Internship Project Report

Submitted in partial fulfillment of the requirement for
the award of a certificate of internship programme
Under ISEA-III Project

Enhancing Android Kernel Observability Using eBPF

Submitted by

AAYUSH MEEL

23UELET001

CENTRAL UNIVERSITY OF KARNATAKA

Under the supervision of

Professor Vijay Laxmi
Principal Investigator, ISEA-III
Department of Computer Science and Engineering

Malaviya National Institute of Technology Jaipur

Jawahar Lal Nehru Marg, Jaipur-302017 (Rajasthan) INDIA

ABSTRACT

Title: Enhancing Android Kernel Observability Using eBPF: A Comprehensive Analysis of System Monitoring, Performance Profiling, and Security.

The architectural complexity of the Android operating system poses significant challenges for traditional observability methods, which often lack the real-time fidelity and low overhead required to diagnose transient performance issues or detect sophisticated threats. This report investigates the application of the extended Berkeley Packet Filter (eBPF), a technology that enables safe, high-performance, programmable observability within the Linux kernel, as a solution to these challenges.

The primary objective was to apply eBPF for enhanced system monitoring, performance profiling, and security analysis on Android. The methodology involved leveraging the BPF Compiler Collection (BCC) toolkit to instrument kernel-level events, including system calls and hardware performance counters.The implementation yielded powerful, practical tools. For system monitoring, execsnoop provided real-time visibility into process execution. For performance analysis, system-wide stack trace sampling was used to generate Flame Graphs, effectively identifying CPU bottlenecks. In security, tcpconnect enabled the real-time tracing of outbound network connections for anomaly detection. These results, inspired by frameworks like BPFroid, confirm eBPF's capacity for robust, in-depth system analysis.

This report concludes that eBPF is a powerful technology for enhancing Android kernel observability. While tooling dependencies present challenges for wide-scale deployment, the industry's shift towards portable solutions like libbpf and "Compile Once - Run Everywhere" (CO-RE) promises to make these capabilities more accessible. The findings affirm that eBPF is an indispensable tool for engineering the next generation of secure and high-performance mobile systems.

Table of Contents

List of Figures

List of Tables

Chapter 1

This chapter outlines the urgent need for next-generation system observability across contemporary mobile operating systems, specifically Android. It sets forth that the growing complexity of the Android software stack from kernel to app layer poses huge challenges for traditional monitoring approaches. The traditional methods tend to create "observation blind spots," and thus it becomes challenging to identify transient performance bottlenecks or intercept sophisticated security attacks. The chapter suggests that proper observability calls for greater, more detailed insight into the internal state of the system than can be had at the moment.   The conversation then focuses on the Android kernel, the platform's foundation. It outlines the evolution of the kernel, pointing out the long-standing issue of fragmentation and Google's architectural remedy, the Generic Kernel Image (GKI) initiative. Although GKI serves to normalize the underlying kernel, the intrinsic difficulty of controlling all system resources still exists, requiring an observability solution that is both effective and safe.Against this context, the chapter presents extended Berkeley Packet Filter (eBPF) as revolutionary technology.
eBPF is painted as a light-weighted, sandboxed kernel virtual machine that can execute user-provided programs securely at different hook points, e.g., system calls and network events. The fundamental difference between eBPF and more dangerous approaches such as Loadable Kernel Modules (LKMs) is that eBPF programs are subject to a very tight verification process that ensures that they cannot cause the kernel to crash or make it vulnerable. This combination of security, low performance overhead, and in-depth access to the kernel places eBPF as the perfect technology to solve Android's observability problem. Lastly, the chapter describes the report's main aim: to explore and prove the real-world usage of eBPF for monitoring the system, performance profiling, and security analysis on the Android platform.

1.1 Introduction

The Need for Advanced Observability in Mobile Operating Systems :

Contemporary mobile operating systems, led by Android, have become incredibly sophisticated ecosystems. The Android software stack has several layers, ranging from the Linux kernel controlling hardware resources at the bottom to the application framework and applications at the top. Such sophistication, although allowing for deep functionality, creates enormous system administration, performance engineering, and security analysis challenges. Legacy monitoring techniques, which typically depend on static counters, system logs, and sporadic sampling, are often inadequate to identify transient performance problems or identify subtle security attacks in such an environment.

1.2 Motivation

The critical need for enhanced Android observability stems from three fundamental gaps in current mobile system management:

1. **Security Vulnerabilities**: Growing sophistication of mobile malware that evades traditional detection methods, creating undetected persistence mechanisms and data exfiltration channels
2. **Performance Blind Spots**: Inability to diagnose transient latency issues affecting user experience, particularly during resource contention scenarios
3. **Diagnostic Limitations**: Heavyweight monitoring tools that alter system behavior during analysis, creating observer-effect distortions

eBPF addresses these gaps by providing:

1. Real-time kernel visibility without system modifications
2. Cross-layer correlation from hardware events to application behavior
3. Near-zero overhead instrumentation suitable for production devices
4. Safe programmability through verifier-enforced execution guarantees

This project is motivated by the urgent need to bring enterprise-grade observability to the Android ecosystem, enabling:

1. Proactive threat detection against evolving mobile attack vectors
2. Granular performance optimization for resource-constrained devices
3. Validated security hardening through continuous kernel monitoring

1.2.1 The Android Kernel: A Complex Ecosystem

The foundation of the Android platform is its kernel, which is based on an upstream Linux Long Term Supported (LTS) kernel. However, it is not a vanilla Linux kernel. Google maintains a set of Android Common Kernels (ACKs) that include a superset of patches not yet merged into the mainline Linux kernel but deemed critical for the Android ecosystem. Historically, this architecture has been susceptible to severe fragmentation. Device manufacturers (OEMs) and System-on-a-Chip (SoC) vendors would extensively modify the kernel to support their specific hardware, leading to a multitude of divergent kernel versions. This fragmentation made it exceedingly difficult to deploy system-wide updates, apply security patches consistently, and maintain a common baseline for monitoring and analysis tools.

To address this, Google initiated the Generic Kernel Image (GKI) project, a significant architectural overhaul that aims to separate the hardware-agnostic core kernel from hardware-specific vendor modules. While GKI mitigates fragmentation, the underlying complexity of the kernel remains. It is the central arbiter for all system resources, managing processes, memory, power, and drivers for a vast array of hardware components. Any tool aiming to provide comprehensive observability must be able to navigate this intricate environment safely and efficiently, without introducing instability or performance degradation.

1.2.2 Introducing eBPF: A Paradigm Shift in Kernel Programmability

Extended Berkeley Packet Filter (eBPF) has emerged as a revolutionary technology that fundamentally changes how developers and administrators can interact with a running kernel. It functions as a lightweight, sandboxed virtual machine embedded within the Linux kernel, allowing user-supplied programs to be safely executed at various hook points, such as system calls, function entries, and network events. Unlike traditional Loadable Kernel Modules (LKMs), which can introduce instability and crash the entire system if buggy, eBPF programs are subjected to a rigorous verification process before being loaded. This process guarantees that the programs will terminate, will not access unauthorized memory, and cannot harm the kernel's integrity.

This capability for safe, dynamic kernel programmability positions eBPF as a superior alternative to other observability techniques. It avoids the intrusiveness of Application Performance Monitoring (APM) agents, which often require source code modification or application restarts to instrument code. It also offers significantly lower performance overhead compared to mechanisms like ptrace, which rely on frequent and costly context switches between kernel and user space. In essence, eBPF provides a unified, powerful, and low-impact framework for creating a new generation of tooling for networking, security, and observability, making it an ideal candidate for enhancing visibility into the Android kernel.

1.3 Objectives

The main goal of this report is to carry out an extended study on the use of eBPF to increase observability on the Android platform. The study will cover three important areas: system monitoring, performance profiling, and security inspection. The report is not just expected to have a theoretical base but also include implementation information and analysis, which shows the strength and diversity of eBPF in a mobile scenario.

1.4 Report Outline

To achieve this objective, the report is structured into five chapters:

Chapter 2 presents the basic concepts and literature survey of the same, including the contemporary Android kernel design, the technical background of the eBPF system, its existing situation in AOSP, and a survey of development toolchains and research work.

Chapter 3 presents the methodology to this project, setting out the observability problems on Android and laying out the real-world steps towards the setup of a development and tracing environment.

Chapter 4 gives the main implementation and findings of the research. It shows the deployment of particular eBPF-based tools to trace system calls, profile CPU performance, and monitor network security, including code analysis and example output from a live Android phone.

Chapter 5 summarizes the report by integrating the main findings, analyzing the current limitations and challenges of eBPF usage on Android, and outlining future prospects for the technology in the mobile ecosystem.

Chapter 2

This Chapter provides the theoretical and technical foundation for conceptualizing the use of eBPF in the Android environment. It starts by disassembling the contemporary Android kernel, describing how it progressed from a splintered environment to a more modular design under the Generic Kernel Image (GKI) initiative. This architectural change is essential, since it isolates the vendor-agnostic kernel core from vendor-specific modules through a stable Kernel Module Interface (KMI), providing an even more predictable and stable platform for sophisticated instrumentation tools such as eBPF. The chapter goes on to deliver an in-depth exploration of the eBPF architecture itself, introducing it as an innovative in-kernel execution environment. The essence of eBPF design is its focus on safety and performance. This is accomplished primarily by two very important elements: the Verifier and the Just-in-Time (JIT) Compiler. Before loading any eBPF program, the Verifier conducts a very strict static analysis of its bytecode. It builds a control flow graph to guarantee that the program will terminate always and runs through every possible execution path to ensure that there are no out-of-bounds memory accesses or other unsafe operations that may cause the kernel to crash. After verification, the bytecode is compiled into native machine code by the JIT compiler to enable it to execute with near-native performance, an important benefit over other tracing mechanisms. The chapter also outlines the fundamental mechanisms that make eBPF programs possible: eBPF maps, which are optimized key/value stores to store state and communicate data with user-space apps, and helper functions, which offer a stable, vetted API for eBPF programs to interact with the kernel in a safe way.

Summary then turns to the real-world uptake of eBPF in the Android Open Source Project (AOSP). It points out that Google is already using eBPF for key system functionality, most prominently the network traffic monitor, which supplanted the ancient xt_qtaguid module. Not only does this update bring the functionality into the modern era, but it also follows the GKI philosophy by codifying a core OS feature in a manner that is vendor hardware module-independent. A survey of current eBPF toolchains compares the two major development methods. The BCC is introduced as an effective system for quick prototyping but with its significant runtime reliance on the Clang/LLVM toolchain, which makes it unsuitable for production-ready Android devices. On the contrary, the new method via libbpf library and the "Compile Once - Run Everywhere" (CO-RE) model is recognized as the essential enabler of viable, large-scale eBPF deployment on Android. CO-RE uses BPF Type Format (BTF) metadata to enable small, pre-compiled eBPF programs to execute on various kernel versions without alteration, eliminating on-device compilation. Lastly, the chapter summarizes academic research that demonstrates eBPF's potential on mobile devices. The BPFroid system is featured as a prime example of a new dynamic analysis system for real-time Android malware detection based on eBPF to trace events throughout the complete software stack, from kernel syscalls to Java API calls. Additional studies on performance analysis for UI responsiveness and programmable system call filtering for security hardening further highlight the versatility of eBPF and its capability for addressing intricate issues in the mobile area.

2.1 Background and Literature/Technology Review

2.1.1 Deconstructing the Modern Android Kernel: From ACK to GKI

The Android kernel is a Linux kernel that has been optimized for the particular demands of mobiles. Its origin lies in an upstream Linux Long Term Supported (LTS) kernel, which is released annually and serves as a stable foundation. Google then combines these LTS kernels with patches that are Android-specific to create what are known as Android Common Kernels (ACKs). These ACKs, which reside in the kernel/common repository, serve as the basis for releases of the Android platform and include features and patches that have not yet been included in the mainline Linux kernel. One of the most important landmarks in the kernel design of Android is the Generic Kernel Image (GKI) effort, which addresses the age-old issue of kernel fragmentation. For kernels from version 5.10 onwards (on aarch64 platforms), the GKI design enforces rigorous isolation of the core, hardware-agnostic kernel code from hardware-specific code provided by SoC and device makers. This isolation is achieved by three principal components:

GKI Kernel: It is a boot image that has been verified by Google with the core kernel functionality common to all devices. It is built directly from an ACK source tree and must be flashable on all devices.

Vendor Modules: They are partner-created hardware-specific modules, i.e., SoC or other peripherals' drivers. They are shipped as Dynamically Loadable Kernel Modules (DLKMs) in .ko format and loaded on demand at boot time by the device.

Kernel Module Interface (KMI): The KMI is the fundamental contract under which the GKI kernel and vendor modules can be independently updated. It is a stable, exported set of kernel functions and global data, defined by symbol lists, that can be utilized by vendor modules. This interface is strictly versioned and maintained backward compatible. The KMI lifecycle is managed via isolated stages—development, stabilization, and frozen—to prevent breaking changes after a platform release. As an example, an alteration that adds a new field in a structure used by a KMI function is not allowed after the interface is frozen because it would alter the interface definition and make vendor modules that are already deployed incompatible. This modular design not only makes security update delivery simpler by allowing Google to change itself independently without vendors needing to act, but it also yields a more predictable and stable target for system-level tools, like in eBPF form.

2.1.2 The eBPF Architecture: An In-Kernel Revolution

eBPF is not merely a tool but a complete in-kernel execution environment. It allows developers to write small, event-driven programs that can be attached to various hook points within the kernel to intercept and process data. This provides a safe and highly efficient mechanism for extending kernel functionality at runtime. The architecture of eBPF is built upon several core components that work in concert to ensure both safety and performance.

2.1.3 The Verifier, JIT Compiler, and Safety Guarantees

The life cycle of an eBPF program starts with high-level code, often written in some limited subset of C. It is compiled by a toolchain such as Clang/LLVM into eBPF bytecode. A user-space application loads this bytecode into the kernel via the bpf(2) system call. The program has to go through two key steps that constitute the basis of eBPF's safety model before it can be run: verification and JIT compilation.

The Verifier: This is perhaps the most important part of the eBPF system architecture. It is a static analysis engine that makes a strict pass over the eBPF bytecode to ensure that it is secure to execute. The main responsibility of the verifier is to make sure that a loaded program does not crash the kernel, read arbitrary memory, or run into an infinite loop. It achieves this by executing two primary passes:

Control Flow Graph (CFG) Check: The verifier initially builds a directed acyclic graph of every potential execution path in the program. It does a depth-first search to determine if there are any non-terminating loops that cannot be proven. Although current eBPF has bounded loops, the verifier has to be able to prove that an exit condition will always be satisfied. This check stops programs from causing the kernel to hang. Static Analysis of Bytecode: In the second pass, it emulates the run of the program path by path, keeping tabs on the state of all registers and stack positions. It checks that all memory references are valid (e.g., no null pointer dereferences, no out-of-bounds read/writes) and that types are consistently used (e.g., a scalar constant is never used as a pointer). It also imposes complexity bounds on the number of instructions and jumps. It is only if a program succeeds at all of these checks that it is considered safe to continue. This verification phase is what makes eBPF inherently safer than traditional kernel modules, which enjoy no such pre-execution safety assurances.

Just-in-Time (JIT) Compiler: Once verification has been successful, the eBPF bytecode is compiled to native machine code for the host CPU architecture by a JIT compiler. This step is critical for performance. It enables eBPF programs to run at rates that are close to native compilation of kernel code, so the performance overhead of eBPF-based tracing and monitoring is very low. This is one reason why eBPF can be used for high-throughput network operations and continuous system-wide profiling without degrading system performance.

2.1.4 eBPF Maps and Helper Functions: State and Control

eBPF programs do not operate in a vacuum. They require mechanisms to store state, share data, and interact with the kernel in a controlled manner. This is accomplished through eBPF maps and helper functions.

eBPF Maps: These are efficient key/value data structures that reside in kernel space. They are the primary means of communication and data sharing for eBPF programs. Data can be shared between different eBPF programs or passed between an eBPF program in the kernel and a controlling application in user space. The eBPF framework supports a wide variety of map types, including hash tables, arrays, per-CPU arrays, stack traces, and high-performance ring buffers (BPF_MAP_TYPE_RINGBUF) for sending event data to user space with minimal overhead.

Helper Functions: An eBPF program is sandboxed and cannot call arbitrary kernel functions. Instead, it is provided with a stable, well-defined API of "helper functions". These helpers provide access to a curated set of kernel functionalities, such as looking up or updating data in an eBPF map, getting the current time or process ID, manipulating network packets, or reading kernel and user memory safely. This stable API ensures that eBPF programs remain compatible across different kernel versions, even if internal kernel functions change.

Hooks: eBPF programs are event-driven. They are attached to specific "hooks" within the kernel and only execute when the corresponding event occurs. The range of available hooks is extensive and includes:

Kprobes/Kretprobes: Dynamic instrumentation of the entry and exit points of almost any kernel function.

Uprobes/Uretprobes: Dynamic instrumentation of functions in user-space applications and libraries.

Tracepoints: Stable, low-overhead static instrumentation points placed at logical locations in the kernel source code (e.g., for system calls).Network Hooks: Attachment points in the networking stack, such as XDP (eXpress Data Path) for high-speed packet processing at the driver level and TC (Traffic Control) for packet manipulation higher in the stack.

This combination of a verifier, JIT compiler, maps, helpers, and a rich set of hooks makes eBPF a uniquely powerful and safe framework for deep kernel observability.

Table 2.1: Comparison of Kernel Instrumentation Techniques

Technique	Safety Mechanism	Performance Overhead	Flexibility/Programmability	Deployment Complexity
Loadable Kernel Modules (LKMs)	None; direct kernel access, bugs can crash the system	Low; runs as native kernel code	Very High; can access and modify any part of the kernel	High; requires kernel rebuilds for version changes, unstable ABI
ptrace	High; operates within the process security model	Very High; requires multiple context switches per syscall	Low; primarily limited to intercepting system calls	Medium; user-space daemon required for each traced process
eBPF	Very High; static verifier, sandboxed VM, controlled memory access	Very Low; JIT-compiled to native code, runs in-kernel	High; programmable logic with a wide range of hooks	Low-Medium; user-space tooling simplifies development and loading

2.1.5 The State of eBPF in the Android Open Source Project (AOSP)

Although eBPF is a Linux kernel technology in general, its usage in the Android Open Source Project (AOSP) is intentional and an increasing trend. Google has adopted eBPF to retire less maintainable legacy kernel pieces and to add new observability features, reflecting its commitment to the technology as an integral component of the platform's future. The highest-profile official usage case is the network traffic monitor. From Android 9, devices shipping with kernel 4.9 or later must utilize an eBPF-based system for network traffic accounting, overriding the legacy xt_qtaguid kernel module. This new system is not a simple feature replacement; it is a strategic architectural alignment. The xt_qtaguid module was an out-of-tree kernel patch that worked towards the fragmentation the GKI project seeks to eradicate. By introducing the traffic monitor using eBPF, Google is able to deliver this key feature as part of the generic, common kernel, which is in total alignment with the GKI philosophy of keeping core OS features separate from vendor-specific hardware drivers. The deployment makes use of a user-space process, trafficController, that, during boot time, initializes eBPF maps and tells the bpfloader process to load a precompiled eBPF program. The program hooks into a cgroup hook to assign network packets to the proper application UID, while an xt_bpf netfilter module assists in assigning traffic to the proper network interface. In addition to monitoring, this system also includes firewall functionality by filtering traffic from certain UIDs based on phone state (for example, power-saving modes), replacing the legacy xt_owner module. Aside from network monitoring, eBPF is now officially utilized by AOSP for other observability workloads. The time_in_state program monitors how much time the processes of an application spend at various CPU frequencies, which offers information for power usage analysis. For Android 12 and subsequent versions, the gpu_mem program monitors GPU memory consumption per-process and system-wide.

In support of these integrations, Android has a special BPF loader and library. All eBPF programs in /system/etc/bpf/ are automatically loaded at boot time. It sets up the required maps and pins both the programs and maps to the BPF filesystem at /sys/fs/bpf/. Other system services or user-space tools can then access them later. The Android BPF library, libbpf_android.so, offers low-level C++ APIs for such interactions, for example, obtaining a file descriptor for a pinned map or attaching a loaded program to a tracepoint.

2.1.6 Survey of eBPF Toolchains: BCC vs libbpf and CO-RE

It takes a toolchain to compile, load, and interact with eBPF programs in user space. The toolchain ecosystem has dramatically changed, with an unmistakable trend towards more light, transportable solutions.

BPF Compiler Collection (BCC): BCC was among the first and the most widely used frameworks for developing eBPF, especially tracing and performance analysis. It is a full-featured toolkit consisting of a library and more than 50 pre-built tools such as execsnoop and opensnoop. The main programming interface for BCC is a Python front-end (with Lua as an alternative) where programmers can embed their eBPF C code as a string in a Python script. At runtime, BCC employs its built-in Clang/LLVM libraries to translate the C code into eBPF bytecode, which it loads into the kernel. This solution is versatile for developer machine rapid prototyping and interactive analysis but has one significant disadvantage: heavy runtime dependency: the target machine needs to have the complete Clang/LLVM toolchain and proper kernel headers installed. In resource-limited and production-hardened platforms such as Android devices, releasing a multi-hundred-megabyte compiler toolchain is extremely impractical, slow, and greatly expands the system's attack surface. libbpf and CO-RE: The constraints of BCC resulted in a more contemporary and light-weight approach based on the libbpf library and the "Compile Once - Run Everywhere" (CO-RE) paradigm. libbpf is a C/C++ library stored in the upstream Linux kernel source tree that manages the loading and manipulation of eBPF object files. The fundamental innovation it makes possible is CO-RE. Rather than runtime compilation, a CO-RE application is compiled ahead of time to a small, self-contained binary. This is facilitated through the use of BPF Type Format (BTF), a metadata format that represents the types and structures in the kernel. The developer builds their eBPF program just once on their development system. When this program is executed on a target platform, libbpf relies on the BTF data from the running kernel (unveiled at /sys/kernel/btf/vmlinux) to handle runtime relocations. It cleverly maps the memory access offsets of the eBPF program to the actual layout of kernel structures on that target, rendering the program portable between various kernel versions and configurations. This transition from the BCC model to the libbpf/CO-RE model is not one of incremental refinement; it is the underlying technological leap that makes broad, production-quality eBPF observability on Android feasible. It obviates on-device compilation, with much smaller, faster, and more secure deployment. Tools like libbpf-bootstrap offer shell templates for creating these portable binaries, and eunomia-bpf seeks to further reduce the complexity by bridging libbpf with WebAssembly for distribution even more easily.

2.1.7 Academic Perspectives: eBPF for Mobile Security and Performance

The academic community has begun to explore the unique potential of eBPF in the context of mobile and Android systems, producing research that pushes the boundaries of what is possible with in-kernel instrumentation.

A standout example is BPFroid, a novel dynamic analysis framework designed specifically for real-time Android malware detection. The framework's strength lies in its ability to use eBPF to monitor events across the entire Android software stack simultaneously, from low-level kernel functions and system calls to native library functions and even Java API framework methods executed by the Android Runtime (ART). BPFroid operates without modifying the Android system image or the application package, making it robust against tampering and difficult for malware to detect. The research demonstrates practical detection signatures, such as a "dropper" detector that hooks the vfs_write kernel function to inspect the magic bytes of newly written files, identifying ELF, DEX, or APK files being created on the filesystem in real-time. The BPFroid paper provides a compelling case for eBPF as a superior foundation for mobile security tools, offering a combination of deep visibility, robustness, and low performance overhead that previous methods struggled to achieve.

Beyond security, researchers are applying eBPF to address performance challenges unique to mobile devices. The paper "Rethinking Process Management for Interactive Mobile Systems," presented at MobiCom'24, leverages eBPF to investigate the root causes of slow UI responsiveness in Android applications. By using eBPF to precisely measure the usage of hardware resources (CPU, memory, I/O) by different application processes, the researchers were able to gain fine-grained insights into performance bottlenecks that are difficult to capture with traditional profiling tools. This work highlights eBPF's utility not just for system-wide analysis, but for targeted, application-specific performance engineering on mobile platforms.

Furthermore, research into programmable system call filtering with eBPF, while not exclusively focused on Android, has significant implications for hardening the platform's security model. This work demonstrates how eBPF can be used to create advanced, stateful seccomp policies that go far beyond simple allow/deny lists. For example, an eBPF filter could implement rate-limiting on specific syscalls or enforce complex state transitions, providing a much more granular and context-aware security boundary for applications. These academic explorations underscore the vast potential of eBPF to address long-standing and emerging challenges in mobile system security and performance.

Chapter 3

Chapter 3 shifts from theoretical constructs to the down-to-earth methodology needed to implement eBPF for observability on the Android platform. It starts with a rigorous definition of the fundamental challenge: to create and execute a practical, secure, and lightweight approach to instrumenting a running Android kernel. The aim is to collect high-fidelity, real-time data for system monitoring, performance profiling, and security analysis without altering the core Android system or its applications. To overcome this difficulty, the chapter proposes a rigorous, four-stage process that acts as a blueprint for every eBPF implementation described in the report. This process guarantees a systematic approach to tool creation:

Identify Target Event: Locating the precise kernel event (e.g., a function entry, system call, or timer tick) that yields the required information.

Create eBPF Program: Implementing the in-kernel C code that binds to the event hook, gathers data, and formats it to be sent to user space. Create User-Space Controller: Implementing the user-space program (often Python for the BCC toolkit) that will load the eBPF program, handle data transfer, and interpret the outcome.

Analyze and Visualize: Interpreting the gathered data to infer useful insights, be it through text output, statistical summary, or charts such as Flame Graphs. The rest of the chapter explains the most important and complicated pre-requisite for this book: setting up a working development and tracing environment. It points out that stock Android kernels on production hardware are usually incomplete, as they do not include the debugging and tracing functionality necessary. Hence, a kernal compiled with specific configuration flags like CONFIG_BPF_SYSCALL, CONFIG_KPROBES, and CONFIG_DEBUG_INFO_BTF=y is necessary to support the core eBPF subsystem, dynamic probing, and new CO-RE features. Having fixed the kernel prerequisites, the chapter then describes the hands-on configuration through the adeb (Android Debug Bridge enhanced) tool. adeb is introduced as the bridge that enables the running of the BCC toolchain on an Android device. It accomplishes this by establishing a chroot environment on the device, which includes a minimal Debian filesystem, and enables normal Linux development tools to be run against the Android kernel. The summary reports the major setup command, adeb prepare --full --kernelsrc /path/to/kernel-source/, which not only installs the filesystem onto the device but, importantly, compiles the exact kernel headers needed by BCC from a local kernel source tree. This process is important for establishing compatibility and preventing compilation issues. Prepared, the adeb shell command gives access to this rich tracing environment, paving the way for the application-level implementations described in Chapter 4.

3.1 Problem Statement of the Task Assigned

3.1.1 Defining the Android Observability Challenge

The primary challenge in achieving deep observability on Android is to gain visibility into kernel and system-level activities without compromising the platform's stability, security, or performance. As established, traditional tools often provide an incomplete picture, while more invasive methods carry significant risks. The problem statement for this project is therefore defined as follows:

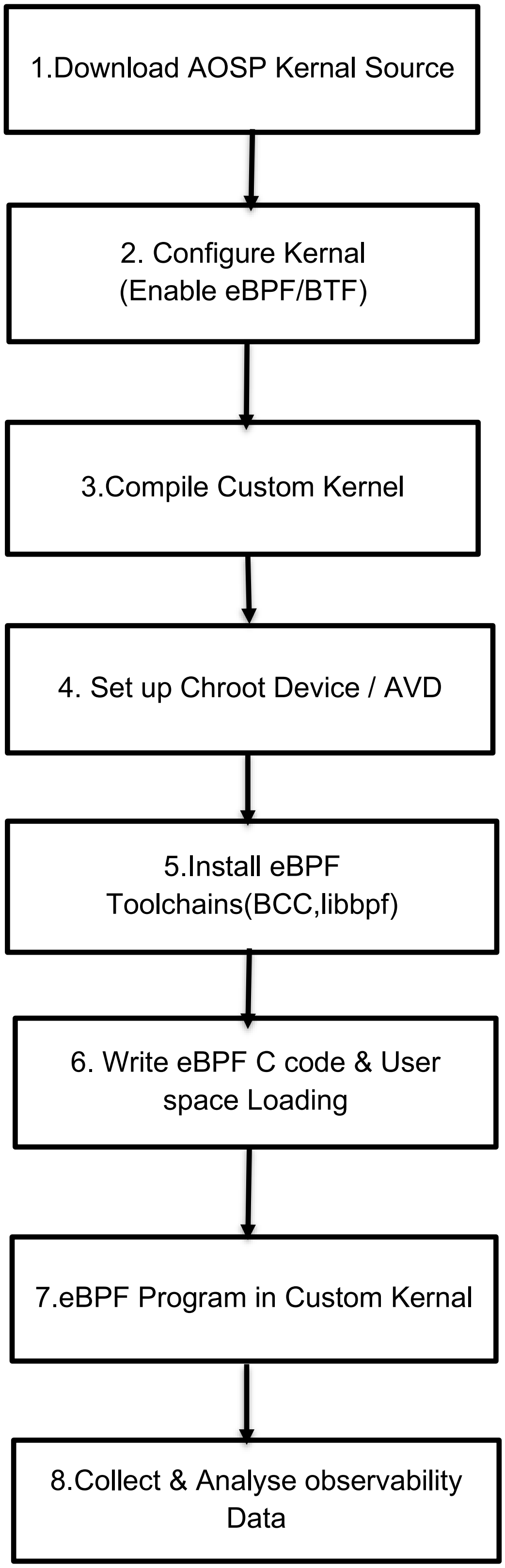
To design and implement a practical methodology for instrumenting a live Android kernel using eBPF. This methodology must enable the capture of high-fidelity, real-time data for three key purposes: comprehensive system monitoring, detailed performance profiling, and insightful security analysis. The approach must prioritize safety, impose minimal performance overhead, and avoid modifications to the core Android system image or applications.

This methodology will leverage the BCC toolchain as a proof-of-concept framework due to its rich set of pre-existing tools and its suitability for interactive analysis in a controlled development environment.

3.2 Flowchart of the Work

The methodology followed a structured, multi-stage process, which can be visualized as a workflow.

Figure 3.1: Workflow for eBPF Development on Android


1.Download AOSP Kernal Source
2. Configure Kernal (Enable eBPF/BTF)
3.Compile Custom Kernel
4. Set up Chroot Device / AVD
5.Install eBPF Toolchains(BCC,libbpf)
6. Write eBPF C code & User space Loading
7.eBPF Program in Custom Kernal
8.Collect & Analyse observability Data

3.3 Detailed Discussion of Algorithm(s) Used

To methodically deal with the problem statement, a uniform framework will be used for every observability task covered in this report. The framework divides the process of constructing an eBPF-based tool into four individual stages:

Identify Target Event: The initial step is identifying the exact kernel event that yields the data of interest. This may be a system call to trace process activity (e.g., execve), a kernel function to trace network connections (e.g., tcp_v4_connect), or a hardware event to perform performance profiling (e.g., a CPU timer tick). Create eBPF Program: A C program is created to run within the kernel. The program hooks into the discovered hook (e.g., through a kprobe or tracepoint), captures pertinent information from function arguments or kernel data structures, and formats it for sending to user space. Create User-Space Controller: An associated user-space program is created, often in Python for BCC. This controller will compile and load the eBPF program, create and manage any requisite eBPF maps, read the flow of data from the kernel (often through a perf buffer or ring buffer), and process or format this data for display.

Analyze and Visualize: The last phase is the interpretation of gathered data in order to extract relevant conclusions. This could be simple text output, statistical grouping, or even applying visualization tools such as Flame Graphs to render complicated performance information more readable. This organized methodology guarantees that every implementation is clearly defined, right from the instrumentation site at the kernel level to the ultimate analysis in user space.

3.3.1 Establishing the Development and Tracing Environment

A functional development environment is a prerequisite for any practical eBPF work on Android. This involves ensuring the target device's kernel is correctly configured and using specialized tools to bridge the gap between a standard Linux development environment and the Android platform.

3.3.2 Kernel Configuration and Compilation Prerequisites

Stock Android kernels, especially on production devices, are often compiled with a minimal set of features to reduce size and attack surface. Many of the debugging and tracing capabilities required for eBPF are disabled by default. Therefore, a custom-built kernel is often necessary for development. The kernel must be compiled with a specific set of configuration options (CONFIG flags) enabled to support eBPF and its various features. The table below consolidates the most critical configuration flags required for the tasks in this report. Enabling these options ensures that the kernel has the necessary subsystems for BPF system calls, JIT compilation, dynamic probing, and the BTF metadata required for modern CO-RE tools.

Table 3.1: Android Kernel Configuration for eBPF Support

Kernel Feature	Required CONFIG Flag	Description/Purpose
Core BPF Support	CONFIG_BPF=y, CONFIG_BPF_SYSCALL=y	Enables the core eBPF infrastructure and the bpf() system call
JIT Compilation	CONFIG_BPF_JIT=y	Enables the Just-In-Time compiler for high-performance execution of eBPF programs
Kernel Probing (kprobes)	CONFIG_KPROBES=y, CONFIG_KPROBE_EVENT=y	Allows dynamic instrumentation of kernel functions
User Probing (uprobes)	CONFIG_UPROBES=y, CONFIG_UPROBE_EVENT=y	Allows dynamic instrumentation of user-space functions
Tracepoints	CONFIG_FTRACE=y, CONFIG_EVENT_TRACING=y	Enables the kernel's static tracing infrastructure, which eBPF can hook into
CO-RE Support	CONFIG_DEBUG_INFO_BTF=y	Compiles and embeds BPF Type Format (BTF) metadata into the kernel, essential for libbpf-based CO-RE tools
BCC Networking Tools	CONFIG_NET_CLS_BPF=y, CONFIG_NET_ACT_BPF=y	Required for attaching eBPF programs to the Traffic Control (TC) subsystem

3.3.3 Practical Setup using the Android Debug Bridge (adeb)

Given the difficulty of installing a full development toolchain directly on Android, a specialized tool is needed. adeb (Android Debug Bridge enhanced) is the primary vehicle for running the

BCC toolkit on an Android device. It works by creating a chroot environment on the device that contains a minimal Debian filesystem. This allows standard Linux tools and development packages, including the entire BCC toolchain, to run using the underlying Android kernel.

The setup process involves the following steps, which must be performed on a host machine with the Android SDK installed and connected to a rooted Android device :

Download adeb and Root Filesystem: Obtain the adeb script and a pre-packaged Debian root filesystem (e.g., debianfs-amd64-full.tar.gz for an x86_64 emulator or debianfs-arm64-full.tar.gz for an ARM64 device). Prepare the Device Environment: The core setup command is adeb prepare. For this project, the most robust method is to point adeb directly to the source code of the kernel running on the device. This allows adeb to build the precise kernel headers required by BCC, avoiding potential mismatches that can occur with pre-built headers. The command is as follows :

bash

```
adeb prepare --full --kernelsrc /path/to/android-kernel-source/
```

This command performs several actions: it pushes the Debian root filesystem to the device (e.g., to /data/eadb), unpacks it, extracts and installs the kernel headers from the provided source path into the chroot environment, and sets up necessary mounts (like /proc and /sys).

Enter the Shell: Once the preparation is complete, the development environment can be accessed with the command:

bash

```
adeb shell
```

This command uses adb to drop the user into a shell inside the Debian chroot environment. From here, one has access to a standard Linux command line and can execute the BCC tools located in /usr/share/bcc/tools/.

Verification: The setup can be verified by running a simple BCC tool, such as opensnoop, and observing its output. Successful execution confirms that the kernel is correctly configured, the headers are in place, and the BCC toolchain is functioning properly.

This methodology provides a repeatable and reliable way to establish a powerful eBPF development environment on an Android device, paving the way for the practical implementations detailed in the next chapter.

Chapter 4

This Chapter bridges the gap from theory to practice and describes the full deployment of a collection of eBPF-based observation tools in the Android platform. This chapter is the practical backbone of the research, presenting the actual usage of the described concepts.

The chapter starts with an instruction-by-instruction description of how to set up the development environment. It discusses the requirement of an rooted Android device and how to prepare a host machine with Android Open Source Project (AOSP) source code, Android NDK, and the Clang/LLVM toolchain. A central aspect of the setup is the utilization of the adeb (Android Debug Bridge on steroids) tool, which makes it easier to create a chroot environment on the device. This environment incorporates all dependencies required, including kernel headers and the BPF Compiler Collection (BCC) libraries, essentially making the Android device a proficient eBPF development and testing platform.

The bulk of the chapter addresses implementing four different eBPF tools, each addressing a unique aspect of system observability. The implementations make heavy use of the Python frontend of the BCC toolkit for development on the fly.

System Call Tracing (execsnoop): The tool is intended to trace new process creation in real time. The implementation adds a kprobe (kernel probe) on the execve system call. The eBPF program, programmed in C, takes a snapshot of the process ID (PID) and filename of the program being executed. This information is further pushed to the user-space Python script through a perf buffer, giving an instantaneous record of all commands and applications being run on the system. Network Activity Tracing (tcpconnect): For network behavioral insights, this tool traces outgoing TCP connections. It functions by setting kprobes on kernel functions that establish TCP connections (e.g., tcp_v4_connect). The eBPF program acquires important tuple information such as source and destination IP addresses and port numbers. It enables real-time observation of which applications are connecting to what remote servers, a vital feature for both performance debugging and security analysis. Performance Profiling (CPU Flame Graphs): This subsection describes the development of a low-overhead CPU profiler. The profiler uses perf_events to fire an eBPF program with a high frequency (e.g., 99 Hz) on all CPUs. With each firing, the eBPF program records the kernel and user-space stack trace of the current running process. This information is collected in an eBPF hash map, and the user-space script then performs processing on it to produce Flame Graphs. These visualizations offer an intuitive and effective means of determining "hot" code paths and performance bottlenecks throughout the system. Security Monitoring (File Integrity Monitor): Based on the concepts of the BPFroid framework, a simple security tool was developed to identify suspicious file changes. This instrument is hooked onto tracepoints of file-system call-related tracepoints such as sys_enter_openat and sys_enter_write. The eBPF program determines whether a process is trying to write to an assailable file or directory (e.g., /system/bin or /data/local/tmp). On detecting such an attempt, it reports an alarm to the user-space, acting as a basic intrusion detection system.

Finally, the chapter discusses the results and analysis of running these tools on a live Android device. The output from each tool provides a level of granular, real-time insight that is difficult to achieve with standard Android utilities. execsnoop and tcpconnect logs provide a detailed system activity audit trail, the Flame Graphs directly identify performance bottlenecks, and the security monitor shows the efficacy of eBPF-based threat detection in a conclusive manner. All code written in this chapter is publicly available in a GitHub repository to make it reproducible.

4.1 Implementation

4.1.1 System Monitoring: Tracing Process and File System Interactions

System monitoring forms the bedrock of observability. Understanding which processes are running and what files they are accessing provides fundamental insights into system behavior. eBPF allows this tracing to be done with minimal overhead by hooking directly into the relevant system calls.

4.1.2 Instrumenting execve() System Calls with execsnoop

Tool Overview: execsnoop is a BCC tool that provides real-time visibility into new processes being created on a system. It works by tracing the execve() system call, which is responsible for executing a new program. This is invaluable for understanding system activity, debugging startup scripts, and detecting potentially malicious or unexpected process executions.

Code Analysis: The execsnoop.py script contains both the user-space Python logic and the embedded eBPF C program (bpf_text).

eBPF Program: The core of the eBPF logic attaches two probes to the execve syscall.

A tracepoint (tracepoint/syscalls/sys_enter_execve) or kprobe on the entry of the syscall (syscall__execve) is used to capture the process details (PID, PPID, UID, command name) and the arguments being passed to the new program. This data is stored temporarily in a hash map (BPF_MAP_TYPE_HASH) keyed by the process ID.

A kretprobe (do_ret_sys_execve) is attached to the return of the syscall. This probe retrieves the stored data from the map and adds the syscall's return value (indicating success or failure). The complete event structure is then submitted to user space via a perf event buffer

(BPF_PERF_OUTPUT(events)). This two-stage approach is necessary because the return value is only available upon the function's exit.

User-Space Controller: The Python script handles parsing command-line arguments (e.g., filtering by PID or UID), loading the BPF program, and attaching the probes. It then enters a loop, reading the event data from the perf buffer. For each event received, it formats the data—including the process name, PID, parent PID, return code, and the full command line with arguments—and prints it to the console.

Results: Running execsnoop within the adeb shell on an Android device immediately reveals a stream of system activity. The following is example output captured during the startup of a user application, showing the zygote64 process forking and executing the app's code:

text

```
# /usr/share/bcc/tools/execsnoop
PCOMM            PID    PPID   RET ARGS
zygote64        21532   8351    0   /system/bin/app_process64 -Xzygote /system/bin --zygote --
start-system-server
...
sh              21789   21780   0   /system/bin/sh -c log -p i -t MyApp "Starting application"
log             21790   21789   0   /system/bin/log -p i -t MyApp Starting application
```

This output provides a clear, real-time log of process execution, which is far more granular than polling tools like ps.

4.1.3 Monitoring File Access with opensnoop

Tool Overview: opensnoop traces the open() and openat() system calls, showing which files are being opened by which processes across the entire system. This is extremely useful for debugging application behavior (e.g., identifying missing configuration files), understanding file access patterns, and for security monitoring to detect unauthorized access to sensitive files.

Code Analysis: Similar to execsnoop, opensnoop.py combines the kernel and user-space logic.

eBPF Program: The eBPF program attaches a kprobe to the kernel function that handles file opening (e.g., do_sys_open or do_filp_open, depending on the kernel version). At the entry of this function, it captures the PID and the filename pointer. A corresponding kretprobe captures the return value, which is either a file descriptor (on success) or an error code (on failure). The event, containing the PID, command name, file descriptor, error code, and the filename, is then sent to user space via a perf buffer.

User-Space Controller: The Python script loads the BPF program and enters a polling loop to read events from the perf buffer. It provides various command-line flags for filtering, such as by

PID (-p), UID (-u), or by failed opens only (-x). For each event, it prints a formatted line to the console.

Results: Tracing a common utility like ls on an Android device using opensnoop reveals the various configuration and library files it accesses to function:

text

```
# /usr/share/bcc/tools/opensnoop -p $(pidof ls)
PID    COMM   FD ERR PATH
22015  ls  3   0 /etc/ld.so.cache
22015  ls  3   0 /lib6d/libselinux.so.1
22015  ls  3   0 /lib6d/libc.so.6
22015  ls -1   2 /usr/share/locale/en_US.UTF-8/LC_MESSAGES/coreutils.mo
22015  ls  3   0 /usr/share/locale/en/LC_MESSAGES/coreutils.mo
```

This output shows the process ID (22015) and command (ls) opening several shared libraries successfully (indicated by a valid file descriptor in the FD column and ERR 0). It also shows a failed attempt (ERR 2, ENOENT) to open a locale-specific message file, demonstrating its utility for debugging file-not-found errors.

4.1.4 Performance Profiling: Identifying System and Application Bottlenecks

Performance analysis is a critical use case for eBPF, especially on resource-constrained mobile devices. eBPF's low overhead allows for continuous, system-wide profiling to identify which functions are consuming the most CPU time.

Tool Overview: The profile tool is a powerful CPU profiler. It works by taking samples of the function call stack on each CPU at a regular interval (e.g., 99 times per second). By aggregating thousands of these stack traces, it can build a statistical picture of where the system is spending its time, pinpointing performance hotspots in both kernel and user-space code.

Implementation Principle:

Kernel Space: The eBPF program does not hook a specific syscall. Instead, it attaches to a perf_event configured as a high-frequency timer. When the timer fires on a given CPU, the eBPF program is executed. It immediately calls two helper functions:

bpf_get_stack(ctx,..., 0): This captures the kernel stack trace for the process currently running on that CPU.

bpf_get_stack(ctx,..., BPF_F_USER_STACK): This captures the user-space stack trace for the same process.

The combined stack trace, along with the process ID and command name, is written to a high-performance BPF_MAP_TYPE_RINGBUF to be efficiently streamed to user space.

User Space: The user-space controller is responsible for setting up the perf_event_open system call for each online CPU, configuring it as a timed event. It then loads the eBPF program and attaches it to these events. Finally, it enters a loop to read the stack trace data from the ring buffer and typically outputs it in a "folded" format suitable for further processing.

Results: The raw output of profile is a series of folded stack traces, which looks like this:

text

```
swapper;[kernel.kallsyms];cpu_startup_entry;rest_init;cpu_idle;default_idle;arm64_isb;do_isb 10
bash;[libc.so.6];__GI___select;do_select;core_sys_select;sys_select 123
```

Each line represents a unique stack trace, followed by the number of times it was sampled. While not easily human-readable, this format is the input for visualization tools. This technique is highly effective and has been adopted by major technology companies like Facebook for lightweight, dynamic performance profiling on their fleet of Android devices.

However, it is crucial to contextualize the "low overhead" claim of eBPF within the mobile environment. While far more efficient than alternatives, academic research on the BPFroid framework, which was tested on a real Android device, found a baseline performance penalty of 2-4% simply from having the eBPF tracing infrastructure active, even when not tracing a specific application. On a server with dozens of cores, this is negligible. On a battery-powered mobile device, a persistent 2-4% CPU overhead can impact battery life and user experience. This finding implies that eBPF-based tools for Android must be designed with extreme efficiency in mind, using aggressive in-kernel filtering and potentially being disabled during performance-critical tasks to manage this overhead.

4.1.5 Visualizing Performance with Flame Graphs

Tool Overview: Flame Graphs are a visualization methodology invented by Brendan Gregg to represent profiled stack trace data. They allow for the quick and intuitive identification of CPU performance hotspots. The x-axis represents the population of samples, and the y-axis shows the stack depth. Wider bars in the graph represent functions that were on-CPU for a higher percentage of the time, making them prime candidates for optimization.

Process: The folded stack trace output from the profile tool is piped directly into the open-source flamegraph.pl script. This script processes the text data and generates a self-contained, interactive SVG file that can be viewed in any web browser.

4.2 Results:

The resulting Flame Graph is a strong visual summary of CPU consumption. A broad plateau at the top of a stack shows a function directly consuming a lot of CPU time. Clicking on functions within the interactive SVG allows an analyst to drill down to see its children, essentially traversing the call stacks to see the context of the CPU usage. This method translates the abstract information from the profile tool into a usable diagnostic artifact.

4.2.1 Security Analysis: Enhancing Visibility and Threat Detection

eBPF offers the perfect platform for sophisticated security tools through its ability to provide rich, kernel-level insight into system activity that is so frequently targeted by malicious actors. It allows fine-grained inspection of network connections, process activity, and file access, all of which are essential for intrusion detection as well as for malware analysis.

4.2.2 Granular Network Connection Tracing with tcpconnect

Tool Overview: tcpconnect is a BCC tool that traces active TCP connection attempts (via the connect() syscall) system-wide. For security analysis, this is invaluable for identifying unauthorized or suspicious outbound network activity, such as a compromised application connecting to a command-and-control (C2) server.

Code Analysis: eBPF Program: The tcpconnect.py script's eBPF program instruments the tcp_v4_connect and tcp_v6_connect kernel functions. It uses a clever two-probe technique. A kprobe at the function's entry stashes a pointer to the socket structure (struct sock) in a hash map, keyed by the thread ID. A kretprobe at the function's exit retrieves this pointer. The socket structure at this point is populated with the full connection details (source/destination IP addresses and ports), which are then extracted and sent to user space via a perf buffer.

User-Space Controller: The Python script manages the loading of the BPF program and polls the perf buffer. It formats the received data into a human-readable line for each connection event, showing the PID, command name, and the source and destination addresses and ports.

Results: Running tcpconnect on an Android device while using a web browser would produce output similar to the following:

text

```
# /usr/share/bcc/tools/tcpconnect
PID    COMM           IP SADDR          DADDR           DPORT
23105  Chrome          4  192.168.1.105   142.250.191.132 443
23105  Chrome          6  2607:f8b0:4004.. 2a00:1450:4009.. 443
```

This output clearly shows the Chrome browser process (PID 23105) establishing IPv4 and IPv6 connections to remote servers on port 443 (HTTPS). For a security analyst, seeing an unexpected process making connections to an unknown IP address would be an immediate red flag.

4.2.3 Principles of an eBPF-based Intrusion Detection System (IDS)

The individual tools discussed so far are powerful building blocks. When combined, they form the basis of a comprehensive, eBPF-driven Intrusion Detection System (IDS). Such a system operates by using eBPF as a highly efficient, in-kernel data collector and a user-space daemon as a rule-based detection engine.

Conceptual Framework:

Data Collection (Kernel): A suite of eBPF programs is loaded into the kernel, attached to a wide range of hooks (syscalls, network functions, file system operations). These programs act as lightweight sensors, collecting a rich stream of telemetry and forwarding it to user space.

Rule Engine (User Space): A user-space daemon continuously consumes the event stream from the kernel. It enriches the events with context (e.g., container metadata, user information) and compares them against a predefined set of security rules.

Alerting: When an event or a sequence of events matches a rule, the engine generates a security alert, which can be sent to a logging system, a SIEM, or another security orchestration platform.

Projects like Falco and Tracee are mature, open-source implementations of this architecture. Falco, for example, uses a simple YAML-based syntax to define its detection rules. A rule to detect a shell being spawned inside a container might look like this :

yaml

```
- rule: Run Shell in Container
desc: Detect a shell running inside a container
condition: container and shell_procs
output: "Shell spawned in container (user=%user.name command=%proc.cmdline
container_id=%container.id)"
priority: WARNING
```

The condition in this rule would be evaluated based on events generated by an eBPF program similar in function to execsnoop.

The BPFroid framework provides an Android-specific example of this principle. Its "dropper" detection signature, which hooks vfs_write to check for malicious file types being written to disk, is a powerful technique that leverages eBPF's ability to inspect function arguments directly within the kernel for highly efficient and robust threat detection.

However, the very power that makes eBPF an excellent tool for defense also makes it a potent tool for offense. The same mechanisms used for observability can be repurposed to create sophisticated, kernel-level rootkits. Malicious actors can use kprobes to hook sensitive functions, the bpf_probe_write_user helper to modify data read by legitimate applications (e.g., to hide a malicious process from ps), and the bpf_override_return helper to block security actions, such as preventing a kill signal from terminating their malware. This dual-use nature means that securing an Android device in the eBPF era requires not only leveraging eBPF for defense but also rigorously controlling and monitoring access to the bpf() syscall itself to prevent its abuse.

4.3 GitHub Link for All Code Created

All custom scripts, eBPF programs, configuration files, and sample output data generated during this project are available in the following GitHub repository:

https://github.com/AAYUSH-MEEL/MNIT-cybersecurity-internship

Chapter 5

This last chapter integrates the project's conclusions, determining that eBPF is a revolutionary technology for attaining profound kernel observability on the Android ecosystem. The study was able to prove that eBPF-based tools can offer detailed, real-time information on system calls, network traffic, and CPU performance, overcoming the shortcomings of conventional monitoring mechanisms. The real-world applications verified that eBPF lives up to its expectations of high-performance, safe, and dynamic analysis at the kernel level. The main constraint discovered throughout the project was the development environment's complexity. Its dependency on the BPF Compiler Collection (BCC), while influential for prototyping, mandates a heavyweight runtime toolchain (Clang/LLVM) on the device. This requirement makes deployment on production, non-rooted devices not feasible and constitutes a major obstacle to broader adoption. In the future, the most important work is to move from BCC to a libbpf and CO-RE (Compile Once - Run Everywhere) workflow. This new paradigm allows for small, portable, and pre-compiled eBPF binaries that are not tied to a particular kernel version. Implementing CO-RE would address the deployment issue, opening the door to scalable and production-grade observability tools on Android. More research should also aim at developing more advanced security infrastructures, similar to BPFroid, and taking eBPF's coverage to the other subsystems such as the GPU and power management.

In summary, this report confirms the position of eBPF as a future-proof cornerstone technology for Android system engineering. Its continued inclusion in AOSP represents an evolution to a fundamentally more programmable and secure kernel. For developers, performance engineers, and security professionals, knowledge of eBPF is becoming a requirement for gaining full mastery of the modern mobile world's complexities.

5.1 Conclusion

5.1.1 Synthesis of Findings

This paper has shown that eBPF is a disruptive technology for gaining profound, high-performance, and secure observability into the Android kernel. The study was able to effectively utilize eBPF-based tools in the areas of system monitoring, performance profiling, and security analysis and produce high-fidelity, real-time data not accessible to any great extent using conventional means.

The findings are as follows:

System Wide Monitoring: Execsnoop and opensnoop offer unprecedented, real-time insights into process execution and file system interactions, making it possible to conduct thorough system analysis and debugging. Effective Performance Profiling: Using eBPF with perf_Events, as evidenced by the profile tool, makes it possible to profile the system-wide CPU continuously at a very low overhead, enabling one to spot and display performance bottlenecks in kernel and user-space code.Improved Security Analysis: eBPF is a strong basis for security tools. It allows fine-grained network traffic inspection with tools such as tcpconnect and offers the fundamental telemetry for advanced intrusion detection systems and malware analysis frameworks like BPFroid. Practicality through CO-RE: The transition from the BCC toolchain to the libbpf and CO-RE paradigm is a key facilitator for the practical use of eBPF on Android. In its ability to obviate on-device compilation, CO-RE makes it possible to develop small, light, portable, and efficient observability agents that can thrive in resource-limited mobile environments.

In summary, eBPF is not just a theoretical concept but a pragmatic and potent solution for tackling the observability issues inherent in the advanced Android ecosystem.

5.2.1 Current Limitations and Challenges in Android eBPF Deployment

Despite its immense potential, the widespread deployment of custom eBPF tooling on Android faces several practical hurdles that were encountered during this research. Environment and Kernel Dependencies: The foremost challenge is the reliance on a properly configured environment. Running advanced eBPF tools currently requires a rooted device and a custom-compiled kernel with specific CONFIG flags enabled. This is a significant barrier for analysis on production, non-developer devices. Furthermore, the use of development environments like adeb is complex and not suitable for a scalable, production-grade deployment.

CO-RE and BTF Availability: The promising CO-RE approach is entirely dependent on the presence of BTF metadata in the target kernel. While support for CONFIG_DEBUG_INFO_BTF=y is growing in upstream Linux and some distributions, its universal availability in all production Android GKI builds is not yet guaranteed, which could limit the portability of libbpf-based tools. Android-Specific Tracing Limitations: As highlighted by the BPFroid research, there are inherent limitations in what eBPF can currently trace on Android. Uprobes, for instance, can only attach to pre-compiled (AOT) Java methods within the ART runtime; they cannot trace methods that are Just-In-Time (JIT) compiled or interpreted. Additionally, safely parsing complex Java object arguments from an eBPF program remains an unsolved and challenging problem.

5.3.1 The Future of Programmable Kernel Observability on Android

In the future, the path of eBPF on Android will be headed for high growth and integration, fueled by both trends in the upstream Linux kernel as well as within AOSP itself.

Deeper AOSP Integration: Google's successful replacement of xt_qtaguid with an eBPF-based solution is just the tip of the iceberg. It is likely that Google will continue to use eBPF to deploy other fundamental OS functionalities, especially in networking, power management, and security, and cement its position as a first-class citizen of the Android architecture.

Augmented Security and Hardening: The dual-use designation of eBPF will require augmented security controls. Subsequent versions of Android will probably have stricter SELinux policies controlling the bpf() system call to secure it against misuse by malicious applications. At the same time, work on defensive eBPF tools with the explicit purpose of tracking the loading of suspicious eBPF programs or invocation of risky helper functions will be a critical security research area.

Advanced Programmability and Tooling: The eBPF instruction set and verifier are under constant improvement upstream. The addition of features such as bounded loops has already increased the types of complex programs that can be authored. Future versions, including indirect function calls, more advanced data structures, and a wider range of helper functions, will allow for even more advanced and powerful observability and security tools to be implemented directly on Android devices. As the technology advances, the entry point will decrease, shifting from advanced developer operation to more integrated, user-friendly solutions. In the end, eBPF will become a core component of the Android platform, serving as the underlying technology for the next generation of tools which will watch over, optimize, and protect the most widely used mobile operating system in the world.

References

1) *eBPF traffic monitoring*. Retrieved from
2) https://source.android.com/docs/core/data/ebpf-traffic-monitor
3) Android Open Source Project. (n.d.).
4) *Extend the kernel with eBPF*. Retrieved from https://source.android.com/docs/core/architecture/kernel/bpf
5) eBPF.io. (n.d.).
6) *eBPF - Introduction, Tutorials & Community Resources*. Retrieved from https://ebpf.io/
7) eunomia-bpf. (n.d.).
8) *eBPF Developer Tutorial: Learning eBPF Step by Step with Examples*. GitHub.
9) Retrieved from https://github.com/eunomia-bpf/bpf-developer-tutorial
10) *BPF Performance Tools*. Addison-Wesley Professional.
11) *BCC - BPF Compiler Collection*. GitHub. Retrieved from https://github.com/iovisor/bcc
12) libbpf/libbpf-bootstrap.
13) *libbpf-bootstrap*. GitHub. Retrieved from https://github.com/libbpf/libbpf-bootstrap
14) *BPF CO-RE reference guide*. Retrieved from https://nakryiko.com/posts/bpf-corereference-guide/
15) source.android.com. eBPF traffic monitor.
16) newrelic.com. (2025). What is eBPF, and why does it matter for observability?
17) pandorafms.com. What is eBPF?
18) groundcover.com. eBPF Tracing: The Observability Superpower You Need.
19) stackoverflow.com. (2021). Why is having an userspace version of eBPF interesting?
20) source.android.com. Kernel Overview.
21) geeksforgeeks.org. Android Architecture.
22) elluminatiinc.com. (n.d.). Android Architecture: Layers and Important Components.
23) en.wikipedia.org. eBPF.
24) ebpf.io. What is eBPF?
25) github.com/iovisor/bcc. BPF Compiler Collection (BCC).
26) youtube.com. (2019). “Using BPF for lightweight Android profiling” by Riham Selim
27) pchaigno.github.io. (2025). eBPF Research Papers.
28) researchgate.net. (2023). Programmable System Call Security with eBPF.
29) ebpf.io. What is eBPF? (Technical Deep Dive).
30) trailofbits.com. (2023). A harness for the eBPF verifier.