Distributed, Linux Kernel Integrated Security Framework for Real-Time Prevention of DNS Data Exfiltration

Vedang Parasnis

Submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2025

Project Committee:

Dr. Geetha Thamilarasu, Chair

Dr. Munehiro Fukuda, Committee Member

Dr. Robert Dimpsey, Committee Member

Program Authorized to Offer Degree:

Computer Science and Systems

University of Washington

Abstract

Distributed, Linux Kernel Integrated Security Framework for Real-Time Prevention of DNS Data Exfiltration

Vedang Parasnis

Chair of the Supervisory Committee:
Committee Chair Dr. Geetha Thamilarasu
Computing & Software Systems

Data exfiltration via Domain Name System (DNS) remains a critical and sophisticated threat, especially for hyperscalers managing massive AI workloads and sensitive client data across distributed cloud environments. DNS is inherently vulnerable due to its ubiquity and fundamental role in enterprise networks, contributing to an average data breach cost exceeding $4.8 million (IBM 2024 Data Breach Report). DNS-based exfiltration, particularly challenging to detect in real-time across scaled infrastructures, poses catastrophic and stealthy risks.

This project introduces a novel, scalable framework for real-time prevention of DNS-based data exfiltration, neutralizing even advanced Command-and-Control (C2) techniques. This system employs lightweight, agent-based, endpoint-centric enforcement, leveraging deep packet inspection via eBPF directly within the Linux kernel's network stack. These kernel-resident programs enable high-speed, per-packet parsing of outbound DNS traffic, with userspace dense neural networks performing real-time lexical analysis to detect obfuscated or malicious payloads. Active defense is enforced directly within the operating system by integrating with the core kernel security layer, allowing immediate termination of malicious processes upon detection to accelerate incident response and contain compromise at the source.

Telemetry, including process identifiers, lifetimes, and DNS usage patterns, is exported per node to Kafka brokers, enabling dynamic domain blacklisting and enforcement of DNS policies across the network. The framework scales horizontally and enforces real-time, cross-node security policies without relying on external middleware or firewalls. Experimental results confirm its scalability and suitability for production DNS security. By disrupting sophisticated command-and-control (C2) attacks - including those modeled by top-ranked adversary emulation frameworks in the C2 Matrix - the system terminates highly evasive processes directly from the kernel. This research and implementation has earned recognition from the Linux kernel community, leading security researchers, and premier security conferences. Combining deep in-kernel inspection, AI-assisted detection, dynamic policy enforcement, and process-level active defense, this framework reduces attacker dwell time, blocks lateral movement, and improves both visibility and response across large-scale cloud and on-premise environments.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation and Goals

Modern threat actors continuously evolve, employing increasingly sophisticated techniques and covert communication channels to maintain persistence on compromised systems and exfiltrate data before detection or remediation. A common entry point in such attacks involves the deployment of lightweight implants or command-and-control (C2) clients. These are often compiled in formats like COFF (Common Object File Format) and delivered to targeted endpoints through phishing campaigns, social engineering, or other initial access vectors. Once a system is compromised, these implants use beacon intervals, strong encryption, and protocol tunneling to remain hidden, effectively bypassing volumetric and time-based detection mechanisms at the firewall. This silent phase of data exfiltration is both stealthy and resilient, allowing adversaries, such as advanced persistent threats (APTs), to maintain long-term control, steal undetected sensitive data, and move laterally within the network. The Domain Name System (DNS) remains one of the most effective channels for attackers to run covert C2 communication and exfiltrate data. As a core protocol responsible for domain-to-IP resolution, business operations, and service discovery, DNS is rarely deeply monitored or filtered at firewalls, making it an ideal backdoor, offering attackers a discreet pathway for unauthorized data transfer and remote command execution on infected systems. This exploitation can cause massive damage to enterprises, as demonstrated by some of the cyber-espionage groups. Hexane, a major threat actor in the Middle East and Asia, used a custom system called DNSsystem to stealthily exfiltrate data from energy and telecom sectors through encrypted DNS tunnels, beacon obfuscation, and adaptive payloads. Likewise, MoustachedBouncer leveraged the Nightclub implant to exploit DNS redirection at the

ISP level, using DNS as a resilient covert channel for long-term espionage in eastern Europe and Central Asia. These campaigns have compromised state institutions and critical infrastructure, underscoring the scale and sophistication of DNS-based threats. Existing solutions primarily rely on passive analysis techniques such as anomaly detection, domain reputation scoring, and static blacklists. However, these approaches are inherently reactive, slow to respond, and often ineffective against stealthy adaptive APT malware. As a result, they offer no guarantees of preventing data loss before exfiltration occurs: By time detection triggers, malicious commands may have already been executed, and significant damage is inflicted. To address these limitations and the evolving sophistication of DNS exfiltration attack vectors, this project aims to develop a robust endpoint-centric defense mechanism that enforces DNS security from within the operating system. The design and implementation are guided by the following core goals:

- Enforce DNS exfiltration protection from within the Linux kernel to enable in-line, real-time traffic inspection and eliminate reliance on external security middleware.
- Instantly detect and terminate malicious implants and DNS-based C2 communication, reducing response time without manual intervention.
- Enable adaptive detection of obfuscated exfiltration techniques by integrating userspace deep learning with kernel-level enforcement.
- Provide distributed scalable enforcement by streaming threat events across nodes for dynamic network Layer 3 policy enforcements paired with domain blacklisting for Layer 7 filters.
- Detect and disrupt advanced DNS-based C2 techniques, including Domain Generation Algorithms (DGAs), remote execution, and process side channeling, to protect against a broader class of stealthy threat vectors beyond exfiltration.

Chapter 2

BACKGROUND

This chapter provides a comprehensive overview of the internals of the Linux kernel network stack, the role of eBPF for dynamic security enforcement within the kernel, the complexity and types of DNS data exfiltration, and the fundamental limitations of the DNS protocol that enable various forms of attacks.

2.1 eBPF

The extended Berkeley Packet Filter (eBPF), introduced in Linux kernel 3.15 (2014), is a general-purpose virtual machine in the kernel evolved from the classic BPF [17]. Unlike kernel modules, which risk destabilizing the system, eBPF safely injects verified code into the kernel, enabling dynamic programmability without compromising stability or security. eBPF surpasses other programmable data path technologies such as P4 [11] and DPDK [28], which operate outside the kernel or lack visibility into kernel-level security subsystems. These models cannot access core primitives such as process identity or Linux's Mandatory Access Control (MAC) layers, limiting their ability to enforce deep, context-sensitive security policies. eBPF, on the contrary, operates directly within the kernel network stack and security layers, allowing high-resolution enforcement and real-time analysis of malicious traffic. eBPF programs are written in a restricted C subset, compiled via LLVM to a platform-independent bytecode, and executed by a RISC-like in-kernel VM. The execution model enforces strict safety: a 512-byte stack, bounded loops, 11 64-bit registers, and a cap of one million instructions. Before loading, the BPF verifier ensures control flow integrity and memory safety. Programs are Just-In-Time (JIT) compiled for performance and executed within a sandboxed environment. A core strength of eBPF is its use of BPF maps—persistent, kernel-resident key-value stores that support data structures such as LRU caches, stacks, and queues for

efficient state management and data sharing. In addition, eBPF maps support pinning to the BPF filesystem, allowing data to persist beyond the lifecycle of the userspace program that loaded the eBPF code. All interactions are mediated via the `bpf()` syscall, guarded by `CAP_BPF` or `CAP_SYS_ADMIN` for complex programs which are reserved for privilege users.

2.2 Linux Kernel Network Stack

The Linux kernel network stack processes packets as they traverse ingress and egress paths. Each packet is represented in memory as a socket buffer (SKB), a metadata-rich structure that the kernel manipulates by advancing internal head pointers. SKBs enable layered processing, allowing headers and payloads to be parsed and modified efficiently. Packets go through multiple stages, classification, filtering, scheduling, and forwarding, across different network interfaces via dedicated receive (RX) and transmit (TX) queues. These queues may be managed by software in the kernel or by hardware through NIC drivers, enabling flow processing as packets move up or down the network stack [25]. Egress is especially critical for preventing exfiltration. When a userspace process writes to a socket, the kernel routes the resulting SKB through the TCP/IP stack, Netfilter (link layer), Traffic Control (TC) for shaping and classification, and finally to the NIC driver for transmission. Each stage of packet processing supports runtime hook injection via eBPF, enabling event-driven triggers as packets traverse the stack in either direction. This flexibility allows eBPF programs to be attached at various points, providing powerful reprogrammability of the network stack.

2.3 eBPF Integration with the Linux Kernel Networking Stack

Within the Linux network stack, the Traffic Control (TC) subsystem plays a crucial role in enforcing egress security, extending beyond traditional responsibilities like flow control and quality of service (QoS). TC provides fine-grained traffic management through shaping, scheduling, classification, policing, and packet dropping. These functions are implemented via queueing disciplines (QDISCs), which determine how packets are prioritized and transmitted through the network driver's transmission queues [21]. Among these QDISCs, CLSACT (classless QDISC with actions) is particularly valuable for advanced security en-

forcement. It enables classification and action hooks on both ingress and egress paths without disrupting existing classful (e.g., HTB) or classless (e.g., FQ_CODEL, PRIO_FAST) traffic configurations. This backward compatibility makes CLSACT suitable for production environments, allowing seamless integration of programmable in-kernel logic. eBPF programs attached to CLSACT filters can chain classification and actions with configurable priorities, enabling deterministic and layered packet filtering directly within the kernel [10]. Since CLSACT operates before any default QDISC, it is ideal for embedding additional security logic without affecting existing traffic control behavior. This approach preserves QoS guarantees while enabling real-time, low-latency filtering. While CLSACT is commonly used by Container Network Interface (CNI) plugins in Kubernetes—for overlay routing, IP masquerading, and node-to-node communication—its full potential for enforcing in-kernel security policies remains largely untapped. Beyond TC, eBPF programs can attach across multiple layers of the kernel network stack. These include the high-speed ingress path via XDP, the link layer via Netfilter, socket-layer hooks (e.g., sockops, sk_msg), and even syscall interfaces and kernel security modules. This broad hook coverage enables eBPF to support profiling, observability, rate limiting, deep packet inspection (DPI), and threat detection with minimal overhead and maximum control.

2.4 DNS-based Data Exfiltration

DNS-based data exfiltration is a covert technique that extracts sensitive information from compromised systems using the DNS protocol. Frequently used by memory-resident fileless implants, this method minimizes forensic footprints by abusing the ubiquity and permissiveness of DNS, which is rarely filtered or blocked. Attackers encode payloads into the subdomain section of outbound DNS queries, sending them to attacker-controlled domains or delegated nameservers using standard recursive resolution paths. As shown in Table 2.1, a variety of obfuscation techniques can be applied to embed and disguise exfiltrated data. These include base encoding, compression, and segmentation. Common DNS record types such as `A`, `AAAA`, `MX`, and `HTTPS` are often used due to widespread acceptance, while `TXT` and `NULL` are favored for their ability to carry arbitrary payloads - ideal for command-and-control

(C2) responses that contain attacker instructions. To evade detection, adversaries employ DGA, randomized query timing, and ephemeral key encryption. Some frameworks, such as DNSCat2, tunnel DNS over arbitrary transport ports, encapsulating non-DNS traffic, and bypassing traditional inspection and firewall filters. Figure 2.4 outlines the typical phases of DNS-based exfiltration. The three dominant forms are tunneling, DNS-based C2, and raw exfiltration and are examined in detail below.

Encoding Format	**Exfiltrated Payload**	**Encoded DNS Subdomain**
Base64	TopSecret	VG9wU2VjcmV0.dns.exfil.com
Mask (XOR 0xAA)	TopSecret	DE.D5.F2.F9.E9.C7.CF.DE.dns.exfil.com
NetBIOS	TopSecret	ECPFEDFEFCDCECEEEA.dns.exfil.com
CRC32 (Hex)	TopSecret	7F9C2BA4.dns.exfil.com
AES-CBC (Hex + IV)	TopSecret	IV.A1.B2.C3.D4.E5.F6.07.08.dns.exfil.com
RC4 (Hex)	TopSecret	9A.B3.47.E2.8C.4D.11.6F.dns.exfil.com
Raw (Hex)	TopSecret	546f70536563726574.dns.exfil.com

Table 2.1: DNS Payload Obfuscation Techniques

DNS Tunneling

DNS tunneling encodes other protocols or breached payloads in query fields to bypass firewalls and perimeter defenses. Attackers disguise these payloads as legitimate DNS traffic to covertly connect compromised hosts to remote servers. Tunnels vary in throughput and adjust traffic patterns to avoid anomaly detection. Advanced forms use kernel-level encapsulation (e.g., `TUN/TAP`, VXLAN) via virtual interfaces, requiring privileges like `CAP_NET_ADMIN`. The intermittent and benign appearance of this traffic makes passive detection particularly difficult.

DNS Command and Control (C2)

DNS-based C2 extends tunneling to establish persistent full-duplex covert channels between implants and attacker-controlled servers, following a client-server architecture. Implants poll for encoded commands via DNS queries and return execution results in DNS responses, enabling full remote control. These channels support backdoors, port forwarding, and DNS-based reverse tunnels that expose internal services. Attackers evade detection by varying beaconing intervals and rotating IPs/domains using DGA. Multiplayer C2 attacks extend basic C2 operations by coordinating multiple operators to simultaneously exploit multiple implants, overwhelming passive anomaly detection systems. Static blacklists and rules fail against such adaptive threats. Real-time in-kernel termination of both the DNS C2 channel and its implant process is critical, but currently lacks viable solutions before data loss or command execution.

DNS Raw Exfiltration

Raw DNS exfiltration involves direct leakage of data, such as files or credentials, through rapid bursts of DNS queries. Although noisier than C2 and tunneling, these attacks can still facilate some data theft before alerts trigger or policies are in effect.Since most defenses detect post facto or passively, real-time prevention at the transmission point is essential to ensure zero data loss.

2.5 DNS Protocol Security Enhancements and Their Limitations

Several standardized enhancements improve DNS integrity and privacy:

- **DNSSEC** Add cryptographic signatures to DNS records to ensure authenticity and prevent spoofing or cache poisoning. However, it does not encrypt payloads, leaving queries visible to intermediaries and vulnerable to covert channels or data exfiltration hidden within legitimate DNS traffic.
- **DNS-over-TLS (DOT)**: Encrypt DNS queries to prevent surveillance and man-in-the-middle attacks. While improving privacy, this encryption blinds traditional security tools, limiting deep packet inspection (DPI) and weakening intrusion detection (IDS)

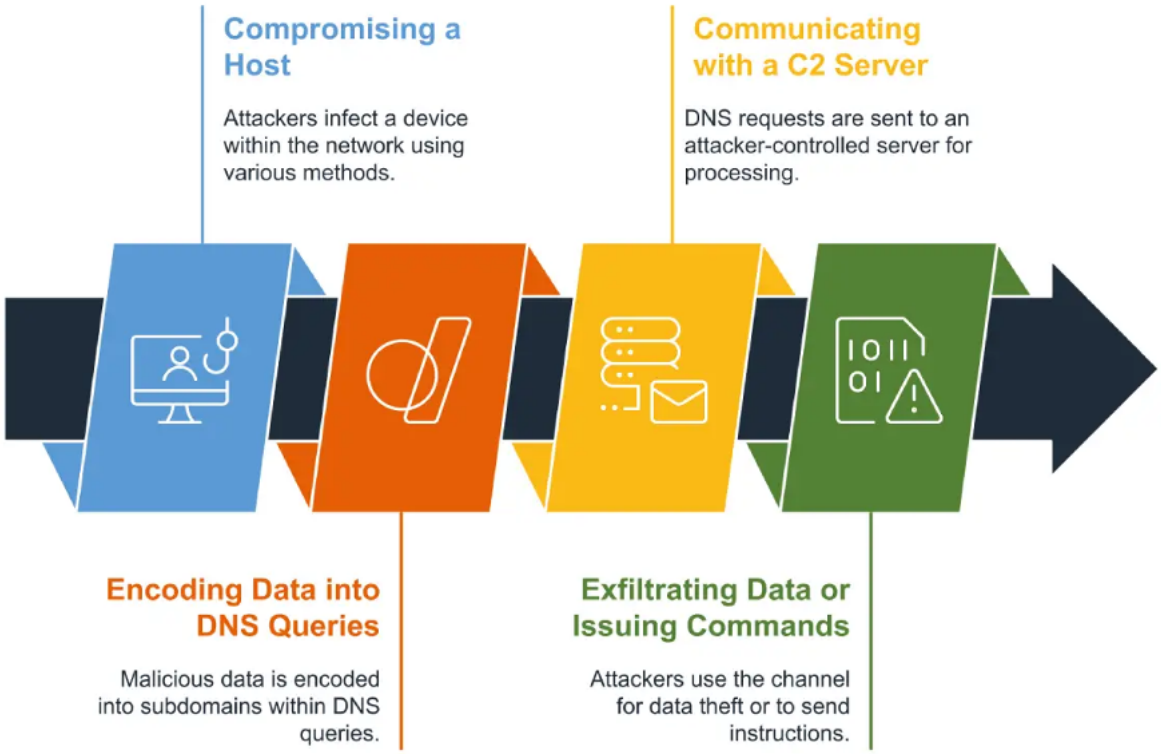


Figure 2.1: DNS Data Exfiltration Phases

and data loss prevention (DLP) systems.

Although effective against some attacks, these protocols do not prevent DNS-based data exfiltration originating from endpoint implants that exploit protocol-compliant DNS structures for covert communication.

2.6 Existing Prevention Mechanisms and Their Limitations

Common DNS exfiltration defenses include:

- **DNS Sinkholing**: Redirects queries for known malicious domains to controlled endpoints.
- **Response Policy Zones (RPZ)**: Apply static filtering rules on DNS servers based on enforced domain access control policies.

Although effective against known threats, traditional defenses are reactive, relying on static

blacklists or passive DPI enforced only after an alert. This delay allows DNS exfiltration and the C2 commands to be successful. They also fail against implants using DGA, which rapidly rotate domains to bypass filters. The stealth and evolving complexity of modern DNS-based C2 make static rulesets too slow to prevent damage in real-time.

Chapter 3

RELATED WORK

This chapter reviews related work on the use of eBPF for network security, research that uses machine learning to detect DNS data exfiltration, and current enterprise solutions.

3.1 Network Security using eBPF

eBPF has emerged as a critical technology for modern networking, security, and observability in Linux. Its ability to inject safe, verifiable code into the kernel makes it ideal for high-performance, in-kernel programmability without compromising system stability. These features have led to widespread adoption by cloud providers, particularly in large-scale data planes and hyperscalers for traffic filtering and enforcement of declarative network policies across multiple layers of the Linux kernel. Most existing research focuses on the ingress path using XDP (eXpress Data Path), a high-speed packet processing mechanism integrated into the network driver, commonly referred to as the high-speed ingress kernel datapath. Initially proposed by Høiland-Jørgensen et al., XDP was later adopted into the Linux kernel to enable early packet drops, hardware offload at the NIC level, and improved throughput. It is often combined with eBPF to support low-latency programmable network processing and security enforcement for DDoS preventions [14, 18]. Vieira et al. studies provide architectural overviews and performance analyses of eBPF in networking contexts [26], similarly Bertrone et al. explains accelerating kernel network firewalls by combining eBPF and iptables [8], yet they predominantly address inbound traffic. In contrast, the egress path—critical for detecting and preventing data exfiltration remains relatively underexplored.

Kostopoulos et al. explored DNS-related defenses using eBPF, leveraging XDP to mitigate DNS water torture DDoS attacks by analyzing queries directly at the network interface of authoritative DNS servers [15]. Although effective for volumetric DDoS mitigation, their

approach is limited in scope and does not address low-volume, stealthy data exfiltration. Similarly, Bertin proposed an XDP-based strategy to mitigate ingress layer DDoS floods, such as TCP SYN and UDP amplification attacks [7]. However, this technique also fails to handle sophisticated or covert DNS-based exfiltration threats. Based on the current literature, Steadman and Scott-Hayward presents the only known eBPF-based system specifically aimed at preventing DNS exfiltration. Their approach combines eBPF and SDN to enforce static rules in the data plane while performing flow analysis in the control plane [24, 23]. However, their design attaches eBPF programs to the XDP layer, which is only suitable for ingress traffic, which limits its effectiveness against exfiltration. Moreover, reliance on static rules increases false-positive rates and restricts adaptability to novel attack patterns. The use of P4 switches and packet mirroring to the SDN controller also introduces latency, hindering real-time enforcement. Moreover, their evaluation was not able to prevent stealthy exfiltration, leading to data loss with more stealthy traffic mirroring to the control plane. Similarly, enterprise tools such as Isovalent Cilium support eBPF-based Kubernetes network policies at layers L3–L7 [27, 6]. Although DNS-aware L7 policies allow domain-level whitelisting, they lack dynamic blacklisting and are not designed to detect exfiltration behaviors. Open-source tools such as Microsoft's Inspector Gadget also rely on static rules defined in the userspace and do not provide deep kernel-level dynamic enforcement mechanisms. These limitations highlight the need for a comprehensive eBPF-based solution that operates at the egress point and supports dynamic security enforcement not only inside the kernel via eBPF but also in a distributed environment with dynamic domain blacklist to combat DGA.

3.2 Machine Learning for Detecting DNS Data Exfiltration

Advancements in network security have significantly improved DNS exfiltration detection, often using machine learning to analyze anomalies in traffic volume and rate, as well as for lexical analysis of exfiltrated DNS queries. Common solutions combine DPI with anomaly detection of traffic volume and timing, identifying potential threats using DNS firewalls or intrusion detection systems. For C2-based exfiltration, Zimba and Chishimba employs behavioral analysis that integrates PowerShell activity and backdoors with DNS tunneling

for APT detection [29]. Similarly, Das et al. trains ML models on real malware samples from financial institutions, while Ahmed et al. uses Isolation Forest for real-time detection [12, 1]. However, these methods focus on detection, not prevention, and struggle against stealthy, persistent C2 channels.

Bilge et al. and Antonakakis et al. introduced DNS server-side solutions like EXPOSURE and NOTOS, which rely on passive analysis of large datasets, extracting domain features to flag malicious activity [9, 4]. While effective in identifying botnet C2 and spamming domains, these approaches lack real-time enforcement capabilities and cannot block payload execution. Similarly, models proposed by Nadler et al. and Mathas et al. use entropy, timing, and anomaly-based techniques to detect low-throughput DNS tunneling [19, 16]. However, these methods remain inherently reactive, relying on historical traffic patterns and often failing against slow, stealthy exfiltration tactics.

Aurisch et al. propose mitigation efforts involving mobile agents that introduce latency, suffer from false positives, and lack scalability [5]. Similarly, Haider et al. propose the C2 Eye framework to detect C2 attacks in supply chains but do not address the damage caused by C2 in distributed environments [13]. Even promising tools like Process DNS, developed by Sivakorn et al., correlate DNS traffic with userspace processes to detect C2 activity [22]. However, these tools remain vulnerable to privilege escalation and evasion due to limited kernel integration and the absence of fine-grained, kernel-enforced mandatory access controls.

Overall, most ML-based DNS security tools are limited by userspace-only architectures. They lack in-kernel inspection, cross-protocol correlation, and visibility into port-layer obfuscation—such as DNS over nonstandard ports. These tools operate passively, disconnected from real-time enforcement, and offer no preemptive response to emerging threats. While lexical payload analysis may help classify anomalies quickly, behavioral models still lag behind, detecting threats only after execution or data exfiltration has occurred. Hence, existing researched solutions fundamentally fail to prevent advanced C2 behavior such as remote code execution, port forwarding, or shell access. They rarely address real-world adversary emulation or modern C2 vectors, and remain ineffective against multiplayer C2 operations,

botnet-based attacks, or DGA. Despite incremental progress, no existing solution offers real-time DNS exfiltration prevention with implant termination, dynamic in-kernel security policy enforcement, negligible or zero data loss, adaptive domain blacklisting, and cloud-native scalability. Userspace-only systems lack the ability to introspect low-level system state—especially near the NIC—and instead rely on passive traffic analysis from centralized locations. This limits their ability to enforce fine-grained controls needed for modern threat defense. Real-time, kernel-level defenses are essential to ensure data sovereignty and integrity with the speed and precision based response to combat emerging threats.

3.3 Enterprise Solutions to Prevent DNS Data Exfiltration

Akamai's ibHH algorithm leverages information heavy hitters for real-time DNS exfiltration detection adopted in production as explained by Ozery et al. by quantifying unique data transmitted from DNS subdomains to their domains, using a fixed-size cache for efficient processing [20]. Although DNS firewalls such as Akamai and AWS Route 53 can detect tunneling and DGA activity using volume thresholds and anomaly rules, they lack direct endpoint prevention, which is essential to minimize data loss and reduce dwell time [3]. These systems often fail against APT malware that employs slow and stealthy C2 patterns and requires static manual policies that are ineffective against DGA. Route 53 also lacks enhanced observability and cross-protocol correlation, limiting its ability to enforce Layer 3 blocks through AWS network firewalls. Similarly, Cloudflare, Akamai, and AWS proprietary DNS firewalls are optimized for DDoS mitigation, but fail against sophisticated exfiltration or insider threats using DNS-based C2. Infoblox adds hybrid agent-based enforcement and centralized threat intelligence, and Broadcom's Carbon Black blocks endpoint processes, but both rely on user-space traffic analysis [2]. In contrast, eBPF enables in-kernel enforcement with fine-grained visibility, offering significantly stronger mitigation and detection capabilities for DNS exfiltration.

Chapter 4

IMPLEMENTATION

This chapter explains the architecture of the security framework and its individual components, first highlighting the overall framework, followed by a detailed breakdown of each component.

4.1 Security Framework Overview

The implemented security framework adopts an endpoint-centric architecture for real-time mitigation of DNS-based C2 and tunneling activity by embedding enhanced DNS exfiltration defenses directly into the operating system using eBPF. It dynamically enforces network policies within the kernel network stack to block C2 communications and terminates malicious processes through tight integration with the kernel syscall layer, coordinated via a lightweight eBPF userspace agent. For scalability, the framework streams threat events asynchronously, enabling cross-node security enforcement across distributed data planes and supports dynamic domain blacklisting on DNS servers to counter DGA-based threats. The following subsections detail the core components of the system.

4.1.1 Data Plane

The data plane is composed of eBPF agents, implemented entirely in Golang for high performance and low memory overhead, and deployed across all endpoint nodes. Operating in userspace, these agents dynamically inject eBPF programs into the kernel's TC layer at the egress hook of each physical network interface, enabling in-kernel DPI to detect and filter exfiltrated DNS traffic, primarily over UDP. Beyond TC filters, the agents also deploy auxiliary eBPF programs at various kernel hook points: kprobes to monitor the creation of new network devices, raw tracepoints to track process termination, and cgroup socket hooks to retrieve internal kernel process representations (task_struct)—particularly on Linux kernels

below version 5.2, where direct access from TC is not supported. Integration with Linux Security Modules (LSM) ensures the integrity of all injected eBPF programs, protecting the kernel against tampering or malicious eBPF programs. Each agent supports two configurable prevention modes, controlled via a dedicated eBPF map injected into the kernel. These modes can be dynamically toggled at runtime from userspace. By default, both are enabled to provide maximum protection against DNS exfiltration attack vectors.

- **Strict Enforcement Active Mode**: DNS packets over standard ports (DNS: 53, MDNS: 5353, and LLMNR: 5355) are scanned in the kernel using eBPF at the TC egress hook. Malicious packets are immediately dropped. If the classification exceeds the limits of the eBPF instruction, the packet is redirected to the userspace for further analysis. The eBPF agent sniffs redirected traffic and checks against the domain blacklist cache or performs ONNX-based deep learning model inference to identify potential payload obfuscation in DNS traffic. Post-userspace inferencing benign packets are retransmitted using high-speed socket options like `AF_PACKET` or `AF_XDP`, while malicious ones are dropped and their domains added to the userspace blacklist cache.
- **Process-Aware Adaptive Passive Threat Hunting Mode**: Designed for non-standard UDP ports overlayed with DNS traffic for exfiltration, this mode clones suspicious packets for userspace analysis while allowing the original packet to continue. If found malicious, the associated process is flagged in the eBPF maps. Subsequent DNS packets from that process are dropped in the kernel, effectively breaking C2 implant communication with the remote server. Once a configurable threshold is reached, the process is terminated. This mode is highly effective against stealthy exfiltration techniques using C2 implants that employ port obfuscation and DNS layering over random UDP ports.

In addition to performing deep packet inspection, the eBPF agents manage network namespaces and virtual bridges using the Linux virtual ethernet bridge driver. The topology -shown in Figure 4.1 - combines Linux network namespaces with multiple pairs, acting as Layer 2

and Layer 3 bridges. Each agent maintains file descriptors to its eBPF maps, enabling efficient kernel-userspace communication and advanced runtime state analysis. The lifecycle of each eBPF program is tightly managed by the agent, which operates with elevated privileges to control the datapath, syscall interfaces, and relevant kernel subsystems. The required kernel capabilities are detailed in Table 4.1. Both prevention modes, active and passive, support real-time enforcement, including the termination of processes responsible for repeated exfiltration beyond configurable thresholds. Filtering logic in the kernel is driven by adaptive policies encoded in eBPF maps and applied immediately after raw DNS parsing from the SKB. On the userspace side, the agent performs low-latency inference using quantized ONNX models, exports telemetry to observability backends, and streams threat events to a centralized message broker for controller-side processing. All agent components, including maps and hooks, are dynamically reprogrammable at runtime via the control plane. To complement egress filtering, agents also inspect ingress traffic to detect C2 response patterns, leveraging the same inference engine and LRU cache used for egress analysis.

Kernel Capability	**Description**
`CAP_BPF`	Load eBPF, manage maps
`CAP_SYS_ADMIN`	Attach BPF, mount BPF FS
`CAP_NET_ADMIN`	Manage netdev creation and tc/xdp/cgroup filters attachment
`CAP_NET_RAW`	Send/receive raw packets from netdev tap RX queues particularly via AF_PACKET sockets
`CAP_IPC_LOCK`	Lock BPF memory

Table 4.1: Linux Kernel Capabilities Required for eBPF Agent at Endpoint

4.1.2 Distributed Infrastructure

In addition to the data plane nodes, the framework includes an open-source PowerDNS setup consisting of a Recursor for upstream resolution and an authoritative server configured with a Postgres backend. Although actual internal domains are not served, the authoritative server is used to carry out DNS C2 and tunneling attacks by generating malicious domains using

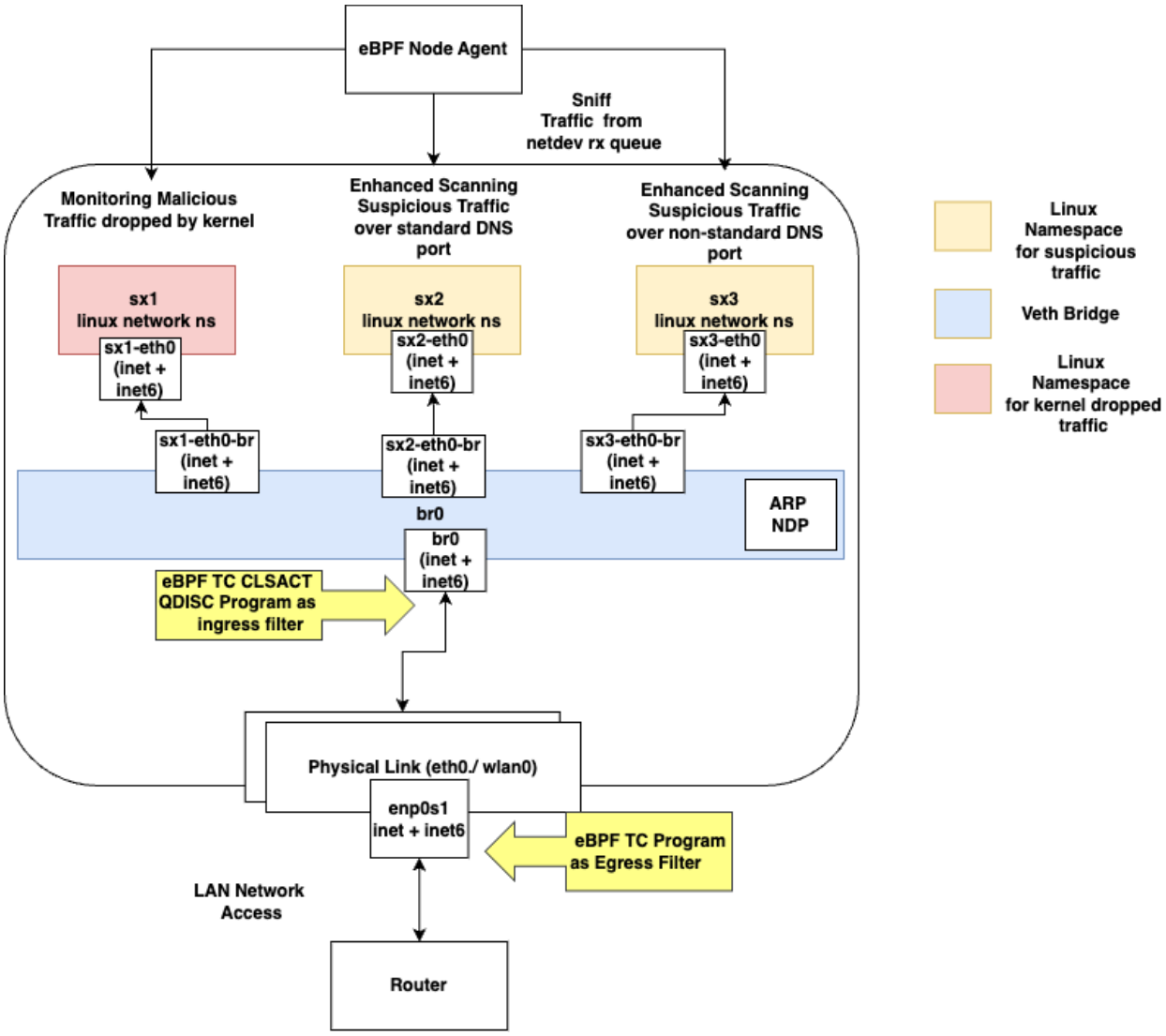


Figure 4.1: eBPF Agent-created network topology at endpoint

eBPF Program Type	Agent Mode	Injection Point	Description
SCHED_ACT	Active, Passive	Physical NICs	Performs in-kernel DNS DPI at TC egress. Interacts with maps and redirects packets to userspace or tracks process info based on mode.
SCHED_ACT	Active	veth bridges	Verifies packet integrity using skb_hash for redirected DNS traffic over namespaces.
KPROBE	Active	Tun/Tap driver kernel functions	Detects virtual device creation to attach DNS filters dynamically.
TRACEPOINT	Passive	process_exit	Cleans up eBPF maps when flagged processes exit before agent-enforced termination.
LSM	Active, Passive	BPF_PROG_LOAD	Intercepts eBPF program loading syscalls. Verifies integrity via kernel keyring to block malicious eBPF code injection.

Table 4.2: eBPF Programs Managed by the eBPF Agent

DGA. This design replicates the behavior of the real-world DNS server commonly deployed by enterprises. All data plane nodes resolve DNS through the PowerDNS Recursor, which is equipped with interceptors that inspect queries before forwarding. These interceptors run ONNX-based deep learning inference to detect threats, specifically on TCP-based DNS traffic offloaded from eBPF agents. Kafka acts as the message broker for streaming threat events. Additionally, the recursor integrates with dynamic RPZ stored in Postgres, allowing the controller to blacklist second-level domains linked to malicious activity, as detailed in the next section.

4.1.3 Control Plane

The control plane consists of a centralized analysis server that consumes threat events from Kafka topics, streamed and updated by eBPF agents running in the data plane. Based on these consumed event payloads, the control plane dynamically blacklists malicious SLDs on the DNS server, thereby safeguarding all endpoints in the data plane that utilize the DNS server. Additionally, the system supports full data plane reprogramming by publishing Kafka topics consumed by eBPF agents, allowing them to rehydrate their local blacklist domain caches and immediately enforce updated policies. This design drastically reduces

DNS resolution hops from data plane nodes to the DNS server by enforcing blacklists locally.

4.2 Data Plane

The eBPF agent implementation on each data plane node is organized into six core components. First, active mode introduces the strict enforcement path, where suspicious DNS traffic is inspected and dropped in real time. Second, passive mode describes the adaptive threat hunting strategy, which correlates DNS behavior with process activity in the userspace. Third, encapsulated phase addresses DNS exfiltration hidden within kernel-encapsulated traffic supported only in active mode. Fourth, feature analysis outlines the extracted features used both in the kernel for heuristic classification and in the userspace for deep learning-based inference. Fifth, datasets lists the data sources used to train and evaluate the detection models. Finally, model architecture and threat event streaming describe the model architecture, its serialization and quantization steps, and how prevented threat events are streamed from the agent to the centralized message brokers and metrics observability backends.

4.2.1 Strict Enforcement Active Mode

In this mode, eBPF programs are injected and attached as direct-action filters to the TC exit hook (CLSACT QDISC) on all physical network interfaces. These eBPF programs are triggered as soon as a packet is queued by the kernel to the CLSACT QDISC for the specific netdev by invoking the (dev_queue_xmit) helper in kernel network stack. At this point, the SKB is fully constructed and ready for the TX queue, having already passed through the upper layers of the networking stack. The eBPF programs run in parallel across multiple CPU cores, with all eBPF maps declared global. The kernel ensures safe concurrent access to these maps through built-in synchronization mechanisms. This mode is implemented through several components. Maps describes the structure and role of eBPF LRU hash maps (shown in Figure 4.2), which track the security state and current policy status enforced by both kernel eBPF programs and userspace eBPF agent. Next, Kernel eBPF filter packet classification explains how incoming packets are classified in the TC egress filter using SKB metadata, and how redirection is handled after deep scans by userspace for enhanced security checks.

Userspace eBPF agent packet handling then details how sniffed packets are processed in zero-copy mode. Finally, Concurrency handling discusses safe access and updates to shared maps between kernel and userspace threads, even under parallel execution across multiple CPU cores.

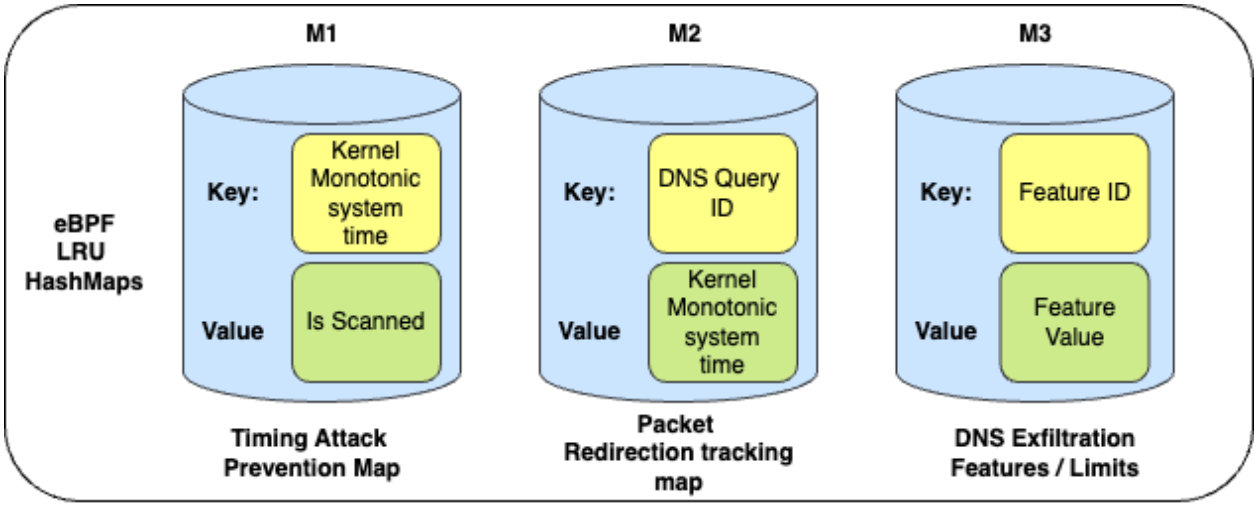


Figure 4.2: eBPF Maps structure for Agent in active phase

eBPF Maps in Active mode

1. **DNS Exfiltration Feature Map:**

   Key: Feature identifier used for DNS traffic analysis.

   Value: Filtering or classification parameter applied by the eBPF TC egress program.

2. **Packet Redirection Tracking Map:**

   Key: DNS transaction ID

   Value: Kernel monotonic timestamp (in nanoseconds). Used to track suspicious packets redirected across interfaces, leveraging NMI-safe timekeeping.

3. **Timing Attach Prevention Map:**

   Key: Monotonic timestamp (in nanoseconds)

   Value: Scanned flag (true/false). Utilized by the userspace eBPF agent to mark pack-

ets as scanned, and by the kernel eBPF program to verify packet integrity prior to retransmission.

eBPF kernel program packet processing in Active mode

When a DNS packet is sent over a UDP socket, it enters the kernel network stack and is intercepted at the TC layer under the CLSACT QDISC. An eBPF direct-action filter inspects the packet by peeling off L2–L4 headers using kernel-native structures. Since DNS is a application protocol handled in userspace, for enhanced inspection the SKB packet payload data above Layer 4 is parsed from raw bytes using RFC 1035-aligned C structures to extract fields like query ID, opcode, flags, and the DNS question section. If the query ID is new and the DNS question section is valid, the program evaluates feature indicators such as multiple questions or unexpected answer counts. These heuristics are flagged as suspicious—especially since C2 implants often encode data in questions and avoid full DNS response behavior. The features are checked against the kernel-defined feature set (Table 4.3) and immediate action is taken: `TC_ACT_SHOT` to drop, `TC_ACT_OK` to forward, or `bpf_redirect` for further inspection.

For packets marked suspicious, the DNS query ID and the current kernel monotonic timestamp (`bpf_ktime_get_ns`) are stored in `dns_packet_redirection_map` for traceability and timing analysis. Before redirection, the eBPF program fetches userspace-provided metadata, such as the bridge's L3 address and `if_index`, stored in shared maps populated at agent startup. It also updates redirection counters for observability metrics. The program determines the IP version and performs DNAT for IPv4, recalculating checksums to redirect the packet to a userspace-controlled bridge interface. For IPv6, static checksums are applied. The suspicious packet is clone-redirected to the RX queue of this virtual netdev for deep inspection in userspace.

When the eBPF agent completes deep inspection and re-emits the packet via an `AF_PACKET` socket, the TC eBPF program is re-triggered. To avoid replay or forged reinjection, it checks the stored timestamp and verification flag in `dns_redirect_ts_verify_map` using the DNS

query ID. If the flag is valid, the packet is forwarded. Otherwise, the packet is dropped to prevent evasion by compromised implants attempting brute-force or timing-based reinjection. This design enforces strict verification using monotonic timestamps, shared state, and minimal userspace/kernel trust boundaries. Algorithm 2 details the full timestamp verification and resend handling logic.

eBPF agent userspace packet processing in Active Mode

The userspace eBPF agent utilizes kernel BPF bindings, primarily through libbpf, to abstract raw BPF syscalls. It spawns dedicated threads to continuously sniff traffic from a separate network namespace configured for handling redirected suspicious DNS traffic in active mode. Using `AF_PACKET` sockets, it reads packets directly from tap interfaces or RX queues in zero-copy mode, eliminating extra buffer allocations and improving performance.

With access to all eBPF maps via file descriptors, the agent parses application-layer DNS payloads redirected from the kernel. Extracted features, described in Userspace Features, are used for detection and policy enforcement. The agent maintains two LRU caches in userspace memory: one for benign SLDs sourced from Cisco's top one million domains, and another for previously identified malicious domains. If a parsed SLD matches an entry in the benign cache, the packet is forwarded immediately—without inference—since DNS protocol semantics prevent adversaries from redirecting malicious traffic through high-reputation DNS zones.

Packets are forwarded using either `AF_PACKET` or `AF_XDP` sockets. For `AF_XDP`, packets are injected directly into the TX queue, bypassing eBPF egress filters. Associated entries in the `dns_packet_redirection_map` are then cleared. For `AF_PACKET`, the agent queries the kernel redirection timestamp via the DNS transaction ID, updates the `dns_redirect_ts_verify_map`, and flags the packet as scanned, enabling kernel passage. If no cache hit occurs, the agent performs live feature extraction and inference. Malicious packets are dropped, blacklisted in the malicious cache, and an event is sent to the controller via Kafka to update DNS blacklists. Benign packets follow the same forwarding path as those with benign cache hits.

Additionally, the agent exports detection events and system metrics—including redirection counts—to Prometheus for observability. It also monitors associated processes responsible for sending flagged packets. If malicious activity exceeds a predefined threshold, the agent terminates the offending process by sending a SIGKILL signal, leveraging kernel-level tracking of DNS query IDs and redirection state.

eBPF Maps concurrency handling in Active Mode

The eBPF programs in this mode use global kernel maps (not per-CPU) to support concurrent reads/writes, protected by BPF spinlocks NUMA-aware for SMP, extensions of kernel spinlocks that ensure cache coherence and atomic access per CPU. Each map tracks atomic reference counts for process access. Every concurrent thread in userpaace processing and sniffing packets parallely from network namespace and if performing updates over shared eBPF map always use RWMutex in userspace for synchronization Since eBPF map FDs are not shared across userspace processes, the maps are exclusively created by the single eBPF agent process in userspace ensuring reference count for all the owned map from being garbage collected. This model, paired with spinlock-based concurrency in the kernel per CPU, ensures consistent parallel packet processing across CPUs. The two primary maps shared between kernel and userspace—`dns_redirect_ts_verify_map` and `dns_packet_redirection_map` each of them always using unique keys (monotonic timestamps and DNS query IDs), preventing stale reads, race conditions, and inconsistent updates that could otherwise cause leaking malicious exfiltrated packets. In addition every update done over eBPF maps in the kernel is performed via built-in LLVM concurrency helpers to ensure strong atomic map updates synchronizing kernel memory address locations for map updates. Thus, strict and reliable control is maintained. The Figure 4.3 details the complete userspace and kernel pipeline for this mode of agent operation.

Algorithm 1: Egress TC-Based DNS Raw SKB Inspection in **ACTIVE** Mode

Input : Socket buffer (`skb`), eBPF LRU hash maps: `dns_limits`, `dns_packet_redirection_map`, `node_agent_config`
Output : Packet Action: `TC_ACT_SHOT`, `TC_ACT_OK`
eBPF Map Updates `bpf_map_updates`

`// Parse skb layers; ensure skb->data_ptr remains memory bound for eBPF verifier`
Parse Layer 2 (Ethernet) from `skb`;
if *VLAN (802.1Q or 802.1AD) is present* **then**
  if *skb->data_ptr exceeds skb->data_end* **then**
    Drop packet via `TC_ACT_SHOT`;
  Extract the inner encapsulated protocol (`h_proto`) from VLAN header;
Parse Layer 3 (Network) from `skb`;
if *skb->data_ptr exceeds skb->data_end* **then**
  Drop packet via `TC_ACT_SHOT`;
Parse UDP Layer 4 (Transport) from `skb`;
if *skb->data_ptr exceeds skb->data_end* **then**
  return `TC_ACT_SHOT`;
if *skb->protocol = IPPROTO_TCP* **then**
  return `TC_ACT_OK`;
if *udp->dest* $\neq$ *53* ***and*** *udp->dest* $\neq$ *5353* ***and*** *udp->dest* $\neq$ *5355* **then**
  `// This is not standard DNS traffic; over MDNS, DNS, LLMNR`
  return `TC_ACT_OK`;
Parse Layer 7 DNS (Application) from `skb`;
if *skb->data_ptr exceeds skb->data_end* **then**
  Drop packet via `TC_ACT_SHOT`;
Extract `qd_count`, `ans_count`, `auth_count`, and `add_count`;
if *qd_count* $>$ *1* ***or*** *auth_count* $>$ *1* ***or*** *add_count* $>$ *1* **then**
  Perform `bpf_map_updates`;
  return;
Parse first question record from `skb`;
`// Extract Kernel DNS features from dns_limits map`
Fetch `n_lbls`, `dom_len`, `subdom_len`, `dom_len_no_tld`, `q_class`, `q_type` from `dns_limits`;
if *n_lbls* $\leq$ *2* **then**
  return `TC_ACT_OK`;
if *Any of (n_lbls, dom_len, subdom_len, dom_len_no_tld) is in [min, max] range* **then**
  Perform `bpf_map_updates`;
  return;
if *Any of (n_lbls, dom_len, subdom_len, dom_len_no_tld) exceeds its maximum threshold* **then**
  return `TC_ACT_SHOT`;
if *q_type* $\in$ {*TXT, ANY, NULL*} **then**
  return `bpf_map_updates`;
return `TC_ACT_OK`;

Algorithm 2: DNS eBPF Map Handling Before `skb_redirect` and After Userspace `AF_PACKET` Socket Writes in **ACTIVE** Mode

Input : `skb` (socket buffer),
eBPF LRU hash maps:
`netlink_links_config`,
`dns_packet_redirection_map`,
`dns_redirect_ts_verify_map`,
`redirect_count_map`
`skb_netflow_integrity_verify_map`
Output : `bpf_redirect` to `bridge_if_index`, `TC_ACT_SHOT`, `TC_ACT_OK`

Extract DNS Layer from the packet application data;
Get DNS transaction ID (`tx_id`) from parsed L7 payload in `skb`;
Determine if packet is IPv4 or IPv6 using `nexthdr` / `h_proto` in Ethernet frame in SKB (`ETH_P_IPV4` / `ETH_P_IPV6`);
`// use the kernel skb destined netdev link index`
Extract `if_index` from `skb`;
`// Fetch Virtualized Bridge netdev information`
Fetch `dst_ip` from `netlink_links_config` with key `if_index`;
`// Use userspace-generated random hash for skb integrity verification post live-redirect across bridge.`
Fetch `skb_mark` from `netlink_links_config` with key `if_index`;
Fetch `dns_kernel_redirect_val` = {`l3_checksum`, `kernel_time_ns`} from `dns_packet_redirection_map` with key `tx_id`;
if *not* ***dns_kernel_redirect_val*** **then**
    `// Packet redirected; arrived at TC hook first time`
    Modify `skb` to replace destination IP with `dst_ip` of virtual bridge;
    if ***ETH_P_IPV4*** **then**
        Recompute Layer 3 checksum and update in `skb`;
        Set `l3_checksum` = computed checksum;
    else
        Set `l3_checksum` = `0xFFFFF`;
    if *not* ***skb_mark*** **then**
        `// Custom skb->mark per netflow computed in-kernel`
        `skb_mark = bpf_get_prandom_u32() // an integrity verify map to verify netflow per redirect from skb mark over bridge`
        `// unique flow index identifier for each skb netflow`
        Update `skb_netflow_integrity_verify_map` with key=`0xFFEF`, value=`skb_mark`;
    Mark `skb->mark = skb_mark`;
    Update `dns_packet_redirection_map` with key=`tx_id`, value=`l3_checksum`, `kernel_time_ns`;
    Increment global redirect count for `if_idx`;
    Update `redirect_count_map` with key=`if_idx`, value=`updated_redirect_count`;
    Perform `bpf_redirect(bridge_if_index, BPF_F_INGRESS)`;
else
    `// Userspace deep-scanned packet re-arrived; verify it is not forged`
    Extract `kernel_time_ns` from `dns_kernel_redirect_val`;
    Fetch and delete `redirect_ts_verify_val` from `dns_redirect_ts_verify_map` with key= `kernel_time_ns`;
    if *not* ***redirect_ts_verify_val*** **then**
        `// Timing attack: userspace agent did not emit packet`
        return `TC_ACT_SHOT`;
    else
        `// Packet rescanned from authorized sender`
        Delete `redirect_ts_verify_val` from `dns_redirect_ts_verify_map`;
        return `TC_ACT_OK`;

Algorithm 3: Userspace eBPF Agent with Deep Learning and Event Streaming for Deep Parsing of DNS Traffic

Input : Sniffed DNS packets from suspicious Linux namespaces via pcap (zero-copy), DNS features,
All kernel eBPF maps of type LRU Hash
Output : Packet Garbage collected (if malicious) or socket write (if benign)

```
Sniff traffic over veth pair interfaces in Linux namespace;
Extract userspace DNS features from DNS packet;
Export metrics from all monitoring maps (redirection_count, loop_time, etc);
Fetch isSLDBenign from eBPF agent's userspace LRU map of top 1M SLDs;
if isSLDBenign then
    Set shouldRetransmit ← true;
else
    Fetch isBlacklistedSLDFound from eBPF agent's malicious map;
    if isBlacklistedSLDFound then
        // Previously blacklisted, drop packet
        return;
    Pass features to ONNX model for inference;
    if Inference result == MALICIOUS then
        Emit event to message brokers with details;
        Blacklist SLD for all DNS query records;
        return;
    else if Inference result == BENIGN then
        Set shouldRetransmit ← true;
if shouldRetransmit then
    Fetch dns_kernel_redirect_val = {l3_checksum, kernel_time_ns} from
     dns_packet_redirection_map with key=tx_id;
    Extract kernel_time_ns from dns_kernel_redirect_val;
    if AF_PACKET then
        Update dns_redirect_ts_verify_map with key kernel_time_ns and value true;
        Replace packet's l3_checksum;
        Serialize packet payload to raw bytes;
        Call syscall.write(AF_PACKET, SOCK_RAW, 0);
    if AF_XDP then
        Delete kernel_time_ns from dns_redirect_map;
        Serialize packet payload to raw bytes;
        Call syscall.write(AF_XDP, SOCK_RAW, 0);
```

4.2.2 Process-Aware Adaptive Passive Threat Hunting Mode

Passive mode reuses the same eBPF program attached to the egress CLSACT TC filter, as introduced in active mode, and attaches it to all physical network interfaces at the endpoint. This mode extends the design by layering additional telemetry and process-aware enforcement. The passive mode architecture is broken down as follows. Maps introduces the key eBPF structures used in this mode, their types, and their roles, as visualized in Figure 4.4. Kernel packet processing then details how the eBPF programs in kernel identifies and drops malicious DNS exfiltration attempts and tracks the responsible userspace processes. It also

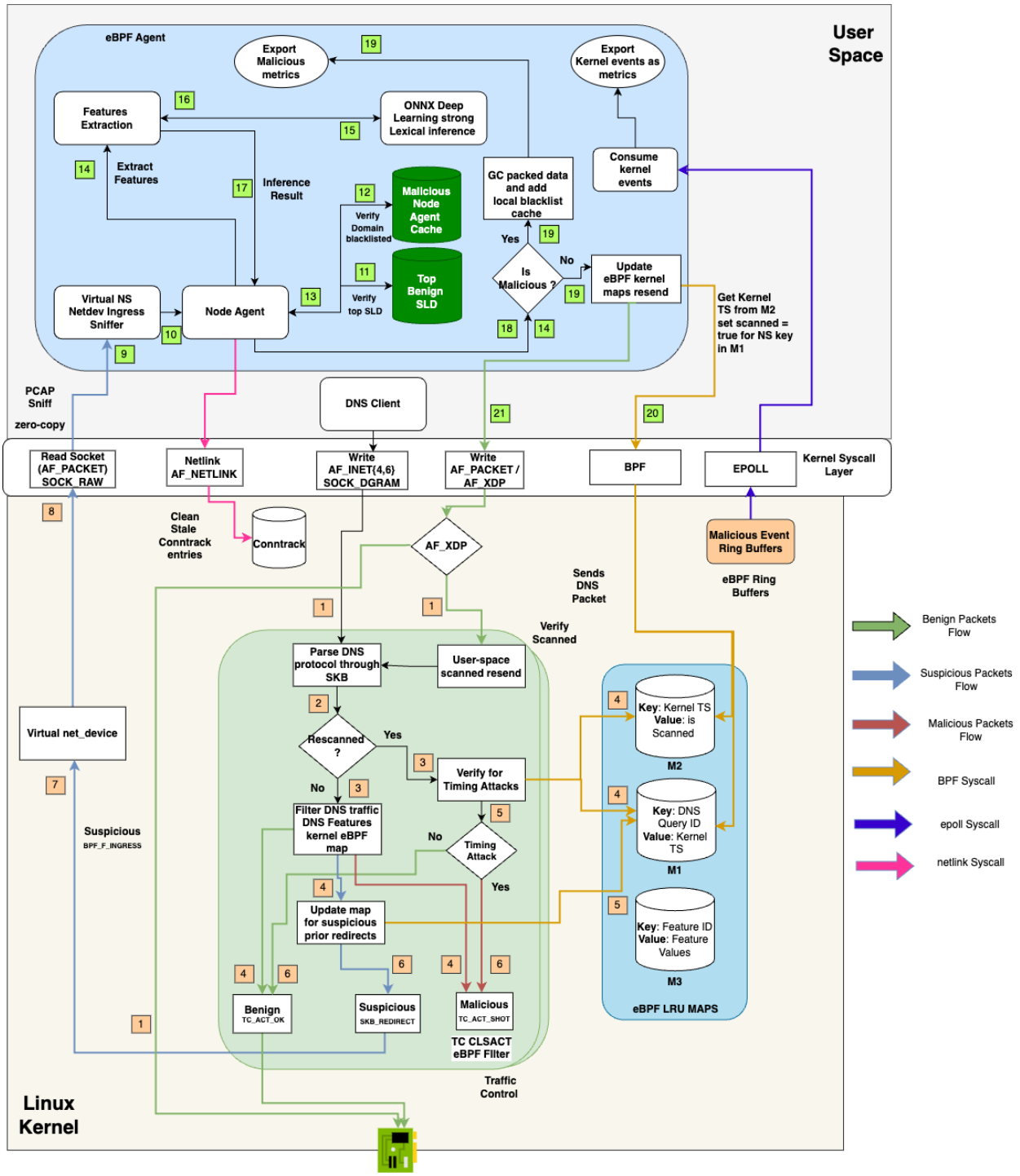


Figure 4.3: eBPF Agent DNS Exfiltration Prevention Flow for active phase

leverages kernel tracepoints—especially from the scheduler—for efficient garbage collection. The next section, Userspace packet processing, describes how the eBPF agent parses and classifies traffic identifying potential malicious process involved in exfiltration. Finally, eBPF map concurrency explains how the kernel and userspace components maintain a consistent map state under concurrent access between CPUs.

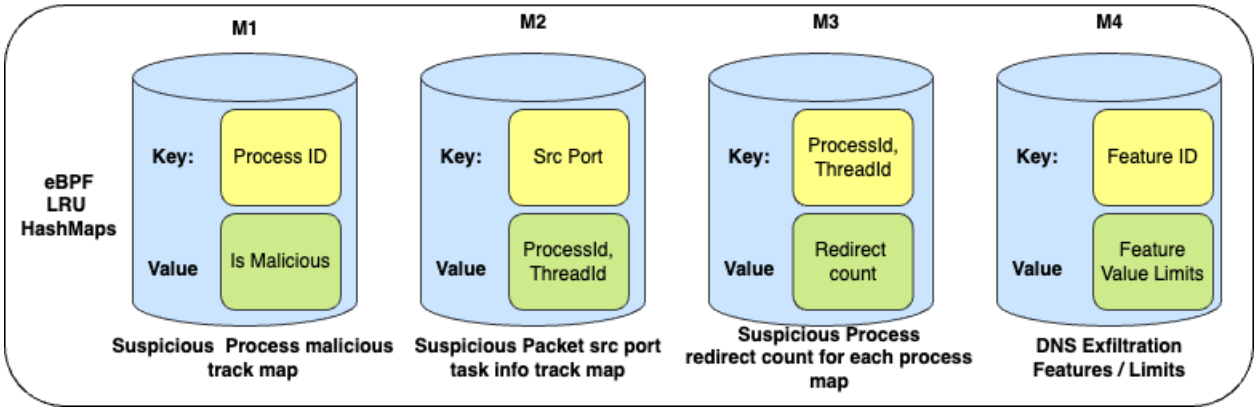


Figure 4.4: eBPF Maps and structure for Agent in passive phase

eBPF Maps in Passive mode

1. **Suspicious Process Redirect Count Map:**

   Key: Process metadata extracted from the kernel `task_struct`.

   Value: Count of DNS redirections per process, specifically for suspicious traffic over non-standard UDP DNS ports.

2. **Suspicious Packet Source Port–Process Map:**

   Key: Source port of a potentially layered DNS packet sent over a non-standard UDP port.

   Value: Process metadata extracted from the kernel `task_struct`.

3. **Malicious Process Tracking Map:**

   Key: Process ID extracted from the kernel `task_struct`.

   Value: Boolean flag indicating whether the process has been identified as malicious.

eBPF kernel program packet processing in Passive mode

In passive mode, the system focuses on preventing DNS exfiltration over arbitrary UDP ports that may bypass active mode protections. Like in active mode, the eBPF program performs packet parsing up to Layer 4. However, this mode deliberately avoids immediate redirection of all suspicious UDP packets to prevent unnecessary congestion or latency for legitimate Layer 7 protocols operating over non-standard ports.

Because the kernel eBPF program cannot reliably determine whether SKB application data on random UDP ports corresponds to DNS, it avoids aggressive redirection strategies used in active mode. Instead, it employs `skb_clone_redirect` to create a full copy of the original packet and redirect it to a netdev interface managed by the eBPF agent. To minimize memory overhead, clone-redirection is only performed for packets whose payload structure heuristically aligns with DNS traffic. The program conducts raw parsing within the SKB bounds (skb-¿data_end) and applies validation based on the DNS protocol limits defined in RFC 1035. If the payload passes these structural checks, clone redirection is triggered. The full procedure is detailed in Algorithm 4.

Before redirection, the TC eBPF program uses `bpf_get_current_pid_tgid`to fetch the current `task_struct`, identifying the process and thread responsible for sending the packet. It then checks the `malicious_process_map` to verify whether the process has been flagged as malicious by the userspace eBPF agent due to prior suspicious transfers. If flagged, the kernel drops subsequent packets while still cloning and redirecting packets from the monitoring process. This design enables dynamic, process-level enforcement, allowing the userspace agent to observe retry behavior, gather telemetry, and ultimately terminate stealthy implants that beacon intermittently and exceed the configured threshold.

If the process is not yet flagged, the eBPF program extracts the source UDP port and updates `src_port_task_struct_map` with the port as the key, and the process/thread ID as the value. It also increments a counter in `task_struct_redirect_ct_map`, keyed by the same PID/TID composite, to track suspicious activity. These updates are shown in Algorithm 5.

This continuous feedback mechanism for threat hunting of malicious processes in the kernel, aided by userspace detection from the eBPF agent, continues until the process either stops sending DNS traffic or is explicitly terminated by the agent in userspace. If the process exits before being flagged, a cleanup step is triggered by an eBPF program attached to the raw tracepoint **tracepoint/sched/sched_process_exit**, which is invoked by the kernel scheduler when a process terminates. This tracepoint eBPF program removes the exited process entries from both `task_struct_redirect_ct_map` and `malicious_process_map`.

The userspace eBPF agent only sends a kill signal when the count of blocked DNS exfiltration attempts from a process exceeds a configured threshold. If the process ends before the threshold is reached, the cleanup ensures that no stale eBPF map entries remain.

eBPF agent userspace packet processing in Passive Mode

The eBPF agent operates passive mode using multiple goroutines, each pinned to an OS thread, to sniff traffic within a dedicated Linux namespace and its associated physical device. Like in active mode, clone-redirected packets are received in zero-copy userspace, containing both application payloads and lower-layer headers. The agent parses each packet to identify embedded DNS structures. If no valid DNS layer is found, the source port is removed from `src_port_task_struct_map` to prevent stale state tracking. When DNS is detected, the agent applies the same logic as in active mode: checking a local LRU memory cache for known blacklisted domains, and if absent, performing feature extraction and deep learning inference. If a packet is classified as malicious, the agent updates the blacklist and streams a threat event to the observability backend. Because these are clone-redirected packets, they are not reinjected. Instead, the agent uses the source port to retrieve the process and thread ID from `task_struct_redirect_ct_map`, flags the process in `malicious_process_map`, and enables the kernel to drop future packets from it. The agent also checks how many packets have been redirected for that process. If the count exceeds a configurable threshold and the process is marked malicious, the agent, running with `CAP_SYS_ADMIN`, terminates it. This process-aware passive mode enables adaptive detection and response for exfiltration over

nonstandard ports. Algorithm 6 details the eBPF agent's passive processing logic.

eBPF Maps concurrency handling

This mode mirrors the concurrency principles of Active Mode by using per-CPU kernel spin locks on shared eBPF maps. This ensures safe, concurrent updates for as long as the userspace agent remains active. For `src_port_task_struct_map`, atomicity is achieved through unique `src_port` keys, allowing the kernel to write and userspace to read without race conditions. Userspace updates to `malicious_process_map` use the `BPF_ANY` flag and are synchronized with a userspace mutex. Since kernel programs only perform reads on this map, the design benefits from the RCU (Read-Copy-Update) model—allowing non-blocking reads for improved performance. In `task_struct_redirect_ct_map`, both kernel writes and userspace reads occur. Here, spin locks protect kernel writers, while a separate userspace mutex guards readers to prevent race conditions. Figure 4.5 illustrates the complete TC pipeline.

Algorithm 4: DNS RAW SKB Parsing over Egress TC CLSACT QDISC in **Passive** Mode

Input : `skb` (socket buffer),
eBPF LRU hash maps: `netlink_links_config`
Output : Packet Actions: `TC_ACT_OK`,
eBPF Map Updates: `bpf_map_updates`

Parse lower layers from `skb`;
Parse DNS header from `skb`;
Extract `qd_count`, `ans_count`, `auth_count`, `add_count`;
`// Verify the DNS count limits within u8 range`
if *qd_count* $\geq$ *256* ***or***
  ans_count $\geq$ *256* ***or***
  auth_count $\geq$ *256* ***or***
  add_count $\geq$ *256* **then**
  return `TC_ACT_OK`;
Extract DNS flags: `raw_dns_flags` from `dns_header`;
`// Verify opcodes and rcode according to RFC 1035`
if *opcode* $\geq$ *6* **then**
  return `TC_ACT_OK`;
if *rcode* $\geq$ *24* **then**
  return `TC_ACT_OK`;
Extract `if_index` from skb;
Fetch `dst_ip` and `bridge_if_index` from `netlink_links_config` with key=`if_index`, value={`dst_ip`, `bridge_if_index`};
Perform `bpf_map_updates`;

Algorithm 5: DNS eBPF Map Handling in **Passive** Mode of Agent

Input : `skb` (socket buffer),
eBPF LRU hash maps:
`malicious_process_map`,
`src_port_task_struct_map`,
`task_struct_redirect_ct_map`,
`dns_packet_clone_redirection_ct_map`

Output : `bpf_clone_redirect` action to `bridge_if_index`

Parse DNS header from `skb`;
Extract DNS transaction ID (`tx_id`) from DNS header;
`// if_index resembles the netdev link index in the kernel`
Fetch `dst_ip` and `bridge_if_index` from `netlink_links_config` with key=`if_index`;
Fetch `skb_mark` from `netlink_links_config` with key=`if_index`;
Fetch process `task_struct` and `process_info` with key=`bpf_get_current_pid_tgid`;
Fetch `is_malicious` from `malicious_process_map` with key=`process_id`;
if *not* *`is_malicious`* *or* *`is_malicious`* *is* *`null`* **then**
    Update `src_port_task_struct_map` with key= `process_id`, value=`task_struct`;
    Fetch `current_suspicious_ct` from `task_struct_redirect_ct_map` with key=`task_struct`;
    Update `task_struct_redirect_ct_map` with key=`task_struct`, value=`current_suspicious_ct + 1`;
    Update `dns_packet_clone_redirection_ct_map` with key=`if_index`, value=`new count`;
    if *`is_malicious`* *is* *`null`* **then**
        `// First DNS attempt by process | not yet marked malicious`
        Update `malicious_process_map` with key=`process_id`, value=`false`;
    Perform `bpf_clone_redirect(skb, bridge_if_index, BPF_F_INGRESS)`;
else
    Update `task_struct_redirect_ct_map` with key= `task_struct`, value=`current_suspicious_ct + 1`;
    Perform `bpf_clone_redirect(skb, bridge_if_index, BPF_F_INGRESS)`;
    return `TC_ACT_SHOT`;
    `// Drop packet since malicious process attempted exfiltration. Continue redirecting clones to monitor retry behavior.`

Algorithm 6: User-Space eBPF Agent for DNS over Non-Standard UDP Ports.

Input : Sniffed DNS packets from suspicious Linux namespaces via `pcap`; all kernel eBPF maps; eBPF LRU hash mapsss (`malicious_process_map`, `src_port_task_struct_map`, `task_struct_redirect_ct_map`)
Output : Updates to `malicious_process_map`

```
Sniff traffic over veth interfaces in isolated namespaces;
if DNS layer not present in skb>data then
    return;
Extract L4 transport ports: src_port, dest_port;
Extract DNS userspace features: tx_id, qd_count, ans_count, query class/type, domain length;
Fetch task_struct from src_port_task_struct_map with key=src_port;
Extract process_id, thread_group_id from task_struct;
Fetch isBlacklistedSLDFound from userspace LRU hash;
if isBlacklistedSLDFound then
    Update is_malicious flag in malicious_process_map with key=process_id, value=true;
    Fetch current_suspicious_ct from task_struct_redirect_ct_map with key=task_struct;
    if current_suspicious_ct > MAX_MALICIOUS_THRESHOLD then
        Send SIGKILL to process_id;
        // Terminate the malicious process previously extracted from src_port_task_struct_map, clean malicious_task_struct_map
        Delete from malicious_process_map with key=process_id;
    return;
Pass features to ONNX model for inference;
if Inference == MALICIOUS then
    Emit malicious threat events to message broker;
    Blacklist SLD for related DNS records in userspace LRU malicious cache;
    Update is_malicious flag in malicious_process_map with key=process_id, value=true;
    Fetch current_suspicious_ct from task_struct_redirect_ct_map with key=task_struct if current_suspicious_ct > MAX_MALICIOUS_THRESHOLD then
        Send SIGKILL to process_id;
        // Terminate the malicious process previously extracted from src_port_task_struct_map, clean malicious_task_struct_map
        Delete from malicious_process_map with key=process_id;
    return;
else if Inference == BENIGN then
    ;    // No immediate action; future packets will be tracked and evaluated
    return;
```

4.2.3 DNS Exfiltration via Encapsulated Traffic

In the Linux kernel network stack, encapsulated traffic is managed through virtualization drivers that extend the netdev core interface with associated RX and TX queues. These drivers support protocol encapsulation across L2, L3, and L4 layers. The current implementation of the eBPF agent focuses on L2-level encapsulation over software network devices, not on tunnels that rely on kernel cryptographic primitives via the keyring, such as those

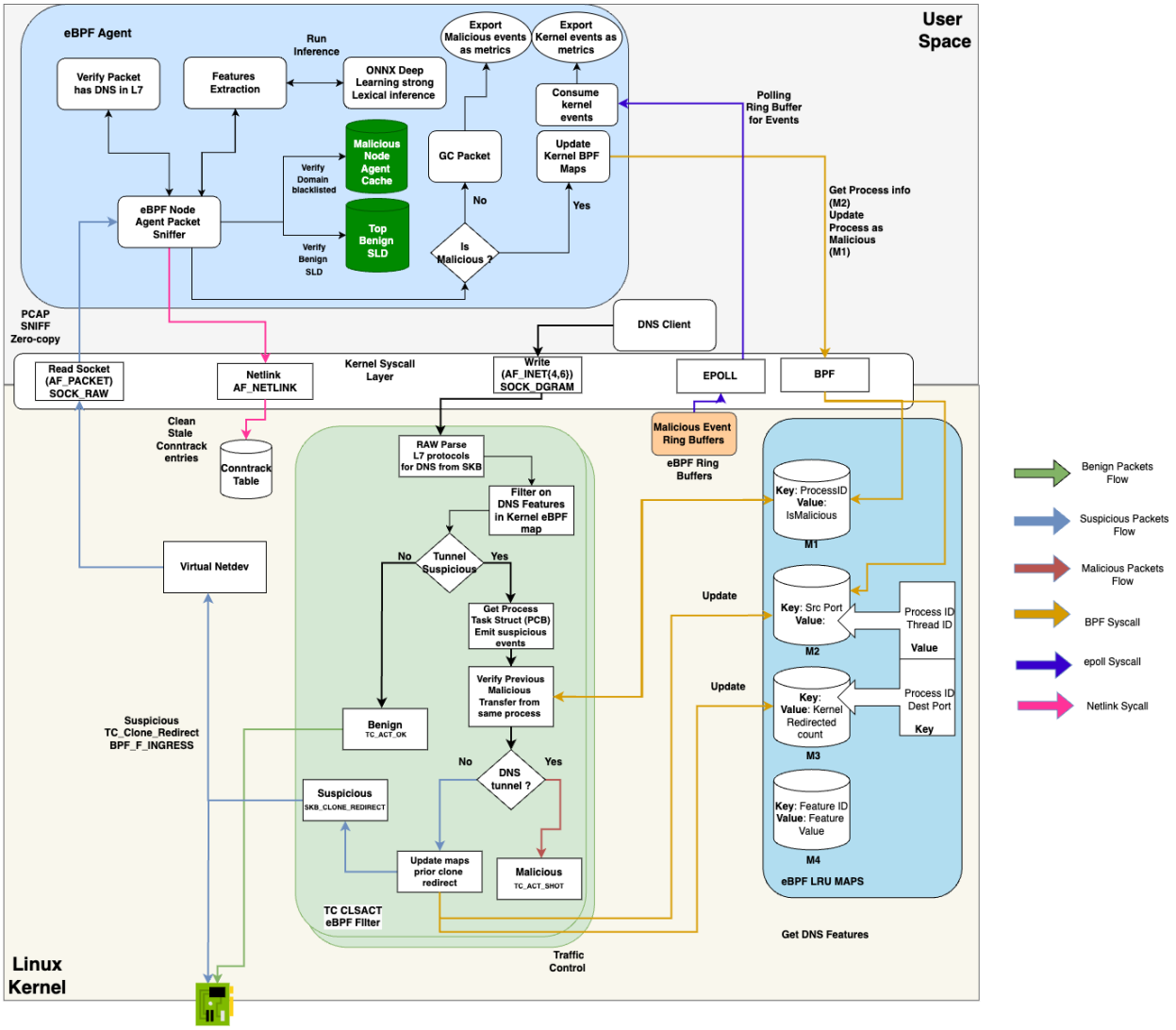


Figure 4.5: eBPF Agent DNS Exfiltration Prevention Flow for passive phase

used by OpenVPN, IPsec, or WireGuard. This design choice aligns with the typical behavior of the DNS protocol, as DNS resolution rarely occurs over VPN tunnels. In this context, DNS exfiltration over encapsulated traffic is primarily limited to VLAN and TUN/-TAP software network devices. VLAN encapsulation is handled during the SKB parsing in the active phase by the TC eBPF program, where L2 headers (e.g., 802.1Q, 802.1AD) are stripped to expose the inner packet for DPI. TUN/TAP interfaces are virtual software devices exposed to userspace as file descriptors. Malicious processes can write tunneled packets containing DNS data directly to these interfaces, bypassing traditional inspection paths. The kernel performs L3 encapsulation on the sender side and L2 decapsulation on the TAP (receiver) side before forwarding the traffic. These interfaces, typically created using iproute2 or netlink, forward traffic through a physical NIC. The eBPF agent currently handles only plaintext-encapsulated traffic. Since encapsulation occurs at higher layers of the network stack, the SKB often lacks visibility into these details—except in VLAN-tagged packets. To address DNS tunneling over TUN/TAP devices, the agent injects kprobes into the tun_chr_open kernel driver function, which is responsible for creating tunnel interfaces. When a new TUN/TAP device is created, an event is pushed to userspace via a kernel ring buffer. The agent responds by attaching the same DPI eBPF program to the TC egress hook on the new interface, enforcing the same detection and prevention logic used in the active phase. The exported ring buffer event also includes rich telemetry about the process that created the tunnel, enhancing monitoring and threat attribution. Figure 4.6 illustrates the detailed flow.

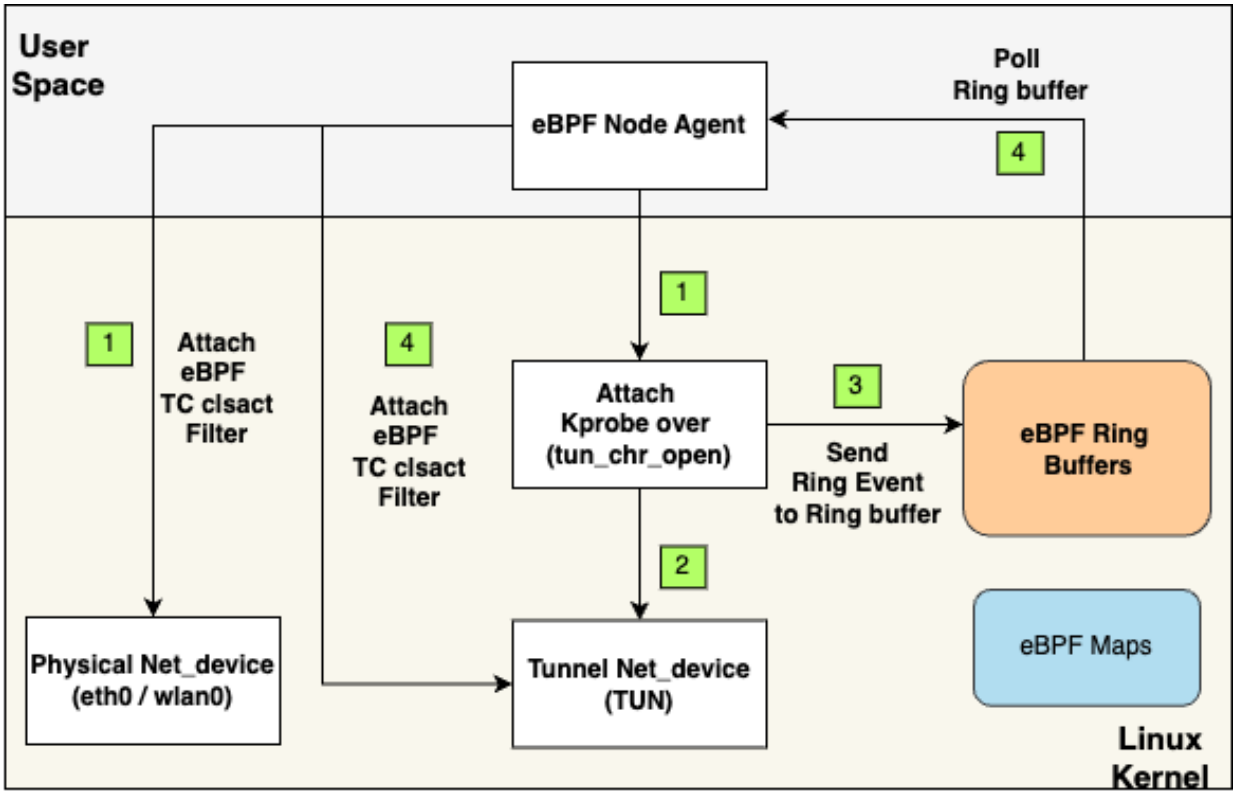


Figure 4.6: eBPF Agent Prevention flow over Tun/Tap Driver kernel function

4.2.4 Feature Analysis in Data Plane

Features used by the eBPF agent in the data plane are derived from real-world DNS exfiltration traffic and conform to DNS protocol standards, with a focus on lexical attributes that differentiate malicious anomalies from normal DNS traffic. According to RFC 1035, DNS packets over UDP are limited to 512 bytes regardless of the maximum transmission unit (MTU) of the network interface. DNS domains may be up to 255 characters long (including periods) and must adhere to label length restrictions of 127 characters maximum per label and 63 characters per individual label for queries such as A (address) records. Other query types like TXT and NULL may carry non-domain information but still fit within the 512-byte UDP limit. Larger payloads are supported through the EDNS extension by fragmentation, while TCP-based DNS transport also allows larger payloads without exceeding these limits.

Taking into account these constraints, feature selection is divided into two parts: first, the kernel features used by the eBPF programs to classify, filter, and redirect suspicious DNS packets within the kernel; second, the features used by the userspace deep learning model to perform enhanced lexical analysis of DNS payloads and detect obfuscation in exfiltrated data.

Kernel-space features

Due to the kernel-level instruction count and complexity limits enforced by the eBPF verifier, DPI is restricted to inspecting only the first DNS question record, which is parsed from the DNS header by the eBPF TC program attached to the egress path. This design aligns with modern DNS behavior, as legitimate queries almost always contain a single question; therefore, the detection of multiple questions is treated as a strong anomaly signal. Feature selection was guided by common patterns in DNS tunneling and C2 abuse while ensuring in-kernel efficiency. Numeric features such as domain length and label count are compared against minimum and maximum thresholds configured by the userspace loader during initialization. Suspicious DNS query types such as NULL and TXT are flagged due to their ability to carry arbitrary payloads, while any query class other than 'Internet' (IN), per RFC 1035, is also considered anomalous. These features are inserted into a kernel feature map for real-time classification. If a feature exceeds its threshold, it is marked malicious; if within bounds, it is flagged suspicious; otherwise, it is benign. This classification logic is uniformly applied across both active and passive modes, even for encapsulated traffic, where tunnel headers are removed before inspection. The features were chosen for their semantic relevance to DNS abuse, verifier safety, and low overhead, allowing fast, accurate classification, and real-time enforcement entirely within the kernel. The kernel features are detailed in Table 4.3.

Userspace deep learning model features

The deep learning model is trained on eight lexical features detailed in These features were selected through an in-depth analysis of DNS exfiltration behavior, based on real-world at-

tack samples, open-source C2 toolkits, and proprietary DNS tunneling frameworks. The focus is on detecting encoded payloads embedded in DNS queries by analyzing structural and statistical anomalies in the query format. Specifically, malicious queries often exhibit either an unusually high number of labels (subdomains) or fewer labels with unusually long lengths, both reaching the peak of the limits, as explained in RFC 1035. Moreover, encoding algorithms as explained before often introduce high entropy and randomness in the payload. These characteristics, derived from protocol-aware inspection and empirical adversary behavior, form the input to the model. Table Table 4.4 summarizes the complete set of features.

4.2.5 Datasets

The trained deep learning model utilizes three main datasets for training. First, Cisco's top 1 million SLDs are not used for model training, but instead are loaded into an in-memory LRU cache by the eBPF agent as the most benign SLD. This design optimizes performance by preventing model inference on these domains, since any sniffed DNS traffic containing them in the SLD will not be an exfiltrated packet because their auth zones are owned by authorized sources. Note that, as explained before, the reliance on domain scoring is used; however, the LRU cache is completely reprogrammable from the control plane. Second, for model training, it uses the DNS exfiltration dataset by Ziza et al., which includes live-sniffed DNS traffic from an ISP, consisting of around 50 million benign and malicious samples [30]. Due to the smaller number of malicious samples compared to benign ones, and to avoid training bias, a synthetic data set was generated using open-source exfiltration tools: DET, DNSExfiltrator, DNSCat2, Sliver, Iodine, and custom DNS exfiltration scripts. These exfiltrated data sets captured all forms of data obfuscation for different encoding or encryption algorithms, as explained Table 2.1, including tunneling and raw exfiltration, in a wide range of file formats including text (markdown, txt, rst, raw configuration files), image (jpg, jpeg, png), and video (mp4), generating a combined data set of around 3.8 million domains, covering all exfiltration patterns, with 50% benign and the remaining malicious.

Feature	Description
subdomain_length_per_label	Length of the subdomain per DNS label.
number_of_periods	Number of dots (periods) in the hostname.
total_length	Total length of the domain, including periods/dots.
total_labels	Total number of labels in the domain.
query_class	DNS question class (e.g., IN).
query_type	DNS question type (e.g., A, AAAA, TXT).

Table 4.3: DNS Features in Kernel

Feature	Description
total_dots	Total number of dots (periods) in DNS query.
total_chars	Total number of characters in DNS query, excluding periods.
total_chars_subdomain	Number of characters in the subdomain portion only.
number	Count of numeric digits in DNS query.
upper	Count of uppercase letters in DNS query.
max_label_length	Maximum label (segment) length in DNS query.
labels_average	Average label length across the request.
entropy	Shannon entropy of the DNS query, indicating randomness.

Table 4.4: DNS Features in Userspace

4.2.6 Deep Learning Model Architecture

The deep learning component in the userspace enhances the detection accuracy of the eBPF agent by recognizing a wide range of obfuscation techniques within DNS payloads: capabilities that are otherwise infeasible to implement in kernel space due to constraints of the eBPF verifier. Built in TensorFlow, the model uses a sequential dense neural network with eight lexical input features in userspace as explained before. It comprises three hidden layers with 16 neurons each and a final sigmoid-activated output neuron for binary classification. ReLU is used as the activation function between layers, and the model is trained using the Adam optimizer with a learning rate of 0.001 and binary cross-entropy as the loss function. A total of 25 epochs were selected to prevent overfitting while ensuring sufficient convergence on a dataset of approximately 3.8 million rows. To ensure efficient training, TensorFlow's

shuffler and autotuned prefetch policies are used to minimize I/O overhead, and the training is parallelized using TensorFlow's MirroredStrategy for multi-GPU acceleration. After training, the model is exported to the ONNX format (Open Neural Network Exchange) for lightweight deployment. The ONNX model is integrated into the eBPF agent as a submodule using the ONNX Runtime, communicating via Unix domain sockets (IPC). Although this design introduces some inference latency, it is mitigated by a first-line caching layer inside the agent, including an LRU-based blacklist and an SLD-level domain cache as explained in userspace eBPF agent filtering section for both modes. ONNX was selected due to its runtime efficiency and broad support, despite limited maturity in the Golang ecosystem. The ONNX-based submodule supports optional hardware acceleration on CPU or GPU, and the model is quantized using ONNX's dynamic quantizer. This reduces float32 weights to lower-precision formats per neuron, minimizing memory usage and inference latency. Compared to formats such as HDF5 or Pickle, ONNX offers a compact and performant graph representation that ensures that the agent maintains a minimal footprint under peak load. Figure 4.7 depicts the ONNX representation of the quantized deep learning model architecture.

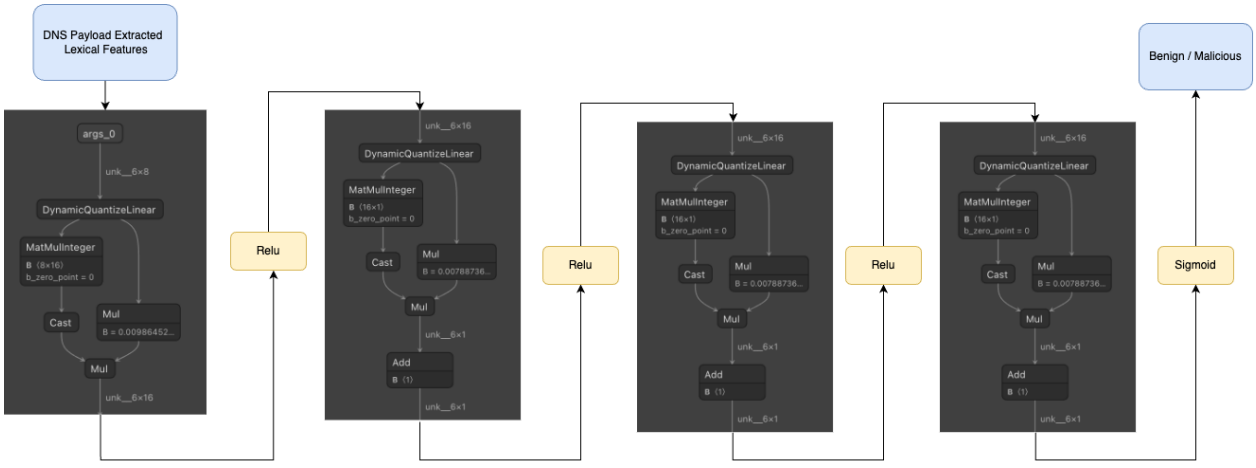


Figure 4.7: DNS Data obfuscation detection Deep Learning Model Architecture

Kafka Topic Name	Description
`exfil-sec`	Kafka topic used by the eBPF agents in data plane to stream prevented DNS threat events.
`exfil-sec-infer-controller`	Topic used by the controller to publish dynamic domain blacklists to DNS servers for data plane eBPF agent update userspace caches.

Table 4.5: Kafka Stream Topics Used in the eBPF agent

4.2.7 Thread Events Streaming and Metrics Exporters

When an eBPF agent flags a DNS packet as malicious in the data plane and contains an exfiltrated payload, the agent streams a threat event using Kafka producers. These producers are embedded in the compiled eBPF userspace binary and configured to send data to a remote Kafka broker. Each eBPF agent also includes Kafka consumers. Each agent is assigned a unique application ID derived from the local node's IP address, combined with a randomly generated ID to form the Kafka consumer group ID. This design ensures strong decoupling between agents, enabling massive horizontal scalability. Data plane nodes can scale independently without relying on a shared consumer group. Kafka producers operate asynchronously, allowing the agent to emit threat events while concurrently consuming control plane topics streamed by the inference controller. These topics carry resolved malicious domains, enabling each data plane node to update its local blacklist even if the exfiltration was detected on a different node, allowing cross-node security enforcement in the data plane. Additionally, consumed events allow eBPF agents across nodes to apply dynamic Layer 3 filters in the kernel, supporting cross-protocol correlation. While threat events focus on detected exfiltration attempts, kernel-space eBPF programs also export deep system-level metrics. These metrics are exposed via Prometheus, allowing the controller to scrape and monitor them across all nodes in real time. This centralized observability supports both the analysis of blocked threats and continuous system behavior tracking. Table 4.5 explains the details.

4.3 Control Plane

As illustrated in the overview of the component components of the control plane, the controller is designed to be entirely stateless, relying on the GPSQL backend used by PowerDNS for DNS state management and RPZ zones to track malicious domains. The control plane currently consists of a single Tomcat web server, with Spring Kafka consumer integrated to consume malicious threat events from the exfil-sec Kafka topic. These events are emitted by all endpoints in the data plane and contain blacklisted domain metadata and node-level context that identify where the threat was detected. Upon consuming these events, the control plane dynamically updates the RPZ backend with the (SLDs) associated with malicious domains in threat events. This update enforces DNS-level security enforcement, which causes the DNS server to start dropping any queries to these domains. To optimize performance and prevent malicious DNS queries from ever reaching the DNS server again once blacklisted, the control plane also writes to another Kafka topic (exfil-sec-infer-controller). This topic is consumed by all data plane nodes to rehydrate their local malicious userspace caches, effectively reprogramming the eBPF agents at each endpoint to drop related packets immediately at the endpoint, reducing network hops. For enhanced security, the control plane additionally performs DNS resolution on the malicious domains to extract their corresponding Layer 3 addresses. These addresses, which represent active C2 or tunneling servers on the public internet, are also streamed in exfil-sec-infer-controller Kafka topic used to dynamically enforce layer 3 network policies inside the kernel post consumed by the eBPF agents deployed across the data plane for cross protocol correlation. With fully asynchronous, bidirectional communication between the control plane and the data plane via Kafka, the architecture enables continuous reprogrammability of data plane nodes to enforce dynamic kernel-level security policies. It also supports the horizontal scalability of individual components crucial for production-grade cloud environments. This design directly targets and stops DGA, providing robust protection against mutation at both Layer 3 and Layer 7.

4.4 Distributed Infrastructure

As described earlier in the security framework overview, the distributed infrastructure consists mainly of a PowerDNS authoritative server, a PowerDNS recursor, a PostgreSQL-based GPSQL backend, Kafka brokers, and Prometheus scrapers. These scrapers collect metrics exposed by eBPF agents deployed in the data plane. Notably, the eBPF agents do not handle malicious DNS exfiltration over TCP due to the complexity of the TCP state machine within the kernel. To address this, TCP-based DNS exfiltration attempts are intercepted in userspace by PowerDNS recursor query interceptors acting as middleware, with details outlined in Algorithm 7. Because the PowerDNS recursor supports only Lua-based interceptors, a custom Lua module was developed. This interceptor extracts features from DNS queries received over TCP and performs inference using a serialized ONNX deep learning model. Leveraging Lua's lightweight, high-performance nature, the interceptor accesses the GPSQL backend's RPZ table, which contains known malicious domains. This enables returning `NXDOMAIN` responses for blacklisted queries, skipping inference to improve throughput. Furthermore, the interceptor employs PowerDNS internal domainsets — fast, in-memory trie-based caches of malicious domains detected over TCP. These caches enable rapid filtering of repeated exfiltration attempts. To maintain accuracy without inference overhead, the domainsets periodically synchronize with the RPZ, ensuring up-to-date enforcement.

Algorithm 7: `PowerDNS DNS Query Interceptor`

```
1  qname ← dq.qname.toString();
2  if dq.isTcp then
3  |   result ← extractFeaturesAndGetremoteInference(qname);
4  |   if result["threat_type"] then
5  |   |   insertMaliciousDomains(qname);
6  |   |   dq.rcode ← NXDOMAIN;
7  |   |   return true;
8  if sf_blacklist.check(getSLD(qname)) then
9  |   dq.rcode ← NXDOMAIN;
10 |   return true;
11 if sf_blacklist.check(getSLD(qname)) then
12 |   dq.rcode ← NXDOMAIN;
13 |   return true;
14 return false;
```

Chapter 5

EVALUATION

This chapter evaluates the effectiveness of the security framework in a distributed setting with comprehensive evaluation results and analysis.

5.1 Environment Setup

The security framework was deployed across eight CSSVLAB nodes (CSSVLAB01–08), each running Ubuntu 24.02 with Linux kernel 6.12 on Intel Xeon Gold 6130. Each node had 8GB RAM and 24GB storage. The systems ran under the hv_netvsc hypervisor network driver, with 100 Gbits/sec network bandwidth and 8 TX/RX hardware queues aligned to CPU cores to enable optimal packet steering and parallel processing for high-throughput netflow handling. In addition, all eight CPU cores and memory resided on a single NUMA node, eliminating memory lookup overhead during kernel benchmarking. The test bench launches a custom PowerDNS authoritative and recursor server on CSSVLAB08. The controller and a Kafka single broker instance run on CSSVLAB01. Nodes CSSVLAB02–07, excluding CSSVLAB06, act as the data plane, each running the eBPF agent and using the custom PowerDNS server as the default resolver via systemd-resolved. CSSVLAB06 is used to simulate DNS exfiltration attacks against data plane nodes, tunneling DNS queries through the same PowerDNS instance. The full deployment is illustrated in Figure 5.1.

5.2 Evaluations Results

The evaluation of this security framework is presented for each individual component in the subsequent sections.

5.2.1 Data Plane

The effectiveness of the eBPF agent at the endpoint of the data plane is evaluated through quantized deep learning metrics, comparison of DNS request throughput in both operational phases (active and passive), and measurement of bandwidth and resource utilization, includ-

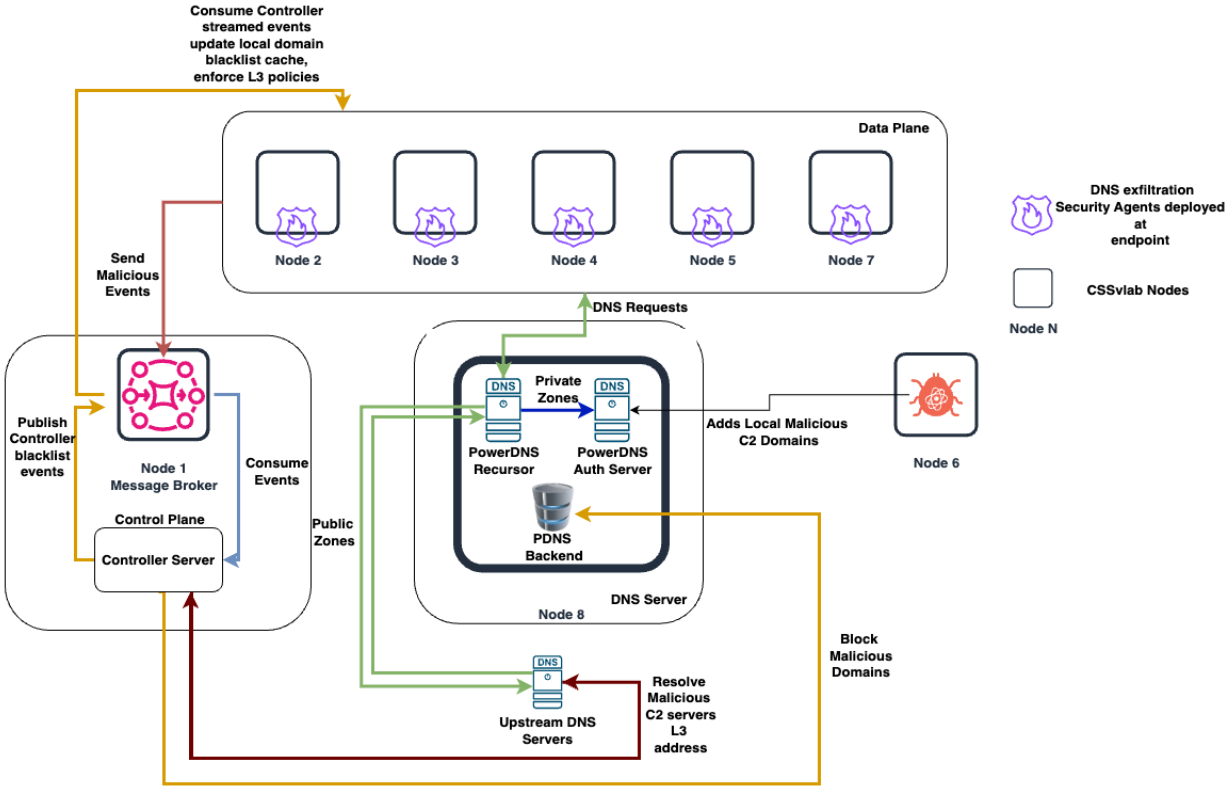


Figure 5.1: Security Framework Deployed Architecture over CSSVLAB Nodes

ing CPU and memory usage, as well as throughput of kernel events processing. Finally, the agent's resilience coverage against advanced adversary emulation frameworks is explained. Performance evaluation is performed on a single selected node within the data plane running the agent.

Deep Learning Model Evaluation

The evaluation of the trained deep learning model was performed on a 3.8 million domain data set, divided into 70% for training and 15% for validation and testing. After training, the model achieved a validation precision of 99.7%, with loss steadily decreasing per epoch and reaching 0.98% at the end of the training. Given the DNS data exfiltration use case, the model performance was evaluated primarily with an emphasis on minimizing false positives.

High false positives would not only result in the eBPF agent dropping benign DNS packets and generating false threat events, but could also terminate legitimate processes introducing operational risks. In contrast, false negatives were considered less critical, as the agent would allow malicious traffic to pass through without taking privilege-level actions. Therefore, **precision** was prioritized over recall. Specifically, precision and recall are defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}},$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Based on these considerations and a dataset engineered to capture a wide range of data obfuscation techniques, the model achieved a high precision of 99.59% and a recall of 99.87%. For runtime inference using ONNX within the eBPF agent, a binary classification threshold of 0.85 was selected. This value was chosen based on the performance of the F1 score on varying thresholds, paying careful attention to the corresponding changes in precision and recall. Since minimizing false positives was a top priority, the threshold was selected to maximize precision while still maintaining a relatively high recall. This high-precision, low–false-positive result was made possible by the selected feature set, which included Shannon entropy across various encoding and encryption schemes—enabling the model to learn effectively. Figure 5.1 and Figure 5.2 present the model's performance on both training and validation datasets, including the confusion matrix. Figure 5.3 and Figure 5.4 illustrate precision vs. recall improvements over 25 training epochs, and the F1 score and precision/recall score under different thresholds sleected to classify in the test dataset.

Metric	Training	Validation
Accuracy	0.9973	0.9997
AUC	0.9997	0.9997
Loss	0.0099	0.0091
Precision	0.9959	0.9959
Recall	0.9987	0.9988

Table 5.1: Model Evaluation Metrics

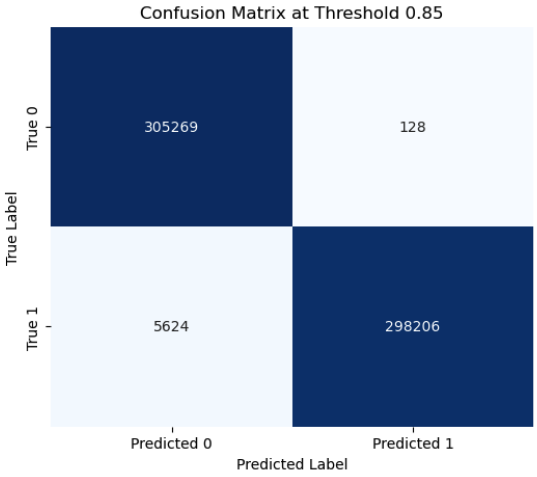


Figure 5.2: Confusion Matrix

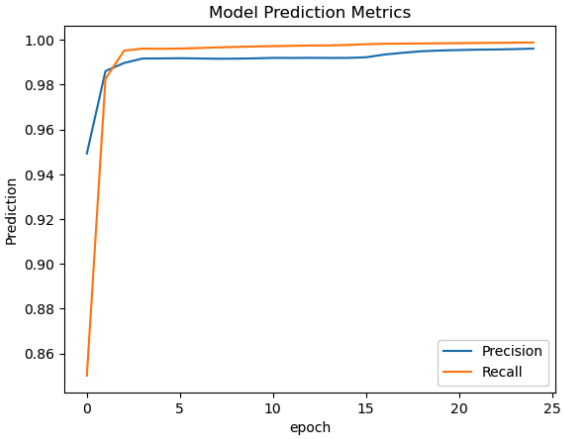


Figure 5.3: Model Precision

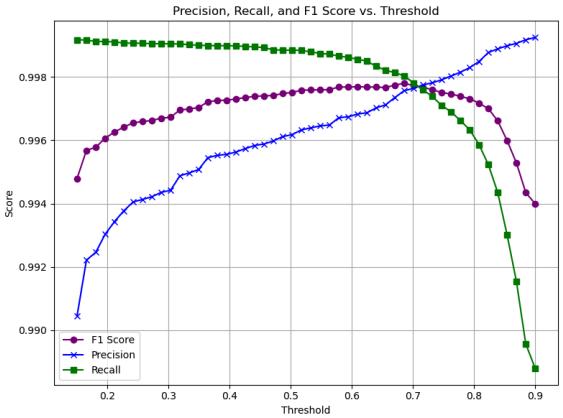


Figure 5.4: Precision, Recall, and F1 Score vs. Threshold

eBPF agent Request Throughput and Latency Metrics in Active Mode

The performance of the system was evaluated in active mode by measuring the impact on benign DNS traffic during standard end-to-end resolution, from a userspace process sending DNS requests, through kernel redirection through eBPF programs, to the network device monitored by the agent. For cache hits, the request is matched against the global SLD

cache; for cache misses, live ONNX inference is performed. The kernel feature thresholds in the eBPF map were intentionally kept stringent, causing most DNS packets to be flagged as suspicious to stress-test the throughput. Throughput was measured using DNSPerf, which quantifies both request throughput and DNS response success rates. The test locked DNSPerf to send 10,000 DNS requests per second over 20 seconds, monitoring packet loss. The recursor server assumed to be healthy with no addon impacy on DNS throughput testing. The testing node relies on the hypervisor-supported `netvsc` virtual driver and operates with a combined 8 RX/TX queues. As a result, it lacks discrete ring buffers required by the kernel for `AF_XDP` sockets, making direct packet injection into TX queues unsupported. As a result, the egress path was forced to rely on `AF_PACKET`. In the cache-hit scenario (100% benign SLD matches), the throughput ranged from 8,100 to 9,820 DNS requests/sec with zero packet loss as presented Figure 5.5. The latency ranged from near 0 ms to a maximum of 250 ms per 10k sample Figure 5.7. However, in the cache-miss path that requires live inference, throughput decreased significantly: minimum of 430 and maximum of 520 requests / sec, as illustrated in Figure 5.6, with peak latencies of up to 750 ms as demonstrated in Figure 5.8. Despite this, no packet loss was observed, which ensures reliability. The latency spike is attributed to the overhead of UNIX domain socket communication with the ONNX inference server and Python's Global Interpreter Lock (GIL), which limits concurrency. In contrast, the eBPF agent is fully compiled and highly concurrent, offering significantly better performance. It was observed that throughput becomes unstable beyond 5,000 DNS requests/sec, though such traffic volumes typically indicate malicious behavior and can be rate-limited in kernel eBPF programs. Overall, for stress testing over a prolonged period of time, the agent successfully processed a maximum throughput of 67.3 million DNS requests per hour with no packet loss, while performing deep parsing across both kernel and userspace.

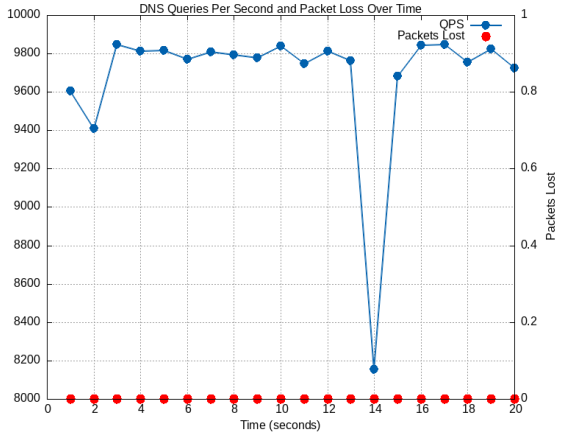


Figure 5.5: eBPF Agent: DNS Throughput for GSLD LRU Hit (10k req/s)

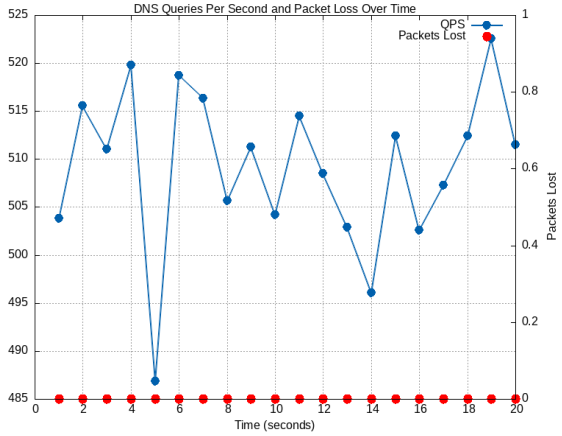


Figure 5.6: eBPF Agent: DNS Throughput, GSLD LRU Miss, ONNX (10k req/s)

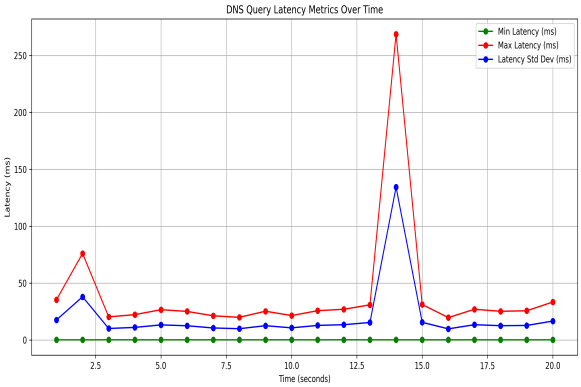


Figure 5.7: eBPF Agent: DNS Latency for GSLD LRU Hit (10k req/s)

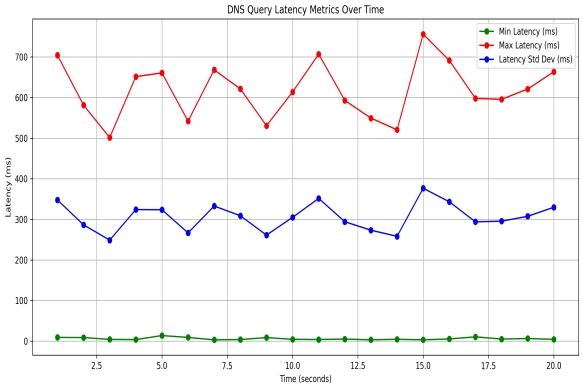


Figure 5.8: eBPF Agent: DNS Latency, GSLD LRU Miss, ONNX (10k req/s)

eBPF agent Request Throughput and Latency Metrics in Passive Mode

The primary evaluation metric in passive mode is the volume of DNS-based data exfiltration successfully prevented before terminating malicious processes. In this mode, the agent employs a clone-and-redirect mechanism, allowing original DNS packets to pass through while kernel programs inspect traffic for signs of malicious activity. Upon detection, the kernel notifies the eBPF agent to kill the responsible process. Traditional throughput and latency

are not emphasized; instead, performance is measured by how effectively the system detects and stops beaconing implants that transmit data over DNS, often across random UDP ports. Figure 5.9 illustrates the total volume of exfiltrated data stopped by the eBPF agent before the end of the process. For example, DNSCat2, configured with a 20 second beaconing interval and exfiltrating through various types of DNS records (MX, TXT, CNAME, HTTP, SRV, etc.), demonstrated the system's strength in delaying termination just enough to observe beacon patterns, empowering administrators to tune termination thresholds for maximum insight.

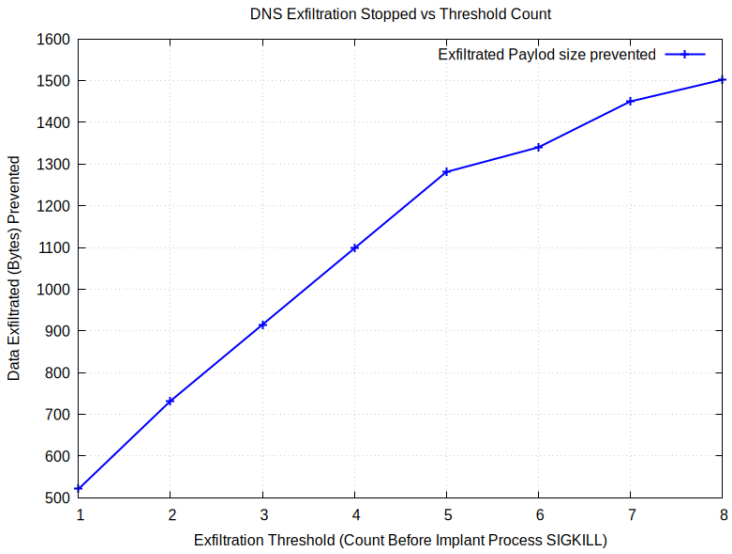


Figure 5.9: eBPF Agent: Volume of DNS exfiltrated data prevented vs various process kill thresholds

eBPF agent Resource Usage

The performance of the eBPF agent was closely monitored while running at the endpoint in the data plane, and the utilization of resources was measured in terms of memory and CPU utilization. During a 10-second DNSPerf benchmark at 10,000 DNS req/sec, with the agent in active mode redirecting all packets to userspace, the agent consumed approximately 310 MB of memory for the main process. This includes heap memory, as all LRU maps loaded

by agent's process for fast lookups. At a higher throughput of 100,000 DNS requests/sec, memory usage remained nearly the same, peaking at 350 MB. Figure 5.10, Figure 5.11 illustrates the memory usage for the previously explained DNS request throughput. Throughout the benchmark, the agent process consistently consumed only 8–10% CPU at peak load and remained below 2% when the system was idle. Memory usage during idle conditions stabilized around 120 MB. These results demonstrate that the agent is highly lightweight, with minimal CPU and memory footprints and no observable impact on other processes running on the endpoint. Considering that the CPU usage for the injected eBPF programs in the kernel peaked at 1.2% at peak load, while for the idle system remained below 0.2%. In addition, the agent binary compiled size with all the explained features is around 22 MB on ARM and 24 MB on x86_64 architectures, ensuring there is minimal storage impact on the endpoint caused by the eBPF agent.

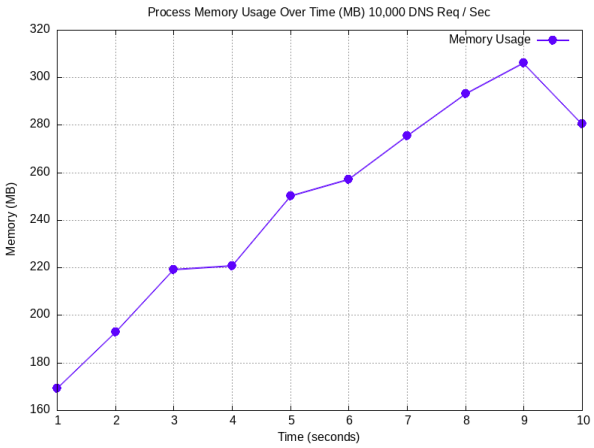


Figure 5.10: eBPF Agent Process Memory Usage for 10k DNS req/sec

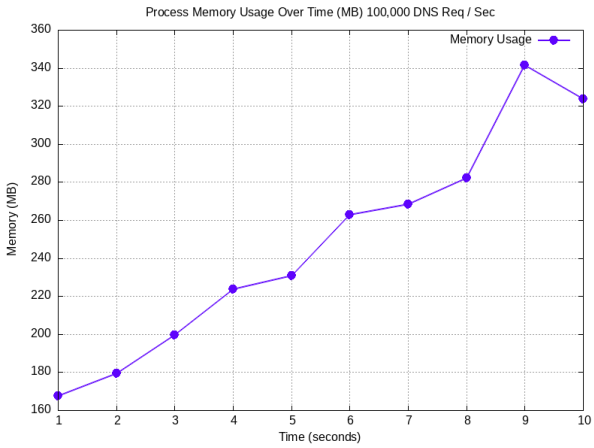


Figure 5.11: eBPF Agent Process Memory Usage for 100k DNS req/sec

5.2.2 eBPF agent effectiveness over DNS exfiltration tools

The eBPF agent was evaluated across all data plane endpoints against widely used, high-reputation DNS-based C2 frameworks commonly leveraged by red teams in production for penetration testing. The tests focused on detecting and disrupting C2 and exfiltration activ-

ities tunneled through DNS. The CSSVLAB06 node acted as the primary attacker, issuing commands through a DNS server to the nodes of the targeted data plane. The Sliver C2 framework, developed by BishopFox, was evaluated not only for basic data exfiltration, but also for its support of advanced C2 operations. These included remote shell access, remote code execution, file transfers, remote port forwarding, and establishing persistent backdoors via dynamically opened ports, all encapsulated within DNS traffic. The eBPF agent enforced kernel-level mitigation from the very first C2 command, immediately breaking the communication channel, forcing implant retries, and ultimately terminating the implant process. This was applied to both beacon-based and session-based implants. Although Sliver does not support DNS C2 over randomized UDP ports, the agent's ability to handle transport layer obfuscation was evaluated using dnscat2, which supports such techniques. The same set of C2 vectors was executed with the agent running in both active and passive modes. In both cases, the agent successfully enforced the mitigation policies, resulting in full-session downtime without exfiltration. For raw data exfiltration scenarios, tools like DET, which lack UDP port randomization capabilities, were used. With the agent operating in active mode, raw exfiltration attempts were immediately blocked with zero data loss, and the offending processes were terminated. In addition, DNS tunneling tools such as iodine were tested for their ability to tunnel arbitrary protocols through DNS utilizing kernel encapsulation through TUN/TAP network interfaces. The agent effectively intercepted and dropped the encapsulated payloads, confirming its ability to prevent a broad spectrum of DNS-based exfiltration and tunneling strategies. In addition, other tools that exploit bulk exfiltration were also evaluated, and the agent successfully prevented all basic raw exfiltration attempts. Figure 5.12 illustrates the attack vectors prevented and all the tools evaluated.

DNS Exfiltration Type	Tool	DNS Overlay Random UDP Port	Kernel traffic encapsulation TUN/TAP	Beacon Implants	Attack Vector tunneled through DNS	Framework prevents in real-time.	Mitigation Action
C2 over DNS	Sliver	No	No	Yes	Remote Shell	Yes	Disrupts C2 communication from first command request, ensures negligible data loss monitors and kill C2 implants
					Remote Code Execution		
					File Upload/Download		
					Remote Port forwarding to tunnel any protocol		
					Open Remote ports for backdoor		
					Remote Screenshot		
	Dnscat2	Yes			Remote shell	Yes	
					File Upload/Download		
					Open Remote ports for backdoor		
					Remote Port forwarding to tunnel any protocol		
Raw exfiltration	DET	No	No	No	Raw file exfiltration	Yes	Prevents exfiltration, and kills process exfiltrating data over DNS
DNS tunnelling	Iodine	No	Yes	No	Raw file exfiltration	Yes	
					Tunnel any protocol		

Figure 5.12: Framework Coverage Against Real-World DNS-Based C2 and Exfiltration Tools

Control Plane

The evaluation of the controller's stateless server focuses on its effectiveness in accurately consuming threat events transmitted from data plane nodes to a Kafka topic, blacklisting domains in RPZ, and redistributing those events to data plane nodes to rehydrate their malicious domain caches. Figure 5.13 illustrates the structure of threat events streamed from eBPF agents in the data plane, serialized as JSON, and published to a Kafka topic. These events are consumed by the controller and used to blacklist domains in the RPZ zone of

the DNS server. Figure 5.14 shows the published threat event structure by the controller on the Kafka topic (exfil-sec-infer-controller) for the data plane nodes to consume. As explained previously, the controller's published events also include Layer 3 (IPv4/IPv6) addresses of remote C2 nodes. This enables agents in the data plane to enforce cross-protocol correlation by dynamically injecting L3 filtering rules into the kernel. This design not only blocks DNS-based DGA communication, but also halts all protocol-level traffic to malicious IPs, offering strong protection from distributed threats and elevating system-level security enforcement directly inside the kernel.


```
{
  "AuthZoneSoaservers": null,
  "AverageLabelLength": 26,
  "Entropy": 4.177708,
  "ExfilPort": "53",
  "Fqdn": "716e039e8200000000cd2d44004d13e25f84e790c44fe3c4b09662534b59
    .9b39fd2238768b366c5b71cf573a2aeb9f2066df80d6bdc8ca1166b96090
    .fe413707839f738e3500a0b7b1.dnscat.strive.io",
  "IsEgress": true,
  "LongestLabelDomain": 60,
  "NumberCount": 102,
  "Periods": 5,
  "PeriodsInSubDomain": 4,
  "PhysicalNodeIpv4": "10.158.82.19",
  "PhysicalNodeIpv6": "2607:4000:700:1003:215:5dff:fe52:3c0d",
  "Protocol": "DNS",
  "RecordType": "CNAME",
  "Subdomain": "716e039e8200000000cd2d44004d13e25f84e790c44fe3c4b0966253
    4b59.9b39fd2238768b366c5b71cf573a2aeb9f2066df80d6bdc8ca1166b96090
    .fe413707839f738e3500a0b7b1.dnscat",
  "Tld": "strive.io",
  "TotalChars": 160,
  "TotalCharsInSubdomain": 152,
  "UCaseCount": 0
}
```


Figure 5.13: Controller consumed threat event


```
{
  "fqdn": "0c470351e00000000038f8eda78b723c294042cee756c0225fb675709f1e
    .5a927edcd7cbeedf0226ae252567185f9b750969c400f032dc033da5b432
    .a2fa46e0869940acc7ef2fc8a5.det.strand.com",
  "tld": "strand.com",
  "recordType": "MX",
  "detectedThreadNodeIpv4": "10.158.82.19",
  "detectedThreadNodeIpv6": "2607:4000:700:1003:215:5dff:fe52:3c0d",
  "resolveAddressMaliciousC2Domains": [
    "10.158.82.53"
  ],
  "forcedUnBlocked": false
}
```


Figure 5.14: Controller streamed threat event

Distributed Infrastructure

The performance evaluation of the distributed infrastructure focuses solely on the DNS server, specifically evaluating the throughput impact of the Lua-based interceptor running on the PowerDNS Recursor. As shown in Figure 5.15, the server was benchmarked under a sustained load of 10,000 DNS requests per second. Due to reuse of the same inference server design as in the data plane - together with reliance on UNIX sockets for interprocess communication and Python's internal concurrency limitations - throughput dropped to as low as 490 DNS requests per second. The latency measurements, illustrated in Figure 5.16,

peaked at approximately 750ms, with the mean deviation stabilizing around 380ms. All TCP traffic benchmarks in the kernel were conducted with `TCP_FAST_OPEN` enabled, allowing application data to be sent with the initial SYN packet. This reduced the impact of the TCP 3-way handshake on throughput and enabled accurate latency measurements for DNS-over-TCP traffic. In addition, Figure 5.17 shows the resulting blacklisted domains stored in the PowerDNS GPSQL backend. The controller supplements this list with additional metadata, allowing for selective unblocking of domains based on operational requirements. In such cases, the controller initiates a forced reprogramming of all data plane nodes, overriding local suspicion heuristics to permit DNS traffic for the specified domain.

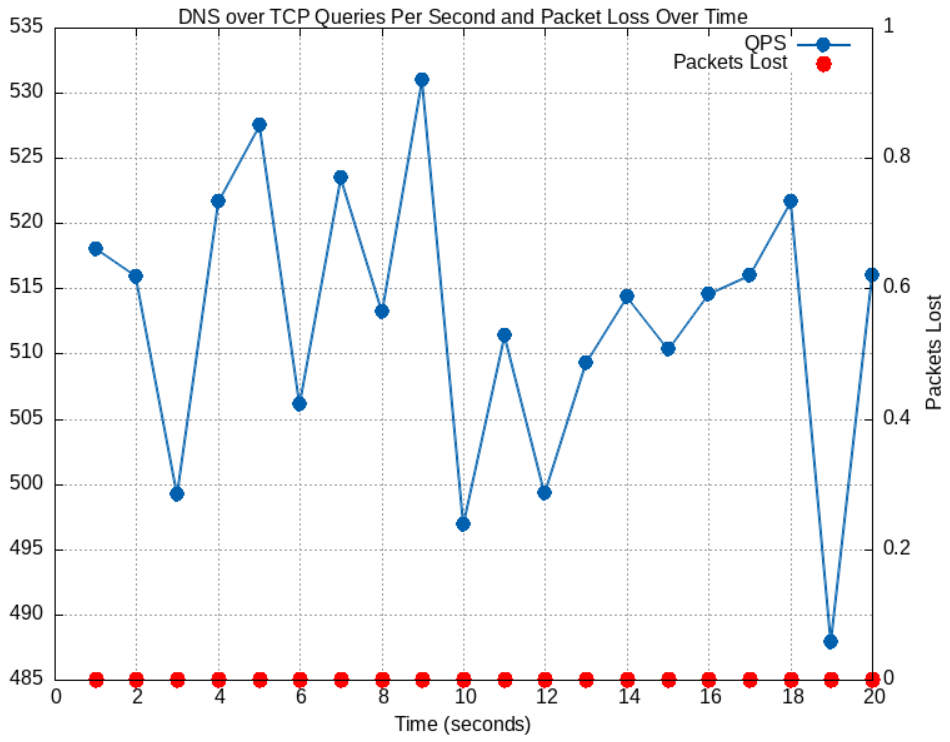


Figure 5.15: DNS Server Throughput for 10k DNS req/s over TCP

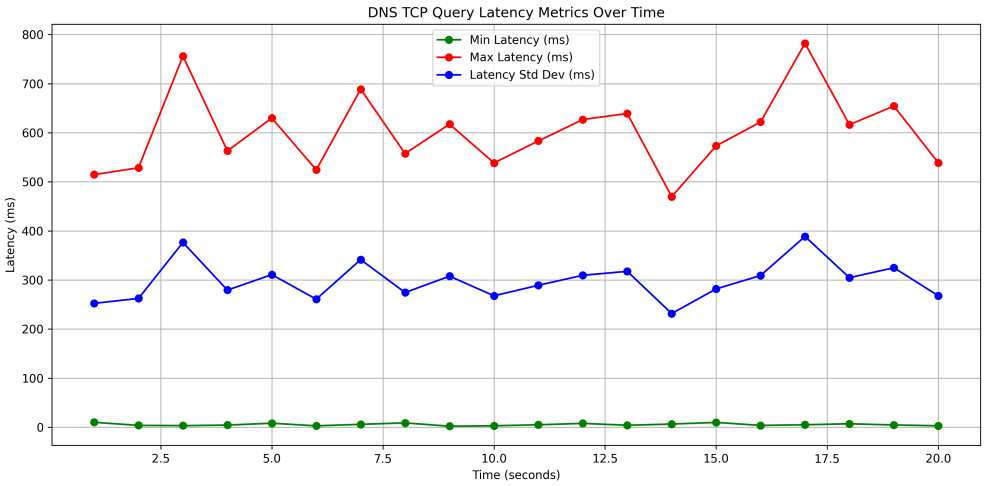


Figure 5.16: DNS Server Latency for 10k DNS req/s over TCP

sld	fqdn	forced_unblocked	is_transporttcp
bleed.io	ytbw2z.t.bleed.io	FALSE	FALSE
soft.de	f21a03d91c000000009ce6ce661c35a3725562b917b27a2…	FALSE	FALSE
spooky.io	0b1c0369c7000000006ac410c42714c5ab794569e9cace9…	FALSE	TRUE
strand.com	92d90351e00000000038f8eda78b723c294042cee756c02…	FALSE	FALSE
strive.io	21a1039e8200000000cd2d44004d13e25f84e790c44fe3c…	FALSE	FALSE

Figure 5.17: Blacklisted domains in RPZ zone on DNS server

Chapter 6

CONCLUSION

This chapter concludes the project by summarizing its contents and outlining future directions.

6.1 Summary

This security framework significantly advances the state-of-the-art by developing a novel architecture to prevent data exfiltration over DNS, directly addressing the critical gaps left by traditional approaches. The existing literature and solutions for DNS exfiltration prevention remain largely stagnant, centered on centralized detection and userspace anomaly detection systems or proxy DPI, which are inherently inadequate and lack the strength to stop sophisticated DNS-based exfiltration, especially those leveraging advanced C2 attack vectors. In contrast, this framework introduces a new paradigm: kernel-enforced endpoint security. It acts as a privileged layer beneath existing endpoint security solutions, enabling strict enforcement inside the operating system. Using eBPF to reprogram the core kernel subsystems, paired with enhanced deep learning based , these across system enforcement demonstrate comprehensive defensive strength, stopping and killing the most advanced DNS C2 implants in real-time with advanced attack vectors commonly used by top-tier adversary emulation frameworks. Furthermore, by combining a layered approach with system security embedded with an endpoint-centric design, this architecture elevates endpoint defense beyond the limitations of userspace alone, providing detailed visibility into malicious activity, and rapid response to malicious implants. The following points summarize the strengths of the security framework.

- **Instant DNS C2 Disruption** – Immediately blocks DNS-based command-and-control channels upon initiation, stopping covert communication at the source.

- **Active Implant Process Detection & Termination** – Detects malicious processes that use DNS for exfiltration and terminates them in real time.
- **Tunnel and Encapsulation-Aware Defense** – Eliminates DNS tunnels, protocol-agnostic payload encapsulation, encapsulated traffic in the kernel, including DNS overlay over random UDP ports.
- **Prevents Sophisticated C2 over DNS** – Effectively stops advanced C2 command attacks - including, but not limited to, remote code execution, reverse tunnels, protocol tunneling, port forwarding, remote file compromise, remote process side channeling, and C2 multiplayer modes.
- **DGA Mitigation with dynamic L3 kernel Network Policies** – Dynamically blacklists domains, reprogram agents in data plane, and enforces layer 3 network policies for cross-protocol coordination.
- **Rich metrics and system observability** – Exports rich metrics to Prometheus, enabling visibility across scaled data planes and providing robust system-level observability at each endpoint.
- **Horizontal Scalability** – Supports horizontal scalability as in production cloud environments

Limitations and Future Work

Limitations

- **Increased Latency for Active mode of Agent** While Active Mode introduces some latency due to live redirection of DNS over UDP traffic from the kernel's TC eBPF program to the userspace for deep scanning, this overhead is still significantly lower than remote proxy-based DPI solutions. If latency is cocnern at endpoint kernel feature values can be eased for kernel eBPF programs to perform less aggressive DPI.
- **Potential Security Bypass for Passive mode of Agent** In passive mode, since the eBPF agent hunts for malicious activity tied to a process and kills post exceeding threshold, malicious process can bypass security by forking the child process to prevent

this agent from tracking malicious activity to parent process from kernel task_struct compared to process id.

- **High Accuracy and Latency in Deep Learning Model Training and Inference:** Although the model achieved high precision with few false positives, its performance could be further improved by incorporating more diverse poisoned samples to address unseen payload obfuscation techniques. Additionally, the use of UNIX socket-based IPC—combined with Python's limitations in true concurrency—reduced inference throughput and increased latency.
- **Absence of Encrypted Exfiltration Prevention:** The framework does not support the prevention of exfiltration through encrypted DNS channels such as DoT or DoH.
- **Absence of Encrypted Encapsulated Tunnels:** The framework does not support prevention of exfiltration over encrypted tunnels relying on kernel `xfrm` such as Wireguard, OpenVPN, IPSec.

Future Work

- **Extend Support for DNS-over-TCP and Encrypted Tunnels:** Implement detection and blocking for exfiltration of DNS over TCP in kernel eBPF programs replicating TCP state machine coupled with envoy as an L7 userspace proxy for analysis
- **Migration away from Python inference server:** Migrate the Python ONNX inference to Rust, with a wasm (web assembly) module for faster inferencing compared to interpreted languages.
- **Add In-Kernel TLS Fingerprinting:** Integrate TLS fingerprinting (e.g. JA3 / JA4) using eBPF to detect encrypted DNS exfiltration over TLS or WireGuard tunnels, supporting userspace deep learning models with detailed system-level metrics for dynamic security policy enforcement by kernel eBPF programs.
- **Rate-Limiting Based on Volume and Throughput:** Integrate egress DNS rate limiting for mass volume breaches using EDT_BPF and HTB QDISC.

BIBLIOGRAPHY

[1] Jawad Ahmed, Hassan Habibi Gharakheili, Qasim Raza, Craig Russell, and Vijay Sivaraman. Real-time detection of dns exfiltration and tunneling from enterprise networks. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 649–653, 2019.

[2] Jawad Ahmed, Hassan Habibi Gharakheili, Qasim Raza, Craig Russell, and Vijay Sivaraman. Monitoring enterprise dns queries for detecting data exfiltration from internal hosts. *IEEE Transactions on Network and Service Management*, 17(1):265–279, 2019.

[3] Ahlam Ansari, Nazmeen Khan, Zoya Rais, and Pranali Taware. Reinforcing security of dns using aws cloud. In *Proceedings of the 3rd International Conference on Advances in Science & Technology (ICAST)*, 2020.

[4] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for {DNS}. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[5] Thorsten Aurisch, Paula Caballero Chacón, and Andreas Jacke. Mobile cyber defense agents for low throughput dns-based data exfiltration detection in military networks. In *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, pages 1–8, 2021. doi: 10.1109/ICMCIS52405.2021.9486400.

[6] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. Netedit: An orchestration

platform for ebpf network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 721–734, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672227. URL https://doi.org/10.1145/3651890.3672227.

[7] Gilberto Bertin. Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, pages 1–5. The NetDev Society, 2017.

[8] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[9] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Ndss*, pages 1–17, 2011.

[10] Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. *Proceedings of netdev*, 1, 2016.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[12] Anirban Das, Min-Yi Shen, Madhu Shashanka, and Jisheng Wang. Detection of exfiltration and tunneling over dns. pages 737–742, 12 2017. doi: 10.1109/ICMLA.2017.00-71.

[13] Raja Zeeshan Haider, Baber Aslam, Haider Abbas, and Zafar Iqbal. C2-eye: framework for detecting command and control (c2) connection of supply chain attacks. *International Journal of Information Security*, pages 1–15, 2024.

[14] 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*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360807. doi: 10.1145/3281411.3281443. URL `https://doi.org/10.1145/3281411.3281443`.

[15] Nikos Kostopoulos, Dimitris Kalogeras, and Vasilis Maglaris. Leveraging on the xdp framework for the efficient mitigation of water torture attacks within authoritative dns servers. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 287–291, June 2020. doi: 10.1109/NetSoft48620.2020.9165454.

[16] Christos M. Mathas, Olga E. Segou, Georgios Xylouris, Dimitris Christinakis, Michail-Alexandros Kourtis, Costas Vassilakis, and Anastasios Kourtis. Evaluation of apache spot's machine learning capabilities in an sdn/nfv enabled environment. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364485. doi: 10.1145/3230833.3233278. URL `https://doi.org/10.1145/3230833.3233278`.

[17] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[18] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018. doi: 10.1109/HPSR.2018.8850758.

[19] Asaf Nadler, Avi Aminov, and Asaf Shabtai. Detection of malicious and low throughput data exfiltration over the DNS protocol. *CoRR*, abs/1709.08395, 2017. URL `http://arxiv.org/abs/1709.08395`.

[20] Yarin Ozery, Asaf Nadler, and Asaf Shabtai. Information-based heavy hitters for real-time dns data exfiltration detection and prevention. *arXiv preprint arXiv:2307.02614*, 2023.

[21] Jamal Hadi Salim. Linux traffic control classifier-action subsystem architecture. *Proceedings of Netdev 0.1*, 2015.

[22] Suphannee Sivakorn, Khae Hawn Jee, Yulong Sun, Livia Korts-Pärn, Zheng Li, Cristian Lumezanu, Zhiyun Wu, Liangzhen Tang, and Ding Li. Countering malicious processes with process-dns association. In *NDSS*, 2019.

[23] Jacob Steadman and Sandra Scott-Hayward. Dnsxd: Detecting data exfiltration over dns. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6, 2018. doi: 10.1109/NFV-SDN.2018.8725640.

[24] Jacob Steadman and Sandra Scott-Hayward. Dnsxp: Enhancing data exfiltration protection through data plane programmability. *Computer Networks*, 195:108174, 2021.

[25] Alexander Stephan and Lars Wüstrich. The path of a packet through the linux kernel. *Technical University of Munich, Chair of Network Architectures and Services, School of Computation, Information and Technology*, 2024.

[26] 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 Comput. Surv.*, 53(1), February 2020. ISSN 0360-0300. doi: 10.1145/3371038. URL `https://doi.org/10.1145/3371038`.

[27] Timothy D Zavarella. *A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters*. PhD thesis, Massachusetts Institute of Technology, 2022.

[28] Wenjun Zhu, Peng Li, Baozhou Luo, He Xu, and Yujie Zhang. Research and implementation of high performance traffic processing based on intel dpdk. In *2018 9th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 62–68, 2018. doi: 10.1109/PAAP.2018.00018.

[29] Aaron Zimba and Mumbi Chishimba. Exploitation of dns tunneling for optimization of data exfiltration in malware-free apt intrusions. *Zambia ICT Journal*, 1:51 – 56, 12 2017. doi: 10.33260/zictjournal.v1i1.26.

[30] Kristijan Ziza, Pavle Vuletić, and Predrag Tadić. Dns exfiltration dataset, 2023.

Chapter 7

APPENDICES

This chapter outlines additional security mechanisms designed to protect the Linux kernel from malicious eBPF programs, as well as enhanced observability features employed by the eBPF agents in the data plane. It also describes the DGA used to simulate the generation of large-scale malicious domains for DNS exfiltration attack testing. The complete security framework codebase—including all configuration files and documentation—is publicly available at GitHub. It is currently licensed under AGPLv3, with plans to migrate to GPL as part of a broader effort to incubate the project under the Linux Foundation. The repository contains all components of the system, covering both kernel and userspace implementations of the eBPF agents in the data and control planes. It also provides Docker deployment manifests for setting up Kafka brokers and scripts to deploy PowerDNS. These scripts configure all nodes in a distributed testbed to use PowerDNS as their default resolver. Alternatively, the entire infrastructure—including PowerDNS and Kafka—can be deployed on any cloud provider, depending on operational requirements. Detailed setup instructions for each component of the security framework are provided in the accompanying docs directory.

7.1 Appendix A

This section focuses on providing additional security details implemented inside kernel for protecting kernel from malicious programs eBPF agents in the data plane.

7.1.1 Kernel eBPF programs and userspace agent profiling

Given the intensive DNS parsing and enforcement logic implemented in the kernel's TC layer, comprehensive benchmarking was conducted to evaluate its performance impact on benign traffic. The documented source code can be found in the kernel/ directory, which includes logic to classify DNS payloads at line rate. Performance evaluation leveraged in-

kernel perf instrumentation and system profiling tools. The eBPF program was executed concurrently across multiple CPU cores and monitored using Netflix's bpftop tool. Under a high-throughput workload of approximately 100,000 DNS requests per second, kernel CPU usage peaked at 4% across eight cores, with a minimum of 2%. The program consistently sustained processing rates of up to 17,063 events per second—excluding hardware interrupt and soft IRQ overhead—demonstrating the efficiency of the TC-attached kernel eBPF logic. Figure 7.2 illustrates the profiling results. Complementing kernel profiling, the userspace eBPF agent—responsible for traffic handling—was profiled using Go's `pprof` tool. Analysis of CPU-intensive function call stacks revealed that approximately 75% of the agent's CPU usage is spent in BPF-related syscalls, reflecting its reliance on frequent kernel interactions to access and update eBPF program internals, particularly eBPF maps. Figure 7.1 presents the agent's flame graph.

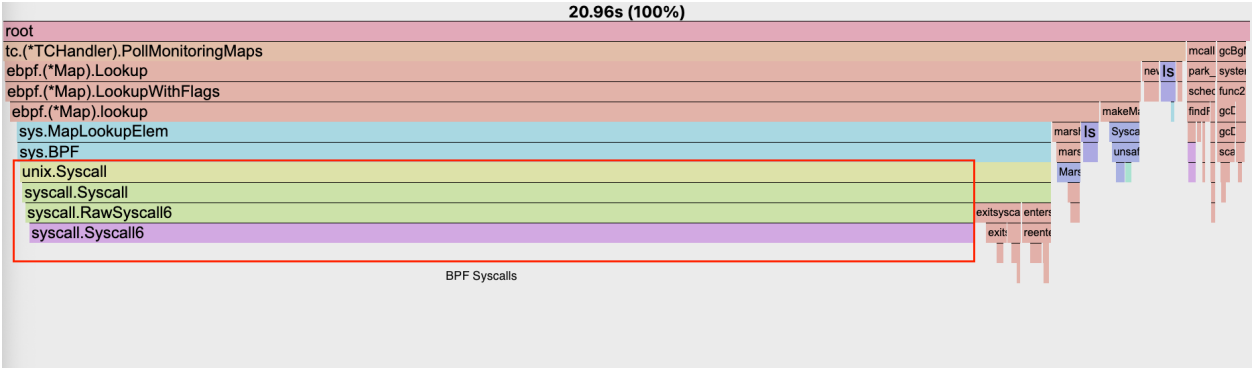


Figure 7.1: eBPF Agent: Flame Graph

7.1.2 eBPF Agent exported metrics

Table 7.1 describes all the details of the execution exported by each eBPF agent at the endpoint in data plane, with Figure 7.6, Figure 7.3, Figure 7.4 detailing some of the enhanced system-level metrics exported by the eBPF agent to Prometheus from the eBPF map and ring buffers to Grafana, a visualization tool to scrape Prometheus metrics.

Metric	**Description**

DNSFeatures	Metadata of detected DNS exfiltration packets, including extracted features.
Tunnel Interface Process Info	Tracks kernel netlink events for virtual network device creation, linked to the process that created them (UID, GID, PID).
DPI_Redirect_Count	Packet redirection count by kernel DPI logic in active mode.
DPI_Clone_Count	Count of cloned packets redirected for inspection in passive mode.
DPI_Drop_Count	Total packets dropped by kernel DPI logic.
MaliciousProcTime	Start time and duration the malicious process was alive before termination.
CPU Usage	CPU utilization of the eBPF agent in userspace.
Memory Usage	RAM usage in MB or percentage of total memory used by the eBPF agent.
DNS Redirect and Processing Time	In active mode, tracks time from kernel redirection to userspace sniffing, model inference or cache lookup, then resend if benign or block if malicious.

Table 7.1: eBPF agent exported metrics in both active and passive modes

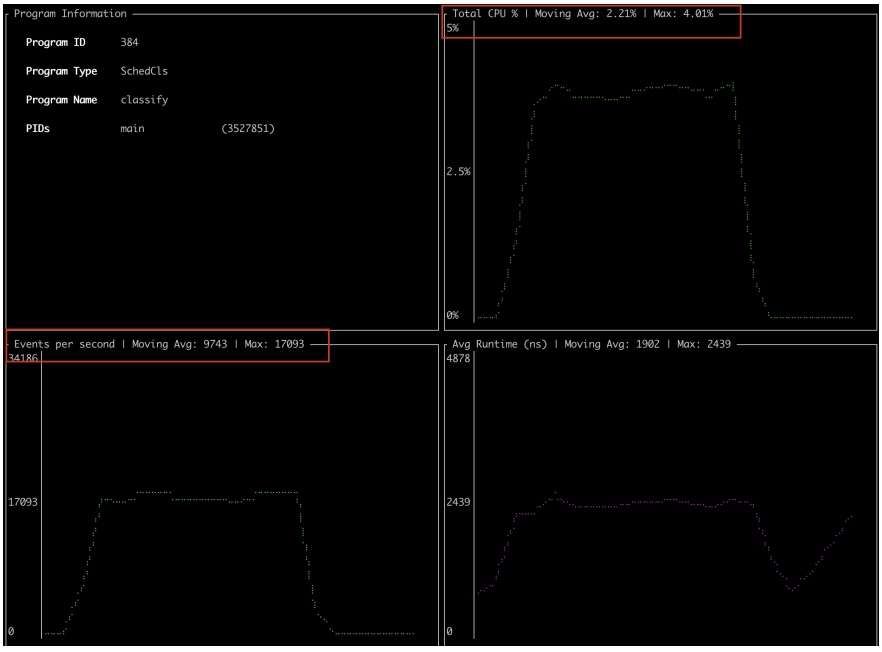


Figure 7.2: Kernel eBPF Programs Profiling

Exfiltration Attempt Count by Process

Exfiltration Port	Node IP	ProcessId	Value (count)
143	10.158.82.19	1630675	63
143	10.158.82.19	1630688	63
143	10.158.82.19	1630726	63
143	10.158.82.19	1630710	63
143	10.158.82.19	1630683	63
143	10.158.82.19	1630663	63
143	10.158.82.19	1630692	63
53	10.158.82.19	1651348	10

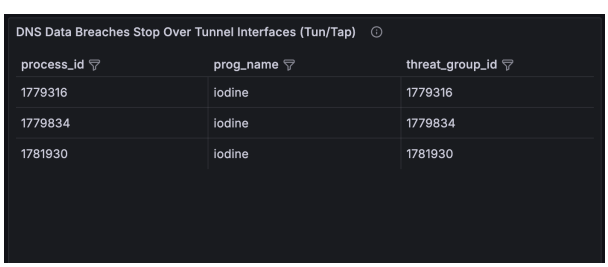
(a) Exfiltration Attempts Prevented per Process

Malicious Process Alive Time in Seconds Before Terminated

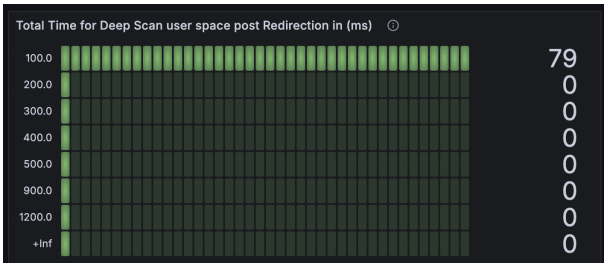
Exfiltration_Attempt_Started_At	Process_Id	Process Alive Time (Sec)
Sunday, 04-May-25 00:49:41 UTC	1630660	6
Sunday, 04-May-25 00:49:41 UTC	1630661	6
Sunday, 04-May-25 00:49:41 UTC	1630662	6
Sunday, 04-May-25 00:49:41 UTC	1630663	6
Sunday, 04-May-25 00:49:41 UTC	1630664	6
Sunday, 04-May-25 00:49:41 UTC	1630665	6
Sunday, 04-May-25 00:49:41 UTC	1630666	6

(b) Process Alive Time Before Termination

Figure 7.3: DNS Exfiltration Prevention Metrics: Process-Level Behavior



(a) Tunnel Interface Exfiltration Metric



(b) Latency in Active Redirect Mode

Figure 7.4: DNS Exfiltration Prevention Metrics: TUN/TAP kernel Network encapsulation and Latency

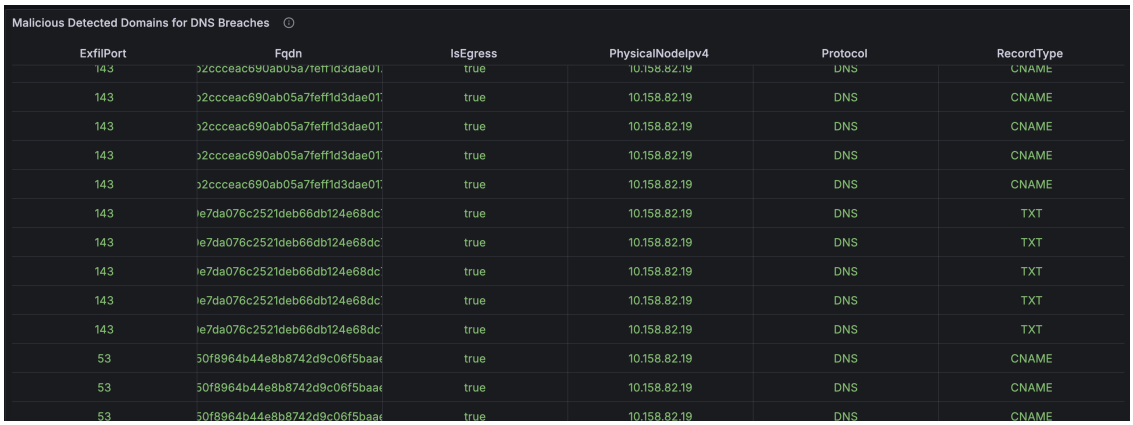


Figure 7.5: Metrics of Prevented DNS exfiltrated packets

7.1.3 Protecting Linux Kernel from malicious eBPF programs

In the data plane, ensuring security beyond kernel-enforced capability checks—such as those near `CAP_SYS_ADMIN` requires guaranteeing the integrity of injected eBPF programs. This prevents tampering or injection of malicious code within the compiled ELF sections of the eBPF bytecode. Without such integrity guarantees, a compromised eBPF agent could load manipulated programs, bypassing exfiltration prevention logic and causing a critical security breach. To mitigate this risk, additional eBPF programs are loaded into the kernel and attached to LSM hooks that intercept BPF syscalls, including `BPF_PROG_LOAD`. These LSM programs verify digital signatures on incoming eBPF bytecode, following a process analogous

to the kernel's verification of loadable modules. Each data plane node's agent bootstraps a local certificate authority (CA) that generates ephemeral elliptic curve certificates and private keys. The agent signs the raw eBPF bytecode prior to injection, establishing an initial trust layer. After JIT compilation, the optimized bytecode is signed again, along with the original raw bytecode signature, creating a two-stage chain of trust. Certificates and keys are securely stored in the kernel session keyring, tied to the user's login session on the endpoint. When the `BPF_PROG_LOAD` syscall occurs, the LSM hook verifies both raw and compiled bytecode signatures against the asymmetric keys in the keyring. These signatures are also maintained within the eBPF maps to support verification logic. This dual-signature approach ensures that neither raw nor compiled bytecode can be tampered with before kernel injection. Furthermore, the core security layer integrates with a centralized control plane connected to the cloud Public Key Infrastructure (PKI), enabling a scalable layered trust model - from cloud PKI to kernel-level mandatory access control. While the keyring currently stores sensitive keys in kernel guarded memory pages, the kernel restricts access to unprivileged userspace processes. In addition, these security primitives support integration with Trusted Platform Modules (TPMs) or Hardware Security Modules (HSMs) - commonly available in cloud environments - allowing the keyring and cryptographic keys to be offloaded to hardware-backed firmware (Intel TDX, AMD SEV-SNP), thereby enhancing security guarantees. The core implementation for the kernel resident, LSM-integrated eBPF verification program is detailed below.

Listing 7.1: Kernel BPF LSM Hook for PKCS7 Signature Verification

```
BPF_PROG(bpf, int cmd, union bpf_attr *attr, unsigned int size) {
    if (cmd != BPF_PROG_LOAD)
        return 0;

    // Look up eBPF program, its original signature, and the modified
       signature
    mod_sig = bpf_map_lookup_elem(&modified_signature, &zero);
    orig_data = bpf_map_lookup_elem(&original_program, &zero);
```

```
    combined_buf = bpf_map_lookup_elem(&combined_data_map, &zero);

    // Copy eBPF program and original signature into a combined buffer
    insn_len = attr->insn_cnt * sizeof(struct bpf_insn);
    bpf_copy_from_user(combined_buf->data, insn_len, attr->insns);
    bpf_probe_read_kernel(combined_buf->data + insn_len, orig_data->
        sig_len, orig_data->sig);

    // Create dynptrs for PKCS7 verification
    // dynptrs work similar to kptr but allowing to point to buffer
        location storing large amount of data as in case of signature for
        eBPF verifier requirements.
    bpf_dynptr_from_mem(combined_buf->data, total_size, 0, &
        combined_data_ptr);
    bpf_dynptr_from_mem(mod_sig->sig, mod_sig_size, 0, &sig_ptr);
    bpf_dynptr_from_mem(orig_data->data, orig_data->data_len, 0, &
        orig_data_ptr);
    bpf_dynptr_from_mem(orig_data->sig, orig_data->sig_len, 0, &
        orig_sig_ptr);

    // Load asymmetric keys from session kernel keyring and verify
        signatures
    // all the kernel keyring access in bpf code is done via bpf_key a
        wrapper over kernel core key structure accessed from userspace via
        keyctl
    trusted_key = load_keyring();
    bpf_verify_pkcs7_signature(&orig_data_ptr, &orig_sig_ptr, trusted_key)
        ;
    bpf_verify_pkcs7_signature(&combined_data_ptr, &sig_ptr, trusted_key);
}
```

7.1.4 Protecting SKB Netflow introspection and eavesdropping

In active mode, the agent enforces advanced security directly in the kernel using a TC-attached eBPF program bound to the veth bridge that manages all network namespaces created by the agent. To prevent malicious processes from analyzing live traffic redirection patterns, an additional eBPF map, `skb_netflow_integrity_verify_map`, is used as explained in Algorithm 2. The core eBPF program assigns a unique SKB mark to each redirected netflow packet, enabling integrity verification at the receiving bridge interface. This bridge, monitored by the eBPF agent in userspace, receives the redirected traffic for further analysis of suspicious behavior. An additional ingress TC eBPF program is attached to the bridge interface. Inspect incoming SKBs to verify that the expected SKB mark is present, confirming that the packet was redirected from the core TC program on a physical netdev and not injected from any untrusted source. This enhanced security design ensures strong SKB integrity and prevents malicious sniffing or brute-force inference of live redirection patterns, effectively enforcing traffic validation across both kernel and userspace components. Algorithm 8 explains the algorithm integrity verification check on the SKB attached to the TC ingress on the bridge.

Algorithm 8: SKB Integrity Verification and Secure Redirection in **Active** Mode

Input : `skb` (socket buffer),
eBPF maps:
`skb_netflow_integrity_verify_map`
Output : TC_ACT_OK
TC_ACT_SHOT

1 Fetch `skb_mark_integrity_mark` from `skb_netflow_integrity_verify_map` with:
- Key: `0xFFEF // unique key same used in core TC program over physical netdev`

if *skb_mark_integrity_mark* $\neq$ *skb->mark* **then**
  `// This is a tampered redirected packet, or sniffed packet and did not originate from the main TC egress program`
2 **return** `TC_ACT_SHOT`;
return TC_ACT_OK;

7.2 Appendix B

This section provides additional internal details about the DGA used for generating domains in mass malicious C2 and tunneling activities.

7.2.1 DGA and malicious C2 and Tunneling domains generation

To carry out advanced DNS C2 attacks or tunnel using open source C2 tools, a DNS server with custom DNS zones (SOA) is required. These zones must include appropriate NS, A, AAAA and glue records pointing to the malicious C2 server's IP address. In this framework, PowerDNS is used as the DNS infrastructure to support such attacks. A custom script simulates a sample DGA by generating random words for the second level domain (SLD), selecting a random top level domain (TLD) and adding a third label that identifies the C2 tool being used. The script automatically generates the PowerDNS recursor forwarder configuration, allowing the recursor to redirect specific queries to a custom authoritative PowerDNS server. It also uses pdnsutil to create the necessary zone files. By DNS design, each label requires a dedicated zone file with an NS record delegating the next label, ensuring the hierarchical resolution of queries through appropriate glue records. Currently, the DGA focuses only on domain name mutation without incorporating IP (L3) address mutation - that is, multiple A or AAAA records per domain for DNS-based load balancing across C2 nodes are not yet implemented. However, this can be extended when more infrastructure is available. Despite the absence of the L3 mutation, the framework controller remains effective. It enforces dynamic domain blacklists and applies in-kernel network policies for cross-protocol correlation, preventing data exfiltration to dynamically generated C2 domains and IPs. Note: Most real-world advanced C2 botnets and multiplayer C2 frameworks do not require a DNS server to layer and forward DNS queries to the C2 server. Instead, they inherently support the start of their own DNS server on the standard port or a random UDP port, allowing the implant to communicate directly with the C2 server. Tools such as Sliver force DNS servers to forward all queries to the C2 DNS server, eliminating the need for additional DNS hops. The DGA implementation is detailed below.

Listing 7.2: Domain Generation Algorithm

```
# generate number of malicious C2 server domains and add create zones,
   child zones NS links inside DNS server
```

```
# all the attacker tools to use
# this attacker tool also generate the third label of C2 domain
exfil_tools: List[str] = ['dnscat', 'sliver', 'iodine', 'det']

# get the random TLD
def gen_randomTLD() -> str:
    return random.choice(['live','com','de','io','se','bld', 'val', 'def',
        'head'])

def DGA_MASS_DOMAINS_GEN(args):
        r = RandomWord()
        dga = gen_c2_exfil_domains(tldDomains=[base64.b64encode(r.word()).
            lower() + "." + gen_randomTLD()
                                        for _ in range(1 << int(args.count)
                                            )],
                                c2_tool_domains=exfil_tools)
        ff = open(DGA_FILE, 'w', encoding='utf-8')
        ff.write('\n'.join(dga))

        append_zone_data_in_zoneFiles(dga) # create zone and child zones
        gen_exil_forward_zones_file(dga) # generate the forward zone file
```