusion detection and prevention systems, and filtering by upstream or cloud-based providers. Their suitability depends on where they are deployed, the traffic volume they must process, and the resources available at that location [12, 13]. On a constrained endpoint or gateway, packet inspection, state maintenance, rule management, and alert processing all compete with the device’s primary workload. Processing and memory limits therefore affect how much defensive logic can be placed locally, while added latency or excessive filtering can impair legitimate service delivery [2]. This does not make conventional mechanisms ineffective; rather, it makes their operating cost and placement important design considerations for IoT edge deployments.

IoT gateways and Linux-based edge devices offer useful enforcement points between local devices and external networks. Local filtering can apply device-specific policy after traffic reaches the access network, complementing upstream protection. Its value depends on the gateway remaining responsive while monitoring and enforcing policy within a constrained resource budget. IoT-specific flow features have also been studied for gateway-side detection of consumer-device DDoS traffic [14].

Feraudo et al. provide a closely related example of policy- and rate-oriented mitigation at an IoT gateway [5]. Their approach uses Manufacturer Usage Description (MUD) information to express expected, device-specific communication behavior and augments those policies with traffic-rate constraints. Enforcement at the gateway uses programmable filtering, including an eBPF implementation, to restrict nonconforming traffic. Unlike that MUD-centered design, this evaluation focuses on per-source rate detection, dynamic blocking, and operational behavior on constrained Linux edge hardware.

IoT DDoS defense can therefore combine endpoint, gateway, and upstream controls. For constrained gateways, suppression must be assessed together with resource cost, latency, legitimate traffic, and service responsiveness.

2.2 eBPF/XDP for High-Performance Network Security

Prior research establishes eBPF as a programmable execution environment for verified programs attached to supported Linux kernel hooks. In networking applications, programs can inspect packet metadata and headers, invoke permitted helper functions, and exchange or retain state through maps that are also accessible to user-space applications. Before loading, the verifier checks program properties intended to protect kernel safety; where supported, just-in-time compilation can translate eBPF instructions into native machine code. These mechanisms permit packet-processing logic to be changed without developing a conventional kernel module, although program constraints, verifier acceptance, and platform-dependent implementation details remain practical development considerations [6, 15].

Within this execution model, XDP provides an early hook in the receive path, typically at the network-driver level before an incoming packet traverses much of the conventional Linux networking stack. An XDP program can inspect a packet and return an action such as passing it onward, dropping it, redirecting it, or transmitting it back through an appropriate interface. Performing such decisions early can avoid allocating and executing later networking-stack processing for packets that need not proceed further. Høiland-Jørgensen et al. describe this architecture as a basis for programmable, performance-oriented packet processing while retaining integration with the Linux networking environment [7]. The availability of an early hook, however, does not by itself guarantee a particular throughput or make every packet-processing policy suitable for XDP.

Empirical work further shows that XDP performance must be interpreted in relation to the program and its execution environment. Processing cost can vary with packet rate and size, the complexity of parsing and decision logic, the frequency and type of map operations, and the amount of state maintained. CPU and memory characteristics, driver support, queue configuration, and the use of native, generic, or other supported execution modes can also influence the observed behavior. Performance studies therefore complement the XDP architecture by demonstrating that implementation choices and test conditions matter when evaluating the attainable packet-processing rate [7, 8]. XDP consequently offers a high-performance execution point, but its operational cost and capacity must be measured for the specific hardware, workload, and program—a requirement of particular importance for resource-constrained edge devices. Edge-server studies similarly show that XDP and AF_XDP behavior depends on traffic destination and hardware interaction [16], while GALETTE examines an XDP dataplane specifically on Raspberry Pi hardware [17].

Miano et al. show that eBPF network functions can span kernel and user-space components [9]. Maps share state and configuration, allowing packet handling to remain in the kernel while lifecycle, policy, and longer-running control remain in user space.

Security research applies these capabilities to mechanisms that require more than static rule matching. iKern demonstrates an eBPF-based framework for kernel-level intrusion detection and prevention, including packet or flow processing and the enforcement of security responses within a programmable architecture [10]. This demonstrates that compact security logic can remain near the packet path while more complex control and analysis remain in user space. At the same time, analyses and fuzzing of the eBPF subsystem show that verifier and runtime safety should not be treated as an absolute guarantee [18, 19].

The relevance of this approach to IoT security is also reflected in work on gateway enforcement. Feraudo et al. use eBPF-based programmable filtering in an IoT-oriented gateway design that combines communication policy with rate-related constraints [5]. As discussed in Section 2.1, their policy basis and

objectives differ from those of the present study; for the networking context here, the important result is that eBPF traffic enforcement can be incorporated into a gateway positioned near IoT devices. This placement makes early local filtering feasible, while also making the gateway’s own processing overhead and continued availability part of the security assessment.

The literature establishes XDP as an early processing point, maps as shared state, and kernel/user-space partitioning as a practical architecture. Because behavior depends on program, execution mode, hardware, and traffic, edge evaluation must include resources, latency, legitimate traffic, and responsiveness as well as packet mitigation. Work on decoupled verification also identifies deployment costs that become more visible on embedded Linux platforms [20].

2.3 Positioning of the Present Work

The closest research context comprises several related but distinct uses of eBPF and XDP. Høiland-Jørgensen et al. establish XDP as an early, programmable packet-processing facility in the Linux receive path and examine its performance-oriented design [7]. Hohlfeld et al. further demonstrate that observed XDP performance depends on the program, execution mode, hardware, and workload rather than following from the hook alone [8]. These studies provide the packet-path and performance foundations for the present work, but their primary objective is not an operational evaluation of DDoS mitigation on constrained IoT edge hardware.

Parola et al. examine XDP and AF_XDP network functions in edge data-center conditions [16], and Simpson et al. evaluate a lightweight XDP dataplane on Raspberry Pi hardware [17]. These studies provide closer platform and deployment context, but do not evaluate the same hybrid DDoS monitoring, controller, and iptables response path.

Other work broadens the context from packet-processing primitives to complete network and security functions. Miano et al. organize eBPF-based network functions across kernel and user-space components, showing how fast-path processing, shared state, and management logic can be combined in a programmable architecture [9]. iKern applies eBPF to kernel-level intrusion detection and prevention, providing a security-oriented example in which packet or flow processing supports detection and enforcement logic [10]. These systems motivate the separation of frequent kernel-level operations from control or analysis functions, but they address broader network-function or IDS/IPS objectives rather than the specific resource and availability questions examined here.

Feraudo et al. are particularly relevant to the IoT setting because they combine MUD-derived communication policies, rate constraints, and eBPF-based gateway enforcement to mitigate IoT botnet traffic [5]. Their work demonstrates that programmable traffic filtering can be placed close to an IoT network and informed by device-specific policy. The present study does not reproduce the MUD policy model. It instead examines per-source rate detection and dynamic source blocking on a protected Linux edge device, with emphasis on how the defensive workflow behaves under a controlled volumetric UDP workload.

Table 1 summarizes these distinctions. This study combines eBPF/XDP observation and counting with user-space blocking and reporting, then evaluates the workflow in Docker and on a Raspberry Pi 4. It does not claim that eBPF-based mitigation or hybrid processing is new. Instead, the study is positioned as a network and systems management evaluation of a local security-control loop in which monitoring, decision, actuation, resource consumption, and service continuity are considered together.

The contribution is therefore a deployment-oriented assessment of mitigation, legitimate traffic, resources, latency, and responsiveness under the documented Docker and Raspberry Pi UDP workloads, with emphasis on the operational cost of managing security locally on constrained edge hardware.

Study	Objective and context	Kernel/control organization	Evaluation emphasis
Høiland-Jørgensen et al.	XDP architecture for programmable packet processing	XDP programs integrated with the Linux networking environment	Packet-processing design and performance
Miano et al.	General eBPF network functions in microservice environments	Kernel processing with user-space management	Network-function composition and operation
Feraudo et al.	MUD- and rate-policy enforcement for IoT botnet mitigation	eBPF gateway filtering with policy management	IoT communication-policy and rate enforcement
Hadi et al. (iKern)	Kernel-level intrusion detection and prevention	eBPF security processing with control/analysis support	IDS/IPS functionality
Simpson et al. (GALETTE)	XDP dataplane on Raspberry Pi hardware	XDP fast path with platform-aware packet handling	SBC dataplane performance and resource use
Present work	Volumetric UDP mitigation on edge-class Linux hardware	eBPF/XDP monitoring; Python control and iptables enforcement	Docker and Raspberry Pi mitigation, resources, latency, legitimate traffic, and responsiveness

Table 1: Positioning of the present work against closely related eBPF/XDP studies.

3 Management-Oriented System Architecture

3.1 Management Objectives and Threat Model

The protected Linux edge device hosts both the monitoring mechanism and a network service, so defense competes with its primary workload. Service continuity is therefore part of mitigation effectiveness.

The threat model covers compromised IoT sources flooding an external target or another local device. The implementation observes incoming traffic at the protected endpoint, with volumetric UDP as the principal workload. Application-layer floods, encrypted-payload classification, low-and-slow behavior, source-address spoofing, and distributed sources below the threshold are not evaluated.

Table 2 maps five goals to evaluation indicators: timely detection, attack suppression, legitimate-traffic preservation, resource awareness, and operational continuity. They are experimental objectives, not universal guarantees.

Goal	Description	Evaluation indicator
G1	Timely detection	Detection and trigger behavior
G2	Attack suppression	Malicious packet mitigation
G3	Legitimate traffic preservation	Goodput, responsiveness, and observed benign triggers
G4	Resource awareness	CPU and memory utilization
G5	Operational continuity	Latency, service responsiveness, and system stability

Table 2: Design goals and corresponding evaluation metrics.

3.2 eBPF/XDP Packet Monitoring

The XDP program inspects IPv4 UDP packets on the protected interface. It checks Ethernet and IPv4 header boundaries, passes nonmatching traffic, and avoids reads beyond the packet region required by the verifier.

For UDP, the source IPv4 address indexes a BPF hash-map counter. Existing counters are incremented and new sources are initialized to one. The program uses no payload or application model, keeping per-packet work to validation, protocol selection, a map operation, and an increment.

When the configured counter condition is reached, the documented runtime program emits a message with the source address through the kernel trace mechanism. That message is consumed by the user-space

controller described in Section 3.3. Importantly, this program returns `XDP_PASS` after processing; it does not implement the complete mitigation action as an `XDP_DROP` path. Packets therefore continue into the Linux networking stack until the user-space response installs a blocking rule. The evaluated Raspberry Pi workflow is accordingly described as eBPF/XDP-assisted mitigation, not as fully kernel-resident XDP enforcement.

The implementation records contain two threshold descriptions that are not treated as one algorithm. The conceptual design gives an example one-second window and an 800-packet-per-second threshold, whereas the documented runtime program emits an event when a persistent source counter reaches a multiple of 50. The confirmed mechanism is therefore stated as per-source counting with threshold-triggered event generation; the differing threshold formulations remain a reproducibility limitation discussed in Section 7.

This implementation choice has two consequences. First, computationally expensive traffic classification is avoided, which is consistent with the resource constraints of the target platform. Second, a source-rate rule alone cannot establish malicious intent: legitimate high-rate or bursty traffic can cross the same condition, while a distributed attack can divide traffic among sources. The benign workload and legitimate-traffic measurements are therefore needed to characterize behavior for the tested configuration rather than to claim general detection accuracy.

3.3 User-Space Control, Enforcement, and Management

The Python controller forms the control and enforcement layer of the physical implementation. It continuously reads the kernel trace stream and searches for the source-address field included in threshold messages. After decoding the address, the controller checks a per-source cooldown state so that repeated counter events do not cause an unrestricted sequence of identical mitigation and notification actions. The controller code configures a 30-second alert interval; this value governs repeated controller actions and should not be interpreted as the packet-counting window used by the conceptual detector.

For a newly actionable event, the controller dynamically inserts an `iptables` rule that matches the reported source IPv4 address and uses the `DROP` target. Blocking in the documented Raspberry Pi workflow is therefore enforced through the Linux filtering rule installed from user space. The rule affects subsequent traffic from that address after installation, whereas the XDP program itself continues to report `XDP_PASS`. This division is central to interpreting the system: eBPF/XDP supplies early observation and detection state, and `iptables` supplies source-based enforcement.

The controller also logs the blocked source and sends a Telegram notification. These interchangeable management functions provide an audit trail and operational awareness but do not affect classification.

Separating these functions keeps the per-packet kernel program compact and allows policy actions, string handling, persistent storage, notification, and cooldown management to remain in a conventional user-space environment. The trade-off is that mitigation is not applied until the event has crossed the kernel/user-space boundary and the filtering rule has been inserted. The methodology therefore evaluates the end-to-end workflow as implemented rather than attributing all measured suppression to an XDP drop action.

3.4 End-to-End Monitoring and Actuation Workflow

Figure 1 shows the interaction among the monitoring, control, and enforcement components. Traffic first reaches the protected network interface, where the XDP program validates the relevant headers and passes non-IPv4 or non-UDP traffic without updating the detector. For a UDP packet, the source address indexes the BPF map and the per-source count is updated. Packets that do not produce a threshold event continue through the normal Linux network stack toward the IoT or edge service.

When the configured counter condition is met, the kernel program emits a detection record. The Python controller reads the record, reconstructs the source address, and applies its cooldown check. If an action is due, it inserts the source-specific `iptables` `DROP` rule. Later packets from that source are then discarded by the Linux filtering path. In parallel with enforcement, the controller writes the local incident record and attempts administrator notification. These management outputs do not delay the kernel program’s return path, although the initial blocking action necessarily follows event delivery and controller processing.

Thus, XDP passes legitimate or not-yet-blocked packets, installed `iptables` rules discard matching sources, and threshold events drive controller actions. Figure 1 distinguishes these data and control paths.

From a management viewpoint, the implementation realizes four local functions: *observation* through XDP packet inspection and BPF counters, *decision* through threshold-triggered events and controller

cooldown logic, *actuation* through source-specific `iptables` rules, and *assurance/visibility* through service checks, logs, and administrator notification. The loop is reactive and threshold-driven rather than autonomically adaptive: thresholds are not learned online, and the controller does not dynamically reallocate CPU or bandwidth. This distinction is important when interpreting the system as a management mechanism rather than as a fully autonomous defense platform.

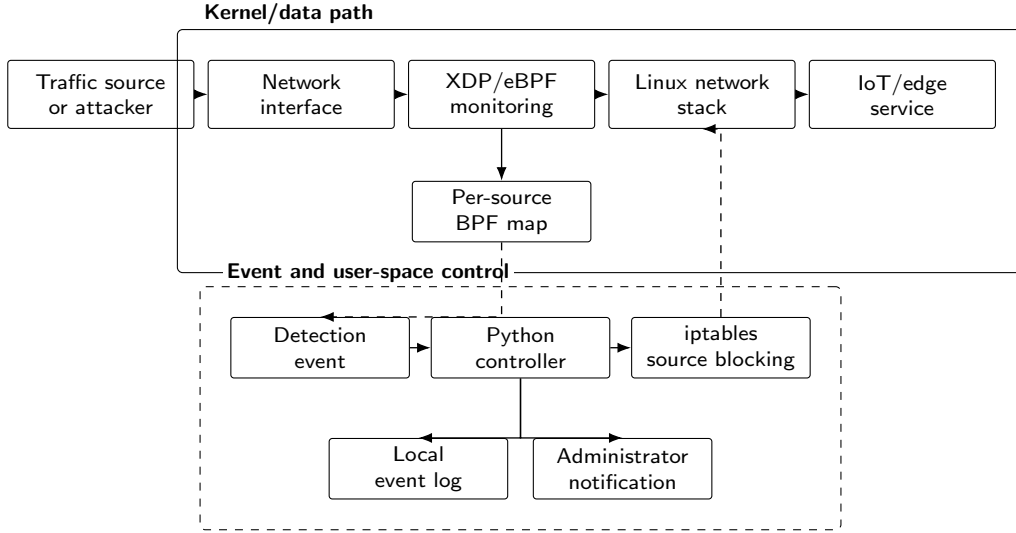


Fig. 1 Hybrid eBPF/XDP monitoring and user-space mitigation architecture

4 Experimental Methodology and Management Metrics

4.1 Evaluation Objectives and Management Metrics

The evaluation examines mitigation, service continuity, and management cost in complementary environments: Docker supports controlled functional testing, whereas the Raspberry Pi exposes physical edge-hardware constraints. The metrics are interpreted as management indicators: suppression reflects policy effectiveness; CPU and memory reflect resource cost; latency and legitimate throughput reflect service impact; and responsiveness reflects operational continuity.

Where documented, measurements cover benign traffic, unmitigated attack, and mitigated attack. The A/B comparison does not specify randomized order, fixed independent repetitions, confidence intervals, or hypothesis tests.

Table 3 defines mitigation, legitimate throughput, CPU, memory, latency, responsiveness, and stability measures. Benign triggers and responsiveness are reported descriptively rather than converted into a false-positive probability or long-duration availability measure.

Metric	Purpose	Measurement interpretation
Malicious traffic mitigation	Security effectiveness	Attack traffic suppressed by the active workflow
Legitimate throughput	Availability	Legitimate traffic delivered during mitigation
CPU utilization	Processing cost	Protected-platform resource consumption under load
Memory utilization	Memory cost	State and controller memory demand
Latency	Response delay	Delay experienced by legitimate communication
Device responsiveness	Operational continuity	Ability to remain reachable and functional
Stability and log behavior	Operational robustness	Kernel/system continuity and recorded mitigation events

Table 3: Management-oriented evaluation metrics and interpretation.

4.2 Docker-Based Test Environment

Docker containers on an isolated virtual bridge represented victim, attacker, and benign-client roles (Figure 2(a)). The victim hosted a lightweight service and the eBPF/XDP component, allowing malicious and benign traffic to be generated independently.

Attacker-side tools included `hping3` and custom Python socket scripts, with UDP floods as the principal reported workload. TCP SYN generation was available in the Docker setup but is not treated as a comprehensively evaluated attack class because the cross-platform comparison centers on UDP flooding. The benign client generated low-rate HTTP or publish/subscribe traffic to verify that the victim service remained accessible.

Victim monitoring covered packet counts, CPU, throughput, and service responsiveness with the eBPF component active or inactive. The environment validated behavior before hardware deployment but did not reproduce Raspberry Pi CPU, driver, or interface constraints.

The implementation records do not establish an identical loader and enforcement path across the two stages. The Docker description includes LLVM/Clang compilation, a libbpf loader, native XDP, and direct `XDP_DROP`, whereas the documented Raspberry Pi runtime uses trace events, `XDP_PASS`, and user-space `iptables` blocking. Docker is therefore reported as a controlled eBPF/XDP evaluation without asserting that its execution path was identical to the physical workflow.

4.3 Raspberry Pi Hardware Testbed

The second stage used a Raspberry Pi 4 Model B with 4 GB RAM as the protected edge-class platform. The recorded operating environment was Raspberry Pi OS 64-bit with Linux kernel 5.15.76-v8+. A dedicated computer generated traffic over a local, Gigabit-capable network. The hardware arrangement is summarized in Figure 2(b), and Table 4 lists the confirmed platform properties.

GRO and LRO were disabled because receive aggregation could alter the individual-packet counts used by the detector. This was a measurement choice, not a general deployment recommendation.

The exact interface identifier is not material to the reported procedure and is omitted. Attachment records refer to both native-XDP tooling and a Python BCC interface, so no single loader is asserted. The confirmed runtime path is an eBPF/XDP program attached to the receiving interface, with the controller consuming kernel trace output and managing `iptables`, logging, and notification.

Measurements were collected using `top`, `vmstat`, `bpftool`, packet counters, service checks, and logs; other intensive Raspberry Pi workloads were avoided.

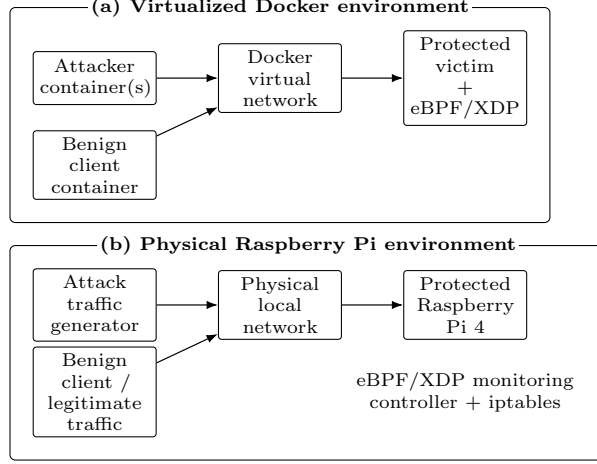


Fig. 2 Experimental environments used for virtualized and hardware-based evaluation

Component	Docker evaluation	Raspberry Pi evaluation
Protected platform	Victim container with lightweight service	Raspberry Pi 4 Model B, 4 GB
Operating environment	Isolated Docker environment	Raspberry Pi OS 64-bit; Linux 5.15.76-v8+
Network context	Docker virtual bridge	Local Gigabit-capable network
Traffic sources	Attacker container(s) and benign client	Dedicated traffic generator and benign client where used
Mitigation path	Victim-side eBPF/XDP; controlled active/inactive comparison	eBPF/XDP monitoring; Python controller; iptables enforcement
Observation tools	BPF counters, sar or top , throughput and service checks	top , vmstat , bpftool , counters, service checks, and logs
NIC configuration	Not specified	GRO and LRO disabled

Table 4: Experimental platform configuration.

4.4 Traffic Generation and Experimental Procedure

Table 5 lists the principal documented workloads. In the Docker configuration, up to three attacker containers generated UDP flood traffic, with a representative configured rate of 15,000 packets per second per attacker. A benign client generated low-rate application traffic. The separate results summary gives a higher aggregate Docker rate. Because the relationship between the configuration value and that measured condition is not fully specified, the two are retained as distinct configurations.

The Raspberry Pi procedure began with a benign UDP transfer generated with **iperf3**. Omitting the private address, the command was **iperf3 -c <victim-IP> -u -b 10M -t 20 -l 1200**. It configured UDP traffic at 10 Mbit/s for 20 s with the **-l 1200** datagram-length parameter. This condition established the documented normal-traffic behavior and whether the active event path was triggered at that load.

The controlled flood used the same generator and datagram setting with **iperf3 -c <victim-IP> -u -b 100M -t 20 -l 1200**. The configured sending rate was 100 Mbit/s for 20 s. Because the physical experiment used a single controlled traffic generator, it is described as a UDP flood or DDoS-like volumetric workload rather than as a literal distributed botnet. Its purpose was to stress the receiving and mitigation path reproducibly, not to reproduce Internet-scale source diversity.

Baseline and mitigated conditions recorded packet behavior, legitimate delivery, resources, responsiveness, and events. On the Pi, XDP counted UDP packets, emitted an event, and returned **XDP_PASS**; the controller then installed and logged the source-blocking rule.

The environments implement the same comparison concept but are not identical benchmark hosts; results are reported as representative controlled observations.

Workload / environment	Protocol	Configuration	Purpose
Benign application traffic; Docker	HTTP, MQTT	Low rate; configuration lists 50 requests/s; duration not specified	Service-access check
UDP flood; Docker	UDP	15 kpps per attacker (up to three); duration not specified	Controlled attack traffic
Benign UDP; Raspberry Pi	UDP	10 Mbit/s; 20 s; 1200-byte length setting	Normal delivery and trigger check
Controlled UDP flood; Raspberry Pi	UDP	100 Mbit/s; 20 s; 1200-byte length setting	Volumetric stress and mitigation comparison

Table 5: Principal traffic workloads used in the evaluation

5 Results

This section reports Docker and Raspberry Pi observations at the precision supported by the experimental records; distinct configurations are not merged.

5.1 Docker Evaluation

The Docker experiments showed a clear difference between the mitigated and unmitigated conditions under high-rate UDP traffic. With mitigation active, most traffic identified as malicious was suppressed and the victim service remained available. In the unmitigated condition, the reported workload impaired the victim and the cross-environment summary characterizes the service as unresponsive. Legitimate application traffic in the mitigated summary condition was reported as essentially unchanged, with no application packet loss recorded in that comparison.

The Docker records contain more than one traffic configuration. A lower-rate plotted scenario shows the victim received approximately 1,000 packets per second without mitigation; following the trigger in that plotted sequence, nearly 700 malicious packets per second were reported as dropped. Separately, the cross-environment summary reports a representative UDP flood condition of approximately 100,000 packets per second and approximately 99% malicious packet mitigation. The latter values are used as the principal Docker summary in this article and in Figure 3; the lower-rate plot is not combined with them as though all values came from the same run.

CPU observations also appear in two reporting contexts. In the detailed Docker plot, the unmitigated condition is described at approximately 45% CPU utilization, whereas the active condition is shown at approximately 12%. The principal cross-environment table instead reports filtering CPU at approximately 30% of one host core for its representative Docker condition. These values are retained as distinct observations because the records do not establish that they share a packet rate or run. No benign trigger was reported, but the tests do not define a false-positive probability.

Latency and reachability followed the same descriptive pattern. The detailed plot reports ping latency near 4 ms without mitigation and approximately 1 ms with the filter active in that plotted scenario. The separate summary table characterizes added latency for the representative Docker condition as approximately 1–2 ms. Accordingly, the principal result is reported as a 1–2 ms mitigation-associated increase for the summary workload, while the absolute ping observations remain specific to the detailed plot. The service was reported as responsive with mitigation and unavailable under the unmitigated representative attack.

Figure 3 summarizes the representative 99% mitigation, 30%-of-one-core filtering CPU, 1–2 ms added latency, and responsive service.

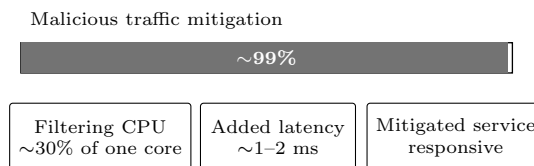


Fig. 3 Principal Docker mitigation outcomes under the reported representative workload

5.2 Raspberry Pi Evaluation

The Raspberry Pi evaluation began with the documented benign UDP condition. The configured transfer rate was 10 Mbit/s for 20 s with the `iperf3` 1200-byte length setting. The measurements report approximately 20,833 packets processed during this test, with no packet loss and no mitigation trigger observed. This observation is limited to the recorded benign workload; it is not used to calculate a general false-positive rate.

The controlled flood used a configured sending rate of 100 Mbit/s for 20 s with the same length setting. More than 208,000 packets were reported for that experiment, and the source triggered the detection and mitigation workflow. A separate comparison reports a representative Raspberry Pi attack rate of approximately 30,000 packets per second in its cross-environment summary. The configured bit rate, total packet count, and summarized packet rate are separate descriptions. They are not converted into one another because the complete framing and measurement basis needed for an exact conversion is not documented.

During the flood, the trace event, source-blocking rule, log, and notification were observed in the documented order; component-level timing was not reported.

Under the representative condition of approximately 30 kpps, the summarized comparison reports that approximately 94% of malicious packets were mitigated. More than 90% of legitimate throughput was maintained, and the Raspberry Pi remained responsive with minor packet loss. Under the corresponding unmitigated heavy load, the device was reported as becoming unresponsive. The detailed prose similarly records loss of network connectivity in an unmitigated trial and continued operation with mitigation active. The principal hardware result is therefore reported as approximately 94%, consistent with the explicit cross-environment comparison.

CPU load during the mitigated Raspberry Pi condition was substantial. The cross-environment table reports filtering CPU at approximately 85% of one core. A separate per-core observation records approximately 72%, 81%, 45%, and 59% utilization for Cores 1–4, respectively, with mitigation active. For the unmitigated condition, the same comparison describes all four cores at approximately 98% and the device as unresponsive. Figure 4 reproduces only these explicitly reported representative values. They are individual observations, not averages over a stated number of trials.

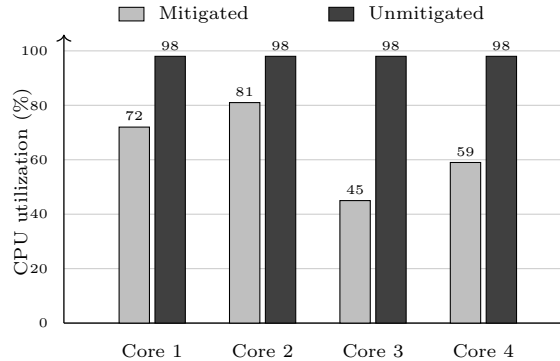


Fig. 4 Representative Raspberry Pi CPU utilization under mitigated and unmitigated attack conditions. Values are representative observations, not averages across repeated trials

Map memory was reported below approximately 1 MB, with no kernel crash during the mitigated test. This excludes controller, operating-system, and application memory and is not a long-duration stability measure.

Two latency quantities are reported for the Raspberry Pi. During the mitigated attack, the detailed prose gives average observed ping latency of approximately 4–5 ms. The cross-environment summary separately characterizes the additional mitigation-associated latency as approximately 3–4 ms. These values are not combined: the former is an absolute ping observation during attack, whereas the latter is the summarized added latency. Without mitigation, pings were reported as timing out or showing very high latency.

An approximate one-second trigger time is reported, but no per-run latency series is available. It is therefore not treated as a measured latency distribution or conflated with the 30-second controller cooldown.

5.3 Management-Relevant Cross-Environment Comparison

Table 6 gives the principal comparison. Relative to Docker, the Pi has a lower representative rate and mitigation proportion and higher filtering CPU and latency. The environments are not normalized hardware benchmarks.

Metric	Docker	Raspberry Pi 4
Representative attack rate	~100 kpps UDP	~30 kpps UDP
Malicious traffic mitigation	~99%	~94%
Mitigated service	Responsive; no application packet loss in summary	Responsive; minor packet loss
Unmitigated service	Unresponsive	Unresponsive
Filtering CPU	~30% of one host core	~85% of one Pi core
Added latency	~1–2 ms	~3–4 ms
Legitimate traffic effect	Essentially unchanged	More than 90% throughput maintained
Observed benign triggers	None reported in documented tests	None in the documented 10-Mbit/s benign test
Memory observation	Not reported comparably	eBPF maps below ~1 MB

Table 6: Summary of principal experimental outcomes.

The comparison characterizes transfer from a virtualized environment to a constrained platform; no cross-platform degradation factor is calculated.

For management purposes, the comparison shows why a virtual validation stage cannot be used as a direct capacity model for an edge deployment. The same general control concept remained effective on the physical device, but its resource margin narrowed substantially. Admission of the mechanism into an operational edge node should therefore be based on platform-specific CPU and latency budgets rather than on mitigation percentage alone.

Table 7 maps the observations to the design goals. G4 is only partially satisfied on the Pi because filtering used substantial CPU; the remaining goals were observed for the tested workloads, subject to the measurement limitations in Section 7.

Goal	Evidence	Outcome within evaluated conditions
G1	Threshold events occurred during controlled floods	Trigger behavior observed; exact latency distribution not reported
G2	~99% Docker and ~94% Raspberry Pi mitigation	Attack suppression observed for the tested workloads
G3	Docker traffic essentially unchanged; Pi maintained more than 90%; no trigger in documented benign Pi test	Legitimate traffic largely preserved
G4	Docker ~30% of one core; Pi ~85% of one core; Pi maps below ~1 MB	Satisfied in Docker; partially satisfied under tested Pi load
G5	Both mitigated systems remained responsive	Operational continuity observed under evaluated attacks

Table 7: Observed outcomes relative to the design goals.

6 Discussion and Management Implications

The results are interpreted here as a network and systems management problem rather than solely as a filtering benchmark. The relevant operational question is whether a local security-control loop can observe abnormal load, trigger policy, actuate a response, and preserve the managed service within the resource budget of the edge node.

6.1 Interpretation of the Results

The evaluation addressed whether an eBPF/XDP-assisted mechanism could improve the availability of a constrained Linux edge device during a volumetric UDP workload without consuming so many local resources that the defense became a second source of service degradation. Mitigation remained functional after transfer from Docker to a physical Raspberry Pi 4, and the protected service remained reachable in both mitigated conditions; the corresponding unmitigated conditions were reported as unresponsive. Continued operation is therefore important even though the hardware mitigation percentage was lower.

The Docker and Raspberry Pi observations should not be interpreted as a controlled performance benchmark between equivalent systems. They differ in processor capacity, network path, interface behavior, traffic generation, and documented loading and enforcement details. Their value is instead sequential: the Docker stage establishes behavior in a controlled virtual environment, while the physical stage shows how the same general detection-and-response concept behaves when CPU headroom and packet-processing capacity are more restricted. The difference between their mitigation values consequently marks outcomes under distinct deployment contexts, not a measured platform penalty.

The Raspberry Pi result is operationally relevant because incomplete mitigation coincided with more than 90% legitimate throughput and continued responsiveness. A local defense can improve availability without removing every malicious packet if it reduces the work reaching protected services sufficiently to preserve useful operation. This interpretation remains limited to the reported workload and provides no guarantee for other platforms or attack intensities.

The results also clarify the role of eBPF/XDP in the evaluated implementation. The XDP program performed early packet observation, protocol selection, and per-source counting through a BPF map. It emitted a kernel event when its counter condition was reached and returned `XDP_PASS`. A Python controller then interpreted the event and installed an `iptables` source-blocking rule. The measured outcome is consequently attributable to the complete hybrid path, not to direct XDP dropping. Early monitoring kept the kernel program compact, while user space accommodated address conversion, policy action, logging, cooldown management, and notification.

This division has an architectural cost: mitigation begins only after event delivery, controller processing, and rule insertion, making the path longer than a map-driven `XDP_DROP` decision could be. The experiments did not isolate this control-path delay. In return, policy and operational functions can be changed in user space without enlarging the restricted per-packet program.

Security effectiveness cannot be interpreted independently of resource use. On the Pi, filtering consumed approximately 85% of one core and added approximately 3–4 ms of latency, leaving limited processing headroom despite continued service. Packet mitigation, service behavior, CPU, memory, and latency must therefore be considered together.

The absence of a mitigation trigger during the documented benign Pi transfer is encouraging for that workload but does not establish general discrimination between legitimate and malicious sources. The detector used source packet rate, not semantic knowledge of an IoT botnet or application intent. A normal device producing a sufficiently large burst can satisfy the same counter condition, while attackers can alter their source distribution or rate. The legitimate-traffic result should thus be understood as configuration-specific evidence that the selected test traffic remained usable, not as a measured false-positive probability for arbitrary IoT applications.

The observed behavior supports use of the mechanism as a local protection layer when thresholds and resource demand are matched to the platform. Its protection remains bounded by CPU capacity, response-path design, traffic characteristics, and the simplicity of the rate detector.

6.2 Security–Performance and Resource-Management Trade-Off

The central trade-off is between spending local processing capacity on defense and preserving capacity for the edge service. Without mitigation, the tested heavy workload was reported to make the Raspberry Pi unresponsive. With mitigation, the device remained operational but filtering consumed approximately 85% of one core. In absolute terms, this is a substantial CPU load. It is more accurate to describe the mechanism as resource-aware than as low-overhead under attack: the resource cost was acceptable within the evaluated workload because useful service continued, but CPU headroom was limited.

The representative per-core readings confirm that considerable packet and kernel work continued after mitigation. The relevant comparison is therefore not whether defense consumed resources, but whether that consumption produced a better service outcome than processing the workload without an active response.

Memory and CPU show different aspects of the cost. The BPF map memory was reported as below approximately 1 MB for the tested Pi configuration, indicating that the counter state itself occupied a small amount of memory. This figure excludes controller and operating-system memory but indicates that map storage was not the dominant reported constraint. Processing was more demanding, although the experiment did not decompose CPU time among packet handling, events, controller work, firewall management, and service activity.

Latency provides a second measure of the performance cost. The Pi summary reports approximately 3–4 ms of added latency, while more than 90% of legitimate throughput was preserved. For the evaluated service, this combination was compatible with continued responsiveness. Other edge applications may place different weight on delay and throughput. A monitoring gateway may tolerate several additional milliseconds, whereas a tightly timed control loop may not. Deployment decisions should therefore use service-level latency requirements rather than treating the measured value as universally small or large.

The hybrid enforcement path also affects the trade-off. Keeping packet inspection and counting in XDP reduces the amount of complex logic executed at the early hook. Moving blocking decisions, logging, and notification to user space simplifies policy management and makes integration with existing Linux tools straightforward. Conversely, events and rule insertion add control-path work and leave a period before the source rule becomes active. Direct map-driven dropping in XDP could shorten this path, but it would move more state and policy responsibility into the kernel program. That alternative is a subject for future evaluation, not a property of the measured architecture.

Rate-oriented detection trades classification richness for bounded per-packet work: it uses simple state, requires no payload inspection, and fits early observation. The same simplicity permits legitimate bursts to cross a threshold and allows low-rate, distributed, spoofed, or rotating sources to evade or stress source-specific blocking. It neither identifies botnets semantically nor addresses application-layer or low-and-slow attacks. The mechanism is therefore a first-line edge component that complements service-specific and upstream controls.

The balance is conditional. Under the tested Raspberry Pi workload, substantial processing preserved service operation and legitimate throughput; at higher rates, with more sources, or beside a CPU-intensive application, the remaining headroom may be insufficient. Resource monitoring and conservative threshold tuning are therefore deployment requirements.

6.3 Operational Deployment and Management Considerations

The evaluated mechanism is best placed as local first-line protection on a Linux-based IoT gateway, edge node, or endpoint that can load eBPF programs and manage host firewall rules. It can react after traffic has reached the local interface, thereby complementing rather than replacing upstream filtering. For attacks that saturate the access link before packets reach the device, a local mechanism cannot recover consumed upstream bandwidth; coordination with network-provider or cloud mitigation remains necessary.

Threshold selection is deployment-specific. Expected legitimate rates, service burst behavior, device CPU capacity, link speed, and the consequences of blocking a source all affect an appropriate setting. The conceptual one-second/800-pps example and the runtime multiple-of-50 event behavior document different implementation descriptions and should not be adopted as universal defaults. A production configuration would require baseline observation and testing with legitimate peaks, including software updates, telemetry uploads, and concurrent client activity. Temporary blocks and explicit expiry policies can reduce the operational impact of an incorrect trigger.

Interface and driver behavior also matter. XDP performance varies with native driver support and execution mode, and not every Linux network interface offers the same attachment capabilities [16, 17]. GRO and LRO were disabled in the Raspberry Pi experiment because receive aggregation could alter packet-counting behavior. A deployment must decide whether to preserve offload performance or to require packet-level visibility, and it must validate that the resulting configuration matches the detector’s counting assumptions. Changes to kernel, driver, or NIC configuration should prompt renewed measurement rather than an assumption that the reported costs remain unchanged.

The user-space controller offers an integration point for expiring stale blocks, deduplicating rules, exporting telemetry, coordinating gateways, or requesting upstream mitigation. These untested extensions introduce their own reliability and security requirements; rule updates in particular should be idempotent and bounded.

Telegram was an implementation-specific management choice, not a dependency of detection or mitigation. Any remote notification should be asynchronous, keep credentials out of source and logs, and fail without disabling the local response.

The mitigation components themselves form part of the trusted computing base. Loading and attaching eBPF programs requires controlled privileges; the controller can modify firewall policy; and its event input can initiate security actions. Verifier and runtime research also supports treating eBPF as part of the trusted computing base rather than assuming complete safety [18, 19]. Program binaries, configuration, and rule-management paths should therefore be protected against unauthorized changes. Least-privilege execution, restricted command interfaces, authenticated updates, log protection, and secret management are important deployment controls. If the edge host or controller is compromised, an attacker may disable monitoring, inject blocks, or suppress alerts.

The management architecture also suggests a natural path toward broader orchestration. The current controller acts only on the local host, but its event and resource signals could feed an external manager or SDN/NFV control layer for policy coordination, temporary rule expiry, escalation to upstream filtering, or migration of mitigation functions. These extensions are not evaluated here; they follow from the separation between the monitoring path and the user-space control plane.

Table 8 summarizes these implications. Deployment should combine resource monitoring, tested fallback behavior, and upstream coordination rather than treating one packet-rate rule as a complete defense.

Consideration	Benefit	Constraint or deployment implication
Early packet observation	Compact processing before much of the network stack	Depends on kernel, driver, and XDP mode
Per-source state	Simple counters and targeted response	Bursts, spoofing, distributed sources, and map pressure
User-space controller	Flexible policy and system integration	Longer response path and privileged control component
iptables enforcement	Familiar Linux rule management	Rule insertion, expiry, duplication, and ruleset growth require control
Threshold selection	Low-complexity trigger	Must reflect legitimate bursts, service needs, and device capacity
NIC/offload configuration	Packet-level observations can be made consistent	Offload choices affect counting and performance
Resource budget	Local defense preserves service when capacity is sufficient	CPU headroom may be limited under attack
Alerting and logging	Operational visibility and audit trail	Must be asynchronous, protected, and free of embedded secrets

Table 8: Practical deployment considerations for eBPF/XDP-assisted edge mitigation.

7 Limitations and Threats to Validity

7.1 Experimental Scope

Volumetric UDP flooding is the principal workload, and the physical experiment used one controlled generator rather than an Internet-scale population of compromised devices. It therefore does not reproduce source diversity, routing variability, background traffic, or upstream congestion. Although the Docker setup supports other traffic, the reported cross-platform results do not support conclusions about additional attack types.

The experiments were short and used a limited set of operating conditions. They show the response during the documented intervals but do not characterize thermal throttling, memory growth, rule accumulation, controller recovery, or service behavior over hours or days. The hardware evidence comes from one Raspberry Pi 4 Model B. It should not be generalized automatically to microcontrollers without Linux, other ARM boards, Wi-Fi-only devices, servers, or provider-scale infrastructure.

7.2 Measurement and Statistical Validity

The experiment records do not specify a fixed number of independent repetitions for every condition, randomized ordering, confidence intervals, or hypothesis tests. The representative observations and approximate summaries support a controlled systems evaluation, but not conclusions about variance, repeatability, or population-level statistical significance.

Some values originate from different reporting contexts. The detailed Docker plot and cross-environment summary describe different packet rates and CPU observations, while the Raspberry Pi bit-rate command, total packet count, and summarized packet rate are not interchangeable measurements. These values are kept separate, and the explicit comparison summary provides the principal cross-environment claims. Raw time series and packet traces would permit a stronger independent reconstruction of the metrics.

7.3 Internal Implementation Consistency

The implementation records contain differences between conceptual design descriptions and the documented runtime path. The conceptual design includes direct `XDP_DROP`, whereas the runtime program emits threshold events, returns `XDP_PASS`, and relies on a Python-controlled `iptables` rule for physical enforcement. Loader records refer to both native XDP/libbpf tooling and a Python/BCC interface, while interface identifiers and threshold formulations also differ across configuration records.

The paper consequently reports only the confirmed common path: eBPF/XDP monitoring, per-source state, kernel event generation, user-space control, and `iptables` enforcement on the Raspberry Pi. The interface name is omitted because it is not material to the procedure, and no single loader is asserted. This preserves internal consistency but limits exact replication of attachment and threshold configuration.

7.4 Construct Validity

The selected measurements are useful proxies but do not fully represent every concept named in the paper. Device responsiveness is based on reachability and service behavior, not a formal availability service-level agreement. No mitigation trigger in one benign workload is not a measured false-positive probability. Packet mitigation percentage describes suppression under the reported workload; it is not equivalent to complete protection from DDoS.

CPU readings are representative observations rather than long-run averages, and the reported BPF map memory does not include total controller, firewall, kernel, or application memory. Ping latency reflects network reachability and round-trip delay but is not a measurement of every application’s response time. Legitimate throughput preservation captures delivery during the tested traffic pattern but does not establish fairness among multiple legitimate clients. Detection events establish that the threshold path ran, while the source does not provide a rigorous distribution of detection or enforcement latency.

7.5 External Validity

Generalization is limited by the Raspberry Pi hardware, Raspberry Pi OS and kernel version, local network, traffic generator, driver behavior, disabled GRO/LRO configuration, and selected thresholds. XDP availability and performance depend on the receiving interface and attachment mode [17, 20]. An edge platform with a different NIC, kernel, or service workload may reach a different CPU limit or show different latency and mitigation behavior. Ambient temperature, power configuration, background processes, and service load can further alter the available processing headroom, but these factors were not varied systematically in the documented experiment.

Broader validation should cover other edge boards, Linux gateways, NICs, drivers, kernels, distributed sources, background traffic, services, longer durations, and repeated trials.

7.6 Security Limitations

Per-source rate detection can be evaded or stressed in several ways. Spoofed addresses can cause blocks to target unrelated sources or make address-based response ineffective. Attackers can rotate addresses, distribute load among many sources below the threshold, or use low-rate traffic that accumulates cost at the application layer. A large number of unique source keys can create state-management pressure. Fixed or simple counter conditions can also be crossed by legitimate bursts and may be evaded at window boundaries, depending on the final threshold implementation.

The mechanism does not inspect application semantics or encrypted payloads and does not address application-layer floods merely because they arrive over a monitored interface. It also assumes that the edge host, eBPF program, controller, and firewall-management path remain trustworthy. Host compromise could allow an attacker to disable monitoring, alter counters, remove rules, or misuse privileged enforcement. These limitations position the mechanism as one layer in a broader defense rather than a self-contained solution.

Table 9 summarizes the principal threats and the future work needed to reduce them.

Category	Threat	Effect on interpretation	Future mitigation
Scope	Limited workload diversity	Results primarily support volumetric UDP claims	Test mixed protocols and attack classes
External	Single hardware platform	Hardware generalization is limited	Evaluate additional boards, NICs, and kernels
Temporal	Short-duration experiments	Long-run stability and rule growth are unknown	Run longer endurance tests
Statistical	No fixed repeated-trial design	Variance and confidence are not quantified	Automate repetitions and report intervals
Security	Source-rate detector limitations	Spoofed, rotating, or distributed sources may evade or stress control	Add aggregate and adaptive detection
Internal	Record differences	Exact loader, interface, and threshold replication remains uncertain	Reconcile code and configuration records
Measurement	NIC/offload dependence	Counting and performance vary with interface configuration	Test modes and document offloads explicitly

Table 9: Main threats to validity and their effect on interpretation.

8 Conclusion and Future Work

This study examined whether eBPF/XDP-assisted packet processing could improve the resilience of a resource-constrained Linux edge device during volumetric UDP traffic while preserving legitimate service and maintaining manageable resource use. The evaluated architecture combined early eBPF/XDP monitoring and per-source counting with kernel event delivery, a Python user-space controller, and dynamic source blocking through `iptables`.

The Docker evaluation reported approximately 99% malicious traffic mitigation and continued service responsiveness. On the Raspberry Pi 4, approximately 94% of malicious packets were mitigated at representative rates up to 30 kpps, while more than 90% of legitimate throughput was maintained and the device remained responsive. The hardware result carried a substantial cost: filtering was associated with approximately 85% of one core and approximately 3–4 ms of added latency. Within this workload, the hybrid defense improved service continuity but was resource-aware rather than overhead-free.

The Docker and hardware environments were not equivalent benchmark platforms. Their comparison nevertheless shows that virtual evaluation alone does not characterize the processing headroom of a constrained physical device.

The approach is most appropriately viewed as a local first-line security-management layer. It can reduce work reaching a protected endpoint after traffic reaches the local interface, but it cannot recover upstream bandwidth already consumed by a flood. Its source-rate detector does not provide semantic botnet detection and does not cover spoofed, distributed low-rate, or application-layer attacks. The absence of a trigger during the documented benign test is configuration-specific and should not be generalized to a zero false-positive probability.

Future work should compare the hybrid path with map-driven enforcement that returns `XDP_DROP`. Direct enforcement could shorten the response path, but it was not the evaluated runtime architecture; comparison should measure mitigation, response delay, CPU, rule-management cost, and legitimate traffic.

Rolling windows, exponentially weighted moving averages, per-service baselines, and adaptive thresholds could better accommodate changing legitimate traffic. Per-prefix state, heavy-hitter algorithms, or compact sketches could address distributed behavior below individual thresholds, subject to verifier, memory, CPU, and state-exhaustion evaluation.

Broader evaluation should include TCP SYN, ICMP, mixed-protocol, low-rate, and distributed-source workloads on additional ARM boards, Linux gateways, kernels, drivers, and native-XDP NICs. Repeated,

longer trials should export raw metrics and report variance and confidence intervals. Automated workloads and versioned manifests should record kernel, driver, offload, threshold, controller, and firewall state.

The control plane also requires least privilege, authenticated configuration, bounded and expiring firewall updates, protected event delivery, and external secret management. Upstream integration could let the detector contribute local evidence without acting as the sole defensive layer.

Implementation materials associated with this study are available in the accompanying public repository: <https://github.com/AbdurrahmanTolay/ebpf-iot-ddos-thesis>. The current repository includes the eBPF program, Python controller, and supporting documentation and configuration examples.

Statements and Declarations

Funding. No funding was received for conducting this study.

Competing interests. The author declares no competing interests relevant to the content of this article.

Author contributions. Abdurrahman Tolay: Conceptualization, methodology, software, validation, investigation, data curation, formal analysis, visualization, writing—original draft, and writing—review and editing.

Data and code availability. The implementation and supporting documentation are available at <https://github.com/AbdurrahmanTolay/ebpf-iot-ddos-thesis>. The repository contains the eBPF program, Python controller, and configuration examples. A separate public archive of the original raw time-series measurements and packet traces is not available.

Ethics approval. Not applicable. The study did not involve human participants, animals, clinical data, or personal data.

References

- [1] Simone Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks*, 76:146–164, 2015. <https://doi.org/10.1016/j.comnet.2014.11.008>.
- [2] Felisberto Pereira, Ricardo Correia, Pedro Pinho, Sérgio Ivan Lopes, and Nuno Borges Carvalho. Challenges in resource-constrained IoT devices: Energy and communication as critical success factors for future IoT deployment. *Sensors*, 20(22):6420, 2020. <https://doi.org/10.3390/s20226420>.
- [3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, 2017. USENIX Association. ISBN 978-1-931971-40-9.
- [4] Jeroen van der Hooft, Maxim Claeys, Niels Bouten, Tim Wauters, Jürgen Schönwälder, Aiko Pras, Burkhard Stiller, Marinos Charalambides, Rémi Badonnel, Joan Serrat, Carlos R. P. dos Santos, and Filip De Turck. Updated taxonomy for the network and service management research field. *Journal of Network and Systems Management*, 26(3):790–808, 2018. <https://doi.org/10.1007/s10922-017-9443-y>.
- [5] Angelo Feraudo, Diana Andreea Popescu, Poonam Yadav, Richard Mortier, and Paolo Bellavista. Mitigating IoT botnet DDoS attacks through MUD and eBPF based traffic filtering. In *Proceedings of the 25th International Conference on Distributed Computing and Networking*, pages 164–173, New York, NY, USA, 2024. Association for Computing Machinery. <https://doi.org/10.1145/3631461.3631549>.
- [6] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 53(1):16:1–16:36, 2020. <https://doi.org/10.1145/3371038>.
- [7] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: Fast programmable packet processing in the

- operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies*, pages 54–66, New York, NY, USA, 2018. Association for Computing Machinery. <https://doi.org/10.1145/3281411.3281443>.
- [8] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan R  th, and Klaus Wehrle. Demystifying the performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 208–212. IEEE, 2019. <https://doi.org/10.1109/NETSOFT.2019.8806651>.
 - [9] Sebastiano Miano, Fulvio Risso, Mauricio V  squez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021. <https://doi.org/10.1109/TNSM.2021.3055676>.
 - [10] Hassan Jalil Hadi, Mubashir Adnan, Yue Cao, Faisal Bashir Hussain, Naveed Ahmad, Mohammed Ali Alshara, and Yasir Javed. iKern: Advanced intrusion detection and prevention at the kernel level using eBPF. *Technologies*, 12(8):122, 2024. <https://doi.org/10.3390/technologies12080122>.
 - [11] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017. <https://doi.org/10.1109/MC.2017.201>.
 - [12] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004. <https://doi.org/10.1145/997150.997156>.
 - [13] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys & Tutorials*, 15(4):2046–2069, 2013. <https://doi.org/10.1109/SURV.2013.031413.00127>.
 - [14] Rohan Doshi, Noah Apthorpe, and Nick Feamster. Machine learning DDoS detection for consumer Internet of Things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 29–35. IEEE, 2018. <https://doi.org/10.1109/SPW.2018.00013>.
 - [15] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzk, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084. Association for Computing Machinery, 2019. <https://doi.org/10.1145/3314221.3314590>.
 - [16] Federico Parola, Roberto Procopio, and Fulvio Risso. Assessing the performance of XDP and AF_XDP based NFs in edge data center scenarios. In *Proceedings of the 17th International Conference on Emerging Networking Experiments and Technologies*, pages 481–482. Association for Computing Machinery, 2021. <https://doi.org/10.1145/3485983.3493352>.
 - [17] Kyle A. Simpson, Chris Williamson, Douglas J. Paul, and Dimitrios P. Pezaros. GALETTE: A lightweight XDP dataplane on your Raspberry Pi. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2023. <https://doi.org/10.23919/IFIPNetworking57963.2023.10186412>.
 - [18] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of Linux eBPF subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 87–92. Association for Computing Machinery, 2023. <https://doi.org/10.1145/3609510.3609822>.
 - [19] Hsin-Wei Hung and Ardalan Amiri Sani. BRF: Fuzzing the eBPF runtime. *Proceedings of the ACM on Software Engineering*, 1(FSE):1152–1171, 2024. <https://doi.org/10.1145/3643778>.
 - [20] Milo Craun, Adam Oswald, and Dan Williams. Enabling eBPF on embedded systems through decoupled verification. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 63–69. Association for Computing Machinery, 2023. <https://doi.org/10.1145/3609021.3609299>.