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

Vedang Parasnis

Submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2025

Project Committee:

Dr. Geetha Thamilarasu, Chair

Dr. Munehiro Fukuda, Committee Member

Dr. Robert Dimpsey, Committee Member

Program Authorized to Offer Degree:

Computing & Software 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

DNS-based data exfiltration remains a critical blind spot in modern infrastructure, especially for hyperscalers operating AI workloads and handling sensitive data across distributed Linux environments. In 2024, the average cost of a data breach exceeded $4.8 million, with DNS emerging as the main exploitation channel. This project develops the first scalable, kernel-enforced DNS exfiltration prevention framework capable of detecting and disrupting advanced Command-and-Control channels in real time. It integrates high-performance eBPF programs for deep packet inspection directly in the Linux kernel, combined with a userspace neural network for low-latency lexical analysis of advanced data obfuscation in DNS. Malicious queries trigger immediate process termination from within the kernel, cutting off exfiltration at the source before data loss or lateral movement occurs. The framework introduces rich system level telemetry that stream process-level observability and domain-based threat intelligence to Kafka, enabling live policy enforcement and domain blacklisting across nodes without relying on external firewalls or proxies. Experimental results show detection and response speed as low as 326 μs even against stealthy, obfuscated payloads generated by industry-grade adversary emulation tools. This significantly reduces attacker dwell time and provides security teams with actionable visibility into DNS-based threats at the endpoint level, at hyperscale.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation and Goals

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

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

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

Chapter 2

BACKGROUND

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

2.1 eBPF

The extended Berkeley Packet Filter (eBPF), introduced in Linux kernel 3.15 (2014), is a general-purpose virtual machine in the kernel evolved from the classic BPF [1]. Unlike kernel modules, which risk destabilizing the system, eBPF safely injects verified code into the kernel, enabling dynamic programmability without compromising stability or security. eBPF surpasses other programmable data path technologies such as P4 [2] and DPDK [3], which operate outside the kernel or lack visibility into kernel-level security subsystems. These models cannot access core primitives such as process identity or Linux's Mandatory Access Control (MAC) layers, limiting their ability to enforce deep, context-sensitive security policies. eBPF, on the contrary, operates directly within the kernel network stack and security layers, allowing high-resolution enforcement and real-time analysis of malicious traffic. eBPF programs are written in a restricted C subset, compiled via LLVM to a platform-independent bytecode, and executed by a RISC-like in-kernel VM. The execution model enforces strict safety: a 512-byte stack, bounded loops, 11 64-bit registers, and a cap of one million instructions. Before loading, the BPF verifier ensures control flow integrity and memory safety. Programs are Just-In-Time (JIT) compiled for performance and executed within a sandboxed environment. A core strength of eBPF is its use of BPF maps, persistent, kernel-resident key-value

stores that support data structures such as LRU caches, stacks, and queues for efficient state management and data sharing. In addition, eBPF maps support pinning to the BPF filesystem, allowing data to persist beyond the lifecycle of the userspace program that loaded the eBPF program in kernel. All interactions are mediated through the `bpf()` syscall, guarded by `CAP_BPF` or `CAP_SYS_ADMIN` for complex programs which are reserved for privilege users.

2.2 eBPF Integration with the Linux Kernel Network Stack

The Linux kernel network stack processes packets through layered ingress and egress paths using socket kernel buffers (SKBs), enabling efficient parsing, filtering, and forwarding across RX/TX queues, with eBPF hooks providing runtime programmability at critical stages such as Netfilter and Traffic Control. Within this stack, the Traffic Control (TC) subsystem plays a crucial role in enforcing egress security, extending beyond its traditional responsibilities like flow control and quality of service (QoS). TC provides fine-grained traffic management through shaping, scheduling, classification, policing, and packet dropping. These functions are implemented via queueing disciplines (QDISCs), which determine how packets are prioritized and transmitted through the network driver's transmission queues [4]. Among these QDISCs, CLSACT (classless QDISC with actions) is particularly valuable for advanced security enforcement. It enables classification and action hooks on both ingress and egress paths without disrupting existing classful (e.g., HTB) or classless (e.g., FQ_CODEL, PRIO_FAST) traffic configurations. This backward compatibility makes CLSACT suitable for production environments, allowing seamless integration of programmable in-kernel logic. eBPF programs attached to CLSACT filters can chain classification and actions with configurable priorities, enabling deterministic and layered packet filtering directly within the kernel [5]. Since CLSACT operates before any default QDISC, it is ideal for embedding additional security logic without affecting existing traffic control behavior. This approach preserves QoS guarantees while enabling real-time, low-latency filtering. Although CLSACT is commonly used by Container Network Interface (CNI) plugins in Kubernetes - for overlay routing, IP

masquerading, and node-to-node communication - its full potential to enforce in-kernel security policies remains largely untapped. Beyond TC, eBPF programs can attach across multiple layers of the kernel network stack. These include the high-speed ingress path via XDP, the link layer via Netfilter, socket-layer hooks (e.g., sockops, sk_msg), and even syscall interfaces and kernel security modules. This broad hook coverage enables eBPF to support profiling, observability, rate limiting, deep packet inspection (DPI), and threat detection with minimal overhead and maximum control.

2.3 DNS-based Data Exfiltration

DNS-based data exfiltration is a covert technique employed by adversaries to extract sensitive information from compromised systems using the Domain Name System (DNS) protocol. Commonly leveraged by memory-resident implants, this method reduces forensic artifacts by avoiding disk writes and capitalizing on DNS's widespread use and typically permissive filtering policies. Attackers encode exfiltrated payloads into the subdomain portion of outbound DNS queries, which are then transmitted to attacker-controlled domains or delegated nameservers via standard recursive DNS resolution—allowing the traffic to blend in with legitimate network activity. As shown in Table 2.1 exfiltrated data is obfuscated using base encoding, compression, and segmentation. Common DNS record types like `A`, `AAAA`, `MX`, and `HTTPS` are used for compatibility, while `TXT` and `NULL` offer flexible payload capacity making them ideal for C2 responses. To bypass inspection, adversaries use DGA, randomized query timing, ephemeral encryption, and even tunnel DNS over arbitrary transport ports using tools like DNSCat2 encapsulating traffic in ways that evade firewall rules.

DNS Tunneling

DNS tunneling evades perimeter defenses by embedding protocol payloads, usually blocked on firewalls within DNS query fields. These payloads are disguised to resemble legitimate DNS traffic, enabling compromised hosts to communicate covertly with remote servers. This technique not only facilitates data exfiltration but also allows encapsulation of other blocked

Table 2.1: DNS Payload Obfuscation Techniques

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

protocols inside DNS, effectively rendering traditional firewall rules for those protocols ineffective. Advanced variants exploit kernel-level encapsulation mechanisms (e.g., `TUN/TAP`, VXLAN) using privileged virtual interfaces (`CAP_NET_ADMIN`), and dynamically modulate throughput and timing to bypass static blacklists and anomaly detection systems.

DNS Command and Control (C2)

DNS-based C2 is an advanced form of tunneling used to establish persistent full-duplex covert channels between implants and attacker-controlled servers. Using a client-server model, implants poll for encoded commands via queries and exfiltrate execution results in responses - enabling remote control, backdoors, port forwarding, and DNS-based reverse tunnels. To evade detection, attackers vary beacon timing and rotate domains/IPs using DGA. Multi-player C2 frameworks coordinate multiple operators that exploit several implants simultaneously, overwhelming passive defenses. Static blacklists and rules are ineffective against such adaptive threats. Real-time in-kernel termination of both the DNS C2 channel and the implant process is critical, but current solutions are slow and reactive, thereby falling short before command execution or data loss occurs.

DNS Raw Exfiltration

Raw DNS exfiltration transmits sensitive data, such as credentials or files, directly through high-volume bursts of DNS queries. Although noisier than tunneling or C2, it can succeed

before alert thresholds are triggered or policy enforcement takes effect. Since most defenses are reactive or delayed, prevention at the point of transmission is essential to ensure zero data loss.

2.4 DNS Protocol Security Enhancements and Limitations

Several standardized enhancements improve DNS integrity and privacy:

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

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

2.5 Existing Prevention Mechanisms and Limitations

Common DNS exfiltration defenses include:

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

Although effective against known threats, traditional defenses are reactive, relying on static blacklists or passive DPI triggered post-alert. This delay allows DNS exfiltration and the C2 commands to be successful. They also fail against DGA-based implants that rapidly rotate domains, making static rulesets too slow to prevent real-time damage.

Chapter 3

RELATED WORK

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

3.1 Network Security using eBPF

eBPF has emerged as a critical technology for modern networking, security, and observability in Linux. Its ability to inject safe, verifiable code into the kernel makes it ideal for high-performance, in-kernel programmability without compromising system stability. These features have led to widespread adoption by cloud providers, particularly in large-scale data planes and hyperscalers for traffic filtering and enforcement of declarative network policies across multiple layers of the Linux kernel. Most existing research focuses on the ingress path using XDP (eXpress Data Path), a high-speed packet processing mechanism integrated into the network driver, commonly referred to as the high-speed ingress kernel datapath. Initially proposed by Høiland-Jørgensen et al., XDP was later adopted into the Linux kernel to enable early packet drops, hardware offload at the NIC level, and improved throughput. It is often combined with eBPF to support low-latency programmable network processing and security enforcement for DDoS preventions [6, 7]. Vieira et al. studies provide architectural overviews and performance analyses of eBPF in networking contexts [8], similarly Bertrone et al. explains accelerating kernel network firewalls by combining eBPF and iptables [9], 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 [10]. 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 [11]. 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 [12, 13]. 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 [14, 15]. Although DNS-aware L7 policies allow domain-level whitelisting, they lack dynamic blacklisting and are not designed to detect exfiltration behaviors. Open-source tools such as Microsoft's Inspector Gadget also rely on static rules defined in the userspace and do not provide deep kernel-level dynamic enforcement mechanisms. These limitations highlight the need for a comprehensive eBPF-based solution that operates at the egress point and supports dynamic security enforcement not only inside the kernel via eBPF but also in a distributed environment with dynamic domain blacklist to combat DGA.

3.2 Machine Learning for Detecting DNS Data Exfiltration

Advancements in network security have significantly improved DNS exfiltration detection, often using machine learning to analyze anomalies in traffic volume and rate, as well as for lexical analysis of exfiltrated DNS queries. Common solutions combine DPI with anomaly

detection of traffic volume and timing, identifying potential threats using DNS firewalls or intrusion detection systems. For C2-based exfiltration, Zimba and Chishimba employs behavioral analysis that integrates PowerShell activity and backdoors with DNS tunneling for APT detection [16]. Similarly, Das et al. trains ML models on real malware samples from financial institutions, while Ahmed et al. uses Isolation Forest for real-time detection [17, 18]. However, these methods focus on detection, not prevention, and struggle against stealthy, persistent C2 channels.

Bilge et al. and Antonakakis et al. introduced DNS server-side solutions such as EXPOSURE and NOTOS, which rely on passive analysis of large datasets, extracting domain features to flag malicious activity [19, 20]. Although effective in identifying botnet C2 and spamming domains, these approaches lack real-time enforcement capabilities and cannot block payload execution. Similarly, the models proposed by Nadler et al. and Mathas et al. use techniques based on entropy, timing, and anomalies to detect low-throughput DNS tunneling [21, 22]. However, these methods remain inherently reactive, relying on historical traffic patterns and often failing against slow, stealthy exfiltration tactics.

Aurisch et al. propose mitigation efforts involving mobile agents that introduce latency, suffer false positives, and lack scalability [23]. Similarly, Haider et al. propose the C2 Eye framework to detect C2 attacks in supply chains but do not address the damage caused by C2 in distributed environments [24]. Even promising tools like Process DNS, developed by Sivakorn et al., correlate DNS traffic with userspace processes to detect C2 activity [25]. However, these tools remain vulnerable to privilege escalation and evasion due to limited kernel integration and the absence of fine-grained, kernel-enforced mandatory access controls.

Overall, most ML-based DNS security tools are limited by userspace-only architectures. They lack in-kernel inspection, cross-protocol correlation, and visibility into port layer obfuscation, such as DNS over non-standard ports. These tools operate passively, disconnected from real-time enforcement, and do not offer a preemptive response to emerging threats. Although lexical payload analysis may help classify anomalies quickly, behavioral models still lag behind, detecting threats only after execution or data exfiltration has occurred. Hence,

existing researched solutions fundamentally fail to prevent advanced C2 behavior such as remote code execution, port forwarding, or shell access. They rarely address real-world adversary emulation or modern C2 vectors and remain ineffective against multiplayer C2 operations, botnet-based attacks, or DGA. Despite incremental progress, no existing solution offers real-time DNS exfiltration prevention with implant termination, dynamic in-kernel security policy enforcement, negligible or zero data loss, adaptive domain blacklisting, and cloud-native scalability. Userspace-only systems lack the ability to inspect low-level system state—especially near the NIC—and instead rely on passive traffic analysis from centralized locations. This limits their ability to enforce fine-grained controls needed for modern threat defense. Real-time kernel-level defenses are essential to ensure data sovereignty and integrity with the speed and precision-based response to combat emerging threats.

3.3 Enterprise Solutions to Prevent DNS Data Exfiltration

Akamai's ibHH algorithm detects DNS exfiltration in real time by identifying information-heavy hitters—quantifying unique data transmitted from subdomains to their parent domains using a fixed-size cache for efficient tracking [26]. While DNS firewalls like Akamai and AWS Route 53 can flag tunneling and DGA activity through volume thresholds and anomaly rules, they lack direct endpoint enforcement, which is critical for minimizing data loss and dwell time [27]. These systems are ineffective against APT malware that leverages low-and-slow C2 patterns or dynamic domain generation. AWS Route 53 also lacks deep observability and cross-protocol correlation, limiting its ability to enforce Layer 3 blocks through AWS Network Firewall. Likewise, cloudflare, Akamai, and AWS proprietary DNS firewalls prioritize DDoS mitigation, but fail to protect against stealthy exfiltration or insider DNS-based C2 threats. Infoblox adds hybrid agent-based enforcement and centralized threat intelligence, and Broadcom's Carbon Black blocks endpoint processes, but both rely on userspace analysis [28]. In contrast, eBPF provides in-kernel enforcement with fine-grained, real-time visibility—offering a significantly stronger defense against DNS exfiltration.

Chapter 4

IMPLEMENTATION

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

4.1 Security Framework Overview

The implemented security framework uses an endpoint-centric architecture to defend against DNS-based C2 and tunneling attacks in real time. It embeds DNS exfiltration defenses directly into the operating system using eBPF, enabling in-line mitigation and malicious process containment. Network policies are dynamically enforced within the kernel network stack to block C2 communication, while malicious processes are terminated through integration with kernel syscall layer all coordinated by a lightweight userspace eBPF agent. To ensure scalability, the framework streams threat events asynchronously, enabling cross-node security enforcement across all nodes in the data plane. It also supports dynamic domain blacklisting on the DNS server to proactively disrupt DGA-based threats. The following subsections describe the core components of the framework in detail. Appendix B provides additional information on DGA.

4.1.1 Data Plane

The data plane consists of distributed nodes running lightweight eBPF agents, implemented entirely in Golang for high performance, optimal concurrency, and minimal memory footprint. Operating in userspace, each agent dynamically injects eBPF programs into the kernel's TC layer at the egress hook of physical network interfaces, enabling in-kernel DPI to

detect and block exfiltrated DNS traffic over UDP. In addition to TC filters, agents deploy auxiliary eBPF programs at various kernel hook points: kprobes to monitor the creation of new network devices, raw tracepoints to track process termination, and cgroup socket hooks to retrieve internal kernel process structures (`task_struct`), especially on kernels below version 5.2 where TC lacks direct access. Integration with Linux Security Modules (LSM) ensures the integrity of all injected eBPF programs, safeguarding the kernel from tampered malicious eBPF bytecode. Each agent supports two configurable prevention modes, managed via a dedicated eBPF map injected into the kernel. These modes can be toggled at runtime from userspace and are enabled by default to provide comprehensive protection against DNS exfiltration attack vectors. Appendix A provides details of eBPF integration with LSM.

- **Strict Enforcement Active Mode**: DNS packets over standard ports (DNS: 53, mDNS: 5353, and LLMNR: 5355) are scanned in-kernel using eBPF at the TC egress hook. Malicious packets are immediately dropped. If classification exceeds eBPF instruction limits, the packet is redirected to userspace for further inspection. The eBPF agent sniffs redirected traffic and either checks it against a domain blacklist cache or performs inference using deep learning model to detect data obfuscation inside DNS. Benign packets are retransmitted using high-speed socket options such as `AF_PACKET` or `AF_XDP` bypassing kernel network stack, while malicious ones are dropped and their domains added to the blacklist cache in userspace.
- **Passive Malicious Process Threat Hunting Mode**: This mode prevents DNS exfiltration when DNS is layered over non-standard UDP ports. Suspicious DNS-overlaid packets are cloned to userspace for analysis while the original packet is allowed to proceed. If the packet is deemed malicious, the originating process is flagged in the eBPF maps, and all subsequent DNS packets from that process are dropped in the kernel. This effectively disrupts communication between the C2 implants and remote servers. The mode is particularly effective against stealth techniques that rely on port obfuscation.

In addition to performing deep packet inspection, the eBPF agents manage network namespaces and virtual bridges using the Linux virtual ethernet bridge driver. The topology shown in Figure 4.1 combines Linux network namespaces with multiple pairs, acting as Layer 2 and Layer 3 bridges. Agents also maintain file descriptors to their eBPF maps, enabling efficient kernel-userspace communication and advanced runtime state analysis. The lifecycle of each eBPF program is tightly managed by the agent, which operates with elevated privileges to control the kernel network stack, syscalls, and the relevant kernel subsystems. Some of the kernel capabilities required for the agent are detailed in Table 4.1. Both prevention modes, active and passive, support real-time enforcement, including the termination of processes responsible for repeated exfiltration beyond configurable thresholds. On the userspace side, the agent performs low-latency inference using quantized ONNX models, exports telemetry to observability backends, and streams threat events to a centralized message broker for controller-side processing. These agents also manage domain caches to improve performance, relying on a cache read-through policy. All agent components, including eBPF maps in the kernel and userspace caches, are dynamically reprogrammable at runtime via the control plane, enabling flexible and real-time reconfiguration of agents in the data plane. To complement egress filtering, agents also inspect ingress traffic to detect C2 response patterns, leveraging the same inference engine and LRU cache used for egress analysis.

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

Kernel Capability	**Description**
`CAP_BPF`	Load eBPF, manage maps
`CAP_SYS_ADMIN`	Attach BPF, mount BPF FS
`CAP_SYS_PTRACE`	Support tracepoint attachment to kernel tracepoints specifically to kernel process scheduler
`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 4.2: eBPF Agent: Injected eBPF programs inside Linux Kernel

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.
`CGROUP_SKB`	Active, Passive	Sockets cgroups	Get the process info for current UDP packet, update information to pinned map shared with core egress TC program.
`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.

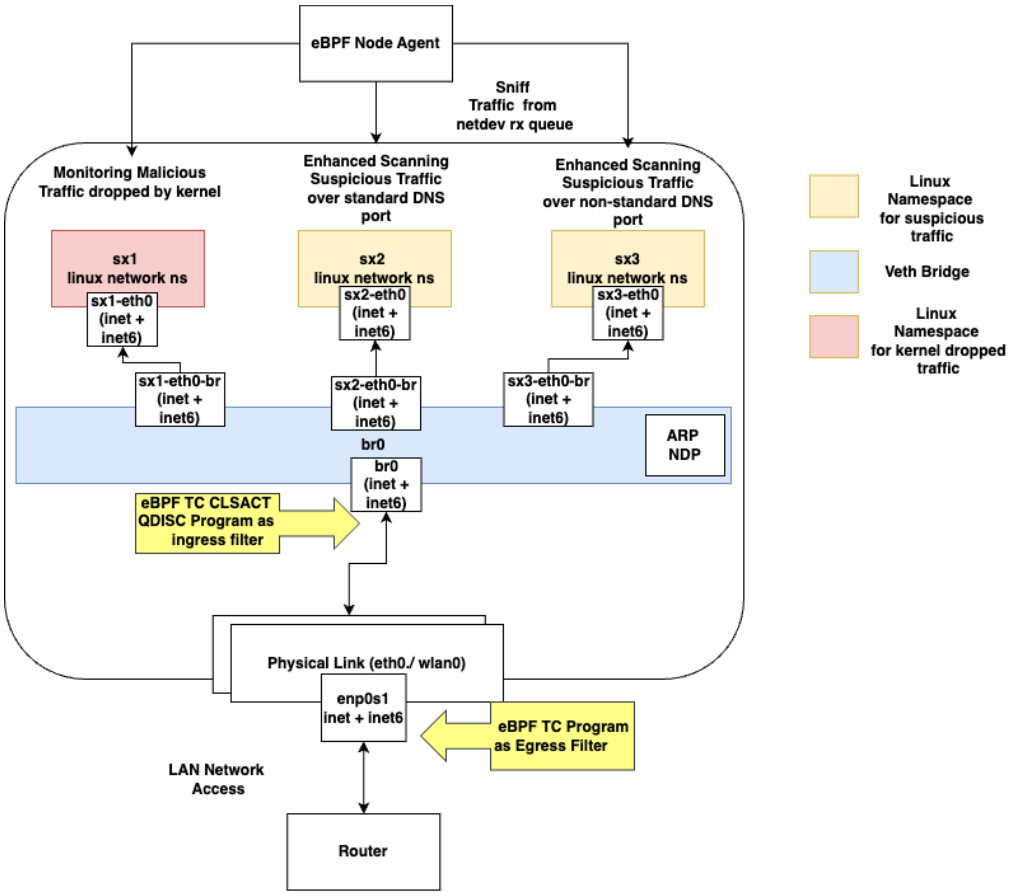


Figure 4.1: eBPF Agent: Network Topology at Endpoint

4.1.2 Distributed Infrastructure

In addition to the data plane nodes, the framework includes an open-source PowerDNS setup consisting of a recursor for upstream resolution and an authoritative server configured with a Postgres backend. Although actual internal domains are not served, the authoritative server is used to carry out DNS C2 and tunneling attacks by generating malicious domains using DGA. This design replicates the behavior of the real-world DNS server commonly deployed by enterprises. All data plane nodes resolve DNS through the PowerDNS Recursor, which is equipped with interceptors that inspect queries before forwarding. These interceptors run ONNX-based deep learning inference to detect threats, specifically on TCP-based DNS traffic offloaded from eBPF agents. Kafka acts as the message broker for streaming threat events. Additionally, the recursor integrates with dynamic RPZ stored in Postgres, allowing the controller to blacklist second-level domains (SLD) linked to malicious activity, as detailed in the next section.

4.1.3 Control Plane

The control plane consists of a centralized analysis server that consumes threat events from Kafka topics, streamed and updated by eBPF agents running in the data plane. Based on these consumed event payloads, the control plane dynamically blacklists malicious SLD 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 agent deployed on each node in the data plane is organized into six core components. First, Section 4.2.1 introduces the *strict enforcement mode*, in

which suspicious DNS traffic is redirected to userspace and dropped in real time if classified as malicious. Second, Section 4.2.2 presents an *passive process threat-hunting strategy* that correlates DNS exfiltration with the originating malicious userspace process. Third, Section 4.2.3 addresses *DNS exfiltration hidden within kernel-encapsulated traffic*, which is currently prevented only in active mode. Fourth, Section 4.2.4 outlines the *features extracted* in the kernel for filtering, and in userspace for deep learning–based inference. Fifth, Section 4.2.5 details the *datasets* used to train and evaluate the detection models. Finally, Sections 4.2.6 and 4.2.7 describe *model architecture*, including serialization and quantization, as well as *streaming pipeline* for threat events from the agent to centralized message brokers and observability backends.

4.2.1 Strict Enforcement Active Mode

In this mode, eBPF programs are injected and attached as direct action filters to the TC egress hook (CLSACT QDISC) on all physical network interfaces at the endpoint. These eBPF programs are triggered as soon as a packet is queued by the kernel to the CLSACT QDISC for the specific netdev by invoking the (`dev_queue_xmit`) helper in kernel network stack. At this point, the SKB is fully constructed and ready for the TX queue, having already passed through the upper layers of the networking stack. The eBPF programs run in parallel across multiple CPU cores, with all eBPF maps declared global. This mode implementation includes several key components. *Maps* describes the structure and purpose of the eBPF LRU hash maps (see Figure 4.2). These maps track the per-packet security state and current policy status, enforced by both the kernel eBPF programs and userspace eBPF agent. Next, *Kernel eBPF filter packet classification* describes the classification of outgoing packets at the TC egress filter using SKB metadata, along with the corresponding TC actions applied to malicious and benign DNS packets. *Userspace eBPF agent packet handling* presents the zero-copy processing of sniffed packets for efficient analysis. Finally, *Concurrency handling* outline the safe access and update mechanisms for shared eBPF maps between eBPF programs inside kernel and user space agent threads.

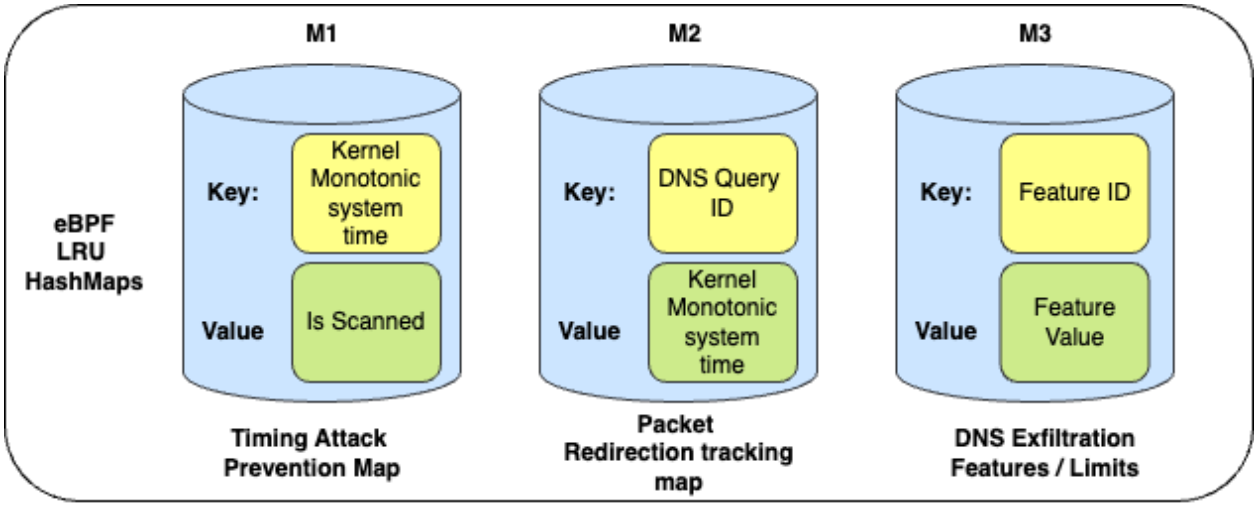


Figure 4.2: eBPF Maps structure for Agent in active phase

eBPF Maps in Active mode

1. **DNS Exfiltration Feature Map:**

   Key: Feature identifier used for DNS traffic analysis.

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

2. **Packet Redirection Tracking Map:**

   Key: DNS query ID

   Value: Kernel monotonic timestamp (in nanoseconds). Used to track suspicious packets redirected across interfaces using non-maskable interrupts (NMI) safe timekeeping.

3. **Timing Attach Prevention Map:**

   Key: Monotonic timestamp (in nanoseconds)

   Value: Scanned flag (true/false). Utilized by userspace eBPF agent to mark packets as scanned, and by the kernel eBPF program to verify packet integrity prior to retransmission.

eBPF kernel program packet processing in Active mode

When a DNS packet is transmitted over a UDP socket, it traverses the kernel's network stack, progressing through the TCP/IP layers (corresponding to Layers 2–4 of the OSI model). At the traffic control (TC) layer, the packet is intercepted via the CLSACT QDISC, where an eBPF program is attached as a direct action filter. This program parses the link (L2), network (L3), and transport (L4) headers using native kernel structures to extract metadata for further analysis. Since DNS operates at the application layer and is not natively parsed by the kernel, the eBPF program manually inspects the payload beyond Layer 4. It interprets the raw SKB data using custom C structures aligned with RFC 1035, extracting fields such as the query ID, opcode, flags, and entries in the DNS question section for deep inspection. The complete inspection logic is shown in Algorithm 1. If the query ID is new and the DNS question section is valid, the program evaluates several protocol-level features—such as multiple questions or anomalous answer counts commonly indicative of covert channels. These features are cross-checked against the kernel-defined security feature set (Table 4.3). Based on the result, the program enforces one of the following actions: `TC_ACT_SHOT` to drop the packet, `TC_ACT_OK` or `TC_ACT_UNSPECT` to forward or delegate to upstream QDISC for further filtering, or `bpf_redirect` to redirect the packet for deeper analysis.

For suspicious packets, the DNS query ID and current monotonic kernel timestamp (`bpf_ktime_get_ns`) are stored in `dns_packet_redirection_map` to prevent timing and brute-force-based evasion. Before redirection, the eBPF program retrieves metadata such as the bridge's L3 address and `if_index` from shared maps initialized by the userspace agent. Redirection counters are updated for observability. Depending on the IP version, the program applies DNAT and recalculates checksums before redirecting the packet to a bridge interface under agent control. For IPv6, static checksum values are used. The packet is then live-redirected to the RX queue of a virtual interface sniffed by eBPF agent for further deeper inspection.

Following inspection, the userspace agent reemits the packet via an `AF_PACKET` socket,

which retriggers the TC-layer eBPF program. To avoid redirection loops or forged reinjection, the eBPF filter verifies the DNS query ID against `dns_redirect_ts_verify_map`, using stored timestamps and verification flags. Only packets resent from authorized userspace paths are allowed to proceed; all others are dropped to prevent evasion. This verification mechanism ensures tight synchronization between userspace and kernel, relying on monotonic timestamps and a minimal trust model. The flow described previously is formalized in Algorithm 2.

eBPF Agent userspace packet processing in Active Mode

eBPF agent leverages kernel BPF bindings, primarily through libbpf, to abstract raw BPF syscalls. It spawns dedicated threads to continuously sniff traffic from a separate network namespace configured to handle redirected suspicious DNS traffic in active mode. Using `AF_PACKET` sockets, the agent reads packets directly from tap interfaces or RX queues in zero-copy mode, eliminating additional buffer allocations and significantly improving performance. With access to all eBPF maps via file descriptors, the agent parses application-layer DNS payloads redirected from the kernel. The extracted features, described in Table 4.4, are used for detection and policy enforcement. The agent maintains two LRU caches in userspace: one for benign SLDs, sourced from Cisco's top one million domains, and another for previously identified malicious domains.

If a parsed SLD matches an entry in the benign cache, the packet is forwarded immediately without inference, as DNS semantics make redirection of malicious traffic through high-reputation zones impractical. Packets are forwarded via either `AF_PACKET` or `AF_XDP` sockets. With `AF_XDP`, packets are injected directly into the TX queue of the device driver, bypassing the kernel TC egress filters, and the associated entry in `dns_packet_redirection_map` is cleared. In contrast, `AF_PACKET` retriggers the kernel eBPF TC program. To authorize forwarding, the agent queries the redirection timestamp using the DNS query ID, updates `dns_redirect_ts_verify_map`, and marks the packet as scanned, allowing the eBPF program in kernel to verify and forward it.

If there is a cache miss, the agent performs live inference using the deep learning model. Malicious packets are dropped, garbage collected in userspace, blacklisted in the malicious cache, and reported via Kafka. Benign packets follow the same forwarding flow as cache hits.

Additionally, the agent exports detection events, system metrics, and kernel tracing data from eBPF ring buffers to Prometheus and monitors processes generating malicious traffic, and if a threshold is exceeded, it terminates the respective malicious process. The complete userspace packet handling logic, as previously described, is detailed in Algorithm 3.

eBPF Maps concurrency handling in Active Mode

In this mode, eBPF programs use global kernel-space maps instead of isolated per-CPU maps. Since packet processing runs in parallel across multiple CPU cores, each executing the same eBPF program, concurrent access to these maps is coordinated using per-CPU kernel spinlocks. On userspace side, multiple threads spawned by the eBPF agent concurrently read and write to these shared maps. Access is synchronized using a read-write mutex, restricted to userspace, to ensure thread safety. Syscalls from userspace that update the maps may be blocked and handled internally by the kernel synchronization mechanisms, as discussed earlier. This combination of kernel spinlocks and userspace locks ensures consistent and parallel packet processing across CPUs. Two primary maps are shared between kernel and userspace: `dns_redirect_ts_verify_map` and `dns_packet_redirection_map`. Each uses unique keys, monotonic timestamps, and DNS query IDs to ensure atomicity and prevent stale reads, race conditions, and inconsistent updates. This ensures that no malicious DNS packets are leaked due to concurrency issues. All kernel-side updates to these maps use built-in LLVM concurrency helpers, which provide atomic operations and memory synchronization guarantees across CPUs. This enforces strict consistency and reliable control over the state of the shared map. The complete ordered pipeline for packet processing in this operational mode, encompassing both userspace and kernel components, is illustrated in Figure 4.3.

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

Input : Socket buffer (`skb`); eBPF LRU hash maps: `dns_limits`, `dns_packet_redirection_map`, `node_agent_config`
Output : Packet action: `TC_ACT_SHOT`, `TC_ACT_OK`; eBPF map updates: `bpf_map_updates`

`// Parse skb layers; ensure skb->data_ptr remains memory safe`
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**
    return `TC_ACT_SHOT`;
  Extract inner encapsulated protocol (`h_proto`);
Parse Layer 3 (Network) from `skb`;
if *`skb->data_ptr` exceeds `skb->data_end`* **then**
  return `TC_ACT_SHOT`;
Parse 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*** *$\neq$ 5353* ***and*** *$\neq$ 5355* **then**
  `/* Not standard DNS/MDNS/LLMNR */`
  return `TC_ACT_OK`;
`/* Parse DNS Header */`
Parse Layer 7 DNS (Application) from `skb`;
if *`skb->data_ptr` exceeds `skb->data_end`* **then**
  return `TC_ACT_SHOT`;
Extract `qd_count`, `ans_count`, `auth_count`, `add_count`;
`/* DNS Header Anomaly Check */`
if *`qd_count` $>$ 1* ***or*** *`auth_count` $>$ 1* ***or*** *`add_count` $>$ 1* **then**
  Perform `bpf_map_updates`;
  return `TC_ACT_SHOT`;
`/* Parse Question Record and Fetch Limits */`
Parse first question record from `skb`;
Fetch `n_lbls`, `dom_len`, `subdom_len`, `dom_len_no_tld`, `q_class`, `q_type` from `dns_limits`;
`/* Label Count Check */`
if *`n_lbls` $\leq$ 2* **then**
  return `TC_ACT_OK`;
`/* Deep Inspection Based on Limits */`
if *`dom_len` $>$ `dns_limits.dom_threshold`* ***or*** *`subdom_len` $>$ `dns_limits.subdom_threshold`* **then**
  Perform `bpf_map_updates`;
  return `TC_ACT_SHOT`;
`/* Mark packet for userspace processing */`
Redirect packet via `dns_packet_redirection_map` to userspace;
return `TC_ACT_OK`;

Algorithm 2: eBPF Map Handling within Kernel for **Active** Mode of Agent

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 and query ID */`

Extract DNS Layer from the packet application data;
Get DNS Query ID (`tx_id`) from parsed L7 payload in `skb`;

`/* h_proto belong to (ETH_P_IPV4 / ETH_P_IPV6) */`

Determine if packet is IPv4 or IPv6 using `nexthdr` Ethernet frame in SKB
Extract `if_index` from `skb`;

`/* Fetch virtual bridge info and skb mark */`

Fetch `dst_ip` and `skb_mark` from `netlink_links_config` with key `if_index`;
Fetch `dns_kernel_redirect_val` from `dns_packet_redirection_map` with key `tx_id`;
if *not* *dns_kernel_redirect_val* **then**
    `/* Packet intercepted first time by TC eBPF filter */`
    Modify `skb` destination IP to `dst_ip`;
    if *ETH_P_IPV4* **then**
        Recompute Layer 3 checksum and update in `skb`;
        Set `l3_checksum` = computed checksum;
    else
        Set `l3_checksum` = `0xFFFF`;
    if *not* *skb_mark* **then**
        Set `skb_mark` = `bpf_get_prandom_u32`;
        Update `skb_netflow_integrity_verify_map` with key=`0xFFEF`, value=`skb_mark`;
    Mark `skb->mark` = `skb_mark`;
    Update `dns_packet_redirection_map` with key=`tx_id`, value=`l3_checksum`, `kernel_time_ns`;
    `/* Global packet redirect count metric for each associated netdev */`
    Increment and update `redirect_count_map` with key=`if_idx`, value=`new_packet_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, clean required map entries */`
        Delete `redirect_ts_verify_val` from `dns_redirect_ts_verify_map` with key=`kernel_time_ns`;
        return `TC_ACT_OK`;

Algorithm 3: Userspace eBPF Agent Packet Processing in **Active** Mode

Input : Sniffed DNS packets from suspicious Linux namespaces (via zero-copy pcap),
Extracted DNS features,
All relevant kernel eBPF LRU hash maps
Output : Packet either garbage collected (if malicious) or retransmitted (if benign)

```
Sniff traffic over veth pair interfaces in Linux namespace;
Extract DNS features from sniffed packets;
Export monitoring metrics (e.g., redirection_count, loop_time) from kernel maps;
Fetch isSLDBenign from userspace LRU map of top 1M SLDs;
if isSLDBenign then
    Set shouldRetransmit ← true;
else
    Fetch isBlacklistedSLDFound from malicious SLD map;
    if isBlacklistedSLDFound then
        return ;                                   /* Drop packet immediately */
    Pass DNS features to ONNX model for inference;
    if inference result is MALICIOUS then
        Emit event to message brokers with packet details;
        Blacklist SLD in malicious map;
        Garbage collect packet data;
        return
    else if inference result is BENIGN then
        Set shouldRetransmit ← true;
if shouldRetransmit then
    Fetch dns_kernel_redirect_val = {l3_checksum, kernel_time_ns} from
     dns_packet_redirection_map by 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;
        Write packet via 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;
        Write packet via syscall.write(AF_XDP, SOCK_RAW, 0);
```

4.2.2 Passive Malicious Process Threat Hunting Mode

Passive mode reuses the same eBPF program attached to the egress CLSACT TC filter, as introduced in Section 4.2.1. This mode extends the design by layering additional security and process-aware security enforcement to prevent DNS-overlaid port obfuscation threats.

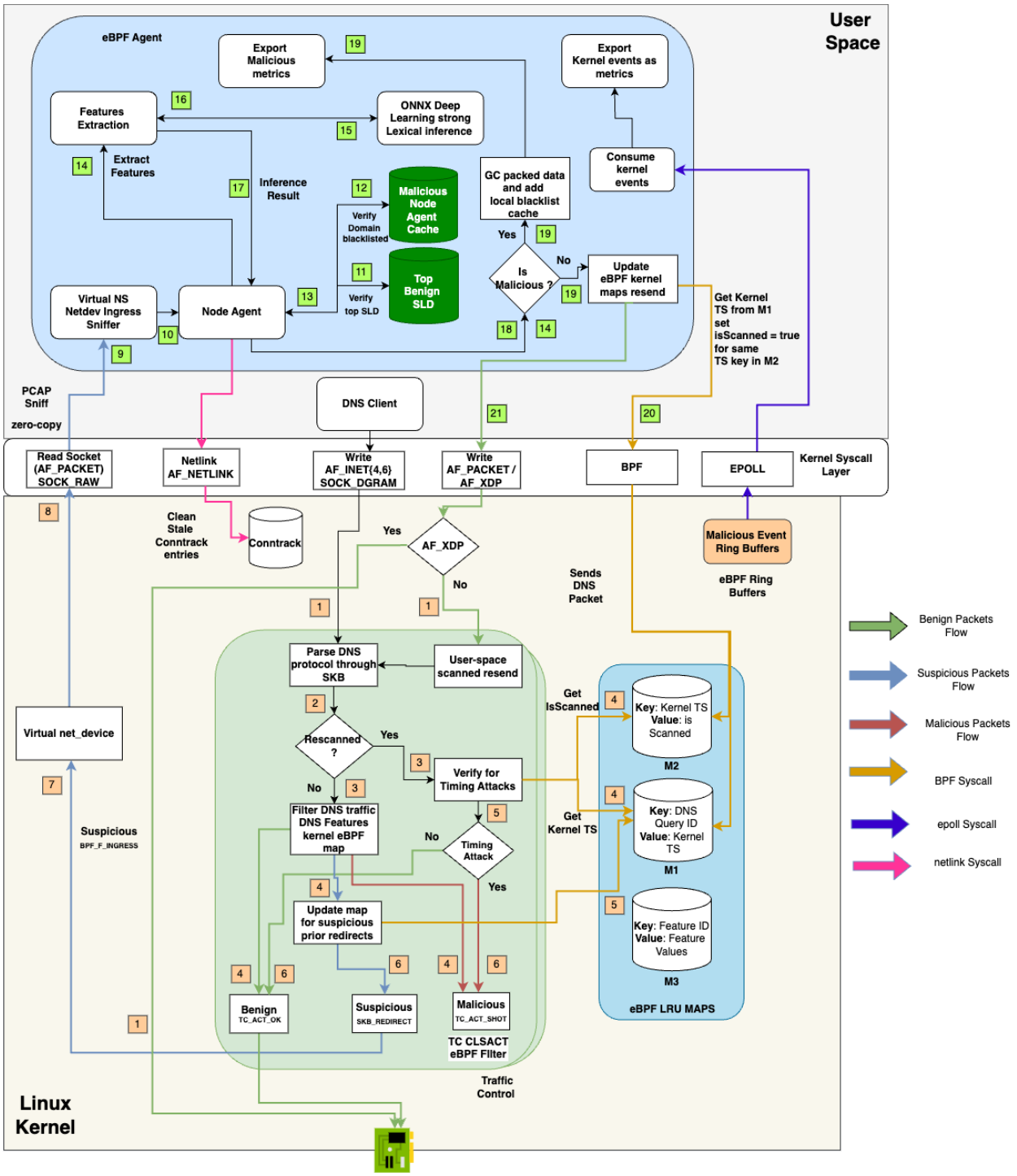


Figure 4.3: eBPF Agent: DNS Exfiltration Prevention Flow for active phase

The passive mode architecture is broken down as follows. *Maps* introduces the key eBPF structures used in this mode, their types, and their roles, as visualized in Figure 4.4. *Kernel packet processing* details the eBPF programs in the kernel to identify and drop malicious DNS exfiltration attempts while tracking the responsible userspace processes. It also leverages kernel tracepoints attached to kernel process scheduler for efficient garbage collection of malicious process tracking. The next section, *Userspace packet processing*, describes the eBPF agent that parses and classifies traffic to identify potentially malicious processes involved in exfiltration. Finally, *eBPF map concurrency* explains the kernel and userspace components maintain a consistent eBPF map state under concurrent CPU access.

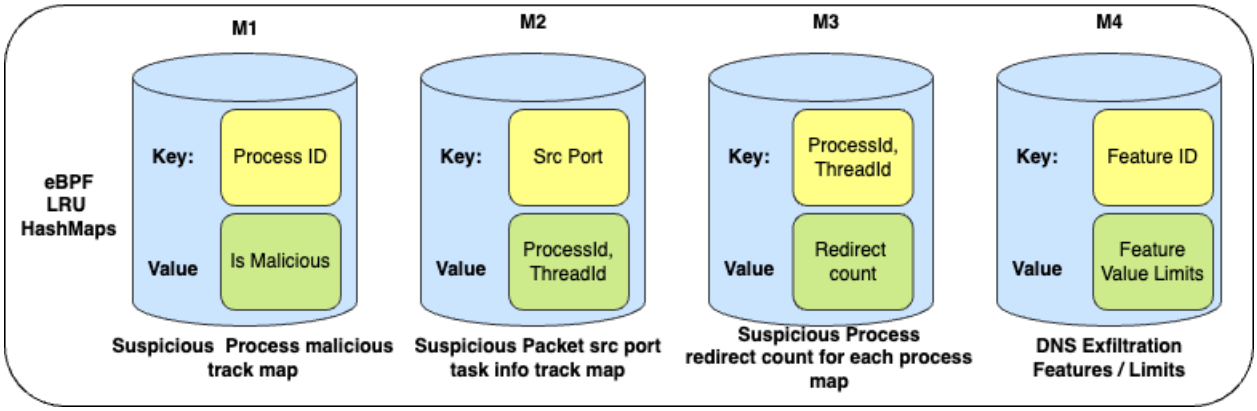


Figure 4.4: eBPF Maps and structure for Agent in passive phase

eBPF Maps in Passive mode

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

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

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

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

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

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

3. **Malicious Process Tracking Map:**

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

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

eBPF kernel program packet processing in Passive mode

In passive mode, the agent targets DNS exfiltration attempts over arbitrary UDP ports that may evade active mode protections. Like in active mode, the eBPF program parses packets up to Layer 4 using SKB metadata. However, it deliberately avoids redirecting all suspicious UDP packets, reducing unnecessary congestion and preserving latency for legitimate Layer 7 protocols using non-standard ports.

Because the kernel eBPF program cannot reliably determine whether application data over arbitrary UDP ports contains DNS, it uses `skb_clone_redirect` to create a full copy of packets that heuristically resemble DNS. These clones are redirected to a netdev interface managed by the eBPF agent. To minimize memory overhead, redirection is only performed if the payload structure heuristically aligns with DNS protocol fields. The eBPF program performs raw parsing within SKB bounds (skb->data_end) and validates structure against constraints defined in RFC 1035. If the payload passes these checks, clone redirection is triggered. The DNS protocol parsing as explained previously from application data of other protocols in SKB is explained in Algorithm 4.

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

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

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

eBPF Agent userspace packet processing in Passive Mode

The parallel packet sniffing procedure from the respective network namespace mirrors the approach described earlier in the active phase. As in active mode, clone-redirected packets are received in zero-copy userspace, preserving both application payloads and lower-layer headers. The agent parses each packet to identify embedded DNS structures. If no valid DNS layer is found, the source port is removed from `src_port_task_struct_map` to prevent stale state tracking. When DNS is detected, the agent applies the same logic as in active mode: checking a local LRU memory cache for known blacklisted domains and, if absent, performing feature extraction and deep learning inference. If a packet is classified as malicious, the agent updates the blacklist and streams a threat event to the observability backend. Because these are clone-redirected packets, they are not reinjected. Instead, the agent uses the source port to retrieve the process and thread ID from `task_struct_redirect_ct_map`, flags the process in `malicious_process_map`, and enables the kernel to drop future packets from it. The agent also checks how many packets have been redirected for that process and are malicious. If the count exceeds a configurable threshold and the process is marked malicious,

the agent terminates the respective process. This process-aware passive mode enables adaptive detection and response for exfiltration over non-standard ports. The passive processing logic of the userspace eBPF agent, as previously explained, is detailed in Algorithm 6.

eBPF Maps concurrency handling

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

Algorithm 4: Egress TC-Based DNS Raw SKB Inspection 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 layer headers from `skb`;
Parse DNS header from `skb`;
Extract DNS header counts: `qd_count`, `ans_count`, `auth_count`, `add_count`;
`/* Verify DNS count fields are within valid range */`
if *qd_count* ≥ 256 ***or*** *ans_count* ≥ 256 ***or*** *auth_count* ≥ 256 ***or*** *add_count* ≥ 256 **then**
    return `TC_ACT_OK`
Extract DNS flags: `raw_dns_flags` from DNS header;
`/* Validate opcode and rcode per RFC 1035 */`
if *opcode* ≥ 6 **then**
    return `TC_ACT_OK`
if *rcode* ≥ 24 **then**
    return `TC_ACT_OK`
Perform necessary `bpf_map_updates`;

Algorithm 5: eBPF Map Handling within Kernel for **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 query ID (`tx_id`) from DNS header;
`/* if_index is netdev link index in 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` using `bpf_get_current_pid_tgid`;
Fetch `is_malicious` flag from `malicious_process_map` using `process_id`;
if *not* *`is_malicious`* *or* *`is_malicious`* *is* *`null`* **then**
    `/* Track src port for the packet and corresponding process transmitting it */`
    Update `src_port_task_struct_map` with key=`process_id`, value=`task_struct`;
    Fetch `current_suspicious_ct` from `task_struct_redirect_ct_map` with key=`task_struct`;
    `/* Suspicious packet redirect count per process */`
    Update `task_struct_redirect_ct_map` with key=`task_struct`, value=`current_suspicious_ct + 1`;
    `/* Global packet clone redirect count metric per netdev */`
    Update `dns_packet_clone_redirection_ct_map` with key=`if_index`, value=`new_clone_redirect_count`;
    if *`is_malicious`* *is* *`null`* **then**
        `/* First DNS packet sent by process; mark initially the process as benign */`
        Update `malicious_process_map` with key=`process_id`, value=`false`;
    Perform `bpf_clone_redirect(skb, bridge_if_index, BPF_F_INGRESS)`;
else
    Update `task_struct_redirect_ct_map` with key=`task_struct`, value=`current_suspicious_ct + 1`;
    `/* Drop packet since malicious process attempted exfiltration; continue clone redirecting clones for monitoring malicious behavior */`
    Perform `bpf_clone_redirect(skb, bridge_if_index, BPF_F_INGRESS)`;
    return `TC_ACT_SHOT`

Algorithm 6: Userspace eBPF Agent Packet Processing in **Passive** Mode

Input : Sniffed DNS packets from suspicious Linux namespaces via `pcap`;
all kernel eBPF maps;
eBPF LRU hash maps: `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 ;                                        /* Ignore non-DNS traffic */
Extract L4 transport ports: src_port, dest_port;
Extract DNS userspace features: tx_id, qd_count, ans_count, query_class, query_type;
Fetch task_struct from src_port_task_struct_map with key=src_port;
Extract process_id and thread_group_id from task_struct;
Fetch isBlacklistedSLDFound from userspace LRU hash;
if isBlacklistedSLDFound then
    /* Mark the process as malicious                                          */
    Update 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
        /* Terminate malicious process and clean maps                         */
        Send SIGKILL to process_id;
        Delete key=process_id from malicious_process_map;
    return;
Pass extracted features to ONNX model for inference;
if inference_result == MALICIOUS then
    Emit malicious threat events to message broker;
    Blacklist SLD for related DNS records in userspace LRU malicious cache;
    /* Mark the process as malicious                                          */
    Update 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
        /* Terminate malicious process and clean maps                         */
        Send SIGKILL to process_id;
        Delete key=process_id from malicious_process_map;
    return;
else if inference_result == BENIGN then
    Garbage collect sniffed cloned packet data;
    /* No immediate action; track future packets                              */
    return;
```

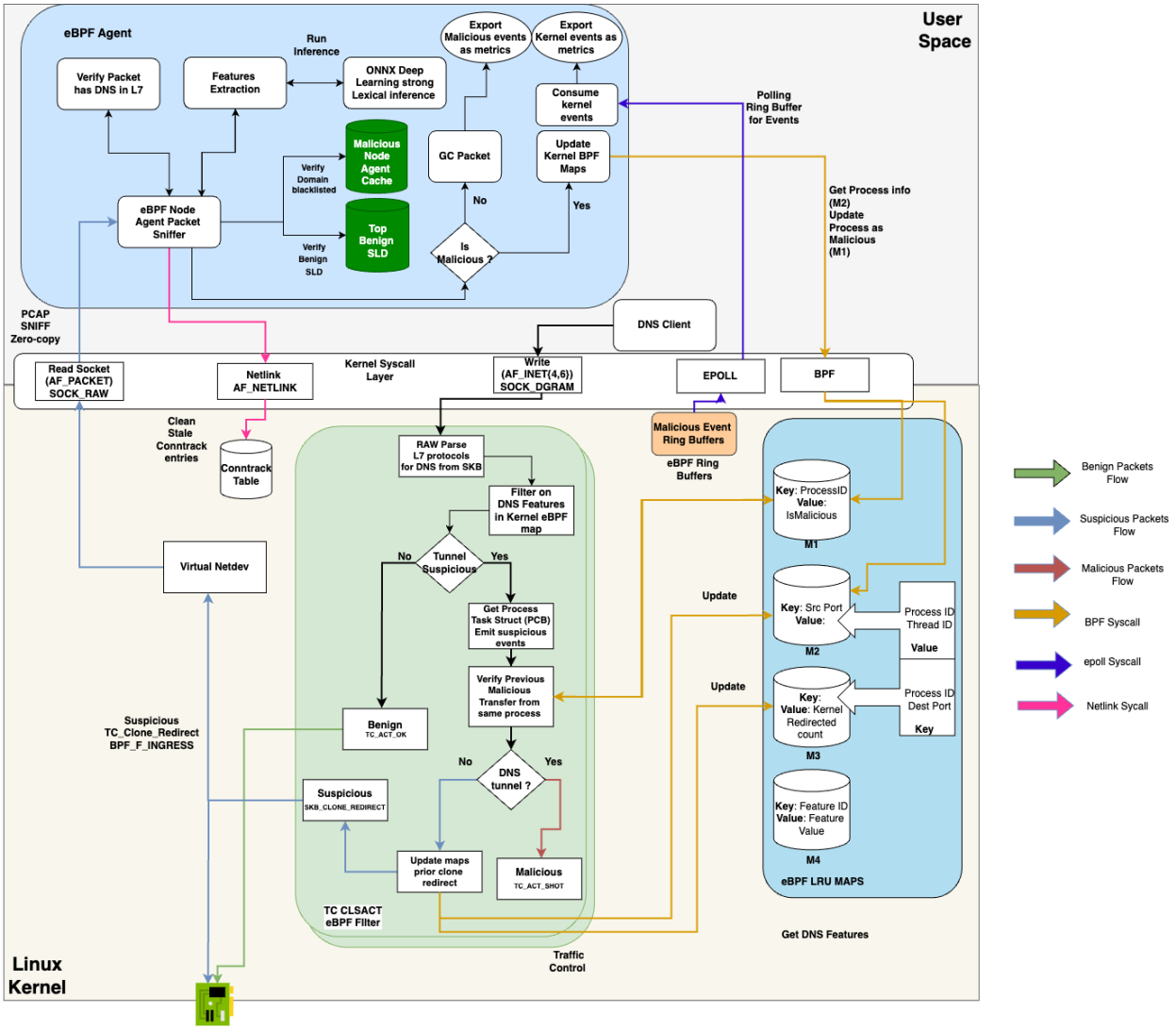


Figure 4.5: eBPF Agent: DNS Exfiltration Prevention Flow for passive phase

4.2.3 DNS Exfiltration via Encapsulated Traffic

In the Linux kernel networking stack, encapsulated traffic is managed through virtualization drivers that extend netdev with associated RX and TX queues. These drivers support encapsulation across the L2, L3, and L4 layers. The current eBPF agent focuses on L2-level encapsulation over software network devices. It does not target tunnels using kernel cryptographic subsystems through the kernel keyring and eXtensible framework (xfrm), such as OpenVPN, IPsec, or WireGuard. This design decision reflects real-world DNS usage: DNS resolution rarely occurs within cryptographic tunnels. In practice, DNS exfiltration over encapsulated traffic is most commonly seen with VLAN-tagged traffic and TUN/TAP virtual interfaces. VLAN encapsulation is handled during SKB parsing by the TC eBPF program in the active phase, as explained earlier in Algorithm 1. Here, L2 headers such as 802.1Q and 802.1AD are stripped to expose the inner packet for deep inspection. TUN/-TAP interfaces, on the other hand, are virtual software devices exposed to userspace via file descriptors. Malicious processes can write raw DNS-tunneled packets directly to these interfaces, bypassing conventional filtering. The kernel applies L3 encapsulation on transmit and L2 de-encapsulation on receive (via TAP), before forwarding to a physical NIC. These interfaces are usually created via iproute2 or netlink. To address DNS tunneling over TUN/-TAP, the agent injects kprobes into the `tun_chr_open` kernel driver. When a new TUN/TAP interface is created, the agent receives a kernel event via a ring buffer. It responds by attaching the same DPI eBPF logic (from the active phase) to the TC egress hook of the new interface as illustrated in Figure 4.6. This ensures consistent filtering across dynamically spawned tunnels. The ring buffer event includes rich telemetry about the creating process, which aids both monitoring and threat attribution.

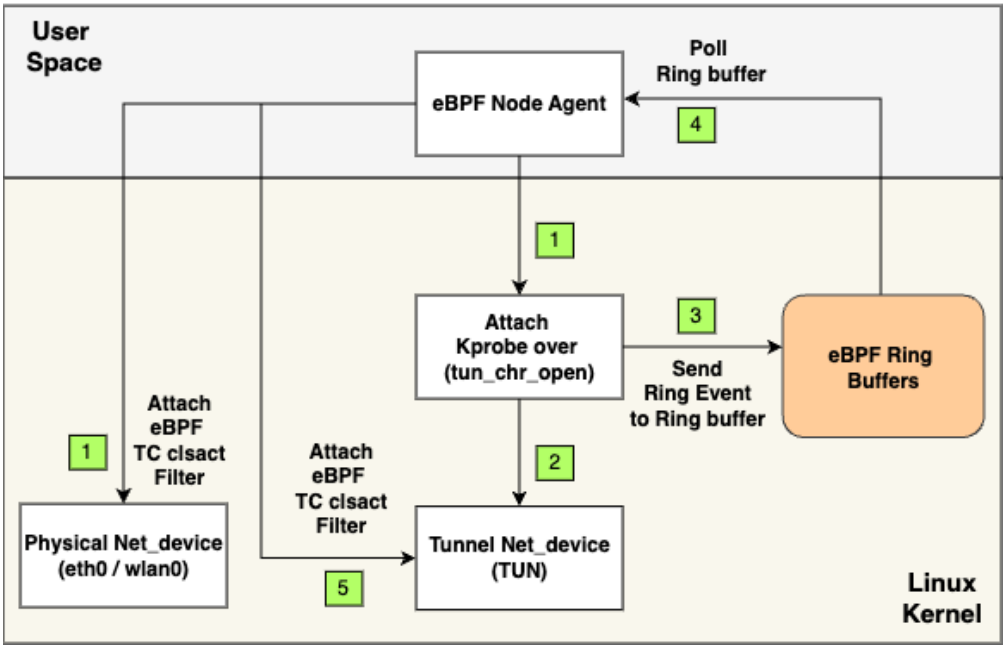


Figure 4.6: eBPF Agent: Prevention flow over Tun/Tap Driver kernel function

4.2.4 Feature Analysis in Data Plane

Features used by the eBPF agent in the data plane are derived from the real world DNS exfiltration traffic. Selection was guided by known patterns in DNS tunneling, C2 channels, and encoding behaviors observed in open source and proprietary DNS exfiltration tools and real-world attack logs. The goal is to enable real-time filtering of malicious queries with minimal overhead while adhering to DNS protocol constraints defined in RFC 1035. DNS packets over UDP are limited to 512 bytes, with domain names capped at 255 characters. Individual labels must conform to structural limits (typically 63 characters), and payload-bearing query types such as TXT and NULL can embed encoded data. Although extensions like EDNS and TCP-based DNS allow larger payloads, most abuse still occurs over standard UDP. To handle this, feature selection is divided into two stages: *kernel-space features*, enforced by kernel eBPF programs for inline filtering and classification; and *userspace features*, analyzed by a deep learning model to detect stealthy or obfuscated payloads.

Kernel eBPF TC program classification and filtering features

Due to the instruction count and complexity limits enforced by the eBPF verifier, DPI is restricted to parsing only the first DNS question record. This aligns with modern DNS usage, where queries typically contain a single question; multiple questions are treated as an anomaly. Numeric features such as domain length and label count are checked against the thresholds configured during initialization. Suspicious query types like 'NULL' and 'TXT'—which may carry arbitrary payloads - are flagged, as are query classes outside 'Internet' (IN), according to RFC 1035. These features are recorded in a kernel feature map and used to classify packets in real time. Packets that exceed thresholds are dropped; those deemed suspicious are redirected or clone-redirected depending on agent mode; all others are marked benign. This logic is applied consistently across both active and passive modes, including encapsulated traffic, where tunnel headers are removed before inspection. The definitions of kernel-level features are shown 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 4.4 summarizes the complete set of features.

4.2.5 Datasets

The deep learning model is trained on a synthetically generated dataset constructed by combining three distinct sources, resulting in a total of 3.8 million domain samples evenly split between benign and malicious.

1. **Benign SLDs from Cisco:** The top 1 million SLDs from Cisco are not used for training, but are loaded into an in-memory LRU cache by the eBPF agent. This avoids unnecessary inference on common and trusted domains and improves performance. The cache is fully reprogrammable from the control plane.
2. **ISP-Captured Dataset:** The core training data set is sourced from Ziza et al., consisting of live-sniffed DNS traffic from an ISP, with approximately 50 million samples that span both benign and malicious traffic [29].
3. **Synthetic Exfiltration Dataset:** To address class imbalance, a custom data set was created using open source tools such as `DET`, `DNSExfiltrator`, `DNSCat2`, `Sliver`, `Iodine`, and internal DNS exfiltration scripts. These samples include a wide range of obfuscation techniques (see Table 2.1) across various file types—text (e.g., `.txt`, `.md`), image (e.g., `.jpg`, `.jpeg`, `.png`), and video (e.g., `.mp4`)

Table 4.3: DNS Features in Kernel

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 4.4: DNS Features in Userspace

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

4.2.6 Deep Learning Model Architecture

The deep learning model operates in userspace to enhance the detection accuracy of the eBPF agent, especially for identifying obfuscated DNS payloads—capabilities that are infeasible to implement directly in kernel space due to eBPF verifier constraints. Built with TensorFlow, the model is a sequential dense neural network using eight lexical input features (as described earlier). It comprises three hidden layers with 16 neurons each, followed by a sigmoid-activated output neuron for binary classification. ReLU is used between layers, and the model is optimized using Adam with a learning rate of 0.001 and binary cross-entropy loss. Training is carried out over 25 epochs on a dataset of approximately 3.8 million entries, balancing convergence and overfitting prevention. To optimize training performance, TensorFlow's shuffler and prefetch mechanisms are used, and the process is parallelized across multiple GPUs using MirroredStrategy. Once trained, the model is exported to the ONNX format (Open Neural Network Exchange) for efficient inference. The ONNX model is integrated into the eBPF agent as a submodule through Unix domain sockets (IPC), using ONNX Runtime. Although this design introduces slight inference latency, it is mitigated by a caching layer inside the agent, consisting of an LRU-based blacklist and a domain-level cache. ONNX was selected for its runtime efficiency and broad interoperability, despite its limited tooling in the Golang ecosystem. The model is dynamically quantized (e.g., float32 $\rightarrow$ int8), which reduces

memory usage and inference time. Compared to formats such as HDF5 or Pickle, ONNX provides a compact, performant graph representation, ensuring that the agent maintains a minimal footprint even under peak load. Figure 4.7 shows the quantized ONNX graph representation of the dense neural network architecture.

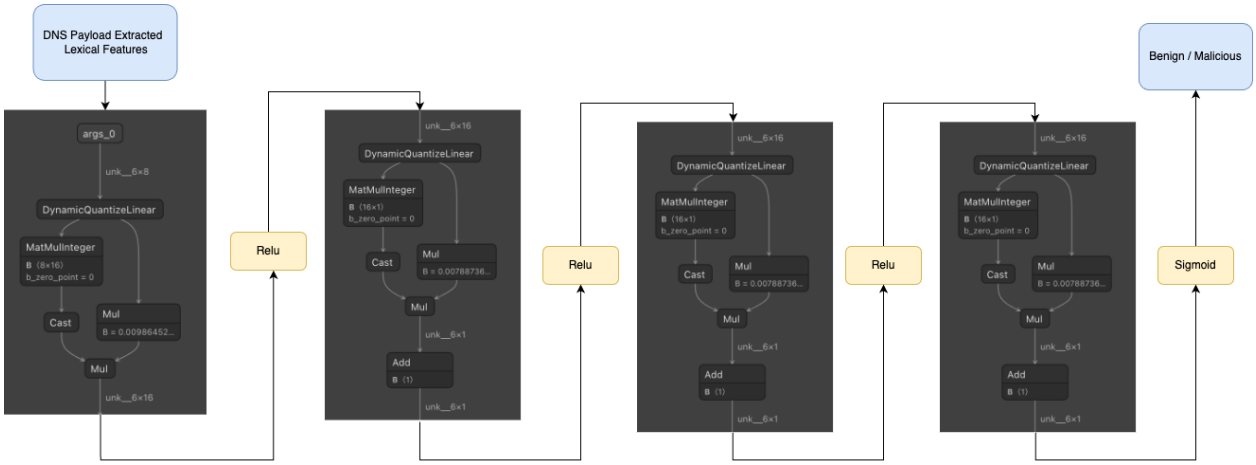


Figure 4.7: Deep Learning Model Architecture (ONNX) for DNS Obfuscation Detection

4.2.7 Thread Events Streaming and Metrics Exporters

When an eBPF agent flags a DNS packet as malicious in the data plane and contains an exfiltrated payload, the agent streams a threat event using Kafka producers. These producers are embedded in the compiled eBPF userspace binary and configured to send data to a remote Kafka broker. Each eBPF agent also includes Kafka consumers. Each agent is assigned a unique application ID derived from the local node's IP address, combined with a randomly generated ID to form the Kafka consumer group ID. This design ensures strong decoupling between agents, enabling massive horizontal scalability. Data plane nodes can scale independently without relying on a shared consumer group. Kafka producers operate asynchronously, allowing the agent to emit threat events while concurrently consuming control plane topics streamed by the inference controller. These topics carry resolved malicious domains, enabling each data plane node to update its local blacklist even if the exfiltration

was detected on a different node, allowing cross-node security enforcement in the data plane. Additionally, consumed events allow eBPF agents across nodes to apply dynamic Layer 3 filters in the kernel, supporting cross-protocol correlation. While threat events focus on detected exfiltration attempts, kernel-space eBPF programs also export deep system-level metrics. These metrics are exposed via Prometheus, allowing the controller to scrape and monitor them across all nodes in real time. This centralized observability supports both the analysis of blocked threats and continuous system behavior tracking. Table- 4.5 explains the details of kafka topics and their significance in the security framework. Appendix A provides additional details on the metrics exported by the eBPF agent.

Table 4.5: Kafka Topics utilized by the eBPF Agent

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

4.3 Control Plane

As illustrated in the overview of the component components of the control plane, the controller is designed to be entirely stateless, relying on the GPSQL backend used by PowerDNS for DNS state management and RPZ zones to track malicious domains. The control plane currently consists of a single Tomcat web server, with Spring Kafka consumer integrated to consume malicious threat events from the exfil-sec Kafka topic. These events are emitted by all endpoints in the data plane and contain blacklisted domain metadata and node-level context that identify where the threat was detected. Upon consuming these events, the control plane dynamically updates the RPZ backend with the (SLDs) associated with malicious domains in threat events. This update enforces DNS-level security enforcement, which causes the DNS server to start dropping any queries to these domains. To optimize performance and

prevent malicious DNS queries from ever reaching the DNS server again once blacklisted, the control plane also writes to another Kafka topic (exfil-sec-infer-controller). This topic is consumed by all data plane nodes to rehydrate their local malicious userspace caches, effectively reprogramming the eBPF agents at each endpoint to drop related packets immediately at the endpoint, reducing network hops. For enhanced security, the control plane additionally performs DNS resolution on the malicious domains to extract their corresponding Layer 3 addresses. These addresses, which represent active C2 or tunneling servers on the public internet, are also streamed in exfil-sec-infer-controller Kafka topic used to dynamically enforce layer 3 network policies inside the kernel post consumed by the eBPF agents deployed across the data plane for cross protocol correlation. With fully asynchronous, bidirectional communication between the control plane and the data plane via Kafka, the architecture enables continuous re-programmability of data plane nodes to enforce dynamic kernel-level security policies. It also supports the horizontal scalability of individual components crucial for production-grade cloud environments. This design directly targets and stops DGA, providing robust protection against mutation in both Layer 3 and Layer 7.

4.4 Distributed Infrastructure

As described earlier in the security framework overview, the distributed infrastructure consists mainly of a PowerDNS authoritative server, a PowerDNS recursor, a PostgreSQL-based GPSQL backend, Kafka brokers, and Prometheus scrapers. These scrapers collect metrics exposed by eBPF agents deployed in the data plane. Notably, eBPF agents do not handle malicious DNS exfiltration over TCP due to the complexity of the TCP state machine and congestion control within the kernel. To address this, TCP-based DNS exfiltration attempts are intercepted in userspace by PowerDNS recursor query interceptors acting as middleware. Because the PowerDNS recursor supports only Lua-based interceptors, a custom Lua module was developed. This interceptor extracts features from DNS queries received over TCP and performs inference using a serialized ONNX deep learning model. Leveraging Lua's

lightweight, high-performance nature, the interceptor accesses the GPSQL backend's RPZ table, which contains known malicious domains. This enables returning `NXDOMAIN` responses for blacklisted queries, skipping inference to improve throughput. Furthermore, the interceptor employs PowerDNS internal domainsets - fast, in-memory trie-based caches of malicious domains detected over TCP. These caches enable rapid filtering of repeated exfiltration attempts. To maintain accuracy without inference overhead, the domainsets periodically synchronize with the RPZ, ensuring up-to-date enforcement. Appendix B explains the Lua interceptor filtering DNS exfiltration over TCP.

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 on 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 `netvsc` hypervisor network driver, with 100 Gbits/sec network bandwidth and 8 TX/RX hardware queues aligned to CPU cores to enable optimal packet steering and parallel processing for high-throughput netflow handling. In addition, all eight CPU cores and memory resided on a single NUMA node, eliminating memory lookup overhead during kernel benchmarking. The test bench launches a custom PowerDNS authoritative and recursor server on CSSVLAB08. The controller and a Kafka single broker instance run on CSSVLAB01. Nodes CSSVLAB02–07, excluding CSSVLAB06, act as the data plane, each running the eBPF agent and using the custom PowerDNS server as the default resolver via systemd-resolved. CSSVLAB06 is used to simulate DNS exfiltration attacks against data plane nodes, tunneling DNS queries through the same PowerDNS instance. The complete deployment is illustrated in Figure 5.1

5.2 Evaluations Results

The evaluation of this security framework is presented for each individual architecture subcomponent in the following sections.

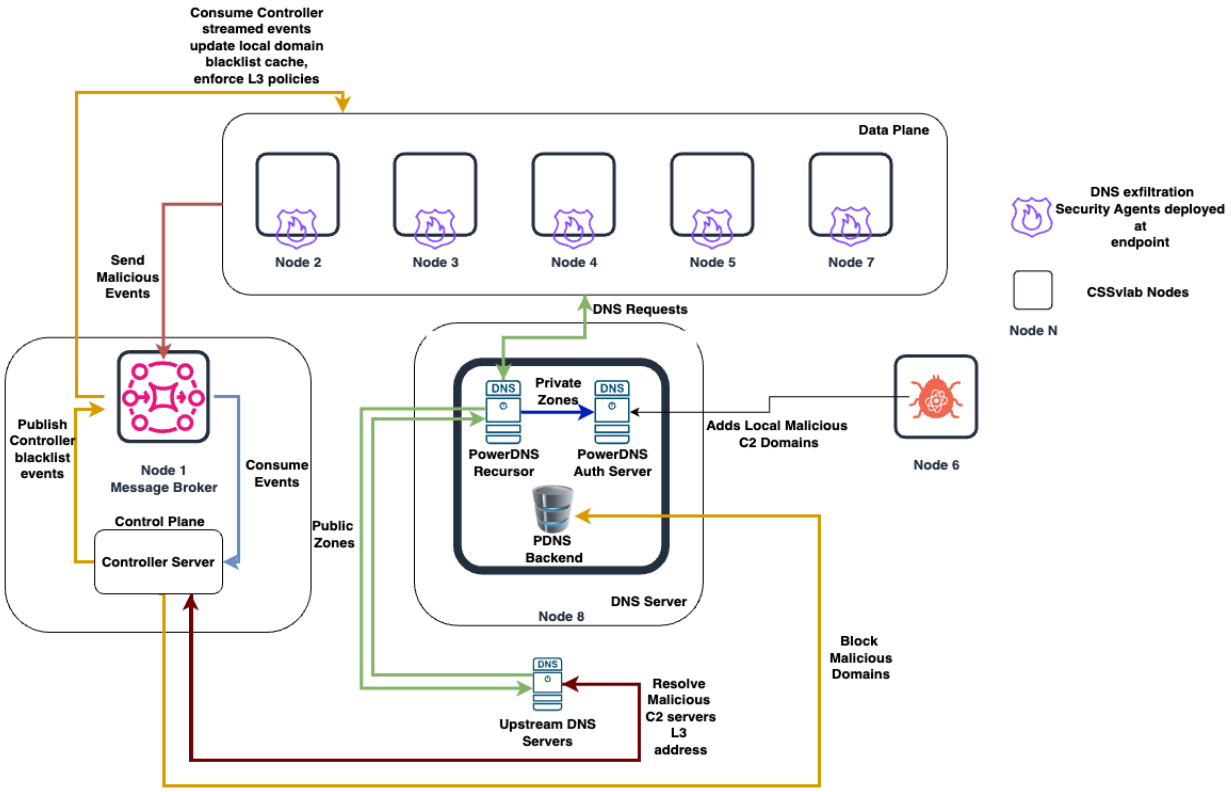


Figure 5.1: Security Framework Deployed Architecture over CSSVLAB Nodes

5.2.1 Data Plane

The effectiveness of the eBPF agent at the endpoint of the data plane is evaluated through quantized deep learning model 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 including agent's response time for these attacks is evaluated. Agent's performance evaluation is performed on a single node within the data plane running the eBPF agent at the endpoint.

Deep Learning Model Evaluation

The trained deep learning model was evaluated on a dataset of 3.8 million domains, divided into 70% for training and 15% each for validation and testing. After training, the model achieved a validation precision of 99.7%, with the loss steadily decreasing throughout the 25 training epochs and reaching 0.98% at the end, as shown in Figure 5.3. Given the DNS data exfiltration use case, the model performance was assessed with a primary emphasis on minimizing false positives. A high false-positive rate would not only cause the eBPF agent to drop benign DNS packets and generate false threat events, but could also result in the termination of legitimate processes, thereby introducing operational risks. In contrast, false negatives were considered less critical, as the agent would allow malicious traffic to pass through without taking privilege-level actions. Therefore, **precision** was prioritized over recall. These metrics are formally defined as:

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

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

Based on these considerations, and using a data set engineered to capture a wide range of data obfuscation techniques, the model achieved a precision of 99.59% and a recall of 99.87% when evaluated on the test dataset, as shown in Figure 5.1, with the corresponding confusion matrix presented in Figure 5.2. For runtime inference using ONNX within the eBPF agent, a binary classification threshold of 0.85 was selected. This value was determined by analyzing F1 score performance across varying classification thresholds, as illustrated in Figure 5.4, with particular attention to the trade-off between precision and recall. Since minimizing false positives was the top priority, the selected threshold maximized precision while maintaining a relatively high recall. This high-precision, low false-positive outcome was enabled by the chosen feature set, which included Shannon entropy across multiple encoding and encryption schemes, allowing the model to effectively learn obfuscation patterns.

Table 5.1: Model Evaluation Metrics

Metric	Train	Test
Accuracy	0.9973	0.9997
AUC	0.9997	0.9997
Loss	0.0099	0.0091
Precision	0.9959	0.9959
Recall	0.9987	0.9988

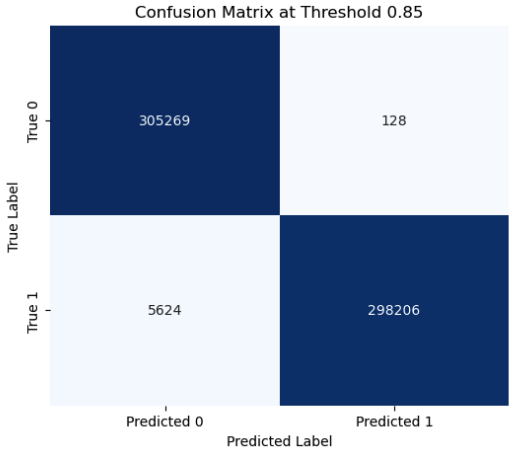


Figure 5.2: Confusion Matrix

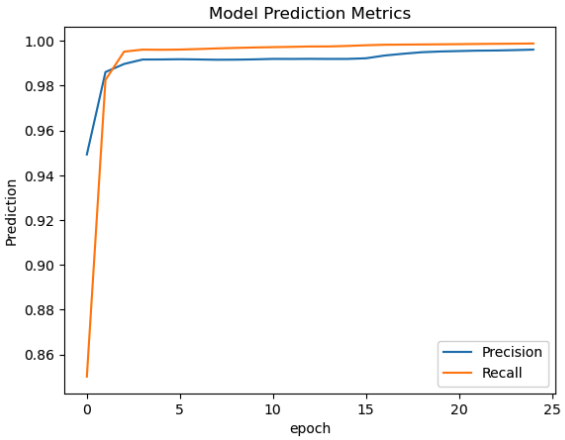


Figure 5.3: Model Precision

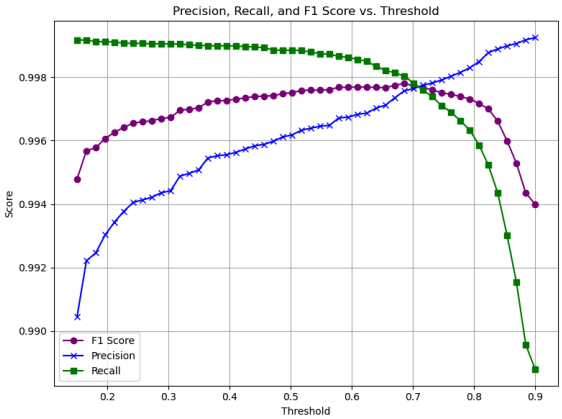


Figure 5.4: Precision, Recall, and F1 Score vs. Threshold

eBPF Agent Request Throughput and Latency Metrics in Active Mode

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

cache; for cache misses, live ONNX inference is performed. The kernel feature thresholds in the eBPF map were intentionally kept stringent, causing most DNS packets to be flagged as suspicious to stress-test the throughput. Throughput was measured using DNSPerf, which quantifies both request throughput and DNS response success rates. The test locked DNSPerf to send 10,000 DNS requests per second over 20 seconds, monitoring packet loss. The recursor server was assumed to be healthy with no additional impact on DNS throughput testing. The testing node uses the hypervisor-supported `netvsc` virtual driver with 8 combined RX/TX queues, but lacks the discrete ring buffers required for `AF_XDP`, making direct packet injection into TX queues unsupported. Consequently, the egress path relies on `AF_PACKET`. In the cache-hit scenario (100% benign global SLD matches), throughput ranged from 8,100 to 9,820 DNS requests/sec with zero packet loss, as shown in Figure 5.5. The X-axis represents the benchmark duration, while the Y-axis shows both packet throughput and loss. Latency remained low, ranging from near 0 ms to 250 ms per 10k sample (Figure 5.7, with X-axis as duration and Y-axis as latency). In contrast, in the cache-miss path that requires live inference, the throughput was reduced to 430-520 requests / sec (Figure 5.6) and showed peak latencies of up to 750 ms (Figure 5.8), using the same axis conventions. Despite this, no packet loss was observed, which ensures reliability. The latency spike is attributed to the overhead of UNIX domain socket communication with the ONNX inference server and Python's global interpreter lock (GIL), which limits concurrency. In contrast, the eBPF agent being compiled binary support true concurrency and parallelism, offering significantly better performance. It was observed that throughput becomes unstable beyond 5,000 DNS requests/sec, though such traffic volumes typically indicate malicious behavior and can be rate-limited in kernel eBPF programs. Overall, for stress testing over a prolonged period of time, the agent successfully processed a maximum throughput of 67.3 million DNS requests per hour with no packet loss, while performing deep parsing across both kernel and userspace.

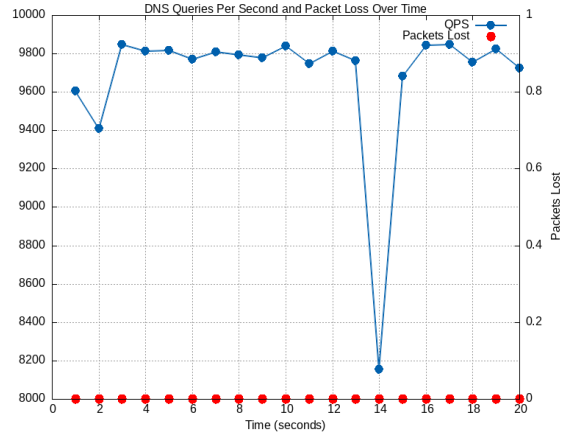


Figure 5.5: eBPF Agent: DNS Throughput for GSLD LRU Hit (10k req/s)

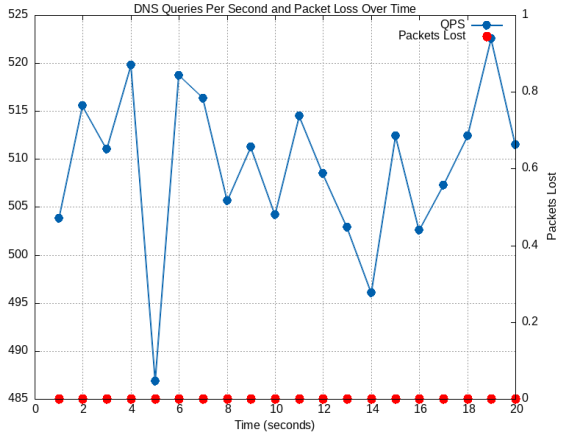


Figure 5.6: eBPF Agent: DNS Throughput, GSLD LRU Miss, ONNX (10k req/s)

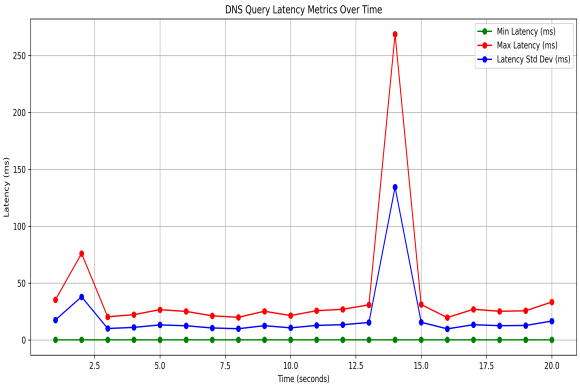


Figure 5.7: eBPF Agent: DNS Latency for GSLD LRU Hit (10k req/s)

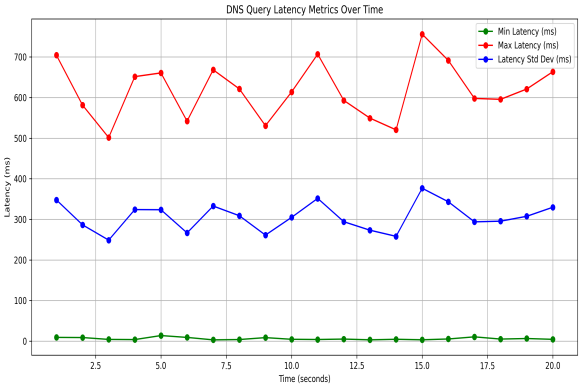


Figure 5.8: eBPF Agent: DNS Latency, GSLD LRU Miss, ONNX (10k req/s)

eBPF Agent Request Throughput and Latency Metrics in Passive Mode

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

are not emphasized; instead, performance is measured by how effectively the system detects and stops beaconing implants that transmit data over DNS, often across random UDP ports. Figure 5.9 illustrates the total volume of exfiltrated data prevented by the eBPF agent before process termination. The horizontal axis represents different process kill thresholds, while the vertical axis shows the amount of data loss prevented. DNSCat2 was used for the benchmark configured with a 20-second beaconing interval and exfiltrating through various types of DNS records (MX, TXT, CNAME, HTTP, SRV, etc.), demonstrating the security strength in delaying termination just enough to observe beacon patterns, allowing network administrators to tune termination thresholds for maximum insight.

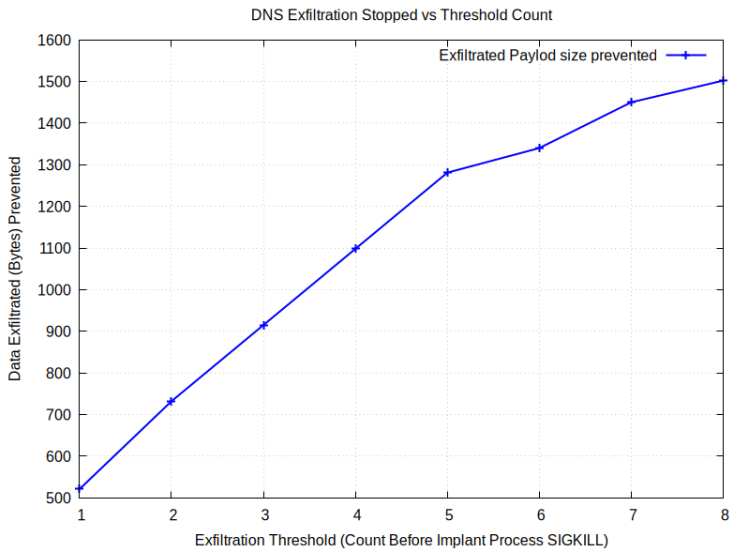


Figure 5.9: eBPF Agent: Volume of DNS Exfiltrated Data Prevented vs. Process Kill Thresholds

eBPF agent Resource Usage

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

MB of memory for the main process. This includes heap memory, as all LRU maps loaded by agent's process for fast lookups. At a higher throughput of 100,000 DNS requests/sec, memory usage remained nearly the same, peaking at 350 MB. Figure 5.10 and Figure 5.11 illustrate the memory usage corresponding to the previously discussed DNS request throughput benchmarks. The horizontal axis represents the benchmark duration in seconds, while the vertical axis shows memory usage over time. 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 it 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. Appendix A provides additional profiling details for eBPF agent in kernel.

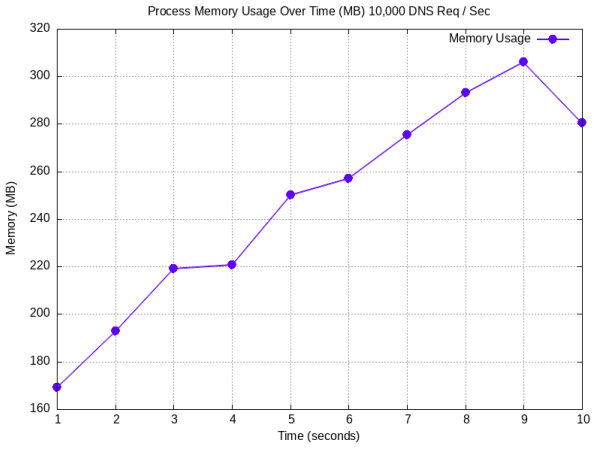


Figure 5.10: eBPF Agent: Process Memory Usage for 10K DNS Req/Sec

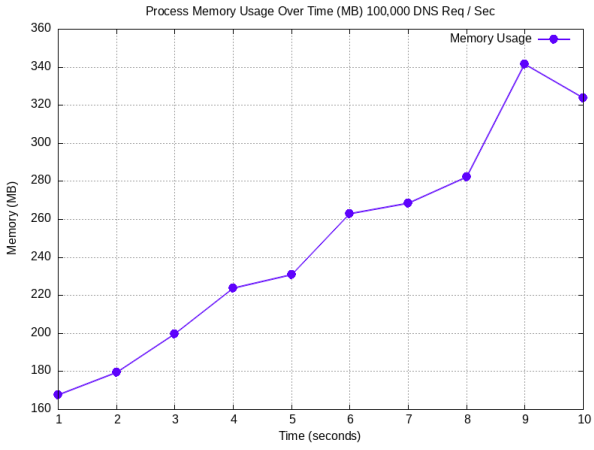


Figure 5.11: eBPF Agent: Process Memory Usage for 100k DNS req/sec

Effectiveness of eBPF Agent Against DNS Exfiltration Frameworks

The eBPF agent was evaluated across all data plane endpoints against widely used DNS-based C2 frameworks commonly employed by red teams. Tests focused on disrupting C2 and exfiltration activity tunneled through DNS, with CSSVLAB06 serving as the attacker node. The BishopFox Sliver framework was tested in agent's active mode for both basic exfiltration and advanced operations, including remote shell access, code execution, file transfers, port forwarding, and persistence, entirely over DNS. The agent intercepted the initial C2 packet at the kernel level, immediately disrupting communication, triggering retries, and terminating implants for both beacon-based and session-based channels. Since Sliver lacks randomized UDP port support, transport obfuscation was assessed using dnscat2. In both active and passive modes, the agent enforced mitigation policies, leading to full-session termination and zero data exfiltration. The average response time for each measured exfiltration attempt was 316.233 microseconds, as shown in Figure 5.12. For raw exfiltration, tools such as DET were immediately blocked. Furthermore, iodine, which tunnels through TUN/TAP interfaces, was effectively mitigated. All other bulk exfiltration tools were similarly thwarted. Table 5.2 summarizes the tools tested and the associated attack vectors prevented.

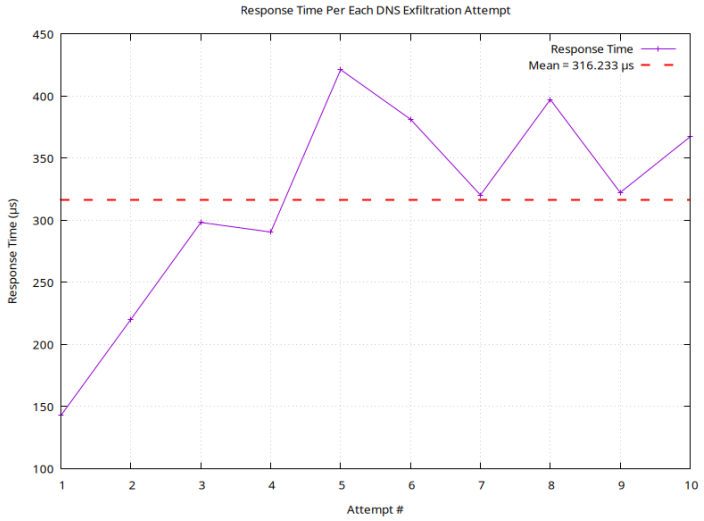


Figure 5.12: eBPF Agent: Response Time for Each DNS Exfiltration Attempt

Table 5.2: Framework Coverage Against Real-World DNS C2 and Exfiltration Tools

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		

5.2.2 Control Plane

The evaluation of the stateless controller server focuses on its effectiveness in accurately consuming threat events transmitted from data plane nodes to a Kafka topic, blacklisting domains in RPZ, and redistributing those events to data plane nodes to rehydrate their malicious domain caches. Figure 5.13 illustrates the structure of threat events streamed from eBPF agents in the data plane, serialized as JSON, and published to a Kafka topic. These events are consumed by the controller and used to blacklist domains in the RPZ zone of the DNS server. Figure 5.14 shows the published threat event structure 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 5.13: Controller consumed Threat Event

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


Figure 5.14: Controller streamed Threat Event

5.2.3 Distributed Infrastructure

Performance evaluation focuses on the DNS server, assessing the throughput impact of the Lua-based interceptor in the PowerDNS Recursor. As shown in Figure 5.15, which plots the benchmark duration on the horizontal axis, with throughput and packet loss on the two vertical axes, the server experienced a drop to 490 DNS requests per second under a load of 10,000 requests / sec. This degradation stems from reusing the same inference server design as the data plane, reliance on UNIX sockets, and Python's concurrency limits. Latency results in Figure 5.16 show the benchmark duration on the X-axis for a period of 20 seconds and associated latency on the Y-axis, peaking at 750 ms with a mean deviation of 380 ms. All TCP traffic benchmarks used `TCP_FAST_OPEN`, allowing data transmission with the initial SYN packet and reducing handshake overhead for accurate DNS-over-TCP latency

measurements. Figure 5.17 displays the blacklisted domains listed in the DNS RPZ table on the server.

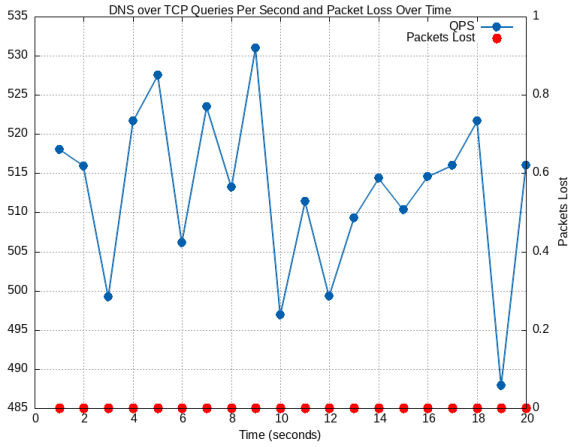


Figure 5.15: DNS Server Throughput for 10k DNS req/sec over TCP

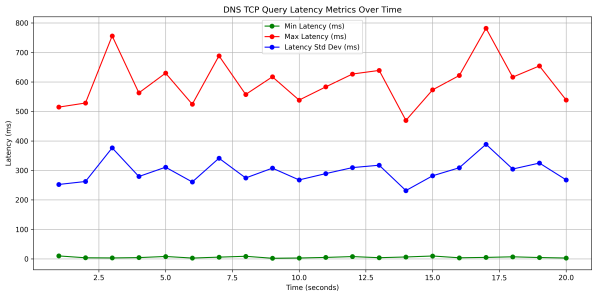


Figure 5.16: DNS Server Latency for 10k DNS req/sec over TCP

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

Figure 5.17: Blacklisted Domains in RPZ Zone on DNS Server

Chapter 6

CONCLUSION

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

6.1 Summary

This security framework significantly advances the state-of-the-art by introducing a novel architecture that prevents DNS-based data exfiltration while preserving horizontal scalability required in production cloud environments, directly addressing critical gaps left by traditional approaches. Existing approaches to DNS exfiltration prevention remain largely stagnant, relying on centralized detection, userspace anomaly systems, or proxy-based DPI, all inherently inadequate against sophisticated, real-world DNS-based attacks, particularly those that take advantage of advanced C2 vectors. In contrast, this framework introduces a new paradigm: kernel-enforced endpoint security. It acts as a privileged layer beneath existing endpoint security solutions, enabling strict enforcement inside the operating system. Using eBPF to reprogram the core kernel subsystems, paired with enhanced deep learning–based inference, this system-wide enforcement demonstrates comprehensive defensive strength: capable of detecting, stopping and killing the most advanced DNS C2 implants in real time, even against sophisticated attack vectors modeled by top-tier adversary emulation frameworks. Furthermore, by combining a layered approach with system security embedded with an endpoint-centric design, this architecture elevates endpoint defense beyond the limitations of userspace alone, providing detailed visibility into malicious activity, and rapid response to malicious implants. The following points summarize the strengths of the security framework.

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

6.2 Limitations

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

it the agent must track malicious activity to parent process from kernel task_struct rather over process id.

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

6.3 Future Work

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

BIBLIOGRAPHY

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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 Comput-

ing Machinery. ISBN 9781450360807. doi: 10.1145/3281411.3281443. URL https://doi.org/10.1145/3281411.3281443.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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`.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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`.

[22] 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`.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

Chapter 7

APPENDICES

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

7.1 Appendix A

This section focuses on providing additional security details implemented within the kernel to protect the kernel from malicious eBPF agents and system profiling of agents in data plane.

7.1.1 Kernel eBPF programs and userspace agent profiling

Given the intensive DNS parsing and enforcement logic implemented in the kernel TC layer, a comprehensive benchmarking was performed to evaluate its impact on the performance of

benign traffic. The documented source code can be found in the kernel/ directory, which includes logic to classify DNS payloads at line rate. Performance evaluation leveraged in-kernel perf instrumentation and system profiling tools. The eBPF program was executed concurrently across multiple CPU cores and was monitored using Netflix's bpftop tool. Under a high-throughput workload of approximately 100,000 DNS requests per second, kernel CPU usage peaked at 4% across eight cores, with a minimum of 2%. The program consistently maintained processing rates of up to 17,063 events per second - excluding hardware interrupt and soft IRQ overhead - demonstrating the efficiency of the eBPF logic attached to the TC kernel. Figure 7.2 illustrates the profiling results. In addition to kernel-level profiling, the eBPF userspace agent responsible for traffic handling was analyzed using Golang runtime profiler. CPU flame graph analysis revealed that approximately 25% of the agent's runtime was spent on BPF-related syscalls, primarily for monitoring and interacting with kernel-managed eBPF maps and ring buffers. These operations are performed using asynchronous I/O via epoll, significantly reducing the overhead compared to busy polling for metrics from userspace. This distribution reflects the agent's dependency on frequent kernel interactions for accessing and updating eBPF map state. Figure 7.1 illustrates the resulting flame graph.

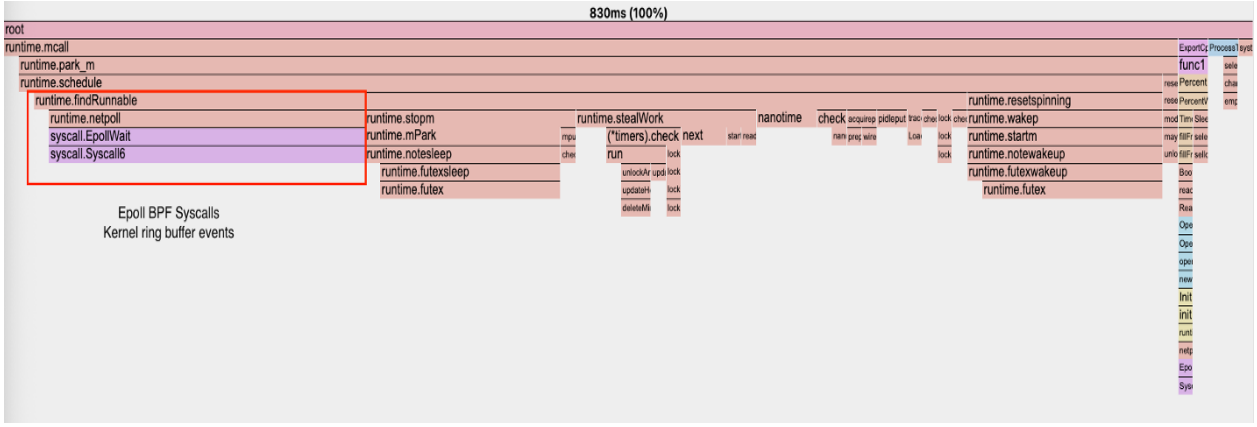


Figure 7.1: eBPF Agent: Flame Graph

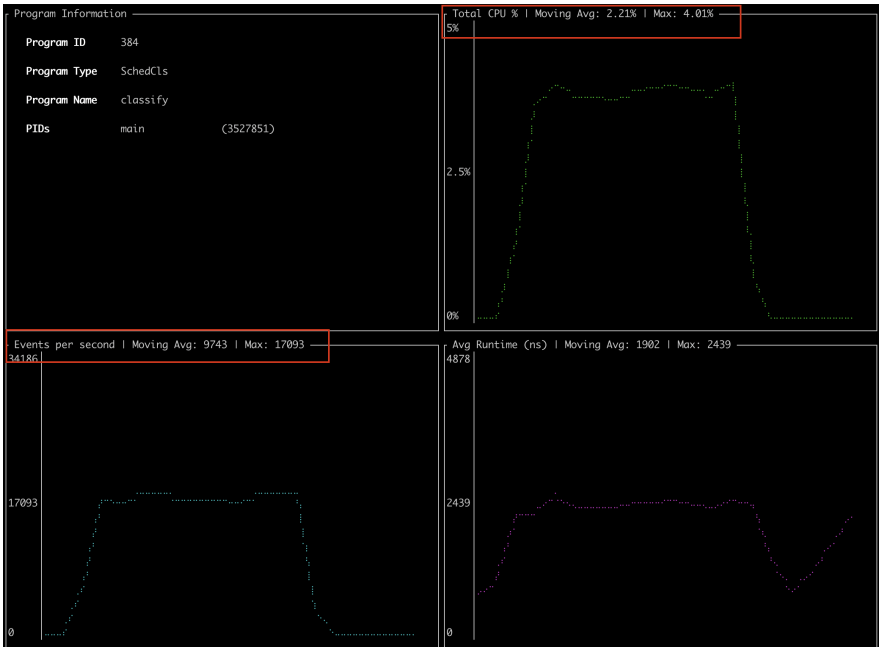


Figure 7.2: Kernel eBPF Programs Profiling

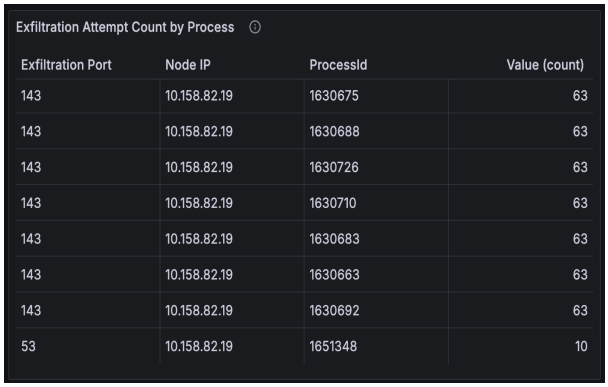
7.1.2 eBPF Agent exported metrics

Table 7.1 describes all the metrics exported by each eBPF agent at the endpoint in data plane, with Figure 7.3, Figure 7.4, Figure 7.5 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.

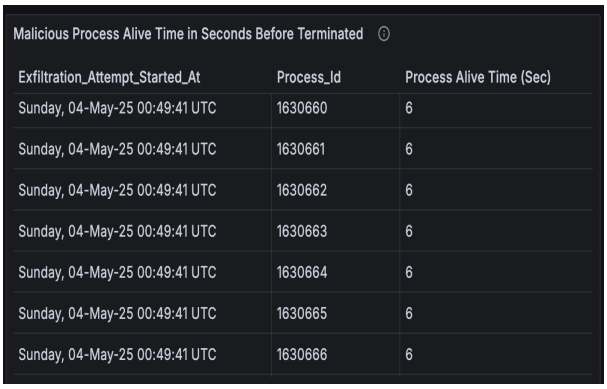
Table 7.1: eBPF agent exported metrics in both active and passive modes

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.

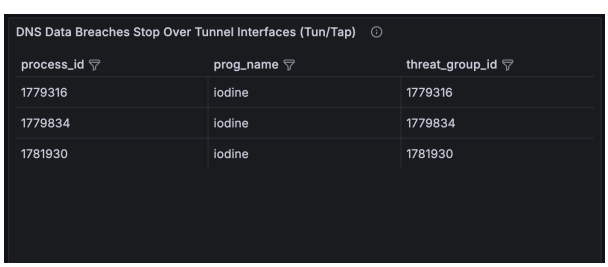


(a) Exfiltration Attempts Prevented per Process

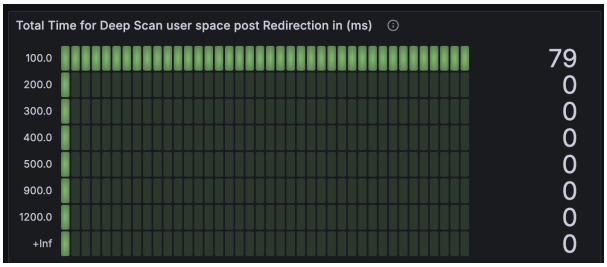


(b) Process Alive Time Before Termination

Figure 7.3: DNS Exfiltration Prevention Metrics: Malicious Process Alive Activity



(a) Tunnel Interface Exfiltration Metric



(b) Latency in Active Redirect Mode

Figure 7.4: DNS Exfiltration Prevention Metrics: kernel network encapsulation and Latency

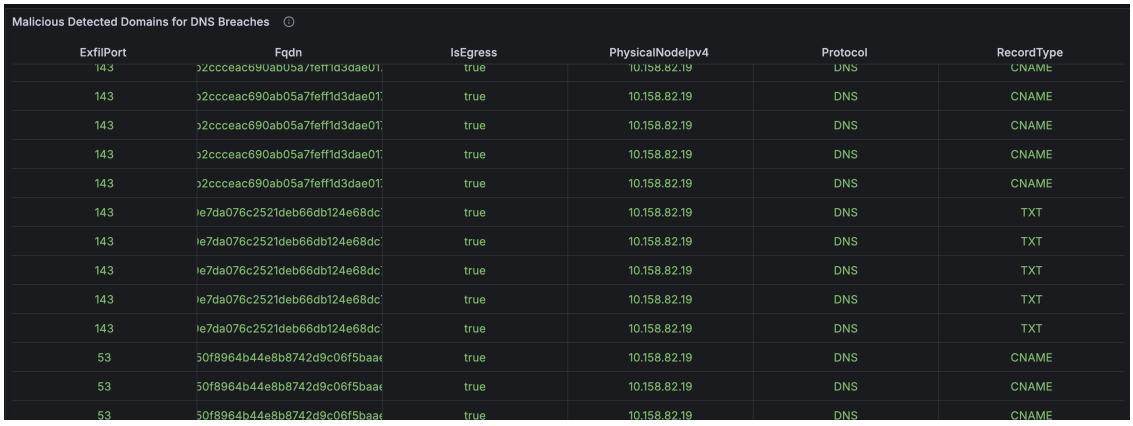


Figure 7.5: Metrics of Prevented DNS Exfiltrated Packets

7.1.3 Protecting Linux Kernel from malicious eBPF programs

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

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

Listing 7.1: Kernel BPF LSM Hook for PKCS7 Signature Verification

```
BPF_PROG(bpf, int cmd, union bpf_attr *attr, unsigned int size) {
    if (cmd != BPF_PROG_LOAD)
        return 0;
    // Look up eBPF program, its original signature, and the modified
       signature
    mod_sig = bpf_map_lookup_elem(&modified_signature, &zero);
    orig_data = bpf_map_lookup_elem(&original_program, &zero);
    combined_buf = bpf_map_lookup_elem(&combined_data_map, &zero);
```

```
    // Copy eBPF program and original signature into a combined buffer
    insn_len = attr->insn_cnt * sizeof(struct bpf_insn);
    bpf_copy_from_user(combined_buf->data, insn_len, attr->insns);
    bpf_probe_read_kernel(combined_buf->data + insn_len, orig_data->
        sig_len, orig_data->sig);
    // Create dynptrs for PKCS7 verification
    // dynptrs work similar to kptr but allowing to point to buffer
        location storing large amount of data as in case of eBPF signed
        paylods to satisfy 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
    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 redi-

rected netflow packet, enabling integrity verification at the receiving bridge interface. This bridge, monitored by the eBPF agent in userspace, receives the redirected traffic for further analysis of suspicious behavior. An additional ingress TC eBPF program is attached to the bridge interface. Inspect incoming SKB 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 7 explains the algorithm integrity verification check on the SKB attached to the TC ingress on the bridge.

Algorithm 7: SKB Integrity Verification and Secure Redirection in **Active** Mode

Input : `skb` (socket buffer)
eBPF map: `skb_netflow_integrity_verify_map`
Output : `TC_ACT_OK` — packet allowed
`TC_ACT_SHOT` — drop tampered packet
; `/* Unique mark used in core TC program on each physical netdev */`
1 Fetch `skb_mark_integrity_mark` from `skb_netflow_integrity_verify_map` with:
Key: `0xFFEF`
2 **if** *skb_mark_integrity_mark* $\neq$ *skb->mark* **then**
`/* Tampered or spoofed redirection; reject packet */`
3 **return** `TC_ACT_SHOT`;
4 **return** `TC_ACT_OK` ; `/* Packet is legitimate */`

7.2 Appendix B

This section provides additional internal details about Lua interceptor to filter malicious DNS over TCP on DNS server and DGA used to generate domains in malicious C2 and tunneling activities.

7.2.1 PowerDNS TCP Lua interceptor

Algorithm 8 details the algorithm implemented in PowerDNS Recursor as a DNS query interceptor.

Algorithm 8: PowerDNS Malicious DNS over TCP Filtering 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 ;                          /* Block malicious DNS over TCP */
8  if sf_blacklist.check(getSLD(qname)) then
9  |   dq.rcode ← NXDOMAIN;
10 |   return true ;                    /* Block known SLD from local blacklist */
11 return false ;                                 /* Allow query if not malicious */
```

7.2.2 Malicious domain generation

To carry out advanced DNS C2 attacks or tunneling with open-source C2 tools, a DNS server configured with custom DNS zones (SOA) is required. These zones must include NS, A, AAAA, and glue records pointing to the malicious C2 server's IP. In this framework, PowerDNS serves as the DNS infrastructure that supports such attacks. A custom script simulates a sample DGA by generating random SLDs, selecting random top-level domains (TLDs), and adding a third label identifying the C2 tool. The script also generates PowerDNS recursor forwarder configurations to redirect specific queries to a custom authoritative PowerDNS server and creates the necessary zone files using pdnsutil. By DNS design, each label requires a dedicated zone file with an NS record delegating the next label, enabling hierarchical resolution through glue records. Currently, the DGA only mutates domain names and does not implement IP (L3) address mutations, such as multiple A/AAAA records for DNS-based load balancing across C2 nodes. This extension is possible with additional infrastructure. Despite the lack of L3 mutation, the framework controller remains effective by enforcing dynamic domain blacklists and applying in-kernel network policies for cross-protocol correlation, blocking data exfiltration to dynamically generated domains and IPs. In particular, most real-world advanced C2 and multiplayer frameworks do not rely on forwarding DNS queries through separate DNS servers. Instead, implants communicate directly with C2 servers that run their own DNS service on standard or random UDP ports. For

example, Sliver forces DNS servers to forward all queries directly to the C2 DNS server, eliminating intermediate DNS hops. The DGA implementation details are explained below.

Listing 7.2: Domain Generation Algorithm

```
# generate number of malicious C2 server domains and add create zones,
    child zones NS links inside DNS server
# all the attacker tools to use
# this attacker tool also generate the third label of C2 domain
exfil_tools: List[str] = ['dnscat', 'sliver', 'iodine', 'det']
# get the random TLD
def gen_randomTLD() -> str:
    return random.choice(['live','com','de','io','se','bld', 'val', 'def',
        'head'])
def DGA_MASS_DOMAINS_GEN(args):
        r = RandomWord()
        dga = gen_c2_exfil_domains(tldDomains=[base64.b64encode(r.word()).
            lower() + "." + gen_randomTLD()
                                        for _ in range(1 << int(args.count)
                                            )],
                                c2_tool_domains=exfil_tools)
        ff = open(DGA_FILE, 'w', encoding='utf-8')
        ff.write('\n'.join(dga))
        append_zone_data_in_zoneFiles(dga) # create zone and child zones
            for PowerDNS Authoritative zone
        gen_exil_forward_zones_file(dga) # generate the forward zone file
            for PowerDNS Recursor
```