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

Vedang Parasnis

A whitepaper
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:
Professor Geetha Thamilarasu, Committee Chair, Chair
Professor 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:
Professor Geetha Thamilarasu, Committee Chair
Professor Munehiro Fukuda, Committee Member

Data exfiltration remains one of the most persistent and sophisticated threats in cybersecurity, with the Domain Name System (DNS) frequently exploited as a covert channel for tunneling and command-and-control (C2) communications. This threat is especially critical for hyperscalers, given the growing data demands of AI workloads and the massive storage of sensitive client data on-premise or in cloud. DNS remains highly vulnerable due to its ubiquity and critical role in enterprise network operations. As highlighted in the IBM 2024 Data Breach Report, the average cost of a data breach exceeds $4.8 million, with DNS-based exfiltration posing catastrophic risks that are extremely difficult to prevent in real time. This paper introduces a novel, scalable framework for real-time prevention of all forms of DNS-based data exfiltration across distributed environments, built around an endpoint-centric security architecture. The framework leverages deep in-kernel packet inspection via eBPF hooks injected into the Linux kernel's network stack, enabling high-speed, per-packet parsing of outbound DNS traffic. It integrates dense neural networks in userspace to perform lexical and structural analysis of DNS queries, identifying obfuscated or malicious exfiltration attempts. These components are packaged within an endpoint security agent, enabling local enforcement directly at the source of exfiltration. Security is enforced proactively by terminating processes in real time once they are flagged as malicious, accelerating incident

response and immediately containing compromise at the endpoint. The system exports granular telemetry from each node—including malicious process identifiers, process lifetimes, and detailed DNS usage patterns—to centralized brokers. This enables dynamic domain blacklisting, supports massive scalability, and facilitates implicit DNS-based cross-node policy enforcement in real time. By combining deep in-kernel inspection, AI-assisted detection for kernel-resident programs, cross-protocol correlation via dynamic in-kernel network policy enforcement, and active process-level defense at the endpoint, the framework reduces attacker dwell time, prevents lateral movement, and enhances visibility and responsiveness for security teams operating in large-scale, distributed production environments.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

Modern threat actors are constantly evolving, using increasingly sophisticated techniques and covert communication channels to maintain persistence on compromised systems and exfiltrate data before detection or removal. 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 sensitive data undetected, 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 across the Middle East and Asia, used a custom system called DNSsystem to stealthily exfiltrate data from energy and telecom sectors via 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 focus on passive analysis via anomaly detection, domain reputation scoring, or static blacklists. However, these methods are less reactive, slow, and often bypassed by stealthy, adaptive APT malware, resulting in slower response time with no assurance of minimal or negligible data loss prior to removal. There is a critical need for a robust, endpoint-centric defense mechanism that enforces DNS security from within the operating system itself, rather than relying on passive userspace monitoring or centralized anomaly detection tools. As cloud providers face increasing demand for secure, high-availability infrastructure, there is a growing requirement for systems that can instantly neutralize malicious implants at the source—reducing response time and actively blocking DNS-based exfiltration in real time supporting dynamic domain blacklisting thereby protecting endpoints across the network without requiring user intervention for safeguarding massively scaled, multi-region cloud environments, where DNS is critical for business operations. The APT Malware Flow illustrates the lifecycle of an implant. By severing DNS C2 links and immediately terminating the implant, lateral movement is prevented and the attacker's control is disrupted at the earliest stage.

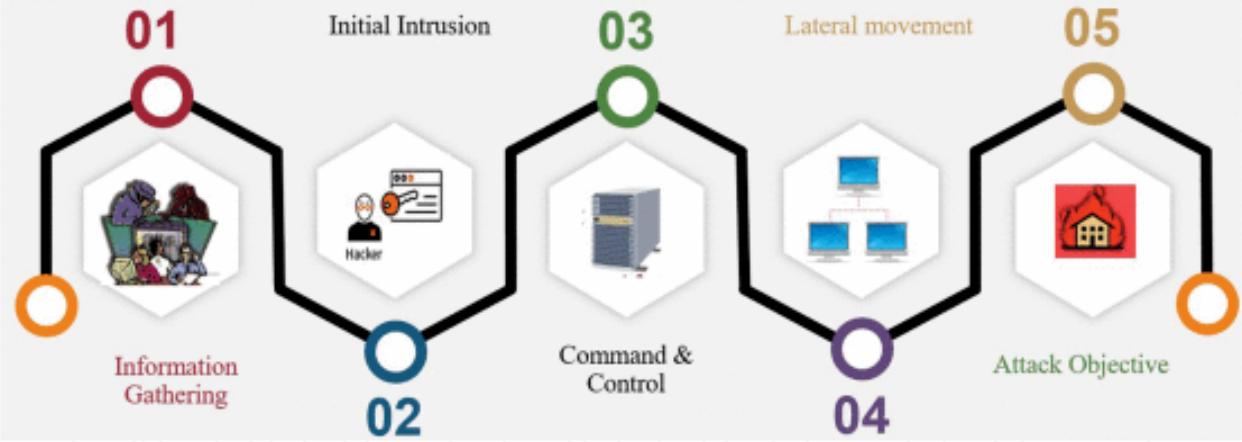


Figure 1.1: APT Malware Exploitation Phases

1.2 Objective

This project develops an endpoint-centric security framework to prevent DNS-based data exfiltration in real time across distributed Linux systems. By embedding detection and enforcement logic directly in the kernel using eBPF, the system targets covert C2 implants, DNS tunneling, and obfuscated payloads—delivering rapid threat response, deep visibility, and zero-trust enforcement across distributed environments.

Chapter 2

BACKGROUND

2.1 eBPF

Extended Berkeley Packet Filter (eBPF) was introduced in Linux kernel version 3.15 (2014) as a general-purpose in-kernel virtual machine. It evolved from classic BPF, which was originally limited to packet filtering through a domain-specific language and lacked runtime flexibility pf injecting kernel code. Unlike kernel modules, which risk destabilizing the system, eBPF enables safe and efficient injection of custom logic into the kernel, offering dynamic programmability without compromising security or stability. eBPF programs are written in a restricted subset of C, compiled to eBPF bytecode using LLVM, and executed by the eBPF virtual machine inside the kernel address space. These programs follow a strict execution model: a RISC-like instruction set, eleven 64-bit general-purpose registers, a 512-byte stack, and a hard cap of one million instructions per program. The bytecode is architecture-agnostic, making it portable across CPUs running the Linux kernel. Before execution, the kernel's BPF verifier performs static analysis on the bytecode to ensure memory safety, bounded loops, and control flow integrity. This verification prevents arbitrary memory access, use of uninitialized registers, and sandbox escapes. Once verified, the kernel can further optimize the program using in-kernel Just-In-Time (JIT) compilation, benefiting from both LLVM optimizations during compilation and runtime JIT enhancements after attachment. eBPF programs interact with userspace via maps—persistent, kernel-resident key-value data structures. These maps allow secure sharing of state and come in various types optimized for different use cases, such as BPF_MAP_TYPE_LRU_HASH, BPF_MAP_TYPE_LPM_TRIE, and BPF_MAP_TYPE_STACK. Maps can be accessed using a map ID or a pinned file descriptor through the BPF virtual file system, enabling persistence beyond the lifetime of

the original userspace loader.Program loading and interaction are tightly controlled through a single syscall: bpf(). Kernel-level Mandatory Access Control and Linux capabilities (e.g., CAP_BPF, CAP_NET_ADMIN) are enforced to reduce the attack surface. For instance, attaching eBPF programs to the network data path requires CAP_NET_ADMIN, especially when managing network interfaces via AF_NETLINK sockets. Once injected, eBPF programs are attached to various kernel hook points—network stack, scheduler, security modules—using kernel probes, tracepoints, or userland hooks. This makes eBPF a powerful foundation for deep observability, fine-grained tracing, and in-kernel security enforcement. The diagram below outlines the three key phases of an eBPF program's lifecycle—development, loading and verification, and runtime attachment—specifically in the context of monitoring socket_write operations initiated by userspace processes.

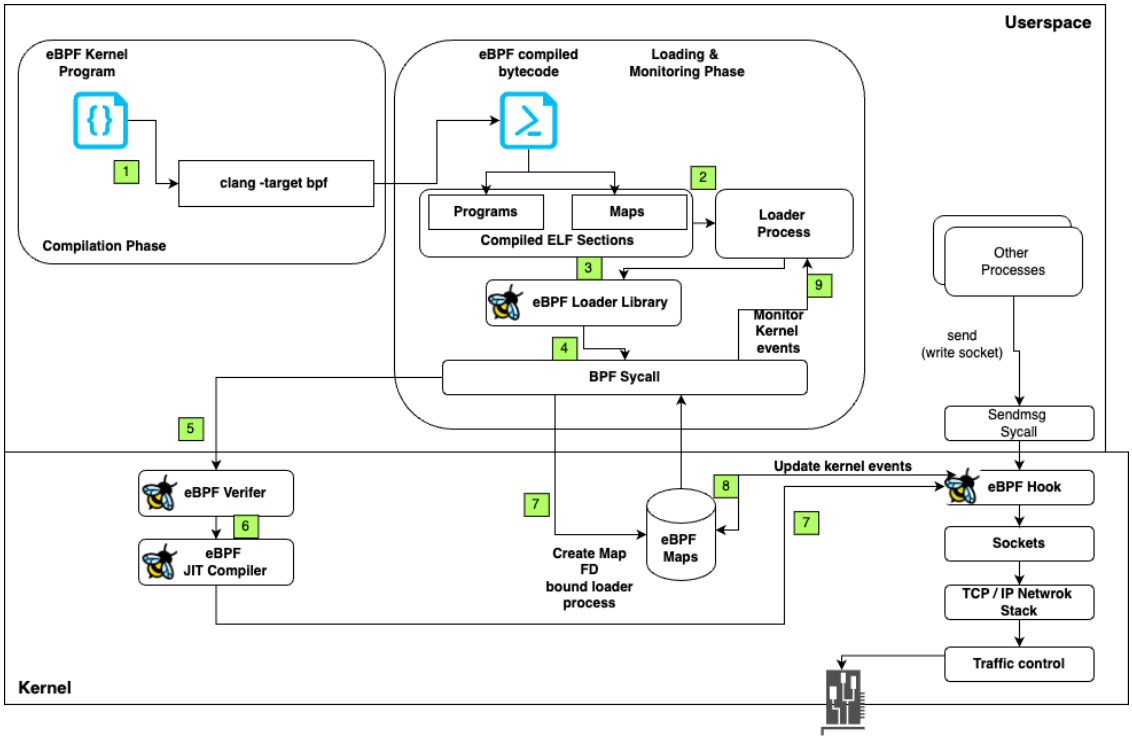


Figure 2.1: eBPF Programs Injection Phases in Kernel

2.2 Linux Kernel Network Data Path

The Linux kernel network datapath refers to the sequence of processing steps that network packets follow as they traverse the kernel's networking stack in both the incoming (ingress) and outgoing (egress) directions. At the heart of this path lies the `sk_buff` (socket buffer), a fundamental data structure used to represent packets in memory. Each `sk_buff` acts as a metadata-rich container, implemented as a doubly linked list, allowing the kernel to manipulate packets as they move through different layers of the stack. Every packet—whether entering or leaving the system—passes through multiple processing stages. These include scheduling, classification, filtering, shaping, and forwarding. The datapath handles these operations across various network interfaces, or `netdevs`, which are kernel abstractions for hardware or virtual network devices. In both ingress and egress directions, packets are managed through dedicated queues, either software-based or hardware-backed, depending on the NIC and driver implementation. The egress datapath is particularly important for detecting and preventing data exfiltration, as malicious packets typically originate from compromised userspace processes. Once a userspace application writes data to a socket, the kernel routes this packet through several stages: the socket layer (TCP/IP stack), the Netfilter subsystem (link layer), the traffic control (TC) system for shaping and classification, and finally to the device driver that transmits the packet. Linux Kernel Data Path The primary components involved in the egress datapath are:

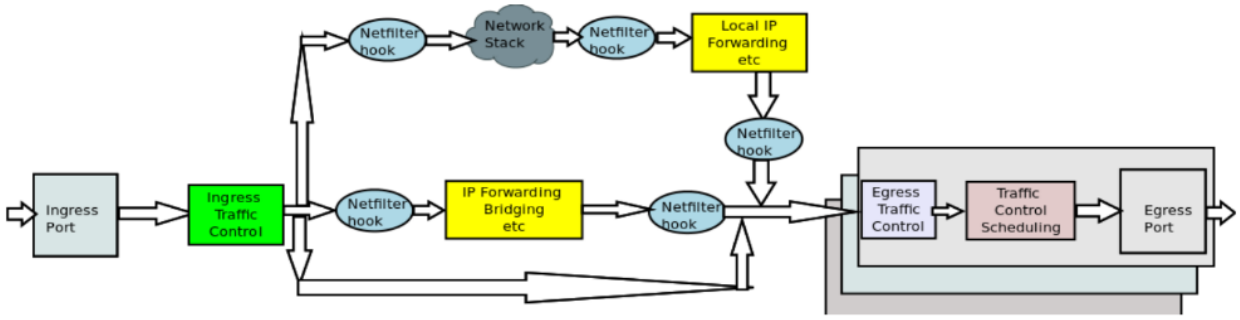


Figure 2.2: Linux Kernel Network DataPath

2.3 eBPF Integration with the Linux Kernel Networking Stack

Among the various subsystems within the Linux kernel network datapath, Traffic Control (TC) is particularly critical for enforcing security in the egress direction, where exfiltration attempts typically occur. TC enables fine-grained control over network traffic through mechanisms such as shaping (rate limiting), scheduling, policing, and filtering. It operates primarily through queuing disciplines (qdiscs)—kernel-level constructs attached to a netdev that define how packets are enqueued, dequeued, and processed before transmission by a network device (TX path steering) [16]. TC supports both classful and classless qdiscs, allowing packets to be classified and processed based on a hierarchy or flat model. Among these, the clsact (classless qdisc with actions) discipline plays a unique role. Unlike traditional classful qdiscs, clsact is a lightweight, classless metadata qdisc still supporting RX, TX software queues, max length like other classful qdisc that can be attached on top of both classful (e.g., htb) and classless (e.g., mq_prio, fq_codel, mq_cake) qdiscs allowing both packet classification and implementing packet forwarding actions post classification. It supports attaching eBPF programs to both ingress and egress routes for processing netflows with varying priorities allowing priority based eBPF programs executions allowing DPI over the sk_buff and security filtering enforcement without altering existing queuing structures [7]. eBPF programs attached via CLSACT execute on every packet passing through the hook, enabling real-time inspection, advanced custom packet filtering compared to legacy TC packet classifer for other QDISC's, and telemetry at the edge of the kernel network stack. This makes it particularly well-suited for DNS data exfiltration prevention, where enforcement needs to happen before the packet lands over NIC driver queues. CLSACT is commonly used in scalable security architectures primarily for bandwidth, and throughput measurement, rate limiting, or packet mangling however, supporting actions over packet post-classification this QDISC IS extensively useful for packet filtering in the kernel TC layer based packet payload characteristics

intercept packets just before transmission or immediately after reception, enabling drops,

header rewrites, metadata tagging, and rerouting to different netdevs, all from inside the kernel. Beyond TC, the deep integration of eBPF with the Linux networking stack allows injection of advanced security logic per packet at multiple strategic hook points: eBPF hooks explains all eBPF different hook points in the kernel network data path. In addition, with faster packet processing socket types introduced in linux kernel (AF_XDP) allows injection of raw packets directly from userspace into the device driver TX queues, allowing fast egress packet processing by passing kernel network stack particularly highly relevant for high egress packet processing.

This broad hook coverage allows eBPF to power everything from performance profiling, network observability, and rate limiting to runtime threat detection, load balancing, and deep in-kernel packet inspection, all with the utmost security.

While cloud native environments such as Kubernetes have adopted eBPF through CNIs like Cilium (from Isovalent, now part of Cisco) for enforcing L3/L7 policies and observability, current solutions focus largely on kernel socket layer for filtering, load-balancing or DDoS prevention via XDP. To date, no industry solution has comprehensively leveraged eBPF across the Linux network stack, syscall interface, and security subsystems to proactively prevent DNS exfiltration or disrupt active Command-and-Control (C2) activity in real time.

2.4 DNS-based Data Exfiltration

DNS data exfiltration is a distinct category of data breaches where sensitive information is stealthily extracted from corporate networks via the DNS protocol. This technique is typically carried out by remote-controlled, fileless malware implants that operate entirely in memory, leaving minimal forensic footprint on infected endpoints. DNS-based exfiltration exploits the protocol's ubiquity and inherent trust to covertly transmit stolen data—either as raw payloads or tunneled traffic from blocked protocols—by encoding it into the subdomain portion of DNS queries. These queries follow the standard hierarchical resolution path and ultimately reach attacker-controlled domains or delegated nameservers. Adversaries leverage DNS for data exfiltration by encoding small chunks of data across multiple queries using

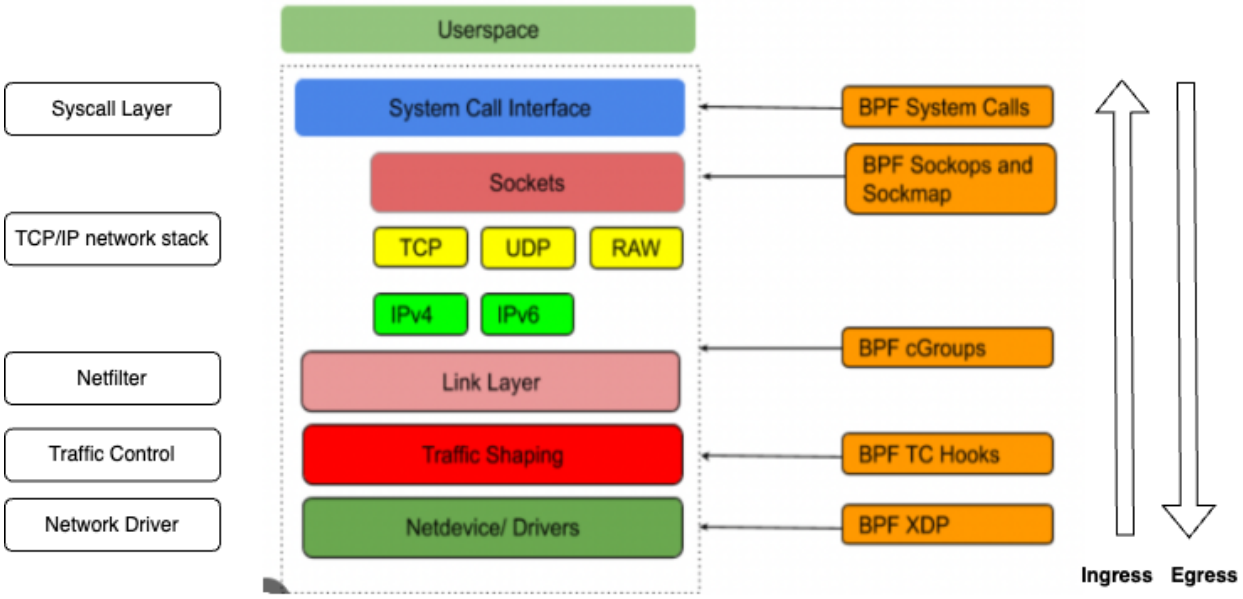


Figure 2.3: eBPF hooks over Linux Kernel Data Path

common query types such as `A`, `AAAA`, `MX`, `SRV`, and `HTTPS`, or by utilizing record types that support arbitrary payloads, including `TXT` and `NULL`. These latter types are frequently used in payload-based exfiltration, allowing queries to contain non-DNS data. In command-and-control (C2) scenarios, attackers can issue commands via DNS responses, with `TXT` and `NULL` records serving as flexible data containers. These mechanisms allow attackers to disguise and transmit sensitive information through traffic that appears benign. To evade detection, adversaries employ techniques such as randomized query intervals, encryption with ephemeral keys, and domain generation algorithms (DGAs) that rotate second-level domains. Advanced implants may also tunnel non-DNS Layer 7 (L7) protocols over DNS, leveraging arbitrary Layer 4 (L4) ports beyond the standard ones (e.g., 53, 853, 5353) as done by DNSCAT2 adversary emulation framework for data exfiltration, complicating both traffic analysis and firewall-based centralized passive monitoring solutions deployed in distributed environments.

DNS Data Exfiltration Phases explains phases of carrying out DNS data exfiltration. The three primary forms of DNS-based exfiltration—tunneling, raw exfiltration, and command-and-control (C2) channels—are described in the subsections 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 abuses the protocol to encapsulate arbitrary data or non-DNS payloads within query fields, allowing attackers to bypass firewalls and traditional perimeter defenses. By disguising malicious payloads as legitimate DNS traffic or blending with benign traffic, it enables covert bidirectional communication between compromised systems and remote servers. Tunneling may operate in low- or high-throughput modes, with traffic patterns modulated to evade anomaly-based detection systems. Sophisticated variants exploit kernel-level encapsulation methods (e.g., `tun/tap`, VXLAN) using virtual interfaces. While these methods require elevated privileges (e.g., `CAP_NET_ADMIN`), the sporadic nature of encapsulated traffic makes detection through passive monitoring particularly challenging.

DNS Command and Control (C2)

DNS-based C2 is an advanced form of tunneling that establishes persistent, covert channels between implants and attacker infrastructure. Malicious implants use DNS queries to poll for encoded instructions and execute responses—forming full-duplex control channels. These

channels can shift between rapid polling and low-frequency beaconing to remain stealthy. Some C2 channels leverage cross-protocol communication, use rotating IPs, and integrate domain generation algorithms to evade static detection. Conventional defenses such as static blacklists and rule-based systems are ill-suited to counter these dynamic threats. Early-stage detection and endpoint-level disruption are critical, as even brief persistence can result in data loss and lateral movement. Currently, no known research or proprietary solutions offer real-time termination of both the DNS C2 channel and the associated implant process at the endpoint—especially before any command execution or exfiltration occurs.

DNS Raw Exfiltration

Raw exfiltration over DNS involves directly leaking files or sensitive data in high-volume bursts of DNS queries over a short duration. While this method often creates detectable spikes in traffic, it can still succeed before monitoring systems trigger alerts or enforcement kicks in. Most existing solutions rely on passive network monitoring and cannot guarantee zero or near-zero data loss. Real-time prevention—prior to transmission—is essential, especially when exfiltration bypasses traditional IDS systems. Unlike tunneling or C2, raw exfiltration is typically unidirectional and lacks duplex communication. Instead, it focuses solely on stealthily transferring obfuscated data through common DNS query fields such as subdomains, often using base32/base64 encodings and chunked transmissions to bypass protocol validation layers.

2.5 DNS protocol transport enforcements

The DNS protocol, as originally specified in RFC 1035, was not designed with security in mind. It imposes several structural limitations designed to reduce complexity and overhead in early network environments. By default, DNS uses UDP as its transport protocol, restricting the payload to 512 bytes per query or response. With the introduction of EDNS0 (Extension Mechanisms for DNS), this limitation was relaxed to allow UDP payloads up to 4096 bytes, enabling support for modern features such as DNSSEC. In addition, DNS also supports for

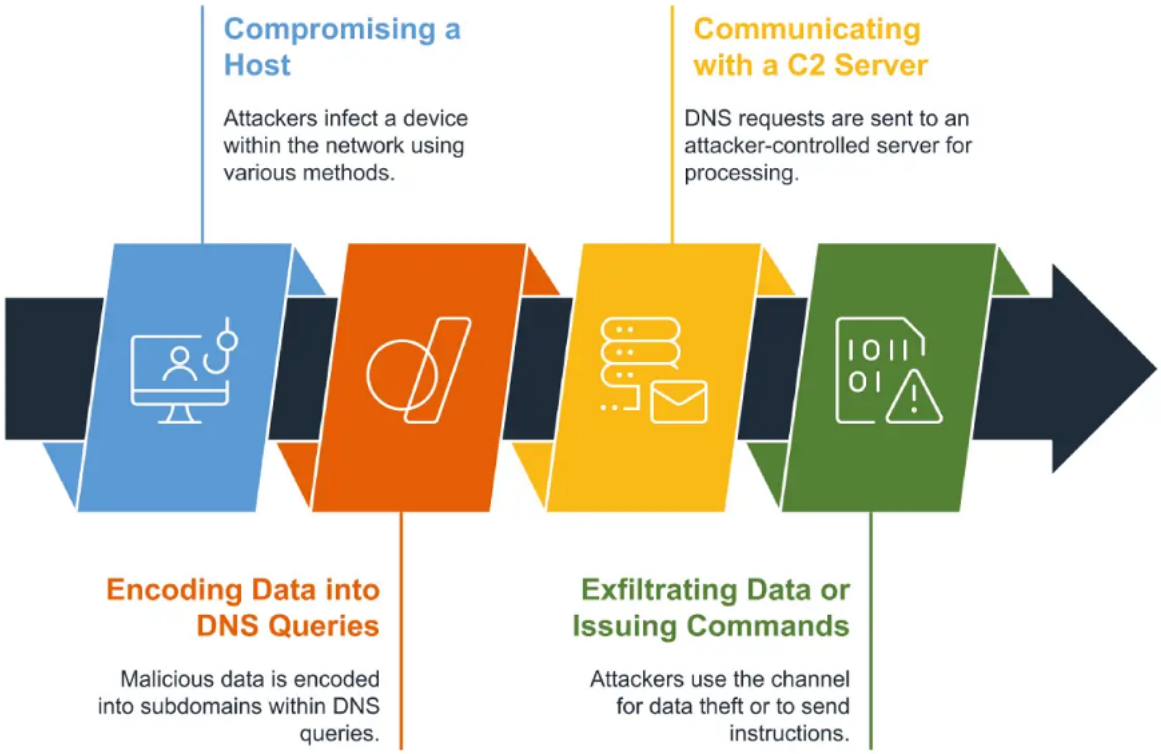


Figure 2.4: DNS Data Exfiltration Phases

transport over TCP for larger payload sizes. As a protocol level enforcement DNS domain names are limited to 255 characters in total length, composed of a maximum of 127 labels / octets, with each label being no more than 63 characters.

2.6 DNS Security Enhancements and Their Limitations

Several protocol-level security enhancements have been proposed and standardized to strengthen DNS integrity and privacy. These include:

- DNSSEC: Adds cryptographic signatures to DNS records to ensure authenticity and prevent spoofing or cache poisoning. However, DNSSEC does not provide encryption or confidentiality, leaving the DNS payload visible to any intermediate observer. As such, it offers no protection against covert channels or data exfiltration mechanisms

embedded within legitimate-looking queries.

- DNS-over-TLS (DoT) and DNS-over-HTTPS (DoH): Encrypt DNS queries to prevent surveillance and man-in-the-middle attacks. While effective at improving privacy, this encryption also impairs traditional security tools from performing deep packet inspection (DPI) on DNS traffic encapsulated within TLS or HTTPS. This blinding effect weakens intrusion detection systems (IDS) and data loss prevention (DLP) tools.

Although these enhancements protect against certain classes of attacks, none are designed to prevent data exfiltration through DNS, particularly when it originates from implants on the endpoint that abuse protocol-compliant structures for covert communication.

2.7 Existing Prevention Mechanisms and Their Limitations

To date, the most widely deployed preventive mechanisms for DNS-based exfiltration include:

- **DNS Sinkholing**: Redirects queries for known malicious domains to controlled endpoints, preventing external resolution.

- **Response Policy Zones (RPZ)**: Enable DNS servers to enforce custom filtering policies based on domain patterns or IP addresses.

While these techniques effectively block malicious domains using declarative policies or access control lists to protect internal networks, they remain fundamentally reactive. Most rely on static blacklists generated after detection via intrusion alerts or passive deep packet inspection. As a result, DNS-based data exfiltration can succeed before server rules are updated and enforced. Moreover, these methods are not equipped to stop C2 implants that use domain generation algorithms (DGA) to mutate both Layer 3 IP addresses and Layer 7 domain names—causing damage before filtering policies can adapt—due to the persistent and evolving nature of C2 infrastructure.

Chapter 3

RELATED WORK

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, from its initial architecture proposed by Høiland-Jørgensen et al. and later adopted in the Linux kernel for high-speed programmable data paths and security use cases, such as eBPF combined with XDP (eXpress Data Path) to improve packet processing throughput via early packet drops and hardware offload at the NIC level [10, 13]. Vieira et al. studies provide architectural overviews and performance analyses of eBPF in networking contexts [20]. Bertrone et al. explains accelerating kernel network firewalls by combining eBPF and iptables [5], 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 [11]. While 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 for mitigating ingress-layer DDoS floods, such as TCP SYN and UDP amplification attacks [4]. However, this technique also falls short in handling sophisticated or covert DNS-based exfiltration threats. Based on current

literature, Steadman and Scott-Hayward presents the only as of know eBPF-based system specifically aimed at DNS exfiltration prevention. Their approach combines eBPF and SDN to enforce static rules in the data plane while performing flow analysis in the control plane [19, 18]. However, their design attaches eBPF programs at the XDP layer—suitable only 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. Additionally, their inspection is restricted to standard DNS ports, missing covert channels using non-standard ports, moreover their evaulation was not able to prevent stealthy exfiltration leading to data loss with more stealthy traffic mirroring to control plane. Similarly, enterprise tools such as Isovalent Cilium (now part of Cisco) support eBPF-based Kubernetes network policies at layers L3–L7. While DNS-aware L7 policies allow for domain-level whitelisting, they lack dynamic blacklisting and are not tailored for detecting exfiltration behaviors. Open-source tools like Microsoft's Inspector Gadget also rely on static rules defined in userspace and do not provide deep, kernel-level enforcement mechanisms. These limitations highlight the need for a comprehensive eBPF-based solution that operates at the egress point and supports dynamic policy enforcement.

3.2 Machine Learning for Detecting DNS Data Exfiltration

Advancements in network security have significantly enhanced the detection of DNS data exfiltration, often leveraging machine learning to analyze packet patterns and payloads. Many solutions combine DPI with anomaly-based analysis of DNS traffic volume and timing, detecting potential exfiltration attempts via DNS firewalls or integrated intrusion detection systems. For C2-based exfiltration, Zimba and Chishimba focuses on behavioral analysis, using system resources like PowerShell and Windows backdoors alongside DNS tunneling for APT detection [21]. However, these methods primarily detect exfiltration and fail to fully prevent DNS tunneling, with no clear response time or method for breaking the C2 communication or terminating the implant. Similarly, Das et al. proposes a machine learning

model that identifies DNS-based data exfiltration using real malware samples from financial institutions, though potential evasion strategies are also discussed [8]. Meanwhile, Ahmed et al. suggests a real-time detection mechanism using the Isolation Forest algorithm [1], but their stateless analysis fails to address protection against prolonged, stealthy APT malware or C2 exfiltration. Several approaches have focused on DNS server-side detection, blending stateful and stateless features. Bilge et al. introduced the EXPOSURE system, which identifies suspicious domains by extracting 15 features from DNS traffic, categorized into query-name, time-based, answer-based, and TTL-based features. Validated using a dataset of 100 billion DNS requests, this system effectively identifies domains involved in botnet C2 and spamming [6]. However, the system's focus is on detection rather than prevention, and it is limited by the reliance on passive analysis of DNS queries. Similarly, Antonakakis et al. proposed NOTOS, a dynamic reputation system analyzing passive DNS queries, extracting 41 features classified into network-based and zone-based categories [2]. While it identifies malicious domain characteristics, it falls short in detecting data exfiltration and lacks real-time detection capabilities, making it ineffective against stealthy C2-based exfiltration. Nadler et al. developed a solution for detecting malicious, low-throughput exfiltration using domain-specific features such as entropy and query time intervals, employing one-class SVM and Isolation Forest. However, their reliance on past traffic limits its real-time detection capabilities [14]. Other studies, such as Mathas et al., explore machine learning models for C2 tunneling detection, but their effectiveness is hindered by the inability to detect stealthy C2 over long exploitation periods [12]. Aurisch et al. uses mobile agents for real-time detection and mitigation, but the approach is prone to false positives and introduces latency due to agent hops across the network, making it unsuitable for real-time security enforcement [3]. While solutions like Haider et al. attempt to address C2-based DNS exfiltration over package distribution systems to prevent supply chain attacks, their approach is efficient for preventing exfiltration from single node but lacks detailed mechanisms for detecting and preventing DNS data exfiltration across multiple nodes or deep system-level observability [9, 17]. These solutions remain vulnerable to policy evasion, privilege escalation, and denial-

of-service attacks, largely due to limited kernel integration and insufficient enforcement of security policies. Process DNS, as introduced by Sivakorn et al., focuses on detecting and countering C2-based DNS exfiltration, specifically by analyzing processes in userspace and terminating malicious ones [17]. Despite its low-latency detection approach, the solution is vulnerable to privilege escalation and denial-of-service attacks due to its lack of kernel integration and inability to handle more sophisticated evasion tactics. The lack of transparency regarding the kernel enforced system access control mechanisms for their agents in userspace further limits its effectiveness, particularly for advanced threat actors that may bypass userspace defenses. Existing machine learning-based solutions typically focus on volumetric or timing anomalies, as they are limited to passively analyzing aggregated DNS traffic in userspace. These systems lack critical capabilities such as instant response, preemptive data loss estimation, cross-protocol correlation, detection of port-layer obfuscation (e.g., randomized DNS over arbitrary ports), and in-kernel inspection of encapsulated traffic. Most remain decoupled from active prevention and dynamic system-level enforcement, relying solely on stateful or stateless analysis, and often lack evaluation against advanced adversary emulation and modern C2 frameworks. While stateless lexical inspection can rapidly classify malicious payloads, userspace solutions still require behavioral or statistical analysis over packet timing to detect slow and stealthy exfiltration tactics resulting in substantial data loss prior enforcement is applied. These approaches are generally effective against bulk data leaks but falter against sophisticated techniques such as C2, multiplayer C2, botnets, and domain generation algorithms. Despite improvements in detection, no existing solution offers real-time DNS exfiltration prevention with the ability to terminate malicious implants instantly, while maintaining fine-grained control and scalability in distributed environments. Userspace-only designs lack direct access to the network interface lowest layers, sacrificing critical system-level visibility. As a result, they are poorly suited for production and cloud environments where data sovereignty, rapid containment, and deep system introspection are essential against evolving DNS 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 [15]. However, like DNS firewalls, it lacks direct prevention at the endpoint, which is essential for minimizing data loss and reducing dwell time. It also struggles against APT malware that uses slow, stealthy C2 patterns to evade detection and volume thresholds, as acknowledged in Akamai's forums. Similarly, Amazon Route 53 Resolver DNS Firewall Advanced detects threats like DNS tunneling and DGA activity by identifying anomalies in queried domain names via DNS volume traffic and customizable rules. However, it suffers from latency due to its reliance on anomaly detection and lacks the capability for automatic prevention or enhanced observability, making it ineffective against the slow, evasive C2 communications used by APT malware. These solutions, while effective for detecting certain types of DNS exfiltration, are limited in their response capabilities. AWS Route 53 Firewall, for example, only blacklists specific domains using static policies that require human intervention, which is inadequate against dynamic and mutating DGA techniques. Additionally, they lack cross-protocol correlation, which is critical for blocking C2 domains and filtering Layer 3 traffic through AWS network firewalls. In contrast, proprietary DNS firewalls from Cloudflare, Akamai, and AWS are effective for DDoS protection but fail to address sophisticated DNS exfiltration or internal threats that exploit advanced C2 channels for DNS-based data exfiltration. Tools like Infoblox integrate hybrid agent-based endpoint security with advanced threat intelligence from a centralized control plane inferencing and Broadcom's Carbon Black at endpoint for blocking exfiltrating processes. However, eBPF kernel programs provide a substantial advantage with deep integration directly within the kernel network stack, offering granular control and much more effective metrics for detecting and mitigating DNS exfiltration compared to these userspace-based solutions that rely solely on traffic pattern analysis.

Chapter 4

IMPLEMENTATION

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

4.1 Security Framework Overview

The security framework implemented for distributed environments using endpoint security approach enables real-time disruption of stealthy DNS C2 channels and DNS tunneling, preventing malicious exfiltrated DNS packets from passing through the endpoint to ensure negligible data loss. It provides robust capabilities for terminating malicious C2 implants, offering deep system observability and cross-protocol protection by dynamically creating in-kernel network policies that block remote C2 server IPs. Designed for massive scalability and production readiness in modern cloud environments, the framework supports threat event data streaming to enable asynchronous communication. This, in turn, allows for horizontal scalability of data plane nodes and addresses DGAs by dynamically blacklisting domains via RPZ directly on the DNS server.

4.1.1 Data Plane

The data plane consists of eBPF node agents built purely in golang for pure performance, lesser memory footprints and ease of concurrent programming via goroutines and channels deployed across all eight CSSVLAB nodes. These agents operate in userspace and dynamically inject eBPF program into the kernel's TC layer at the egress point for all physical network interfaces at the endpoint to perform in-kernel DPI and filter DNS exfiltrated packets transmitted over UDP. The agent also inject other eBPF programs attached kernel hook

points like kprobes (kernel probes) to monitor the creation of new network devices, raw tracepoints to detect process termination, and socket cgroup hooks (for older kernels) to retrieve `task_struct` when native `task_struct` support is unavailable over root kernel TC programs. eBPF program types outlines all eBPF programs and their corresponding kernel injection points utilized by the eBPF node agent.

Complementing egress filtering, the agents monitor incoming packets at ingress for real-time inference and maintain a userspace LRU cache of blacklisted domains. This cache is shared with eBPF programs via eBPF maps to ensure coordinated filtering and detection across data paths.

The node agents support two prevention modes, which can be configured via the eBPF node agent configuration in userspace. A dedicated eBPF map is used to enable or disable either mode when injected into the kernel. By default, both modes are enabled to ensure high security.

- **Strict Enforcement Active Mode**: DNS packets over standard ports (53, 5353, 5355) are scanned in-kernel using eBPF at the TC egress hook. Malicious packets are dropped immediately. If classification exceeds eBPF instruction limits, the packet is redirected to userspace for further analysis. Userspace agent sniffs redirected traffic and checks against the domain blacklist cache or perform 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 cache. This mode effectively halts all forms of DNS exfiltration, while also enabling live disruption of C2 communication and, ultimately, termination.
- **Process-Aware Adaptive Passive Threat Hunting Mode**: Designed for nonstandard UDP port 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 eBPF maps. Subsequent DNS packets

from that process are dropped in kernel effectively breaking C2 implant communication with 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 DPI, node agents manage network namespaces and virtual bridges using the `veth` driver. Network topology explains the network topology, which starts with virtual namespaces and the `veth` bridge driver over the kernel link layer in the datapath. They maintain eBPF map file descriptors to bridge kernel-userspace communication, enabling advanced analysis. The lifecycle of eBPF programs is tied to the agent in userspace, which operates with elevated privileges to interact with the network datapath, syscall layer, and other privileged kernel subsystems. Node agent kernel capabilities explains the kernel capabilities which the eBPF node agent runs in. Both mode of agent operation support active response mechanisms include malicious process termination when userspace agent detect repeated exfiltration exceeding configured thresholds. Filtering of traffic in kernel is driven by adaptive thresholds values or limits configured in kernel eBPF maps utilized by eBPF programs for filtering traffic post raw parsing DNS protocol from socket buffer. In userspace, the eBPF node agent also loads quantized and serialized ONNX (Open Neural Network Exchange) deep learning models for fast inference, and exports system telemetry metrics to observanblity backend, and threat events to centralized message brokers. All the configuration which the node agents in userspace enforce in kernel programs is completely reprogrammable from control plane.

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 Node Agent Functionality

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 bridge netdev's	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 Node Agent

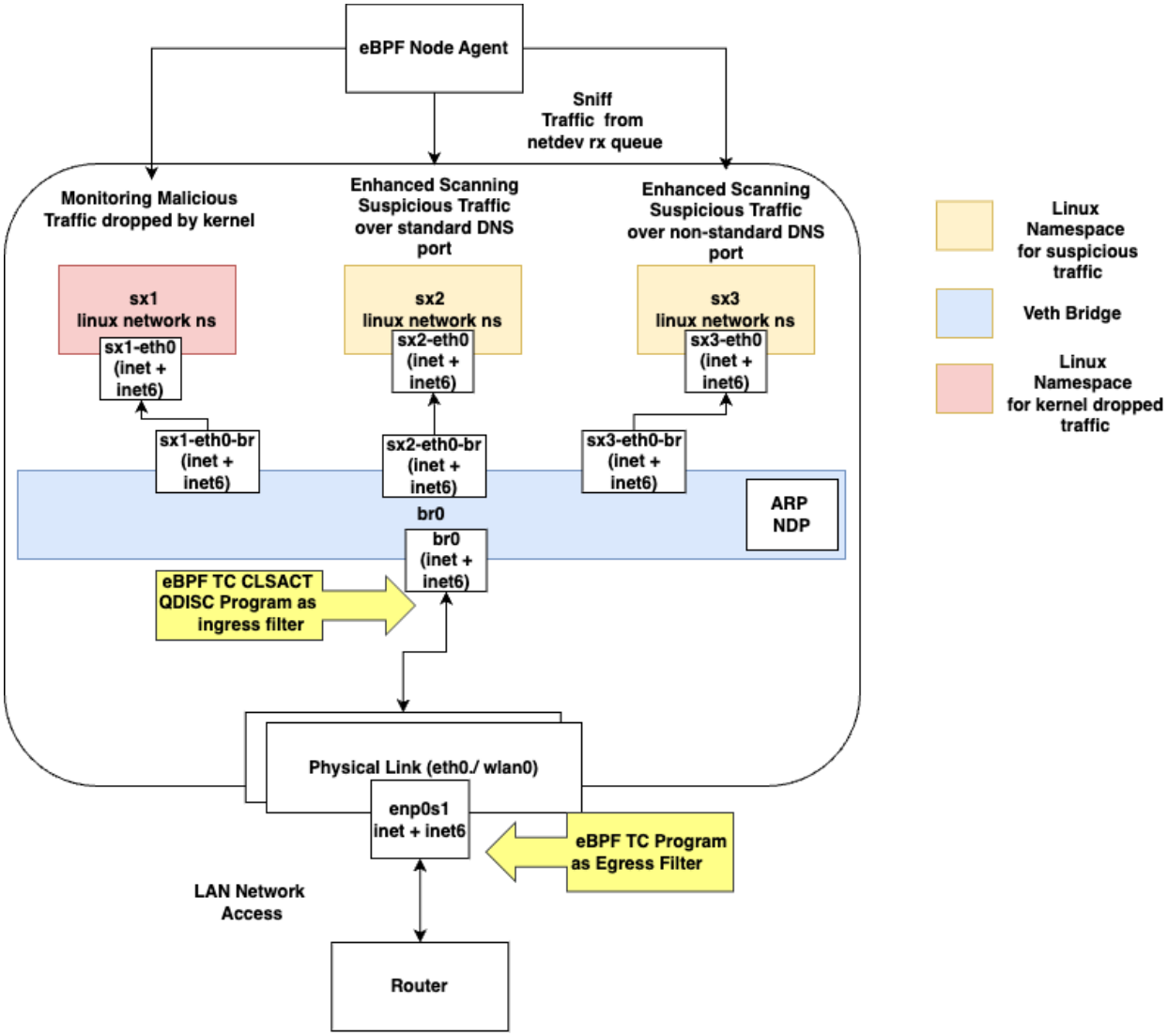


Figure 4.1: eBPF Node Agent created Network Topology at endpoint

4.1.2 Distributed Infrastructure

In addition to the data plane nodes, the framework includes an open-source DNS infrastructure built with PowerDNS. The setup consists of a PowerDNS Recursor for upstream resolution and an authoritative DNS server for handling internal zones. While no actual local domains exist, the authoritative server is primarily used to generate malicious C2 domains

using a domain generation algorithm (DGA) on a single compromised endpoint within the CSSVLAB environment. This design enables DNS-based C2 and tunneling attacks across the data plane. To support DGA-driven malicious domain generation without relying on public cloud DNS providers, both PowerDNS components were deployed locally. The authoritative server uses a Postgres backend to store DNS zones, records, transfers (AXFR, IXFR), TSIG keys, DNSSEC keys, and related metadata. All data plane nodes are configured to use the PowerDNS Recursor as their default DNS resolver. Kafka acts as a message broker for real-time streaming of threat events detected by eBPF agents in the data plane. To enhance DNS-layer defenses, the framework uses PowerDNS Recursor interceptors to inspect DNS queries before resolution or forwarding. These interceptors perform inference using the same ONNX-serialized deep learning model as the data plane, specifically handling TCP-based DNS queries offloaded from the eBPF agents in data plane. Finally, the recursor is integrated with Response Policy Zones (RPZ), backed by Postgres. These zones are dynamically updated by the controller to blacklist second-level domains (SLDs) tied to malicious C2 activity, as described 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 node agents running in the data plane. Based on these consumed events 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, it fully supports reprogramming the data plane by publishing Kafka topics consumed by node agents, enabling them to rehydrate their local blacklist domain cache and immediately enforce updated policies. This design significantly improves performance for real-time inferencing in userspace.

4.2 Data Plane

The implementation of the eBPF node agents deployed on data plane nodes is organized as follows. First, the active mode describes the handling of non-encapsulated DNS traffic, detailing the interaction between in-kernel eBPF programs and the userspace node agent. Second, the passive mode explains the passive mode. Third, the encapsulated phase outlines the handling of DNS exfiltration over encapsulated traffic, currently supported only in the active phase. Fourth feature analysis section covers features extracted by eBPF programs for in-kernel classification, along with userspace feature extraction for deep learning model training and real-time inference on suspicious traffic redirected from the kernel. Fifth, dataset describes the dataset used for model training, and finally, model details the model architecture, its serialization through ONNX, and quantization for optimized inference performance.

4.2.1 Strict Enforcement Active Mode

After injecting the eBPF programs and attaching them as direct-action filters to the kernel TC egress CLSACT QDISC across all physical network interfaces, the eBPF program is triggered immediately upon packet arrival at the netdev layer. At this stage, inside the QDISC egress filter, the kernel provides a fully formed SKB, which has traversed the upper layers of the kernel network stack and is ready for transmission to the NIC tx queues. The eBPF filter is executed for each packet, with the kernel passing the SKB as input to the filter, and the program runs across multiple CPU cores in parallel. All eBPF maps in this mode are global, and the kernel handles concurrent access to these maps to ensure proper synchronization across cores. First, the maps section explains the fundamental structure of the eBPF LRU hash maps used to store state information and manage packet flow.

Next, the netflow across kernel and userspace is explained as follows: kernel eBPF filter packet classify: Describes the packet process and classification on first-time arrival over the TC filter, including actions based on kernel features in the SKB, and handling redirected packets from userspace after timing checks or brute-force verification checks post deep scan.

It also details the flow if a packet is found suspicious, including eBPF map updates and redirection to a different veth bridge owned by the eBPF node agent for upstream Linux namespace processing up to userspace and through the kernel network stack. userspace eBPF agent packet handling: Covers userspace handling of the redirected packet, including sniffing in zero-copy mode. Concurrency handling: Details both userspace and kernel programs concurrency handling over these shared eBPF maps residing in kernel space to userspace eBPF node agent as well internal synchronization across handled in kernel eBPF programs running these programs over differen CPU cores.

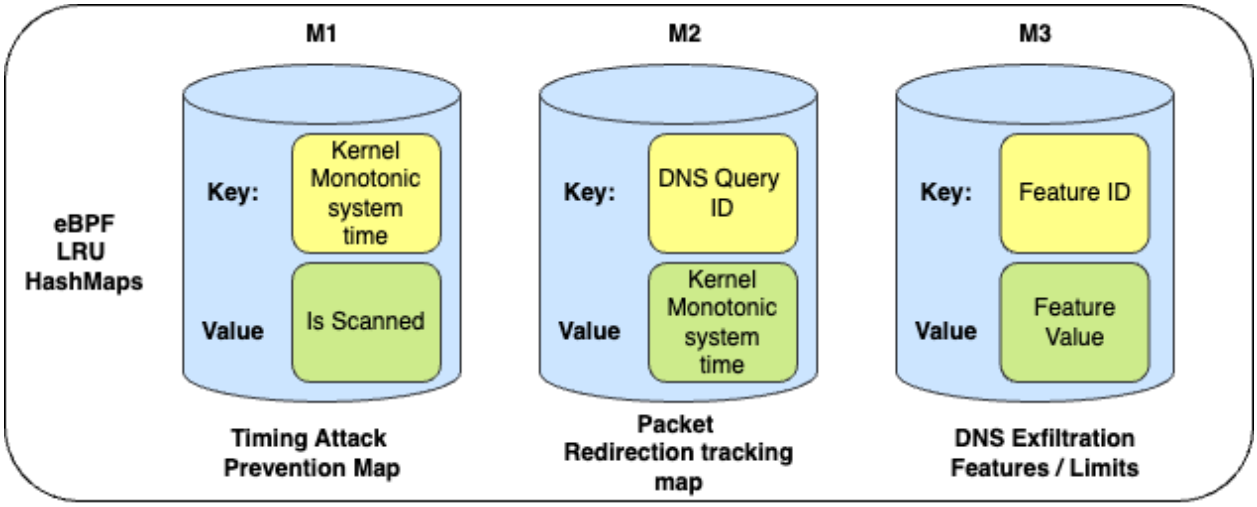


Figure 4.2: eBPF Maps and structure for Node agent in active phase

eBPF Maps in Active mode

1. **DNS Exfiltration Feature Map:**

   Key: Feature identifier

   Value: Filtering or classification parameter used by the eBPF TC egress program to process DNS traffic.

2. **Packet Redirection Tracking Map:**

   Key: DNS query ID

Value: Kernel system monotonic time (nanoseconds). Tracks suspicious packets redirected across interfaces, using NMI(Non Maskable Interrupts) safe timestamps.

3. **Timing Attach Prevention Map:**

   Key: Monotonic timestamp (nanoseconds)

   Value: Scanned flag (true/false). Used by the userspace agent to authorize scanned packets, and by the kernel eBPF program to verify packet integrity before resend.

eBPF program packet processing over egress TC QDISC filter

Once a DNS packet traverses down the kernel network stack after being written to the UDP socket by userspace, it reaches the traffic control (TC) layer, with each CPU handling classification and traffic shaping under the parent QDISC of the respective network interface, and before processing any parent QDISC kernel triggers the CLSACT QDISC where the eBPF direct-action filter is attached. At this point, the eBPF program determines whether the packet is arriving for the first time or has been rescanned from userspace by parsing the raw DNS protocol directly from the kernel SKB. The eBPF program peels off the packet layers from the SKB using the Linux kernel's core protocol structure definitions, with each protocol layer (L2 to L4) parsed based on the corresponding headers. Since most Layer 7 (application) protocols are handled in userspace, the kernel processes application payloads as raw bytes. Like any other application protocol, DNS also has its protocol header and payload. The eBPF program parses the DNS header from the raw application data using custom written C structure definitions for DNS protcol inside the kernel according to RFC 1035. After populating all required fields like query ID, opcode, flags, question count, and answer count, the eBPF program extracts metadata for further evaluation. If the packet is arriving for the first time, there will be no corresponding entry for its query ID in the `dns_packet_redirection` eBPF map. The TC eBPF program proceeds to parse the DNS application data after the DNS header. It extracts features from the first DNS question; if multiple questions are present, the packet is flagged as suspicious and redirected without further checks, since multiple questions per query are typically disallowed in normal DNS

behavior and rejected by most DNS servers. In addition, it is expected that the answer count is zero because the endpoints running these C2 and tunneling implants do not operate full DNS servers and instead communicate with their C2 servers through crafted DNS questions and the additional section of the DNS packet. These extracted features are evaluated against the kernel feature set described in kernel features, and based on the evaluation, appropriate TC actions are enforced, such as `TC_ACT_SHOT` to drop, `TC_ACT_OK` to forward, or `bpf_redirect` to redirect the packet to a different network interface. If the packet is benign or malicious the action is taken immediately according to the Active Mode DNS Parsing and Feature Evaluation Algorithm. For suspicious packets requiring deeper inspection, before redirection to the different netdev, the parsed DNS query ID is stored as a key in the `dns_packet_redirection` map, with the value being the current kernel system monotonic timestamp obtained via `bpf_ktime_get_ns`, providing nanosecond precision to accurately track the time of redirection and detect possible timing-based evasion. At this point, additional map information populated by the eBPF node agent in userspace during bootup is fetched, including the recorded L3 address of the bridge interface to which the packet will be redirected, the interface network `if_index`, and other system observability maps. Redirection counters are also incremented for the eBPF node agent to export system observability metrics. Prior to redirection, the eBPF program determines whether the incoming packet is IPv4 or IPv6; if IPv4, the program performs destination NAT (DNAT) inside the kernel to modify the L3 destination address and incremental checksum calculation for Ipv4 packets to point to the userspace-owned bridge for inspection, while for IPv6, due to the absence of an L3 checksum in protocol, a static checksum is assigned to maintain consistency. The suspicious packet is then redirected to the RX queues of the associated netdev owned by the userspace eBPF node agent. When the userspace node agent completes deeper inspection and sends the packet back through an AF_PACKET socket, the TC eBPF program is retriggered. To prevent infinite loops or reprocessing of malicious resends, the eBPF program reparses the DNS protocol from the SKB, fetches the associated timestamp from the `dns_packet_redirection` map using the DNS query ID as key, and retrieves the scan verifi-

cation flag from the `dns_redirect_ts_verify_map`. If the flag is set, authorizing the packet as legitimately scanned and returned by the userspace agent, the packet is forwarded toward the NIC and exits the kernel. If the flag is unset, suggesting the packet is forged, unsanctioned, or a result of a timing or brute-force attack by a compromised implant in userspace, the packet is dropped immediately by the eBPF program. The detailed algorithms for active redirection timestamp handling and userspace resend verification are described in eBPF Map Handling in Active Phase. Through strict enforcement of privileged map access and precise use of the kernel monotonic clock, the system robustly prevents timing attacks, brute-force attempts, and unauthorized packet exfiltration while deep inspection is performed on the packet, involving context switching between user space and kernel space.

Userspace eBPF Node Agent packet processing

The userspace eBPF node agent utilizes kernel BPF bindings to abstract raw BPF syscalls, primarily through libbpf. The loader runs with the required kernel capabilities as explained earlier, injecting all eBPF programs and owning the file descriptors for eBPF maps both for pinned and unpinned maps to BPF kernel filesystem. It injects the root eBPF program powering the TC filter attached to the egress direction. The agent spawns threads to continuously sniffs traffic from its owned network namespace dedicated for handling redirected suspicious traffic in active mode using `AF_PACKET` sockets, reading directly from tap interfaces or RX queues in zero-copy mode to avoid additional buffer allocations in userspace for performance. Since the agent has access to all eBPF maps via their file descriptors, it relies on pcap, an extension of libpcap, to parse DNS application-layer traffic redirected from kernel for inspection. After sniffing, the agent extracts userspace DNS features as outlined in Userspace Features. It maintains two LRU caches in userspace heap memory: one representing Cisco's dataset of the world's top one million SLDs, and one representing malicious domains previously inferred, preventing re-inferencing of known malicious packets. If the parsed SLD is found in the benign cache, the packet is immediately forwarded without inference; due to DNS protocol properties, malicious C2 servers cannot redirect these legitimate domain

zone files of highest reputed benign domains. The agent forwards benign packets through `AF_PACKET` or `AF_XDP` sockets accordingly. The agent also supports live Kafka consumers, enabling dynamic updates to the userspace LRU caches from a controller in real time supporting eBPF node agent re-programmability from controller. For AF_XDP sockets, packets are injected directly into the TX queues of the device driver, bypassing the eBPF TC egress filter, while cleaning up associated entries in the `dns_packet_redirection_map` added during redirection. For `AF_PACKET` sockets, the agent fetches the kernel redirection timestamp using the parsed DNS transaction ID, updates the `dns_redirect_ts_verify_map`, and marks the packet as scanned so the kernel allows it to pass. If no cache hit occurs, features are extracted live from the packet and passed to the deep learning model; if the packet is inferred as malicious, it is dropped, the corresponding SLD is blacklisted in the userspace malicious cache, and a Kafka event is produced for the controller to update DNS server blacklists. If benign, the packet is resent using the same approach as a benign cache hit. All malicious detection events are exported to Prometheus along with system observability metrics like redirection counts. The userspace agent continuously monitors maliciously detected packets and the parent processes involved in sending these packets, aided by the same eBPF program that tracks all redirected suspicious DNS packets and their query IDs. If these packets exceed a defined malicious threshold, the userspace agent sends a SIGKILL signal to the process, effectively terminating the malicious implant at the endpoint. Userspace Active Mode details the packet processing algorithm utilized by the userspace agent.

eBPF Maps concurrency handling

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 threads in usesrpaace 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 fd's aren't shared across userspace processes, the maps are exclusively owned by the userspace eBPF node agent ensuring reference count for all the owned map fd in kernel's global fd table. This model, paired with spinlock-based concurrency, 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` use unique keys (monotonic timestamps or DNS query IDs), preventing stale reads, race conditions, and inconsistent updates that could otherwise leak malicious exfiltrated packets. In addition every updates done over eBPF maps in kernel are via builtin LLVM concurrency helpers to ensure strong atomic map updates synchronizing kernel memory address locatiosns for map updates. Thus, strict and reliable control is maintained. The active mode details the full userspace and kernel pipeline over TC.

Algorithm 1: DNS RAW SKB Parsing over Egress TC CLSACT QDISC in **ACTIVE** Mode

Input : Socket buffer (`skb`), eBPF LRU hash maps: `dns_limits`, `dns_packet_redirection`, `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 Layer 4 (Transport) from `skb`;
if *`skb->data_ptr` exceeds `skb->data_end`* **then**
  Drop packet via `TC_ACT_SHOT`;
if *`skb->protocol` = IPPROTO_TCP* **then**
  Forward packet via `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` ≤ 2* **then**
  Forward packet via `TC_ACT_OK`;
  return;
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**
  Drop packet via `TC_ACT_SHOT`;
  return;
if *`q_type` ∈ {`TXT`, `ANY`, `NULL`}* **then**
  Perform `bpf_map_updates`;
  return;
Forward packet via `TC_ACT_OK`;

Algorithm 2: DNS eBPF Map Handling Prior to `skb_redirect` and Post-Socket Write over `AF_PACKET` from userspace in **ACTIVE** Mode

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

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);
Fetch `dst_ip` (destination IP), `bridge_if_index` from `netlink_links_config` using `if_index` from `skb`;
Fetch `skb_mark` from `netlink_links_config` using `if_index`;
Fetch `dns_kernel_redirect_val` = {`l3_checksum`, `kernel_time_ns`} from `dns_packet_redirection` using `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`;
    Mark `skb->mark` = `skb_mark`;
    Update `dns_packet_redirection`:
        - Key: `tx_id`
        - Value: {`l3_checksum`, `kernel_time_ns`}
    Update `dns_loop_time`:
        - Key: `tx_id`
        - Value: `kernel_time_ns`
    Fetch current redirect count from `redirect_count_map` using `if_idx`;
    Increment global redirect count for `if_idx`;
    Update `redirect_count_map`:
        - Key: `if_idx`
        - Value: `new_count`
    Perform `bpf_redirect(bridge_if_index), BPF_F_INGRESS`;
else
    `// Userspace deep-scanned packet re arrived, verify there is no timing attack via forged packet through unknown sender except eBPF node agent in userspace`
    Fetch and delete `redirect_ts_verify_val` from `dns_redirect_ts_verify_map` using `kernel_time_ns` as key;
        - Key: `kernel_time_ns`
        - Values: `l3_checksum`
    Using `kernel_time_ns` from `dns_kernel_redirect_val`
    if *not* ***redirect_ts_verify_val*** **then**
        `// Timing attack: user-space agent did not emit packet`
        Perform `TC_ACT_SHOT`;
    else
        Delete `redirect_ts_verify_val` from `dns_redirect_ts_verify_map`;
        Perform `TC_DROP`;

Algorithm 3: User-Space eBPF Node Agent with Deep Learning and Event Streaming for Deep Parsing 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 dropped (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 the monitoring maps (redirection_count, loop_time, etc);
Fetch `isSLDBenign` from userspace agent's LRU hash of top 1M SLDs;
if *`isSLDBenign`* **then**
    Set `shouldRetransmit` ← `true`;
else
    Fetch `isBlacklistedSLDFound` from userspace agent LRU Hash;
    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 `kernel_time_ns` from `dns_redirect_map` using `tx_id` as key;
    if *AF_PACKET* **then**
        Update `dns_redirect_ts_verify_map`:
            - Key: `kernel_time_ns`
            - Value: `l3_checksum`
        Replace packet's `l3_checksum`;
        Serialize packet payload to raw bytes;
        syscall.write(AF_PACKET, SOCK_RAW, 0);
    else if *AF_XDP* **then**
        Delete `kernel_time_ns` from `dns_redirect_map`;
        Serialize packet payload to raw bytes;
        syscall.write(AF_XDP, SOCK_RAW, 0);

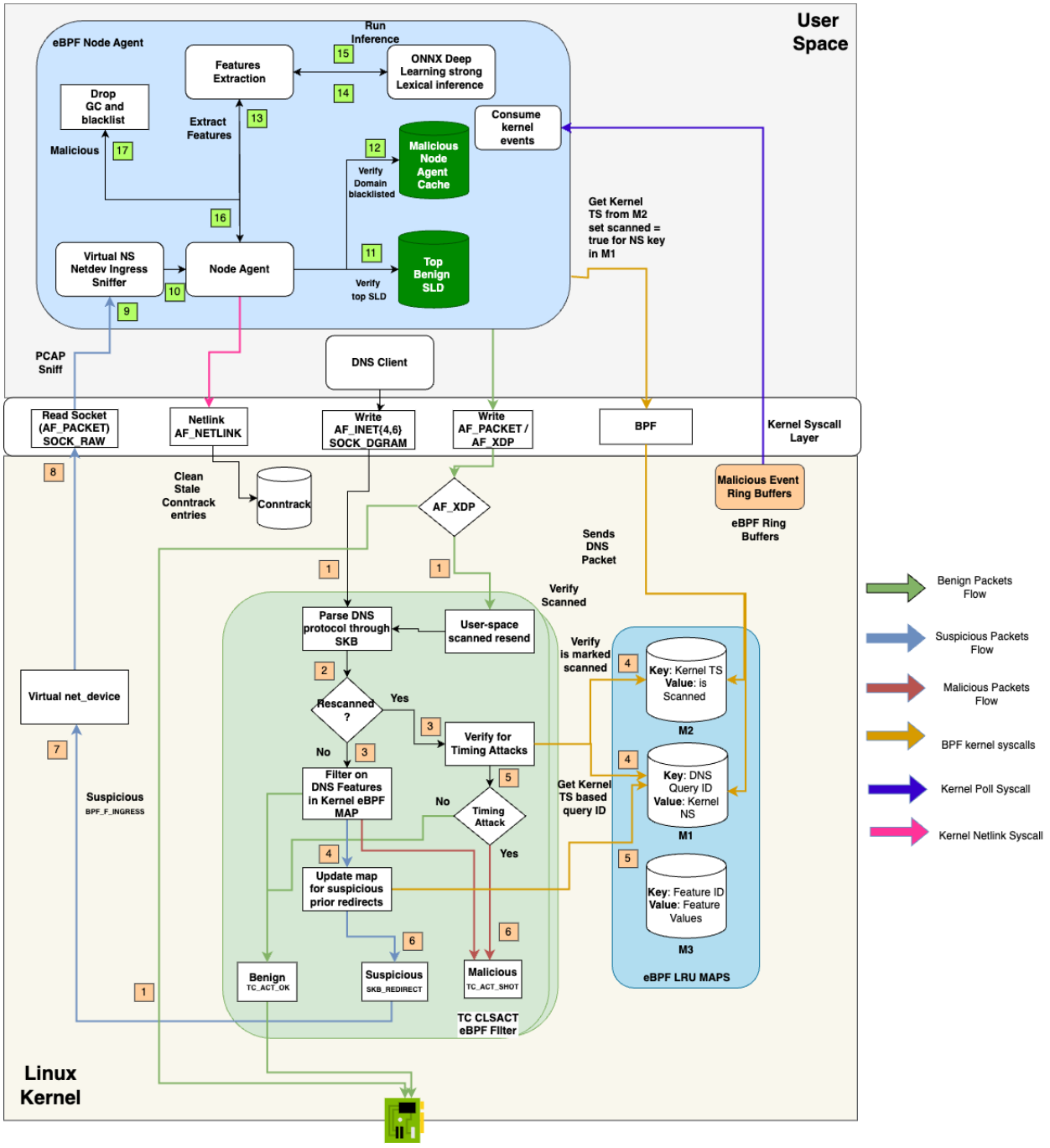


Figure 4.3: eBPF Agent DNS Exfiltration Prevention Flow for Strict Enforcement Active Mode

4.2.2 Process-Aware Adaptive Passive Threat Hunting Mode

The same eBPF program attached to the egress clsact TC filter, as described in active mode, is reused for passive mode. It is attached to all physical netdev interfaces at the endpoint. The implementation of this mode is structured as follows: First, maps describes the eBPF map structures, their types, and their specific significance in passive mode. Second, kernel packet processing details how the kernel eBPF program drops malicious exfiltrated packets, which are uniquely tied to the userspace processes responsible for sending them. This section also explains how additional kernel tracepoints, particularly those related to process scheduling, are used to perform garbage collection on map values. Third, userspace packet processing outlines how the eBPF node agent handles packets in this mode. Finally, eBPF map concurrency covers how the system maintains consistency between the kernel and userspace, even when multiple userspace threads and kernel programs run in parallel across different CPU cores read or updates the eBPF maps .

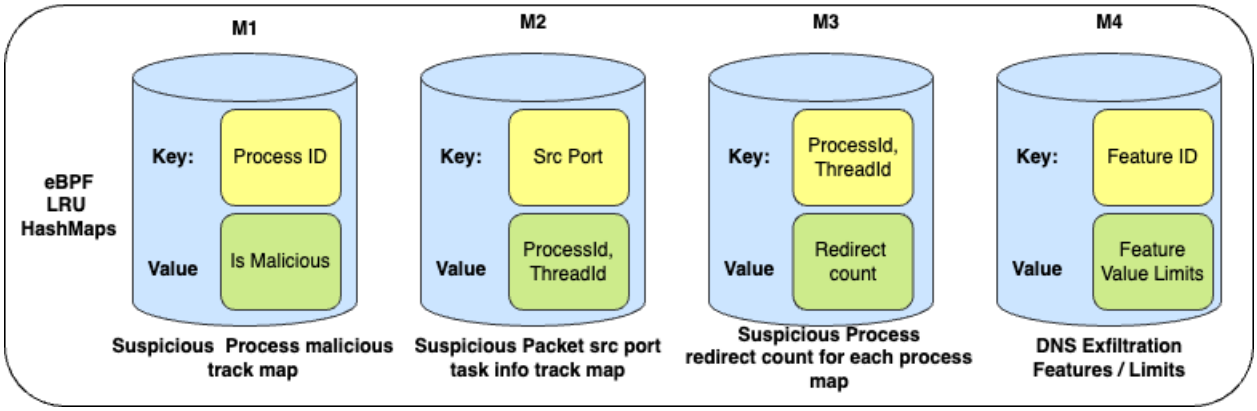


Figure 4.4: eBPF Maps and structure for Node agent in passive phase

eBPF Maps in Passive mode

1. **DNS Exfiltration Feature Map:**

   Key: Feature identifier

   Value: Filtering or classification parameter used by the eBPF TC egress program to

process DNS traffic.

2. **Suspicious Process Redirect Count per process Map:**
   Key: Process Info extracted from kernel task_struct
   Value: Count per process level suspicious redirected DNS layered over non-standard UDP DNS ports.
3. **Suspicious Packe:**
   Key: Source Port of the potential layered DNS packet over random UDP port
   Value: Process Info extracted from kernel task_struct
4. **Malicious Process Track Map** Key: Process Info extracted from kernel task_struct
   Value: Is the Process malicioous based on previous detected malicious transfer through same process.

4.2.3 ***eBPF program packet processing over egress TC QDISC filter***

In passive mode, DNS exfiltration is often attempted by layering DNS payloads over random UDP transport ports, excluding the standard DNS ports that are directly handled in active mode. The eBPF program parses the lower layers up to transports in the same manner as explained in active mode. As a deliberate design decision, these packets are not immediately redirected by the eBPF program, even if they appear suspicious. This prevents unnecessary network congestion and added latency for other legitimate Layer 7 protocols using arbitrary UDP ports. Since the eBPF program cannot guarantee that the application data in the skb truly belongs to DNS, it avoids aggressive redirection. Instead, it permits the packet to pass through the kernel as usual. Internally, however, it invokes `skb_clone`, a kernel helper that creates a full clone of the packet from the original SKB.

In addition, because cloning packets increases memory usage, this mode avoids cloning all packets over random UDP ports. Instead, it attempts to raw-parse the SKB application data for potential DNS structures. If any validation exceeds the SKB limits (e.g., `skb->data_end`), the packet is not considered DNS. If certain bounds align the SKB payload with potential

DNS structures, additional checks are applied to validate that the application payload genuinely conforms to DNS, adhering to the limits defined in RFC 1035. DNS SKB parsing explains these checks before performing clone redirection to the netdev owned by the eBPF node agent.

Prior to clone redirection, the eBPF TC program fetches the kernel `task_struct` (process control block) via the `bpf_get_current_pid_tgid` helper, identifying the actual parent process and thread group ID in userspace involved in sending the packet. It then checks the `malicious_process_map` to determine whether the associated parent process ID has been flagged as malicious by userspace, based on previously detected DNS exfiltration attempts from the same process. If flagged, the eBPF program begins dropping the original packets but continues to clone and redirect them to userspace for monitoring. This behavior is referred to as a trap, where the kernel drops the packets yet keeps redirecting them to the eBPF node agent, allowing the agent to monitor the malicious implant's retry behavior and gather runtime telemetry before taking termination action. This is an extremely robust and advance security design in the case of stealthy beaconing implants with randomized intervals. Once the implant's parent process is flagged as malicious, subsequent beacon attempts are discarded in the kernel, forcing retries. The userspace agent observes these and collects system-level metrics to justify a kill signal against the offending process. Alternatively, if the process is not flagged as malicious, the eBPF program extracts the Layer 4 UDP port from the SKB and updates `src_port_task_struct_map` with the source port as the key and the extracted process ID and thread ID as the value. It also updates `task_struct_redirect_ct_map` with a composite key of process ID and thread ID, and an atomically incremented value representing the count of suspicious redirects. eBPF map's explains the eBPF map updates performed by the eBPF program in passive mode. This continues until the process ceases sending DNS packets. If the process terminates and no malicious DNS activity was detected, cleanup is necessary to avoid dangling entries in `task_struct_redirect_ct_map`. For this purpose, an additional eBPF program is attached to the raw kernel tracepoint `tracepoint/sched/sched_process_exit`, which triggers when a process terminates or is

killed. This tracepoint retrieves the exiting process's information and removes its entries from both `task_struct_redirect_ct_map` and `malicious_process_map`. The userspace eBPF node agent only sends a kill signal once the number of prevented DNS exfiltrations for a single process exceeds a configured threshold defined in the userspace agent's configuration. If the process exits before reaching this threshold, the eBPF tracepoint program still ensures cleanup of `malicious_process_map` entries accordingly.

Userspace eBPF Node Agent Passive Mode

The eBPF node agent launches concurrent, multiplexed goroutines—each pinned to a specific OS thread—to implement passive mode, which differs architecturally from active mode. These goroutines sniff traffic within a dedicated Linux namespace and its associated physical network device. They operate in zero-copy mode in userspace, similar to the active phase, where the kernel eBPF program redirects potentially layered DNS traffic for further inspection. All clone-redirected packets arriving in userspace contain both application data and lower-layer headers up to the transport layer. The userspace eBPF agent first parses the application payload, searching for embedded DNS structures. If no valid DNS layer is found, the extracted Layer 4 transport information is removed from the `src_port_task_struct_map`, using the source port as the key. If DNS data is identified within the tunneled application payload, the same evaluation logic as in active mode is followed. This includes extracting domain names and checking them against a local in-memory blacklist LRU cache. If a match is found, the domain is marked as blacklisted. Otherwise, the packet undergoes feature extraction and is passed to a deep learning model. If the model detects the packet as malicious, the userspace blacklist LRU cache is updated accordingly, and a detected threat event is streamed to the observability backend in the same manner as active mode. Since this is a clone-redirected packet, it is not re-injected into the network. Instead, the eBPF node agent uses the extracted source port to retrieve the associated process ID and thread ID from `task_struct_redirect_ct_map`, identifying a potentially malicious userspace process running alongside the eBPF node agent. After retrieving this information, the node

agent updates the `malicious_process_map`, marking the parent process as malicious with a key-value entry of the process ID and a value of true. This allows the kernel eBPF program to drop all subsequent packets originating from that process. Additionally, the node agent examines rich telemetry from `task_struct_redirect_ct_map` — specifically, the number of times the packet was clone-redirected — using the process ID and thread ID as keys (retrieved via `src_port_task_struct_map`). If the redirected count exceeds a configurable threshold and the associated process has been flagged as malicious, the eBPF node agent — running with `CAP_SYS_ADMIN` capability — sends a kill signal to terminate the identified malicious process. With process-aware adaptive passive mode, the kernel eBPF program effectively aids the userspace agent in threat hunting processes that exfiltrate data over DNS via nonstandard UDP ports. Once identified, the eBPF agent terminates the malicious process in userspace. Userspace packet handling explains in detail how the eBPF node agent processes packets in passive mode.

4.2.4 ***eBPF Maps concurrency handling***

The map concurrency principles remain consistent with those in active mode, relying on per-CPU kernel spin locks with a globally shared eBPF map. This allows the eBPF program, scheduled across different CPUs, to update and reference-count map FDs, which remain alive as long as the eBPF node agent process is active in userspace. However, this mode introduces specific concurrency handling for map updates. For `src_port_task_struct_map`, the unique key—representing the source port—ensures atomic read and write operations, starting in the kernel when a DNS packet first arrives, followed by a corresponding read in userspace. Similarly, `malicious_process_map`, which is updated by the userspace eBPF node agent to mark processes as malicious, always uses the `BPF_ANY` flag (create or update) for updates. As concurrent goroutines in userspace process sniffed packets in parallel, updates to this map are synchronized using a userspace mutex lock to ensure consistency. Since the kernel eBPF program across different CPUs only reads from this map, this design avoids blocking readers and improves packet processing throughput—similar to the RCU (read-

copy-update) concept, which prioritizes performance and high-speed data access. Finally, for `task_struct_redirect_ct_map`, where the eBPF program in the kernel (executing on multiple CPUs) writes the redirect count while the eBPF node agent in userspace reads it, consistency is maintained using the eBPF map's internal spin lock mechanism along with the `BPF_ANY` update flag. Concurrent reads in userspace are synchronized via a separate userspace mutex, decoupled from the kernel's per-CPU spin locks.

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

Input : `skb` (socket buffer),
eBPF LRU hash mapss: `netlink_links_config`
Output : Packet Actions Clone Redirect action: `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 the header variable size (u8)`
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 the opcodes, rcode in DNS flag to vilate RFC 1035`
if *opcode* $\geq$ *6* **then**
  return `TC_ACT_OK`;
if *rcode* $\geq$ *24* **then**
  return `TC_ACT_OK`;
Fetch `dst_ip` and `bridge_if_index` from `netlink_links_config`:
- Key: `if_index` (from `skb`)
- Values: `dst_ip`, `bridge_if_index`

Perform `bpf_map_updates`

Algorithm 5: DNS eBPF map handling prior in **PASSIVE** mode of agent

Input : `skb` (socket buffer),
eBPF LRU hash mapss:
`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`

1 Parse DNS header from `skb`;
2 Extract DNS transaction ID (`tx_id`) from DNS header;
3 Fetch `dst_ip` and `bridge_if_index` from `netlink_links_config`:
- Key: `if_index` (from `skb`)

Fetch `skb_mark` from `netlink_links_config`:
- Key: `if_index`

Fetch kernel `task_struct` and `process_info` using `bpf_get_current_pid_tgid`:
- Key: `bpf_get_current_pid_tgid`

Fetch `is_malicious` from `malicious_process_map` using `process_id`:
- Key: `process_id`

if *not* `is_malicious` **then**
Update `src_port_task_struct_map` with `key=process_id`, `value=task_struct`;
Fetch `current_suspicious_ct` from `task_struct_redirect_ct_map` using `task_struct` as key:
- Key: `task_struct`
- Value: `current_suspicious_ct`

Update `task_struct_redirect_ct_map` with:
- Key: `task_struct`
- Value: `current_suspicious_ct + 1`

Increment global `clone_redirect_ct` counter;
Update `dns_packet_clone_redirection_ct_map` with:
- Key: `if_idx`
- Value: `new count`

Perform $\text{bpf}_{c}lone_{r}edirect(skb, bridge_{i}f_{i}ndex, BPF\_F\_INGRESS)$**else**
return `TC_ACT_SHOT`;
`// Drop the packet exfiltration attempt found over the process`

Algorithm 6: User-Space eBPF Node Agent with Deep Learning and Event Streaming for DNS Traffic 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 : Real-time updates to `malicious_process_map`

Sniff traffic over `veth` interfaces in isolated namespaces;
Verify DNS Layer in packet SKB data
Extract DNS L7 features: `tx_id`, `qd_count`, `ans_count`, query class/type, domain length;
Extract L4 transport ports: `src_port`, `dest_port`;
Fetch `kernel_task_struct` from `src_port_task_struct_map` using key `src_port`;
Fetch `isBlacklistedSLDFound` from userspace LRU hash;
if *`isBlacklistedSLDFound`* **then**
    Update `malicious_process_map` for process ID from `kernel_task_struct`;
    return;
Pass features to ONNX model for inference;
if *Inference == `MALICIOUS`* **then**
    Emit event to message broker;
    Blacklist SLD for related DNS records;
    Update `is_malicious flag` in `malicious_process_map` with key as process ID from `kernel_task_struct`;
    return;
else if *Inference == `BENIGN`* **then**
    ; `// No immediate action; future packets will be tracked and evaluated`
    return;

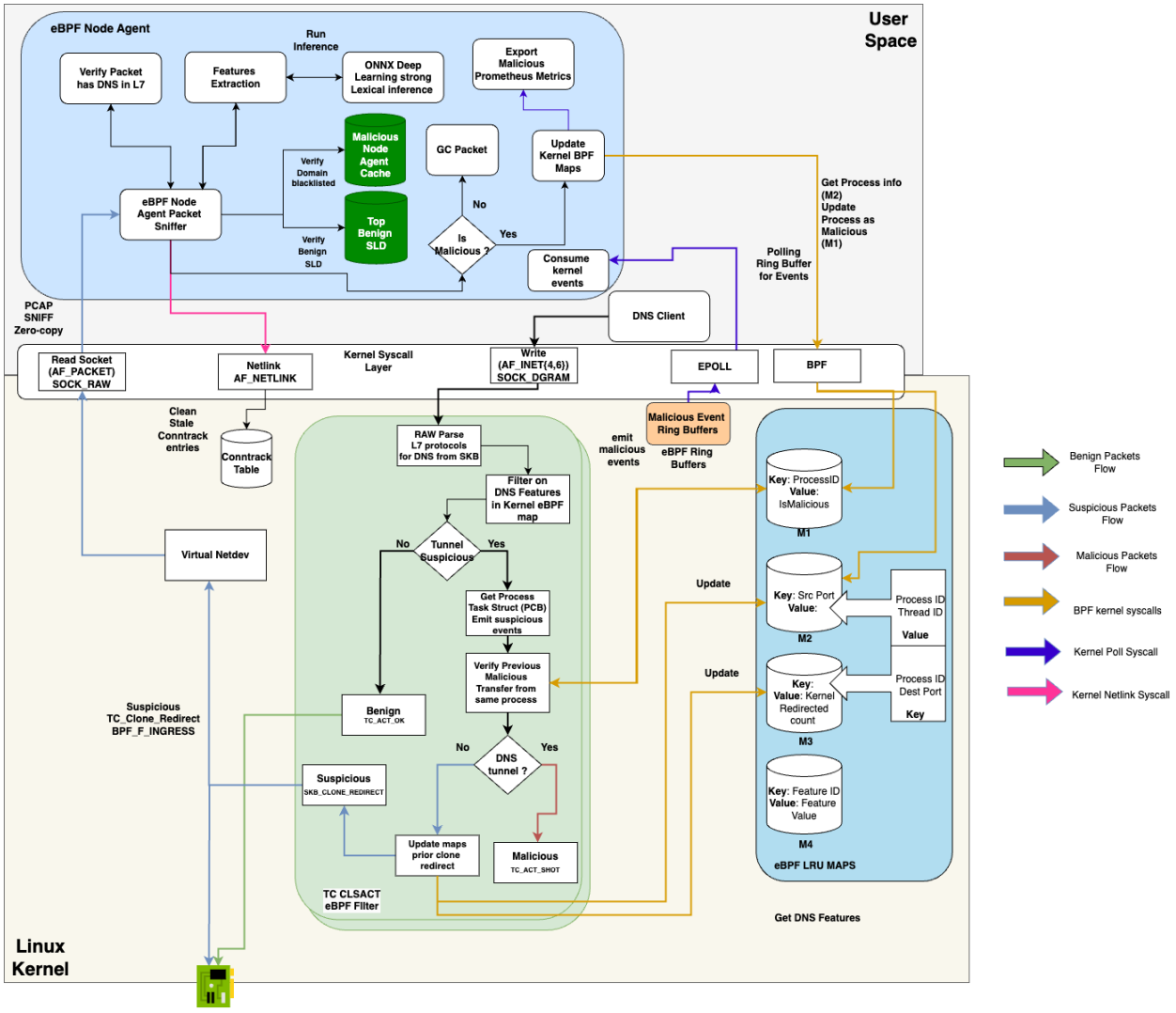


Figure 4.5: eBPF Agent DNS Exfiltration Prevention Flow for Process-Aware Adaptive Mode of Agent

4.2.5 DNS Exfiltration via Encapsulated Traffic

In the Linux kernel network stack, encapsulated traffic is managed through virtualization drivers that extend the core `netdev` interface with associated RX and TX queues. These drivers support protocol encapsulation at various layers—L2, L3, and L4—by wrapping one protocol within another. The current implementation of the eBPF node 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 used by OpenVPN, IPsec, or WireGuard. This design choice aligns with the DNS protocol's typical behavior, as DNS resolution rarely occurs over VPN tunnels. DNS exfiltration over encapsulated traffic in this context is limited to VLAN and TUN/TAP software network devices. VLAN encapsulation is similar and explained before in the SKB parsing in active phase by the TC eBPF program, where the eBPF program removes the L2 encapsulation layer (e.g., 802.1Q, 802.1AD) to expose the inner packet before proceeding with DPI. TUN/TAP interfaces are virtual software devices exposed to userspace as file descriptors. Malicious userspace processes can write tunneled packets containing DNS data directly to these interfaces, bypassing standard inspection paths. The kernel handles L3 encapsulation at the sender side and performs L2 de-encapsulation on the TAP (receiver) side before forwarding traffic upstream. These devices, typically created using `iproute2` or `netlink`, forward traffic through a physical NIC. In its current state, the eBPF node agent handles only plaintext encapsulated traffic. Since encapsulation by these network drivers occurs at higher layers of the kernel network stack, the `skb` often lacks visibility into encapsulation details, except in the case of VLAN-tagged packets. To counter DNS tunneling over TUN/TAP interfaces, the agent injects `kprobes` on 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 eBPF DPI program to the TC egress hook on the new interface, enforcing the same detection and prevention logic described in the active phase. Additionally, the exported ring buffer event includes rich telemetry about the process that created the tunnel, enhancing monitoring and threat attribution. See TUN/TAP interface and encapsulated traffic for a detailed flow.

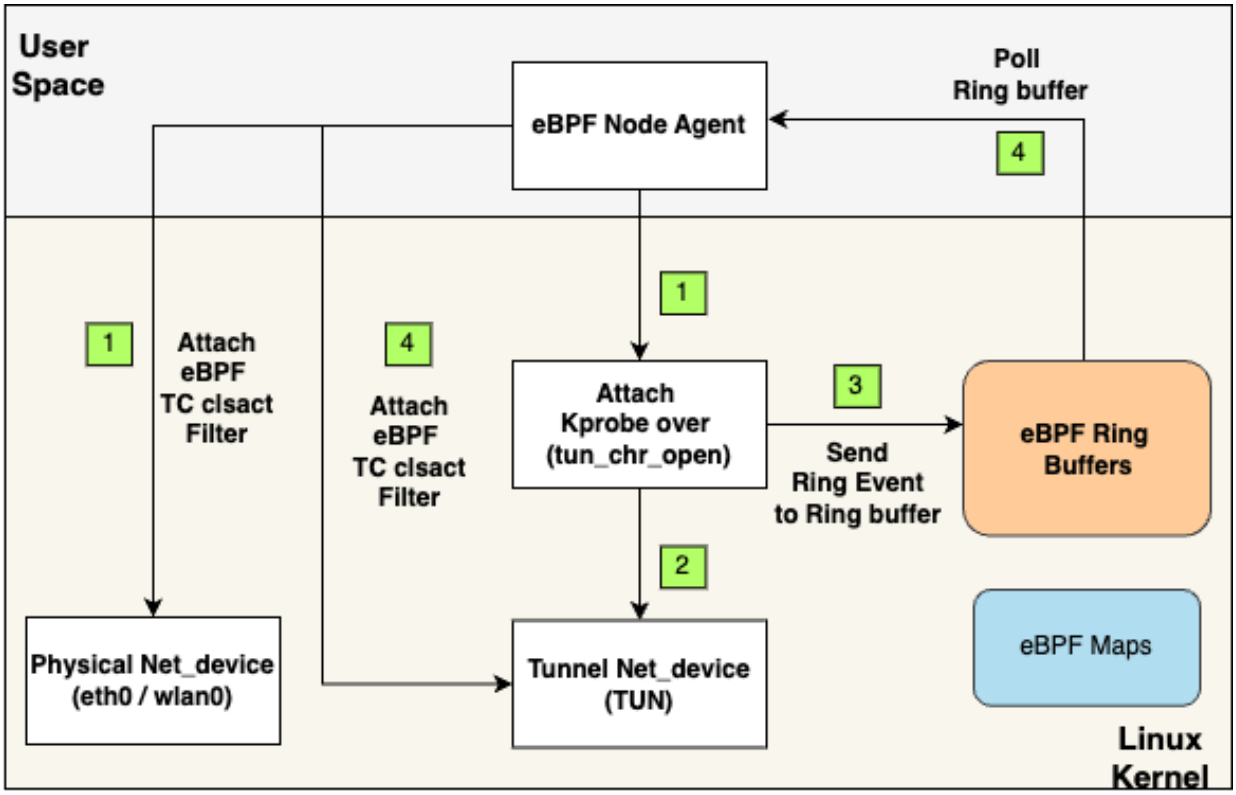


Figure 4.6: eBPF Node Agent Prevention flow over Tun/Tap Driver kernel function

4.2.6 Feature Analysis in Data Plane

The eBPF node agent filters traffic directly in kernel space using statically populated values in eBPF maps. These values modify packet processing and scheduling logic by correlating DNS packet fields within the kernel's socket buffer (sk_buff) with predefined threshold limits. The kernel's traffic control (TC) subsystem enforces this logic at runtime, inspecting and filtering each packet as detailed in the ingress and egress processing sections.

DNS packets over UDP are restricted to 512 bytes of data, regardless of the MTU (Maximum Transfer Unit) of the network interface. According to RFC 1035, DNS domains have a maximum length of 255 characters (including periods) and must adhere to specific label length limits (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 contain

non-domain information, yet still remain within the 512-byte UDP limit. For larger payloads, the EDNS extension allows fragmentation, while TCP-based DNS transport also supports larger payloads without exceeding these limits.

The implementation is divided into two parts. First, the kernel features used by eBPF programs are explained, focusing on classifying, filtering and redirecting suspicious DNS packets. Second, the features used by the userspace deep learning model for enhanced lexical analysis of DNS payloads to detect obfuscation in exfiltrated packets are discussed.

Kernel-space features

Due to kernel-level restrictions imposed by the eBPF verifier, DPI is limited to the first DNS question record, parsed from the DNS header by the eBPF TC program attached in the egress path. This design aligns with modern DNS behavior, as legitimate queries almost always contain a single question; thus, detection of multiple questions (via the qd_count field) 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 like 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 semantic relevance to DNS abuse, verifier safety, and low overhead, enabling fast, accurate classification and real-time enforcement entirely within the kernel. The full logic is detailed in kernel features.

Userspace deep learning model features

The deep learning model is trained over the eight features as outlined in The primary choice for selection of the features are based on analysis of exfiltration samples from open-source data exfiltration toolkits and adversary emulation C2 frameworks. Since this feature primarily focuses on lexical analysis of DNS exfiltrated payload to detect malicious payloaded encoded, The deep learning model extracts eight key features from live DNS traffic, as outlined in Table These features are based on protocol standards and are used for model training and for detecting DNS payload obfuscation.

4.2.7 Datasets

The deep learning model, trained utilizes three main datasets for training. First, Cisco's top 1 million SLDs are not used for model training but are instead loaded into an in-memory LRU cache by the eBPF node agent as top benign SLD. This design optimizes performance by preventing model inference on these domains, since any sniffed DNS traffic containing them in the SLD won't be an exfiltrated packet due to their auth zones being owned by authorized sources. Note as explained before reliance on domain scoring is used however the LRU cache is completely reprogrammable from 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 [22]. Due to the smaller number of malicious samples compared to benign ones, and to avoid training bias, a synthetic dataset was generated using open-source exfiltration tools—DET, DNSExfiltrator, DNSCat2, Sliver, Iodine and custom DNS exfiltration scripts. These exfiltrated datasets captured all forms of data obfuscation for different encoding or encryption algorithms as explained before , including tunneling and raw exfiltration, across a wide range of file formats including text (markdown, txt, rst, raw config files), image (jpg, jpeg, png), and video (mp4), generating a combined dataset of around 50 million domains, covering all the 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 Space

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

Table 4.4: DNS Features in Userspace

4.2.8 Deep Learning Model Architecture

The deep learning architecture in userspace enhances the eBPF node agent's detection accuracy across a broad spectrum of obfuscation techniques within DNS payloads—capabilities that are otherwise infeasible to implement directly within kernel space due to eBPF verifier constraints. The model employs a sequential dense neural network architecture built using TensorFlow over the dataset generated and labelled as explained before, taking 8 lexical input features that aid detection in identifying exfiltrated DNS obfuscation patterns. The input during model training is handled using TensorFlow shufflers with a configured batch size. The pipeline shuffles the data and prefetches data using tensorflow autotune policies for efficient prefetching of data batches minimize I/O for large dataset of around 6 millions rows. Moreover, the model relies on distributed mirrored strategy of tensorflow for efficient

parallel usecase of GPU resources. It consists of three hidden dense layers, each with 16 neurons, and a final output layer with a single neuron for binary classification. ReLU (Rectified Linear Unit) is used as the activation function between layers, while the output layer uses sigmoid activation due to the binary nature of the final output layer. The model is compiled using the Adam optimizer with fine tuned finalized learning rate of 0.001 ensure stable and efficient convergence during training. The binary cross-entropy loss function is used specifically due to binary classification requirement for detecting a specific feature extracted from DNS payload as bengin or malicious. Total of 25 epochs were selected to optimize model weights ensuring enough compared to dataset size to prevent the model from overfitting. Once trained, the model is exported to the ONNX (Open Neural Network Exchange) graph format for use in live inference on kernel-redirected or clone-redirected suspicious traffic. To minimize memory footprint and improve inference speed, the model is integrated into the eBPF userspace agent as a submodule via the ONNX Runtime. Communication between the ONNX-loaded submodule and the core eBPF agent is handled through Unix domain sockets (IPC). Although this design introduces some inference latency, it is mitigated by a first-line caching layer that includes an LRU-based blacklist and a top-level SLD domain cache within the core eBPF agent process, improving efficiency for repeated inferences on identical inputs. This architecture was chosen due to the limited maturity of ONNX ecosystem support in Go. The ONNX-based inference submodule of the eBPF node agent also supports optional hardware acceleration on CPU or GPU when available at the endpoint. To further optimize runtime efficiency, the model is loaded after undergoing dynamic quantization using ONNX's quantizer, which internally applies per-neuron dynamic quantizers and casters to convert float32 vectors into lower-precision formats. This significantly reduces memory usage and computational overhead. Compared to formats such as HDF5 or Pickle, ONNX provides a lightweight, runtime-efficient graph representation, enabling fast, low-overhead inferencing and ensuring the agent maintains a minimal memory footprint even under peak load conditions. Model architecture illustrates the deep learning quantized model architecture as internally represented in ONXX graph.

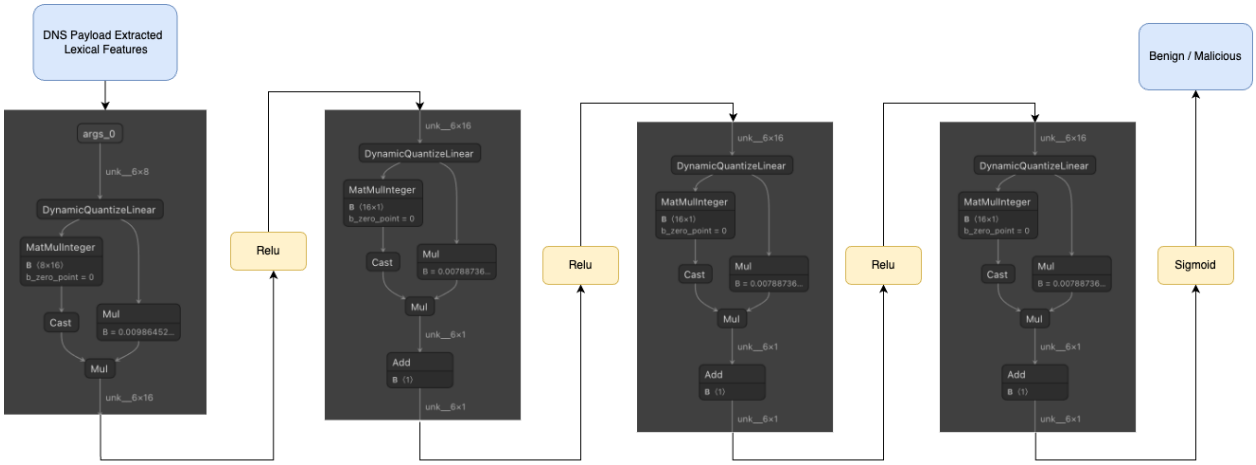


Figure 4.7: DNS Data obfuscation detection Deep Learning Model Architecture

Kafka Topic Name	**Description**
`exfil-sec`	Kafka topic used by the node 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 Node Agent

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 node agent in userspace.
`Memory Usage`	RAM usage in MB or percentage of total memory used by the eBPF node 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 4.6: eBPF Node Agent exported metrics in both active and passive modes

4.2.9 Thread Events Streaming and Metrics Exporters

The eBPF node agent in the data plane operates in both active and passive modes. When a DNS packet is flagged as malicious 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 are configured to send data to a remote Kafka broker. Each eBPF node agent also includes Kafka consumers. Every 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. This improves prevention accuracy across the cluster. Additionally, consumed events allow eBPF node agents to apply dynamic Layer 3 filters in the kernel, supporting cross-protocol correlation. While threat events focus on detected exfiltration attempts, the 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. For details, see eBPF node metrics.

4.3 Control Plane

As illustrated earlier in the control plane component overview, the controller design is entirely stateless, relying on the gpsql backend used by PowerDNS for state management. This backend is specifically used for dynamic domain blacklisting and maintaining the RPZ zone in PowerDNS Recursor to block domains based on malicious events received from the data plane. After consuming threat events from the Kafka topic (exfil-sec), the controller writes to the exfil-sec-infer-controller topic to enable asynchronous communication between data plane nodes, ensuring loose coupling. The writes to this topic are performed by controller nodes, while a single consumer group processes the events produced in the data plane. Additionally, the controller exposes a RESTful server that provides API access to share the current blacklist with all nodes across the data plane.

4.4 Distributed Infrastructure

As explained earlier in the security framework overview, the distributed infrastructure primarily consists of a PowerDNS authoritative DNS server, a PowerDNS recursor, a generic GPSQL backend (PostgreSQL), Kafka brokers, and Prometheus metric scrapers. These scrapers collect metrics exposed by the eBPF node agents deployed in the data plane. The eBPF agents do not handle malicious DNS exfiltration over TCP due to the inherent complexity of the TCP state machine within the kernel, once packets have passed through Netfilter in both ingress and egress paths of the TCP/IP network stack. To address this gap, TCP-based

DNS exfiltration attempts are intercepted in userspace by PowerDNS recursor query interceptors, which function similarly to HTTP middleware. As the PowerDNS recursor currently only supports Lua-based interceptors, a custom Lua interceptor was developed to process DNS queries received over TCP from data plane nodes. This interceptor extracts features from the query and performs inference using a serialized ONNX-based deep learning model. Leveraging Lua's lightweight and high-performance characteristics, the recursor also accesses the GPSQL backend, which contains an additional table maintained by the controller. As previously discussed, this table serves as an RPZ (Response Policy Zone), enabling efficient `NXDOMAIN` responses for queries involving known malicious domains. Before triggering inference, the Lua interceptor checks whether the domain is already blacklisted, allowing it to skip inference for improved throughput. To further enhance performance, the interceptor uses PowerDNS internal Domainsets—a fast, in-memory trie-based data structure—for caching malicious domains detected over TCP. This enables rapid filtering of repeated exfiltration attempts. Additionally, the Lua interceptor periodically synchronizes with the RPZ, rehydrating the local Domainset cache to ensure up-to-date enforcement without incurring inference overhead.

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 CSSVLAB01–08 nodes running over Ubuntu 24.02 on Linux kernel 6.12 on Intel(R) Xeon(R) Gold 6130 (x86_64) architecture, each with 8 GB of memory and 24 GB of storage. 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 node 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 CSSVLAB architecture.

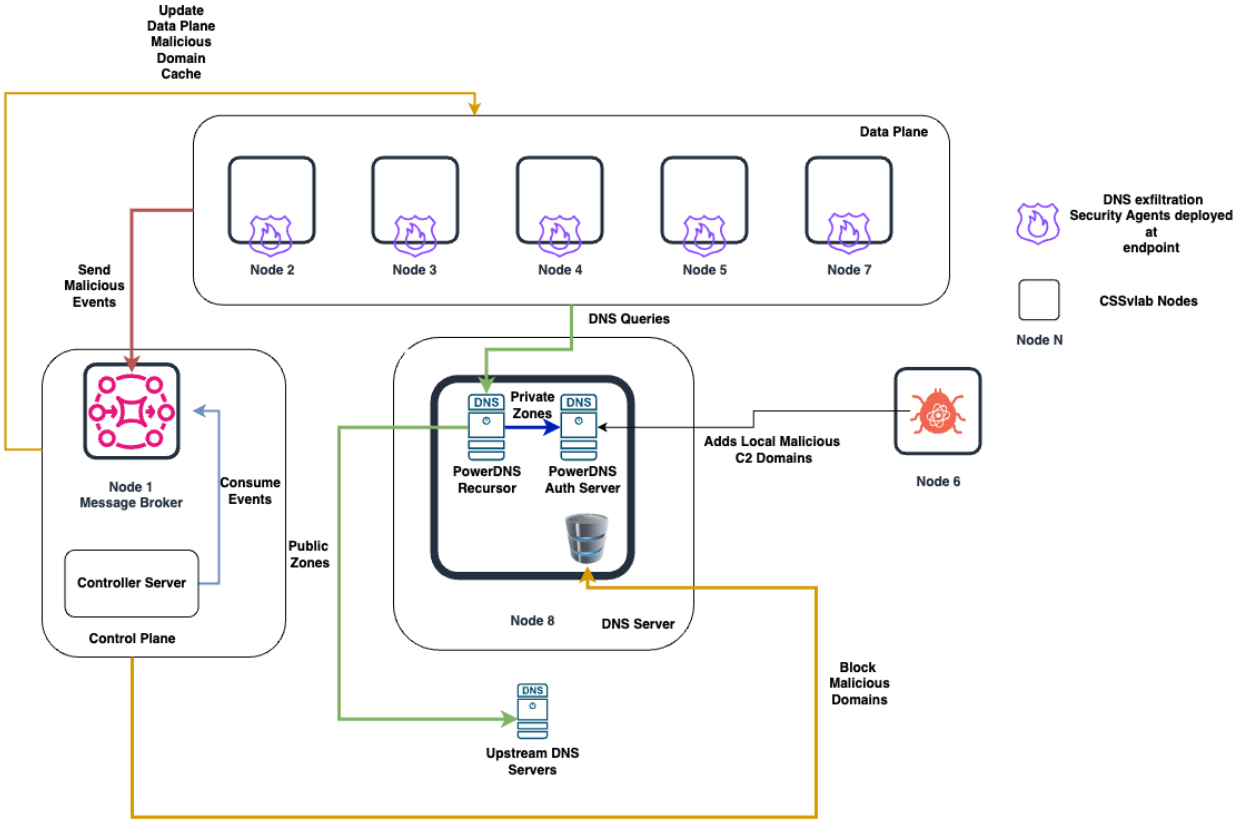


Figure 5.1: Security Framework Deployed Architecture over CSSVLAB Nodes

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 node agent at the data plane endpoint is evaluated through quantized deep learning metrics, DNS request throughput comparison across both operational phases (active and passive), and measurement of bandwidth and resource utilization, including CPU and memory usage, as well as kernel event processing throughput. Finally, the agent's resilience against advanced adversary emulation frameworks is demonstrated us-

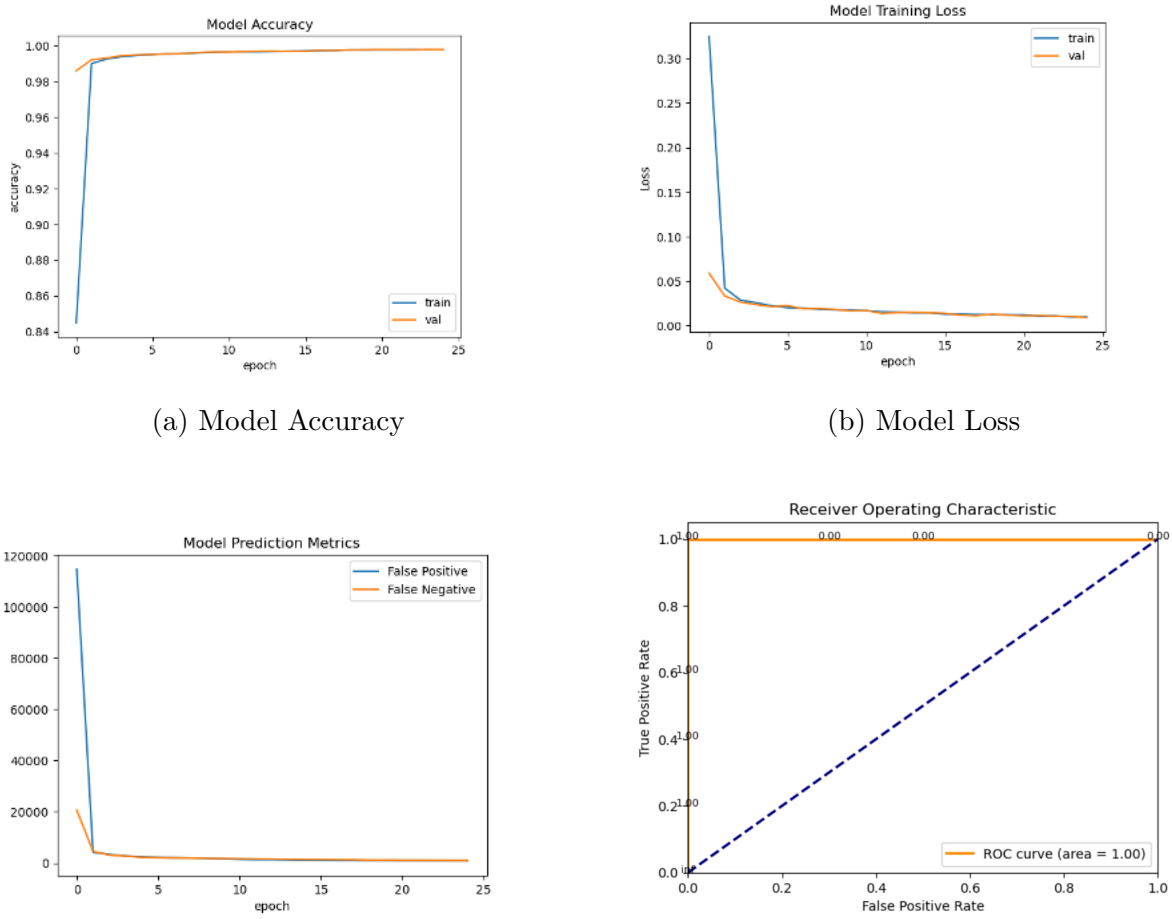
ing detailed dashboard metrics exported by the eBPF node agent. Performance evaluation is conducted on a single selected node within the data plane running the agent.

Deep Learning Model Evaluation

The evaluation of the trained deep learning model was conducted on a dataset of 6 million domains, split into 70% for training and 15% each 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% by the end of training. Given the DNS data exfiltration use case, model performance was primarily evaluated with an emphasis on minimizing false positives. High false positives would not only result in the eBPF node agent dropping benign DNS packets and generating false threat events but could also terminate legitimate processes introducing operational risks. In contrast, false negatives were deemed less critical, as the agent would allow limited malicious traffic through without taking privilege-level actions. Therefore, precision (TP / [TP + FP]) was prioritized over recall (TP / [TP + FN]). Based on these considerations and a dataset engineered to include a wide range of data obfuscation techniques, the model achieved a high precision of 99.79% and recall of 99.76%. For runtime inference via ONNX within the eBPF agent, a decision threshold of 0.98 was selected, as the model demonstrated near-perfect classification (AUC 1). This performance was largely due to the selected feature set, including Shannon entropy over various encoding and encryption schemes, which enabled effective learning. Model metrics and Figure: Model Performance illustrate detailed evaluation results across 25 training epochs.

Table 5.1: Model Evaluation Metrics

Metric	Training	Validation
Accuracy	0.9979	0.9978
AUC	0.9997	0.9997
Loss	0.0094	0.0089
Precision	0.9979	0.9979
Recall	0.9979	0.9976



(a) Model Accuracy

(b) Model Loss

(c) Model Precision

(d) ROC Curve

Figure 5.2: Model performance metrics: accuracy, loss, precision, and ROC curve

eBPF Node 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 via 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 for packet loss. This benchmark was executed on CSSVLAB machines with an upstream PowerDNS recursor server assumed to be healthy, running on a Microsoft Hyper-V hypervisor. Due to the use of SR-IOV virtual NICs, the network stack lacked discrete RX/TX queues, and AF_XDP sockets were unsupported on the egress path—forcing reliance on AF_PACKET. In the cache-hit scenario (100% benign SLD matches), throughput ranged from 8,100 to 9,820 DNS requests/sec with zero packet loss. Latency ranged from near 0 ms to a maximum of 250 ms per 10k sample, validating the lightweight nature of the kernel+userspace eBPF agent pipeline. However, in the cache-miss path requiring live inference, throughput dropped significantly—minimum of 520 and maximum of 5,200 requests/sec, with peak latencies up to 750 ms. Despite this, no packet loss was observed, ensuring reliability. The latency spike is attributed to UNIX domain socket-based communication overhead with the ONNX inference server and Python's GIL (Global Interpreter Lock), which bottlenecks concurrency—unlike the Go-based core agent that remains highly concurrent and compiled for 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, the agent successfully processed a maximum throughput of 67.3 million requests per hour with no packet loss, all while performing deep parsing across both kernel and userspace.

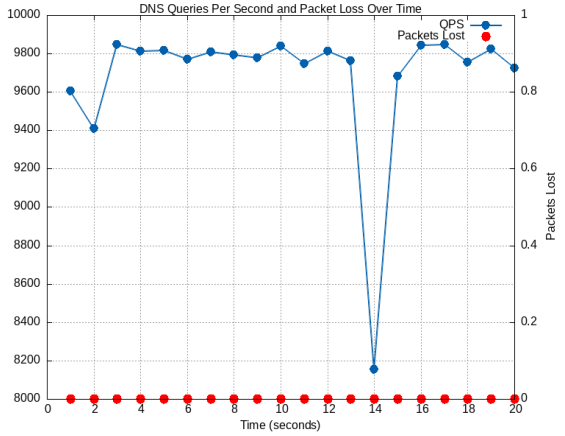


Figure 5.3: eBPF Agent: DNS Throughput for GSLD LRU Hit (10k req/s)

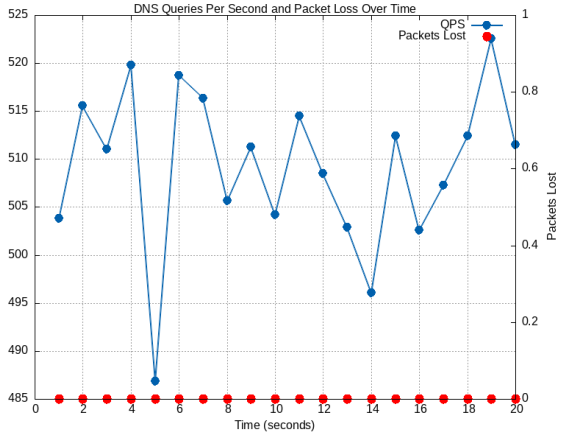


Figure 5.4: eBPF Agent: DNS QPS, GSLD LRU Miss, ONNX (10k req/s)

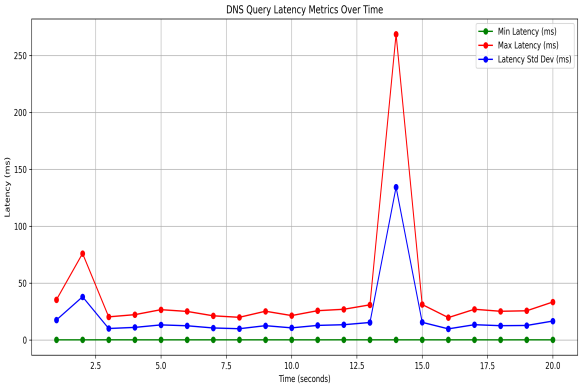


Figure 5.5: eBPF Agent: DNS Latency for GSLD LRU Hit (10k req/s)

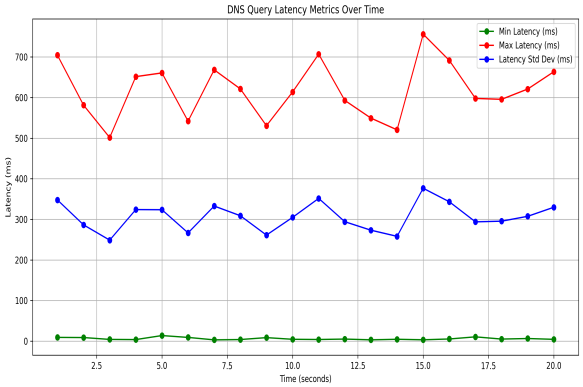


Figure 5.6: eBPF Agent: DNS Latency, GSLD LRU Miss, ONNX (10k req/s)

eBPF Node 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 userspace agent to kill the responsible process. Traditional throughput and la-

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

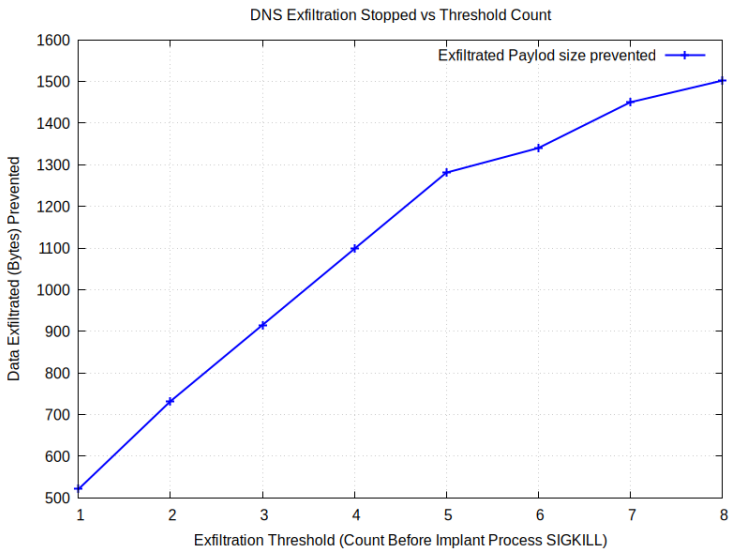


Figure 5.7: eBPF Agent preventing passive DNS exfiltration across varying thresholds prior SIGKILL

eBPF Node Agent Resource Usage

The performance of the eBPF node agent was closely monitored while running at the endpoint in the data plane, with resource usage measured in terms of memory and CPU utilization. During a 10-second DNSPerf benchmark at 10,000 DNS requests per second, 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 are in-

memory and tied to the process heap for ultra-fast lookups. At a higher throughput of 100,000 DNS requests per second, memory usage remained nearly the same, peaking at 350 MB. Throughout the benchmark, CPU usage by the agent process remained under 2%, demonstrating that the agent is extremely lightweight and performant, with no impact on other processes running at the endpoint.

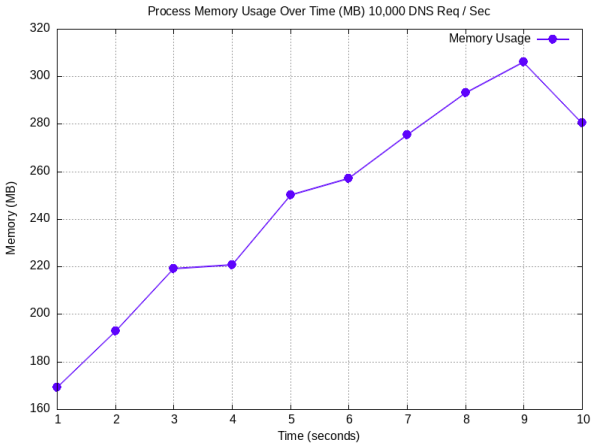


Figure 5.8: eBPF Node Agent Process Memory Usage for 10k DNS req/sec

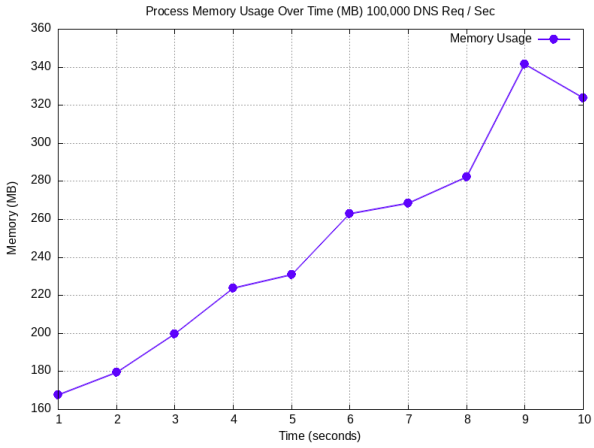


Figure 5.9: eBPF Agent Process Memory Usage for 100k DNS req/sec

Control Plane

The evaluation of the controller's stateless server focuses on its effectiveness in accurately consuming threat events streamed 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.The controller consumed threat events figure 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. The controller produced events figure shows how the controller republishes structured threat data back to the Kafka topic for data plane nodes to consume. As previously explained, 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 5.10: Controller consumed threat events from data plane nodes

```
{
  "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 5.11: Controller streamed threat event rehydrate data plane agents malicious cache

Distributed Infrastructure

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

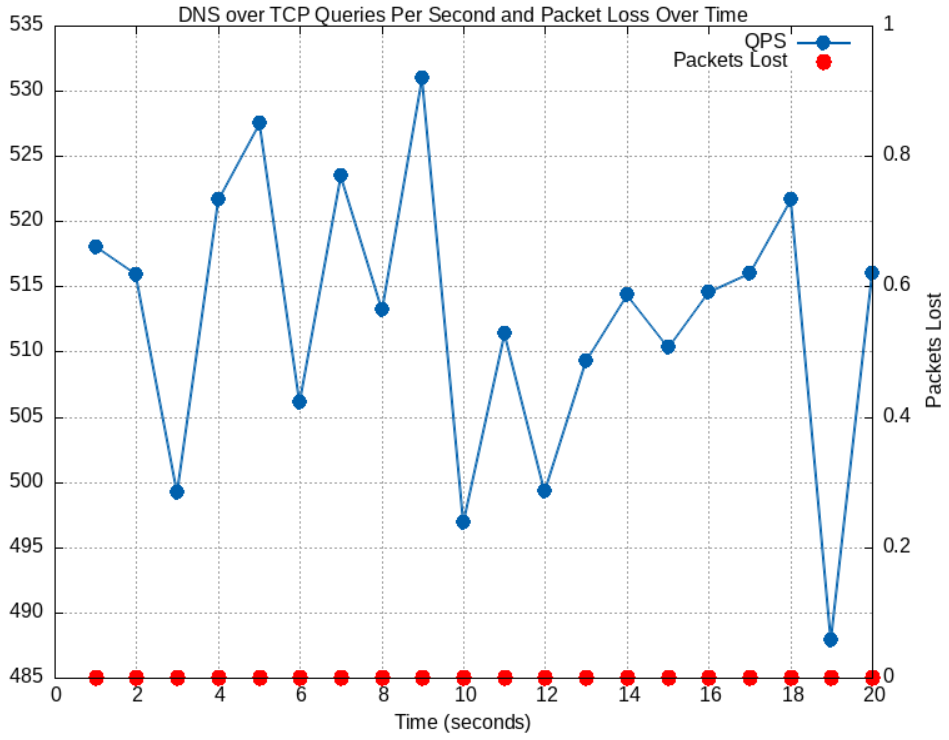


Figure 5.12: DNS Server Throughput for 10k DNS req/s over TCP

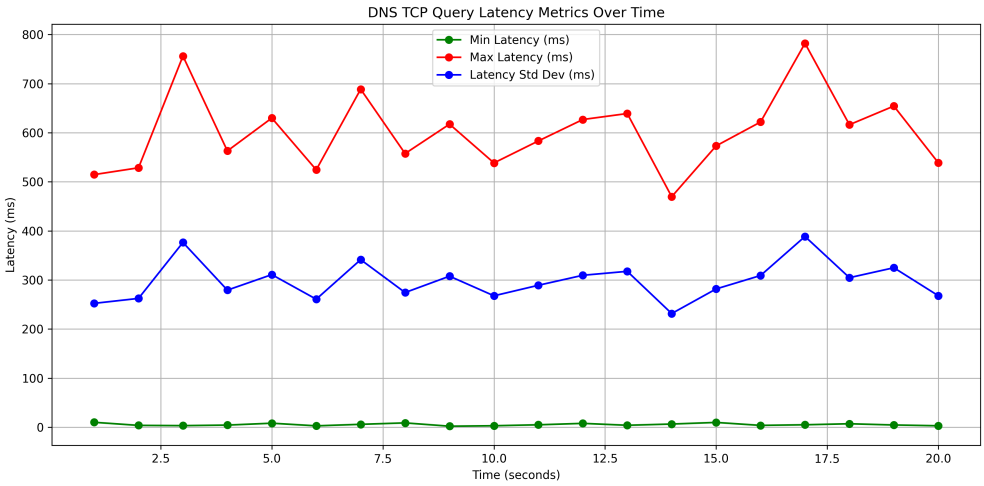


Figure 5.13: DNS Server Latency for 10k DNS req/s over TCP

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 5.14: Blacklisted domains in RPZ zone in DNS server

Chapter 6

CONCLUSION

This chapter concludes the whole paper by summarizing the contents and give future directions.

6.1 Summary

This security framework significantly advances the state of the art by developing a novel architecture for preventing data exfiltration over DNS, directly addressing critical gaps left by traditional approaches. Existing literature and solutions for DNS exfiltration prevention remain largely stagnant, centered around centralized detection and userspace anomaly detection systems—which are inherently inadequate and lack the strength to stop sophisticated DNS-based exfiltration, especially those leveraging C2 implants or APT malware once the system is compromised. 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 in tandem with userspace agents.With security code running inside the kernel, these agents at the endpoint possess unprecedented defensive strength stopping and killing even the most advanced forms of DNS C2 implants, including remote code execution, shellcode exploits, port forwarding, reverse tunnels, and SOCKS tunnels in real-time often used by most reputable adversary emulation tools actively used in production by red teamers. Furthermore, by combining a layered, system-security-first approach with an endpoint-centric design, it significantly enhances endpoint security in ways that were never possible when relying solely on userspace. The result is a deeply integrated, system-level defense that delivers unprecedented visibility into OS activity, enables intelligent threat hunting, and robustly thwarts advanced exfiltration techniques in real time.

1. **Instant DNS C2 Blocking** – Immediately halts DNS-based C2 attempts upon initiation of remote communication.
2. **C2 Implant Detection & Termination** – Identifies and stops processes exploiting DNS for covert exfiltration or control.
3. **DNS Tunnel Elimination** – Prevents tunneling of arbitrary protocols via DNS, eliminating covert exfiltration paths.
4. **Protocol-Agnostic Tunnel Disruption** – Neutralizes exfiltration over any protocol tunneled through DNS on any UDP port.
5. **Elimination of Port-Fowarded, Reverse tnnels** - Neutralizes port forwarded tunnels from comrpomised node to attacker nodes, reverse forwarded tunnels from attacker to compromised nodes, socks5 tunnels, remote shell execution.
6. **Encapsulation-Aware Kernel Defense** – Inspects and blocks threats within kernel interfaces such as VLAN and Tun/Tap.
7. **DGA Mitigation & Deep Observability** – Dynamically blacklists DGA domains and integrates with Prometheus/Grafana for real-time monitoring across large infrastructures.
8. **XDR/EDR Integration** – Natively supports metric export to XDR/EDR platforms for unified threat visibility and response.
9. **Cloud-Scale Horizontal Scalability** – Engineered for production-scale environments with distributed, horizontally scalable design.
10. **System-Level Observability** – Provides fine-grained OS and endpoint visibility, pinpointing processes responsible for DNS exfiltration attempts.

6.2 Limitation and Future Work

Limitations

- **Partial DNS Parsing:** The DNS parser implemented in-kernel currently covers only key sections of the protocol. Further parsing of DNS message structure (e.g., full RR

parsing, EDNS, DNSSEC) is pending.

- **Absence of Encrypted Exfiltration Detection:** The framework does not yet support prevention of exfiltration over 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, IPSsec.
- **Basic Throughput Control:** Egress rate limiting is not yet adaptive to volume-based exfiltration detection at high scale.

Future Work

- **Extend Support for DNS-over-TCP and Encrypted Tunnels:** Implement detection and blocking for exfiltration over TCP-based DNS, DoT (DNS-over-TLS), and encrypted channels.
- **Add In-Kernel TLS Fingerprinting:** Integrate TLS fingerprinting (e.g., JA3/JA4) using eBPF to detect encrypted DNS exfiltration or covert TLS channels over wireguard / mTLS.
- **Enhance DNS Protocol Parsing:** Expand in-kernel DNS parser to cover full protocol depth, including additional sections and response types.
- **XDP-Based Flood Prevention:** Introduce XDP ingress filtering inside kernel to mitigate NXDOMAIN-based DNS water torture and DNS amplification attacks on the endpoint.
- **Rate-Limiting Based on Volume and Throughput:** Integrate egress `TC CLSACT QDISC`-based dynamic rate limiting via token bucket algorithm implementation through kernel BPF timers to prevent high-throughput DNS data breaches completely inside kernel.
- **Layered Cloud and Kubernetes Defense:** Deploy policy enforcement layers across Kubernetes orchestrated environments (L3/L7 filtering via CNI to drop in userspace)

and public cloud cross protocol access control list defending all nodes behind firewall.

Chapter 7

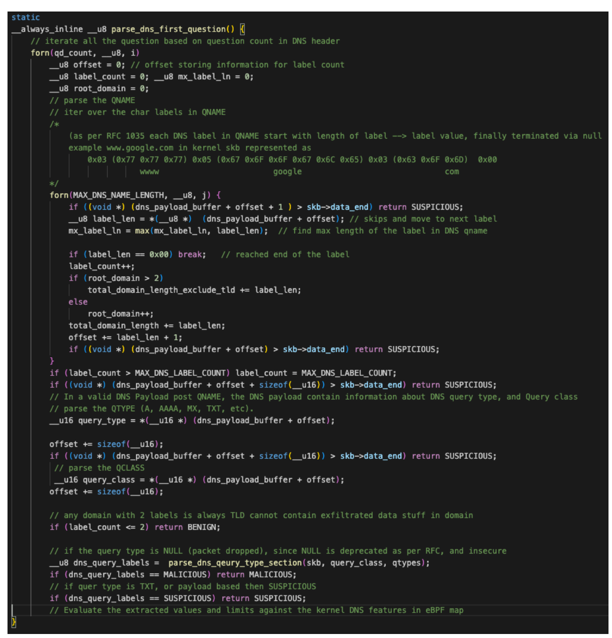
APPENDICES

```
// DNS header struct
struct dns_header {
    __u16 transaction_id; //queryID / transaction ID
    __u16 flags;    //DNS header flags

    __be16 qd_count;   //Number of questions
    __be16 ans_count;  //Number of answer RRs
    __be16 auth_count; //Number of authority RRs
    __be16 add_count;  //Number of resource RRs
} __attribute__((packed));

// DNS flags struct
static
__always_inline struct dns_flags
get_dns_flags(struct dns_header * dns_header) {
    struct dns_flags flags;
    __u16 host_order_flags = bpf_ntohs(dns_header->flags);
    flags = (struct dns_flags) {
        .qr = (host_order_flags & DNS_QR_MASK) >> DNS_QR_SHIFT,
        .opcode = (host_order_flags & DNS_OPCODE_MASK) >> DNS_OPCODE_SHIFT,
        .aa = (host_order_flags & DNS_AA_MASK) >> DNS_AA_SHIFT,
        .tc = (host_order_flags & DNS_TC_MASK) >> DNS_TC_SHIFT,
        .rd = (host_order_flags & DNS_RD_MASK) >> DNS_RD_SHIFT,
        .ra = (host_order_flags & DNS_RA_MASK) >> DNS_RA_SHIFT,
        .z = (host_order_flags & DNS_Z_MASK) >> DNS_Z_SHIFT,
        .ad = (host_order_flags & DNS_AD_MASK) >> DNS_AD_SHIFT,
        .cd = (host_order_flags & DNS_CD_MASK) >> DNS_CD_SHIFT,
        .rcode = (host_order_flags & DNS_RCODE_MASK) >> DNS_RCODE_SHIFT
    };
    return flags;
}
```

Figure 7.1: Caption for first image



```
static
__always_inline __u8 parse_dns_first_question() {
    // iterate all the question based on question count in DNS header
    forn(qd_count, __u8, i)
        __u8 offset = 0; // offset storing information for label count
        __u8 label_count = 0; __u8 mx_label_ln = 0;
        __u8 root_domain = 0;
        // parse the QNAME
        // iter over the char labels in QNAME
        /*
            (as per RFC 1035 each DNS label in QNAME start with length of label --> label value, finally terminated via null
            example www.google.com in kernel skb represented as
                0x03 (0x77 0x77 0x77) 0x05 (0x67 0x6F 0x6F 0x67 0x6C 0x65) 0x03 (0x63 0x6F 0x6D)  0x00
                           wwww                         google                              com
        */
        forn(MAX_DNS_NAME_LENGTH, __u8, j) {
            if ((void *) (dns_payload_buffer + offset + 1 ) > skb->data_end) return SUSPICIOUS;
            __u8 label_len = *(__u8 *)  (dns_payload_buffer + offset); // skips and move to next label
            mx_label_ln = max(mx_label_ln, label_len);  // find max length of the label in DNS qname

            if (label_len == 0x00) break;   // reached end of the label
            label_count++;
            if (root_domain > 2)
                total_domain_length_exclude_tld += label_len;
            else
                root_domain++;
            total_domain_length += label_len;
            offset += label_len + 1;
            if ((void *) (dns_payload_buffer + offset) > skb->data_end) return SUSPICIOUS;
        }
        if (label_count > MAX_DNS_LABEL_COUNT) label_count = MAX_DNS_LABEL_COUNT;
        if ((void *) (dns_payload_buffer + offset + sizeof(__u16)) > skb->data_end) return SUSPICIOUS;
        // In a valid DNS Payload post QNAME, the DNS payload contain information about DNS query type, and Query class
        // parse the QTYPE (A, AAAA, MX, TXT, etc).
        __u16 query_type = *(__u16 *) (dns_payload_buffer + offset);

        offset += sizeof(__u16);
        if ((void *) (dns_payload_buffer + offset + sizeof(__u16)) > skb->data_end) return SUSPICIOUS;
         // parse the QCLASS
         __u16 query_class = *(__u16 *) (dns_payload_buffer + offset);
        offset += sizeof(__u16);

        // any domain with 2 labels is always TLD cannot contain exfiltrated data stuff in domain
        if (label_count <= 2) return BENIGN;

        // if the query type is NULL (packet dropped), since NULL is deprecated as per RFC, and insecure
        __u8 dns_query_labels =  parse_dns_qeury_type_section(skb, query_class, qtypes);
        if (dns_query_labels == MALICIOUS) return MALICIOUS;
        // if quer type is TXT, or payload based then SUSPICIOUS
        if (dns_query_labels == SUSPICIOUS) return SUSPICIOUS;
        // Evaluate the extracted values and limits against the kernel DNS features in eBPF map
}
```

Figure 7.2: Caption for second image

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

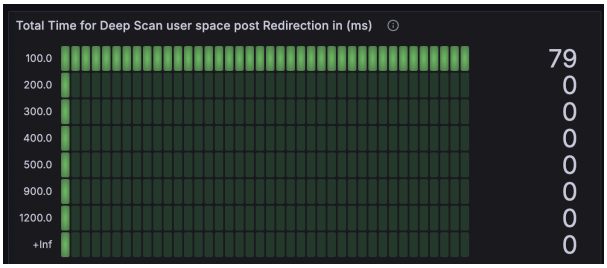
Exfiltration_Attempt_Started_At	Process_Id	Process Alive Time (Sec)
Sunday, 04-May-25 00:49:41 UTC	1630660	6
Sunday, 04-May-25 00:49:41 UTC	1630661	6
Sunday, 04-May-25 00:49:41 UTC	1630662	6
Sunday, 04-May-25 00:49:41 UTC	1630663	6
Sunday, 04-May-25 00:49:41 UTC	1630664	6
Sunday, 04-May-25 00:49:41 UTC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

(b) Process Alive Time Before Termination

DNS Data Breaches Stop Over Tunnel Interfaces (Tun/Tap)

process_id	prog_name	threat_group_id
1779316	iodine	1779316
1779834	iodine	1779834
1781930	iodine	1781930

(c) Tunnel Interface Exfiltration Metric



(d) Latency in Active Redirect Mode

Malicious Detected Domains for DNS Breaches

ExfilPort	Fqdn	IsEgress	PhysicalNodeIpv4	Protocol	RecordType
143	)2cccеac690ab05a7feff1d3dae01.	true	10.158.82.19	DNS	CNAME
143	)2cccеac690ab05a7feff1d3dae01.	true	10.158.82.19	DNS	CNAME
143	)2cccеac690ab05a7feff1d3dae01.	true	10.158.82.19	DNS	CNAME
143	)2cccеac690ab05a7feff1d3dae01.	true	10.158.82.19	DNS	CNAME
143	)2cccеac690ab05a7feff1d3dae01.	true	10.158.82.19	DNS	CNAME
143	)e7da076c2521deb66db124e68dc	true	10.158.82.19	DNS	TXT
143	)e7da076c2521deb66db124e68dc	true	10.158.82.19	DNS	TXT
143	)e7da076c2521deb66db124e68dc	true	10.158.82.19	DNS	TXT
143	)e7da076c2521deb66db124e68dc	true	10.158.82.19	DNS	TXT
143	)e7da076c2521deb66db124e68dc	true	10.158.82.19	DNS	TXT
53	50f8964b44e8b8742d9c06f5baa	true	10.158.82.19	DNS	CNAME
53	50f8964b44e8b8742d9c06f5baa	true	10.158.82.19	DNS	CNAME
53	50f8964b44e8b8742d9c06f5baa	true	10.158.82.19	DNS	CNAME

(e) Detailed Metrics of Exfiltrated DNS Packets

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] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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`.

[11] 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.

[12] 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`.

[13] 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.

[14] 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`.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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`.

[21] 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.

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