

Preventing Unintended Data Access: Information Flow Control in eBPF

Chinecherem Stephanie Dimobi

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Dan Williams, Chair

Daphne Yao

Matthew Hicks

May 08, 2025

Blacksburg, Virginia

Keywords: eBPF, Security, Information Flow Control, Static Analysis

Copyright 2025, Chinecherem Stephanie Dimobi

Preventing Unintended Data Access: Information Flow Control in eBPF

Chinecherem Stephanie Dimobi

(ABSTRACT)

The extended Berkeley Packet Filter (eBPF) technology has become widely adopted by enterprises due to its flexibility and ability to enhance tracing, observability, monitoring, and security within the kernel. However, since the kernel is a critical resource containing sensitive information, eBPF also presents a significant attack surface for malicious actors. One of the challenging-to-detect yet easiest-to-execute attacks is sensitive information leakage, as it does not require additional privileges beyond standard eBPF functionality. Attackers can exfiltrate sensitive data using built-in eBPF mechanisms, such as saving information to a shared data store. Although the eBPF subsystem provides safety guarantees through its verifier, it does not track or restrict access to sensitive data that an eBPF program is not explicitly intended to access. In this research, we propose an information flow control (IFC) system that leverages labels and policies to track and prevent unauthorized access and leakages to sensitive information by third-party eBPF programs. We define sensitive information as any data that a given eBPF program is not explicitly authorized to access. Our approach defines a label-based policy specification that includes an "allow list", IFC-based static analysis to analyze eBPF bytecode, and policy enforcement to prevent malicious programs from loading and attaching into the kernel if they violate predefined security constraints. Results from our implementation show that our framework catches previously undetectable leakage patterns. This work addresses a critical gap in eBPF security by providing a structured mechanism to prevent unintended data access and leakages while maintaining the legitimate use cases of eBPF.

Preventing Unintended Data Access: Information Flow Control in eBPF

Chinecherem Stephanie Dimobi

(GENERAL AUDIENCE ABSTRACT)

Computer systems run the world and are critical for almost everything technology. Operators have to constantly monitor the most critical parts of a system, specifically the operating system (OS) for security and reliability. The OS kernel is the piece of software on every computer system that allows applications and services to safely and efficiently utilize computer hardware. In recent years, operators have turned to running small programs within the OS kernel for monitoring purposes. These programs, called "eBPF" programs, have access to sensitive information of applications from the OS kernel and are trusted not to abuse this information. Recent events have shown that eBPF programs can be used maliciously. Operators can be tricked into running malicious eBPF programs that extract unauthorized sensitive information from the kernel. eBPF programs utilize a special verification process to ensure that they are safe and do not cause systems to crash. Unfortunately, the verification process does not prevent eBPF programs from reading unauthorized data and in turn leaking sensitive information. Our work enhances existing verification to eBPF programs by applying a technique of Information Flow Control (IFC) tailored for eBPF. The system statically analyzes the program and ensures sensitive data is not leaked according to a label-based policy specification we define. Our methodology offers a lightweight, pre-verification step that complements existing kernel protections. Results from our implementation show that our framework catches previously undetectable leakage patterns, providing both security and peace of mind for system operators.

To my family and friends

Acknowledgments

I would like to thank my advisor, Dr. Dan Williams, for being a support throughout my research journey. He guided me in the right direction, not just research-wise but career-wise as well. I am deeply grateful to my lab mates, who became like family during my time here. They helped me fill knowledge gaps and understand most of what I now know about systems. Special thanks to Raj for introducing me to the lab, Siddharth Chantemani for supporting me in the project, and Rahul Tiwari, Milo Craun and Tanuj Rao for their troubleshooting expertise and knowledge guidance. Last but not least, I would like to thank my friends and family for their unwavering support and for constantly checking up on me. I would not have gotten here without all of you.

Contents

List of Figures	x
List of Tables	xi
1 Introduction	1
1.1 The security implications of eBPF programs	2
1.2 Solution to prevent unwanted access	3
1.3 Contributions	3
1.4 Thesis Organization	4
2 Background	5
2.1 The Extended Berkeley Packet Filter (eBPF)	5
2.1.1 Data Access in the eBPF Subsystem	8
2.1.2 Security Concerns Related to Uncontrolled Data Access	8
2.1.3 Information Flow Control	9
2.2 Conclusion	10

3	Malicious BPF	11
3.1	Bad BPF Programs	11
3.2	Sensitive Data Access Experiment	12
3.3	Present Security Mechanisms are not Enough	14
3.4	Consequences of Information Leakage	15
3.5	Conclusion	15
4	Threat Model and Problem Definition	16
4.1	Scope	16
4.2	Assumptions	17
4.3	Sample Attack Scenarios	17
4.4	Security Goals	18
4.5	Conclusion	19
5	Information Flow Control Static Analysis	20
5.1	High-level Architecture Overview	20
5.2	Policy Generation	22
5.2.1	Fine-grained policies	22
5.3	Information Flow Control Static Analysis	26
5.3.1	Control Flow Graphs of the eBPF Bytecode	27
5.3.2	Static Analysis and Taint Propagation	28

5.4	IFC in eBPF	31
5.5	Implementation	32
5.6	Conclusion	32
6	Results & Evaluation	34
6.1	Evaluation setup	34
6.2	Evaluation objectives	35
6.3	Program Evaluation with Verifier and IFC Checks	35
6.4	Evaluation on Real-World Rootkits	38
6.5	Performance Overhead	39
6.6	Summary	41
7	Related Work	42
7.0.1	IFC in Systems	42
7.0.2	Enhancing eBPF Security through Isolation Mechanisms	43
7.0.3	Improving the verifier’s static analysis capabilities	43
8	Discussion & Future Work	45
8.1	Discussion	45
8.2	Future Work	46
9	Conclusion	48

List of Figures

2.1	BPF Subsystem Overview	7
2.2	Data Access in eBPF	9
3.1	Simplified Pixie Program.	13
3.2	Malicious Pixie Program.	14
5.1	Information Flow Control System in eBPF	22
5.2	Data Access Policies	23
5.3	Helper Capabilities	24
5.4	Offsets for XDP Kernel Structure	25
5.5	Data Access Policies After Offset Translation	26
5.6	Label and Taint Propagation in the IFC Component	30
5.7	Detailed Information Flow Control System in eBPF	32
6.1	Detection outcomes under verifier, helper-only IFC, and full IFC	37
6.2	Comparison of kernel verifier and IFC load time	40

List of Tables

2.1	eBPF Program Types and Their Hook Points.	6
3.1	Malicious eBPF Helper Functions	12
6.1	Summary of Evaluated eBPF Programs: Type, Behavior, and Helpers	36
6.2	Detection of Real-World eBPF Rootkits by IFC Enforcement	39

Chapter 1

Introduction

Extended Berkeley Packet Filter (eBPF) is a Linux kernel extension that allows users to safely run custom programs within the kernel without the need to develop or load traditional kernel modules. This approach provides significant safety by allowing user programs to gain access to kernel resources without the risk of crashing the kernel. It also provides flexibility by eliminating the complexities and delays of writing, maintaining, and getting kernel modules approved by kernel maintainers. Additionally, eBPF enhances efficiency by enabling dynamic scaling of kernel capabilities, making it a valuable tool for modern system development.

eBPF operates by attaching user-defined programs to specific hook points within the kernel, such as syscalls, tracepoints, kprobes (kernel probes), and uprobes (user-space probes). These hooks allow eBPF programs to observe and manipulate data in the kernel. Due to its capabilities, eBPF is increasingly used across various industries for observability, monitoring, tracing, network management, and security. Tools like BCC, bpftool, and Eunomia-bpf provide essential development frameworks for eBPF programs [5, 9, 24], while solutions like Cilium, Falco, Pixie, Tetragon, and Pyroscope leverage eBPF to deliver advanced observability and security capabilities [3, 7, 8, 10, 11]. Furthermore, companies such as Netflix,

Google, Android, and Microsoft rely on eBPF to optimize performance and enhance their system monitoring capabilities [16].

1.1 The security implications of eBPF programs

While eBPF's power makes it indispensable for observability and security, it also presents significant risks. Due to its deep integration with the kernel, eBPF can be exploited by malicious actors to perform harmful actions. Examples of such exploits include information theft, denial of service (DoS) attacks, process hijacking, tampering with shared data stores, and deploying stealthy rootkits that evade traditional detection mechanisms. [17]. The severity of these threats is evident from both research and real-world incidents [21]. For these attacks, attackers can leverage certain helper functions like `bpf_probe_write_user`, `bpf_override_return`, `bpf_send_signal`, and `bpf_probe_read` [41] which allow some attack primitives. Preventing the usage of these dangerous helper functions seem like the solution, however, limiting helpers like `bpf_probe_read` is not practical. This is because such functions are essential for benign applications like profiling and performance monitoring. Consequently, information theft remains one of the most challenging threats to address, as restricting these functions would severely limit legitimate use cases. Prior research [6] has attempted to enforce coarse-grained policies that deny or allow eBPF programs based on their capabilities, where capabilities are essentially eBPF helper functions. In practice, this approach can either overly restrict the abilities of eBPF programs, rendering them less useful, or leave enough functionality intact for information leaks to persist.

1.2 Solution to prevent unwanted access

In Chapter 5, we demonstrate how we build an information flow control mechanism capable of detecting all accesses to kernel data within an eBPF program and ensuring that each access is explicitly authorized by a system administrator. We assign labels to input and output data within eBPF and propagate them throughout the program to be used for fine-grained control. By doing so, we prevent potential malicious programs from gaining unauthorized access and leaking sensitive information.

We extend the current state-of-art beyond simply restricting certain BPF helpers by focusing on the specific data the program accesses while also limiting the program to only the BPF helpers it requires. Our approach accurately detects any unauthorized data access, providing a more granular level of security enforcement. The mechanism we design detects unwanted access before the program is verified and attached to the kernel.

1.3 Contributions

In this research, we propose a flexible and practical static information flow control mechanism that prevents sensitive information leakage while preserving the legitimate functionality of eBPF programs. This is achieved by extending the properties of the eBPF verifier to vet security access properties of the eBPF program. System administrators can define access policies following least-privilege best practices and detect policy violations before eBPF program execution.

To summarize the main contributions, in this thesis we:

- Identify the balance between allowing access to sensitive data and preventing leakages

in kernel extensions.

- Design techniques to assign sensitivity labels and policies to inputs and outputs in kernel extensions, propagating these labels during execution.
- Implement a system that distinguishes between safe and malicious kernel extensions that access sensitive data.

This research fills the gap in existing eBPF security measures, ensuring that sensitive information remains protected while maintaining the functionality needed for legitimate eBPF applications.

1.4 Thesis Organization

Chapter 2 provides a background on eBPF, how eBPF accesses data, and information flow control. Chapter 3 discusses the implications of not implementing access controls for eBPF programs. Chapter 4 gives detail of the threat model, the adversary capabilities, sample attack scenarios and the scope of the problem we address in this paper. Chapter 5 presents the design and implementation of our information flow control system, and provides an overview of its integration with the eBPF subsystem. Chapter 6 presents and evaluates results from our information flow control component. Chapter 7 discusses some of the other works focused on the security of eBPF. Chapter 8 discusses important points related to our proposed solution, limitations and future work, while Chapter 9 presents our conclusion.

Chapter 2

Background

This chapter provides background on the eBPF architecture and common use cases of eBPF. The chapter then explains how eBPF programs access kernel or user data. In addition, it presents a high-level overview of the role of the verifier in enforcing security for eBPF and security concerns associated with uncontrolled access. We then present a high-level overview of information flow control. Finally, we discuss the key takeaways that inform the design of our proposed solution.

2.1 The Extended Berkeley Packet Filter (eBPF)

Kernel extensions make the kernel dynamically programmable at runtime. eBPF programs are extensions to the Linux kernel. These programs are dynamically loaded and executed at specific interaction points within the kernel's execution path, known as hook points. The hook points vary depending on the program's intended purpose, for example, network packet processing, system call tracing, or performance monitoring. Table 2.1 gives an overview of some common types of eBPF programs, their hook points and potential use cases.

Table 2.1: eBPF Program Types and Their Hook Points.

Program Type	Hook Point	Use Cases
XDP	Network Driver	Packet filtering, DDoS mitigation
Socket Filter	Socket layer	Packet capturing, firewalling
Kprobe/Kretprobe	Kernel Function entry/exit	Kernel tracing, debugging
Tracepoint	Predefined kernel tracepoints	Performance monitoring
Sched CLS/ACT	TC ingress/egress	Traffic shaping, QoS

Figure 2.1 shows the architectural overview of the BPF subsystem. The process begins when a userspace application compiles a program into eBPF bytecode. The eBPF verifier does a safety check on the eBPF program to ensure it does not crash the kernel. After passing verification, the eBPF program is attached to its designated kernel hook point, where it can intercept, analyze, or modify kernel-level events in real-time.

The verifier: Before attachment, eBPF programs are verified by the in-kernel verifier [15], which performs static analysis to ensure safety and performance. The verification process involves two stages. In the first stage, the verifier performs DAG checks, such as ensuring no cycles (backward edges) exist in the program. This verification step ensures that eBPF programs have no infinite loops and are guaranteed to terminate. In the second stage, the verifier tracks all possible paths taken by a eBPF program by performing register and stack tracking to ensure the program is not executing any invalid operations. During verification, the verifier examines each eBPF program as an individual unit and makes assumptions about the runtime environment (kernel) with which it interacts.

Helper Functions and Kfuncs: The eBPF subsystem provides interfaces for kernel interaction: helper functions and kfuncs. Helper functions are stable interfaces that remain consistent between kernel releases, while kfuncs are not. Both helper functions and kfuncs must register and encode information such as arguments, return types, and compatible program types into the verifier to ensure eBPF programs use these interfaces safely. The verifier

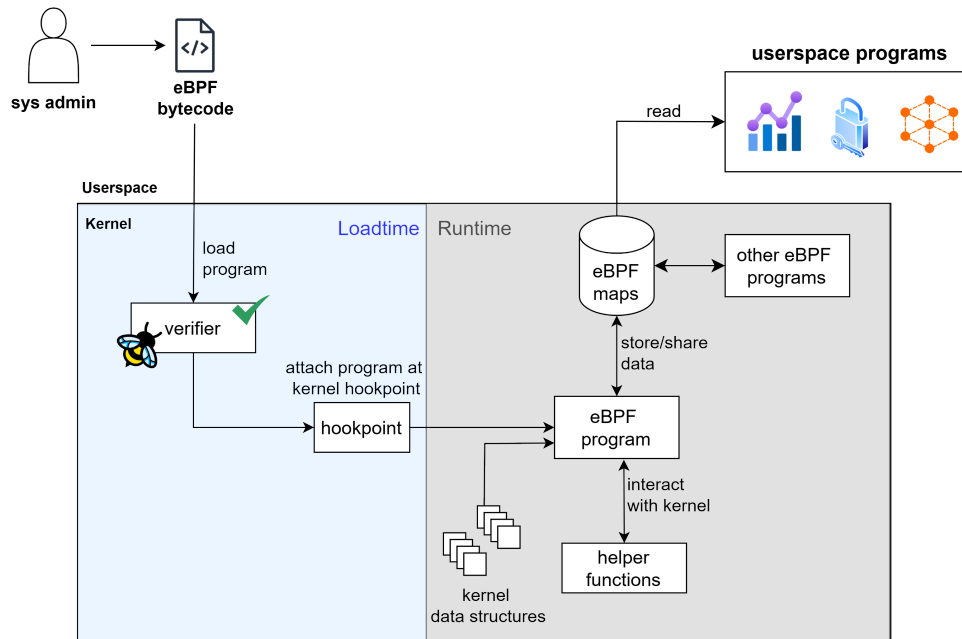


Figure 2.1: BPF Subsystem Overview

assumes that execution within helper functions and kfuncs are inherently safe.

Maps: Maps provide a way to store and share information obtained from eBPF programs. Maps enable communication between eBPF programs and userspace. eBPF programs communicate with maps via helper functions, and userspace communicates with maps via system calls. Maps have varying data structures and are mainly used to store state.

Due to the strong safety guarantees of eBPF programs, they have been adopted across a wide range of use cases. The initial adoption of eBPF was in packet filtering for networking, which is where the name Berkeley Packet Filters originated. Later, because of the safety guarantees and its dynamic nature led to eBPF's expansion into various use cases including tracing [1, 3], networking [7], security modules [2], scheduling [39], etc.

2.1.1 Data Access in the eBPF Subsystem

When an eBPF program runs, it does so within a certain context based on the location it is attached. This context, provided by the kernel, contains specific data structures relevant to that hookpoint. eBPF programs typically get kernel data in two ways: directly from the provided context, or by calling helper functions that return pointers to kernel information. For example, an **XDP** (eXpress Data Path) program receives a pointer to a context called **xdp_md**, which has information about network packets. On the other hand, programs used for performance monitoring, like those attached to tracepoints, often rely on helper functions such as **bpf_get_current_task()** to access details about the current process. These helpers fall into categories like memory-related helpers, process-related helpers, and CPU-related helpers. Importantly, eBPF programs do not directly read arbitrary kernel memory; instead, they interact with kernel data through these contexts and helpers as can be seen in [Figure 2.2](#). Programs can also store or share kernel information using eBPF maps. For instance, one program might save process IDs or related details into a map, which another program can then read or update later. Maps offer a safe and organized way to store kernel data outside of individual eBPF programs.

2.1.2 Security Concerns Related to Uncontrolled Data Access

Allowing eBPF programs unrestricted access to kernel data structures can pose notable security risks. Doing this assumes that all programs are trustworthy, which might not always be true. In enterprise settings where third-party eBPF programs are commonly used, these programs could have malicious intent or become targets of supply chain attacks, making open access to sensitive kernel data risky. In [Figure 2.2](#), we show how sensitive data obtained from the kernel can be stored in eBPF maps and later accessed by potentially malicious

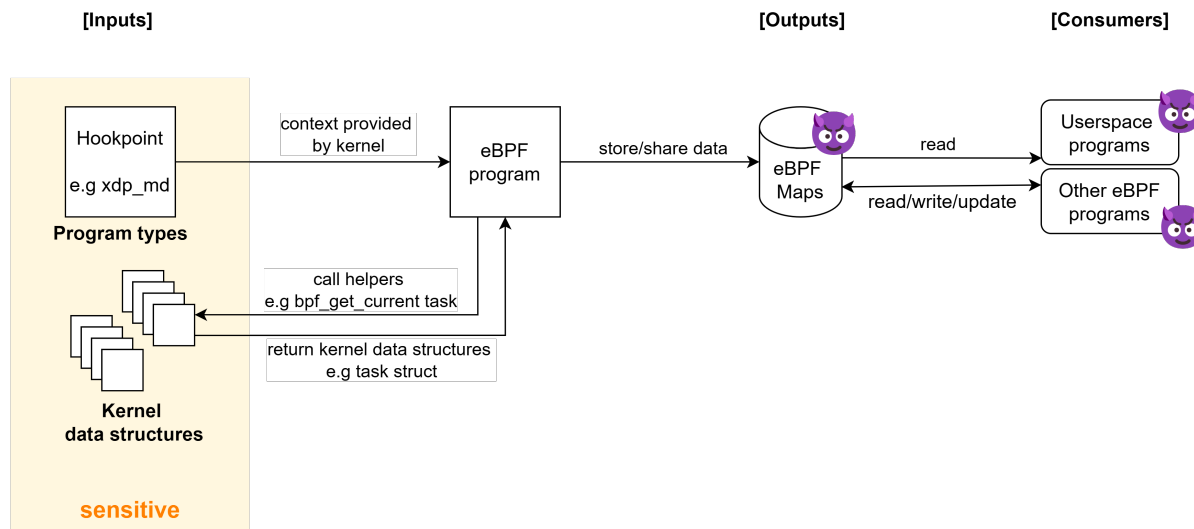


Figure 2.2: Data Access in eBPF

third-party applications or other eBPF programs. The eBPF verifier helps ensure memory safety by checking that programs do not read data beyond the intended limits and confirming that only relevant data structures are accessed. However, as we will explore in Chapter 3, the verifier still has certain limitations. These gaps can lead users to overestimate the verifier’s capability to control data access, creating a false sense of security.

2.1.3 Information Flow Control

Information Flow Control (IFC) [29, 42] has long been used as a security mechanism to track how sensitive data moves through a system and to ensure that this data does not flow into untrusted locations. It has been applied in everything from programming languages to operating systems in order to prevent unauthorized leaks or misuse of confidential information. The core idea is to label data based on its sensitivity and then enforce rules that control how that data flows as the program runs. IFC enforces data flow restrictions throughout the execution of the program, ensuring that sensitive data does not propagate beyond its

intended scope.

2.2 Conclusion

To summarize, we gave an overview of the eBPF subsystem, security concerns of data access control in eBPF and an introduction to information flow control. In the next chapter, we will discuss how unrestricted access to kernel structures even with a BPF program's context may raise serious security concerns.

Chapter 3

Malicious BPF

This chapter presents how eBPF programs have been used maliciously in the past. Then we introduce how unrestricted access for eBPF programs raise serious security concerns. We discuss a BPF program within the scope of normal BPF capabilities and verifier specification can be used to access sensitive information.

3.1 Bad BPF Programs

eBPF programs have been previously exploited for malicious purposes. One notable example is BPFdoor [12], a stealthy backdoor identified by PwC that leveraged eBPF to bypass firewall rules and evade forensic detection. Another, Symbiote [37] is a malware that steal user data and conceals the activities of other malwares deployed with it. It is able to collect user data and exfiltrate to DNS servers. Due to incidents like this, researchers and industry experts have become very interested in understanding how eBPF can be misused [18, 19, 31, 32, 34]. The Linux Foundation conducted a comprehensive security threat modeling of eBPF [17], highlighting possible threats such as malicious use of certain eBPF helpers

Table 3.1: Malicious eBPF Helper Functions

Helper Function	Purpose
<code>bpf_probe_write_user</code>	Write to any process's user space memory
<code>bpf_probe_read_user</code>	Read any process's user space memory
<code>bpf_override_return</code>	Alter return code of a kernel function
<code>bpf_send_signal</code>	Send a signal to kill any process

(outlined in Table 3.1), sensitive information leakage, supply chain attacks, denial-of-service (DoS), rootkits, and detection evasion. Most misuses involving specific eBPF helpers can be stopped by limiting or blocking their use. But one helper, `bpf_probe_read`, is challenging to restrict because many legitimate monitoring tools depend on it. This makes it difficult to detect when attackers misuse it to secretly read sensitive information from the kernel.

3.2 Sensitive Data Access Experiment

To effectively steal sensitive information, the following requirements need to be met: (1) having a resource for the information, (2) accessing the information, and (3) a means to exfiltrate the sensitive information. The core functionality of eBPF inherently satisfies all these requirements by allowing programs to hook into the kernel, a resource that holds some of the most sensitive information available. Additionally, eBPF provides the capability to extract this information into a map or even use the sensitive data for decision-making using conditional logic.

Figure 3.1 shows a sample eBPF program from Pixie [10], an open-source observability tool for Kubernetes applications that utilizes eBPF for monitoring.

A simple eBPF program like this inherently possesses characteristics that make it well-suited for extracting sensitive data. With slight modification, as shown in Figure 3.2, the program

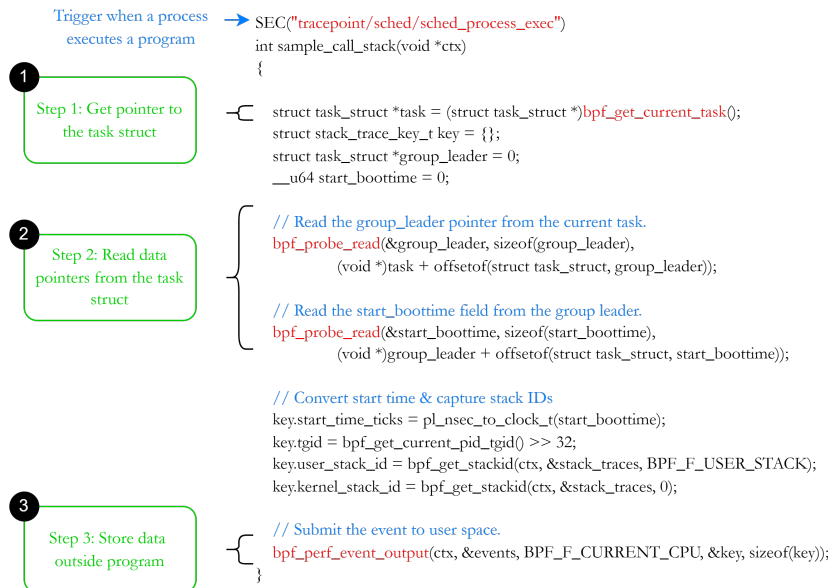


Figure 3.1: Simplified Pixie Program.

accesses additional sensitive information. For example, instead of accessing the start boot time to use as an ID, the modified program now accesses the `request_key_auth` field within the credential structure of the task struct and like the original, the modified program passes the verifier. This small change that could easily happen during a supply-chain attack might not be noticed by a system administrator, since the overall structure of the program and the eBPF helper functions did not change, but only the specific data field accessed. The core issue here is that there is currently no robust mechanism available to detect malicious eBPF hooks or generate alerts when sensitive data outside the intended scope of an eBPF program is accessed.

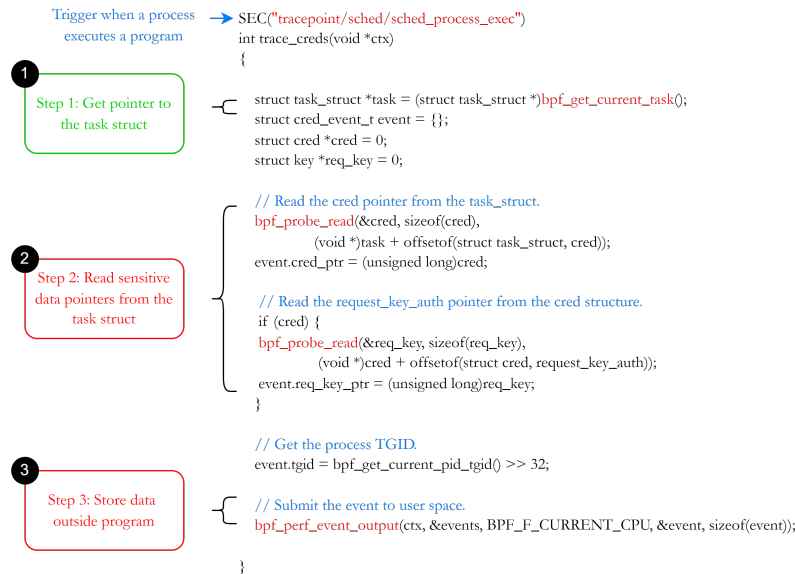


Figure 3.2: Malicious Pixie Program.

3.3 Present Security Mechanisms are not Enough

From the previous experiment, we have seen that eBPF security mechanisms in place for access controls are not enough. Some of the reasons why users of eBPF-based applications are prone to sensitive information leakage are:

1. The eBPF verifier, while ensuring safety by preventing unsafe operations, does not track or trace the data that is read or accessed. This is a significant gap in the eBPF system.
2. Other approaches that involve limiting programs solely based on helper functions are coarse-grained and do not solve the problem of vetted programs that may be accessing data out of scope.
3. Even with manual monitoring of hook points using Linux commands and tools, it is

easy to lose track or fail to prevent sensitive information leakage before it occurs.

3.4 Consequences of Information Leakage

Information leakages are critical and have several real-world consequences. Leaking sensitive data like passwords, authentication keys, user data or unintended data can lead to data breaches or violations of privacy regulations such as GDPR, HIPAA, or CCPA. These leaks can also provide attackers with more information that may aid better understanding of the system or create targeted attacks, potentially leading to escalation of privileges.

3.5 Conclusion

In this chapter, we have given instances of how eBPF have been used maliciously in the past and a walk-through scenario to understand why fine-grained data access control is necessary for environments that use eBPF. We discussed the consequences that may come about from sensitive information leakage. In future chapters, we will discuss the threat model that we address in this research paper and how information flow control can fill the gap of enforcing data access control.

Chapter 4

Threat Model and Problem Definition

In this chapter, we discuss the scope of the thesis in addressing information leakage in eBPF. We discuss what the adversary is capable of doing and the attack scenarios we consider. Lastly, we state the security goal of this thesis paper.

4.1 Scope

We focus on scenarios where a **system administrator** employs third-party eBPF programs. The eBPF programs may be used for any combination of the following:

- Monitoring and observability for performance insights
- Security features such as intrusion detection (IDS) or intrusion prevention (IPS)
- Networking enhancements to efficiently route and manage traffic.

Malicious Actors and Attack Vectors. Sensitive information leaks can originate from any of the following:

- A malicious third-party application or external party.
- A supply chain attack on a third-party or external-party application.
- A remote code execution (RCE) vulnerability that allows unauthorized eBPF code injection.
- A malware

The system administrator is trusted not to write their own malicious eBPF program.

4.2 Assumptions

While the adversary is flexible, we assume that the adversary cannot modify the kernel (e.g. modules) except for BPF programs and all kernel-side protections still exist. Additionally, all eBPF programs will pass the eBPF verification stage by the verifier. Hence, there will be no out-of-bound accesses and the program only uses eBPF helper functions concerned with the program type. However, even if these assumptions are not met, our proposed design still works well to identify accesses not intended for an eBPF program because we focus on allow policies rather than deny policies. If the bound is not within the allowed offsets set by the system administrator, the program will be rejected.

4.3 Sample Attack Scenarios

In this section we will discuss three attack scenarios that link to the eBPF usages discussed above.

Networking: BPFdoor is an eBPF-based rootkit that stealthily hides the presence of other malicious programs within the kernel. It attaches to tracepoints related to network packet processing and inspects traffic for specific trigger patterns. When a matching packet is detected, it opens a backdoor shell on the system, giving remote access to an attacker without triggering traditional security alerts. This shows how eBPF programs can be weaponized to create covert communication channels if not properly audited.

System Security: Pamspy is a known eBPF malware that targets the Pluggable Authentication Module (PAM) framework in Linux. PAM is used to handle authentication and authorization flows across the system. Pamspy hooks into PAM-related function calls and silently collects authentication credentials, including usernames and passwords, as users log in. This attack shows how deeply integrated eBPF programs can be abused to intercept highly sensitive data if they are allowed unchecked access to kernel hooks.

Monitoring: In a more subtle but equally dangerous scenario, a legitimate eBPF-based monitoring tool designed to measure network performance is configured to capture and log entire packet payloads. These payloads may include unencrypted credentials, private messages, or sensitive business data. Without proper data flow restrictions, such tools, even if not intentionally malicious, can become a vector for data leakage simply by writing sensitive payloads to user space logs or maps shared with user applications.

4.4 Security Goals

The security goal of this research is to prevent eBPF program from unauthorized access. Unauthorized here refers to data and functionalities that are not explicitly allowed by the system administrator.

4.5 Conclusion

We discussed the scope of this thesis as well as the capabilities of the adversary. We then discussed code samples for attacks in different scenarios and defined the goal of this research. In the next chapter, we will see how we designed our framework to achieve our security goal.

Chapter 5

Information Flow Control Static Analysis

This chapter discusses how we extend the functionality of the eBPF subsystem by adding a layer of security that provides constraints on the inputs (what they can access) and outputs (data that can leave the program), hence reducing system administrators' susceptibility to malicious eBPF programs or supply-chain attacks. We demonstrate how our design uses fine-grained policies alongside Information Flow Control (IFC) static analysis to audit untrusted eBPF programs to ensure they only access what is expected of them. This approach extends the existing verification process and the state-of-the-art approach for auditing eBPF programs.

5.1 High-level Architecture Overview

We use the concept of IFC for our design because it allows us to track how data is accessed within a program and where those data can flow to. We are able to apply IFC to the eBPF

subsystem because of some key insights.

1. The verifier shows that tracking the usage of the register is feasible. Tracking register usage provides a structured way to analyze how data propagates through an eBPF program statically without runtime cost.
2. Maps provide a potential exfiltration channel. Maps act as the communication channel between the kernel and user space as well as other eBPF programs. Since the verifier only checks structural safety and not what data are stored, a malicious program can store sensitive data in a map and later read it from user space or other malicious eBPF programs.
3. Helper functions are well defined and have fixed argument and return types, making it possible to perform precise static analysis of data movement to function calls. We can therefore treat helpers as both inputs (i.e helpers that return kernel data) and outputs (i.e helpers that interact with maps) in static analysis.

There are two major components of our design; the policy generation, which defines what is allowed, and the IFC static analysis engine that checks and enforces the policies in the program. Figure 5.1 shows a high-level view of our design. System administrators set policies based on the policy specification that we have defined. We then ensure that all data accesses and helper function calls in the eBPF program fall within those policy constraints. In this way, we prevent potential malicious programs from leaking or gaining unauthorized access to sensitive kernel information. We place our system at load time, which is after compilation and before run-time execution. By strategically placing our design at this location, we ensure that any eBPF program violating policies is rejected before it is attached to the kernel. Another benefit of this location is that our implementation only

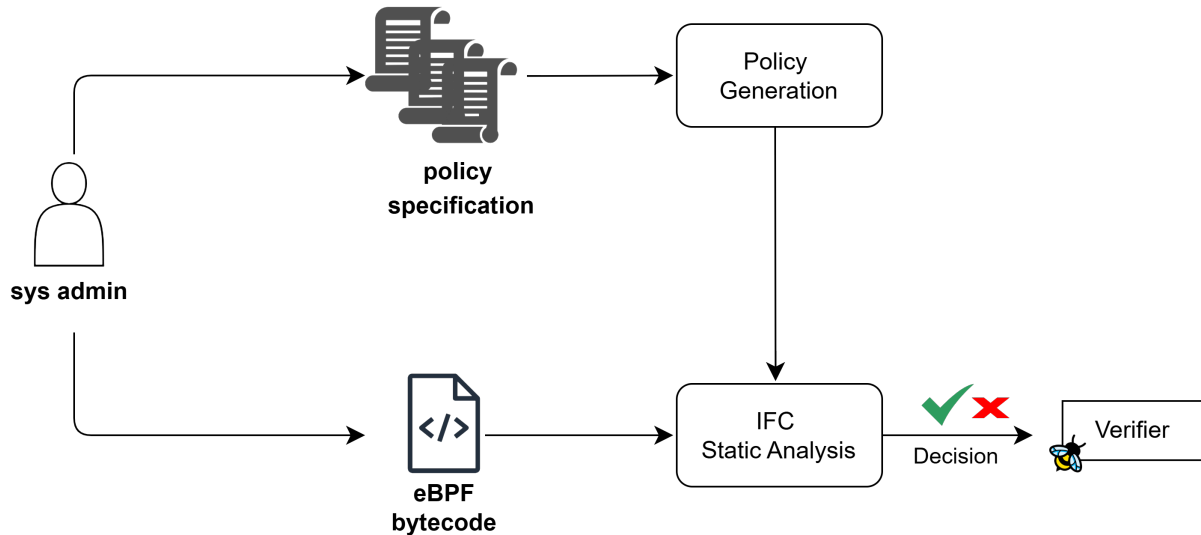


Figure 5.1: Information Flow Control System in eBPF

incurs negligible load time costs and does not affect the performance of the eBPF program during execution.

In the following sections, we will discuss in detail the two key components of our design and explain how they work together to achieve our security goals.

5.2 Policy Generation

In our design, the system administrator specifies policies for each eBPF program introduced into the system using a JSON format.

5.2.1 Fine-grained policies

The policies define the inputs that are allowed in the eBPF program. These inputs fall into two categories: data access and helper function usage. As a result, each eBPF program in

```
{
  "program": {
    "name": "pixie.kern",
    "purpose": "Tracepoint at sched/sched_process_exec"
  },
  "data_access": {
    "task_struct" : {
      "allow": [group_leader, start_boottime],
      "sensitive": [cred]
    }
  }
}
```

Figure 5.2: Data Access Policies

our system has two sets of policies in the policy specification.

Data Access Policies:

One set of policies governs data access, specifying which information an eBPF program is permitted to read. These policies reference fields within the eBPF context and kernel structures that can be accessed.

Each of these fields can be assigned one of three labels: *allow*, *deny*, or *sensitive*. The *allow* label refers to data that may be accessed freely, while the *deny* label refers to data that the program must not access. The *sensitive* label is used for data that may be accessed but must not leave the program.

This label-based system is what makes our policy model fine-grained and more security-aware. It allows us to be particular about the data that a program can access ensuring that it does not operate outside its intended scope or leak sensitive data.

```

{
  "program": {
    "name": "test_pixie.kern",
    "purpose": "Tracepoint: sched/sched_process_exec"
  },
  "capability_profile":{
    "bpf_get_current_task": allow,
    "bpf_get_stackid": allow,
    "bpf_get_current_pid_tgid": allow,
    "bpf_perf_event_output": allow,
    "bpf_probe_read": sensitive,
    "bpf_map_delete_elem": deny
  }
}

```

Figure 5.3: Helper Capabilities

Figure 5.2 shows an example data access policy for the Pixie program discussed in Section 3.2. In this policy, the program is allowed to access the `group_leader` and `start_boottime` fields within the `task_struct` of the running task. Additionally, the `cred` field in the struct is marked as sensitive.

Helper Capabilities: The second set of policies defines the helper functions that the eBPF program is permitted to use. We refer to this as the helper capabilities. Similar to the data access policies, we assign *allow*, *deny*, and *sensitive* labels to helper functions. A helper marked with *deny* is not permitted in the program. Helpers marked with *allow* can be used and are not expected to return sensitive data. The *sensitive* label is applied to helpers that return sensitive data which must be tracked and should not leave the program.

Figure 5.3 shows the capability profile used for the Pixie program we discussed in Section 3.2.

Internally, eBPF subsystem has restrictions for the helper functions that are available to

each eBPF program type. For example, some helpers are only available to networking type programs. Therefore, the helper functions listed in the helper capability policies must fall within the subset of helper functions permitted for the program type, otherwise, the program will automatically fail the verifier check.

Offset Translation: The policies define data fields in a human-readable way which goes through an offset translation process to map the data fields into their corresponding memory offsets, based on eBPF and kernel internal structures. This translation allows the IFC component to properly enforce the policies with the offsets in the eBPF bytecode. Figure 5.4 shows how some data fields within the XDP context (`xdp_md`) map to their memory offsets (`data` at offset 0, `data_end` at offset 4, and `data_meta` at offset 8).

```
struct xdp_md {  
    __u32 data;      /* offset 0 */  
    __u32 data_end; /* offset 4 */  
    __u32 data_meta; /* offset 8 */  
    /* ... and so on ... */  
};
```

Figure 5.4: Offsets for XDP Kernel Structure

After offset translation, the data access policy in Figure 5.2 will look like Figure 5.5

Policy Characteristics: This policy specification follows a deny by default model. If a data field or helper function usage is not explicitly allowed in the policies, then access to it will be denied. By doing this, we inherently implement the principle of least privilege, ensuring that an eBPF program can only interact with data that it has been explicitly authorized to access.

To provide flexibility and simplify policy management for system administrators, we also

```

{
  "program": {
    "name": "pixie.kern",
    "purpose": "Tracepoint at sched/sched_process_exec"
  },
  "data_access": {
    "task_struct" : {
      "allowed": [1472, 3176],
      "sensitive": [1880]
    }
  }
}

```

Figure 5.5: Data Access Policies After Offset Translation

introduce predefined policy groups tailored to common eBPF program types such as tracing and networking. These groups provide reasonable defaults, such as allowing access to commonly used helper functions like those for retrieving process IDs. Similarly, data access policies can be generalized to permit access to all kernel data but apply sensitivity labels to restrict data flow. With this structure, administrators can easily choose between broad, permissive policies by allowing access to all kernel data but marking it sensitive to prevent leakage or more detailed, restrictive ones that permit only specific helper functions and data fields.

5.3 Information Flow Control Static Analysis

The IFC (Information Flow Control) component audits the eBPF program bytecode. For many beneficial reasons, the IFC component uses static analysis rather than dynamic analy-

sis. Static analysis during load time allows for fine-grained information flow control without impacting the runtime performance of the program. The main aim of the IFC component is to understand what kernel data the eBPF program accesses and where the data flows. This component ensures that an eBPF program does not bypass restrictions defined in the policies and does not allow sensitive data to leave the program. The IFC static analyzer component performs two passes on the eBPF bytecode. The first pass breaks down the bytecode into basic blocks used to construct a Control Flow Graph (CFG) and the second pass performs access checks and propagates access and taint labels throughout the program.

5.3.1 Control Flow Graphs of the eBPF Bytecode

A Control Flow Graph (CFG) represents the flow of a program from start to end. It helps in tracking data access and the propagation of sensitive labels across the program. To understand the flow of the program, particularly due to conditional branches within the code, we begin by constructing basic blocks.

A *basic block* [4] is a straight-line sequence of code with no branches into or out of the middle of the block. It has one entry point (the first instruction) and one exit point (the last instruction). To generate basic blocks, we scan the entire code to identify *leader instructions*, which are:

- The first instruction of the code,
- Any instruction that is the target of a conditional or unconditional jump,
- Any instruction immediately following a conditional or unconditional jump.

Each leader marks the beginning of a basic block. In our CFG, each node represents a basic block. To build the CFG, we determine the successors of each block based on branch

instructions. For a conditional statement, both the jump target and the fall-through block are considered successors. The CFG supports the static analysis process that checks access to kernel data and propagation of labels by providing the data flow.

5.3.2 Static Analysis and Taint Propagation

In this step, we track all data accesses to the kernel within the program and where this data flows. We first track data access to ensure that it is permitted by the policies, then we propagate labels from the policies through the program.

Static Analysis Data Access Tracking:

We track data access by monitoring how registers are used in the eBPF code. eBPF programs use eleven registers: R1 to R5 for passing function arguments, R6 to R9 for storing intermediate values, R0 for return values, and R10 as a stack pointer. We track registers from the function entry to the function exit.

As we discussed earlier, there are two major sources of kernel sensitive data: the context at the hookpoint of the eBPF program (kernel data passed directly to the eBPF program) and kernel structures accessed via helper functions such as `bpf_get_current_task()`.

To track the former, we tag R1 as a `ctx` register at the beginning of the program because the context structure is always passed to R1. To track the latter, we track the output of helpers that return kernel data structures.

Policy Enforcement on Data Access:

At each point where the program accesses kernel data, we check whether the access is allowed by the JSON-defined policies for the eBPF program. If it is not, a program violation is raised, and the program is not allowed to load into the kernel. If the access is allowed, we return a

label for the data that indicates whether it is regular kernel data or sensitive, based on the policy. These labels are then propagated throughout the program using the CFG.

Access and Taint Propagation:

We integrate the CFG construction with the data flow analysis that propagates kernel data access information across blocks. Propagation occurs with the help of states. Each basic block carries an *in-state* and an *out-state*, as illustrated in Figure 5.6. Each state stores the following;

1. the set of registers holding kernel values
2. the set of registers that are tainted with sensitive values
3. a flag indicating if the block is tainted because a conditional depended on sensitive data

A worklist-based iterative algorithm propagates the out-state of each predecessor block to its successors along the CFG. At each block, the state is updated based on program's accesses within that block. When a block has multiple predecessors, their states are merged to form the input state of that block.

There are two cases where propagation occurs. In the first case, as seen in 1) of Figure 5.6, when a register holding kernel or context value is used, either read, moved, stored, or passed into helper functions, we update the state of the basic block to include the new kernel data-holding register along with the offset within the structure that is accessed. This state is then propagated along the CFG.

We track both explicit and implicit data flows. For explicit flows, if a register holds kernel data and its contents are copied into another register, the new register inherits the same label.

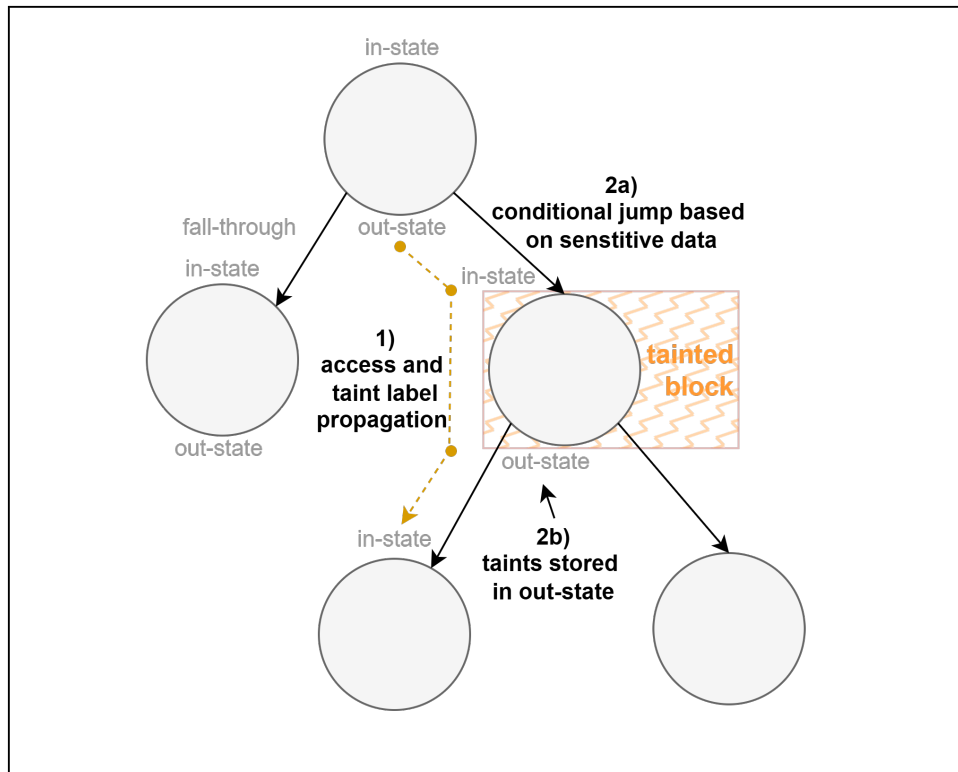


Figure 5.6: Label and Taint Propagation in the IFC Component

Explicit flow example:

```
x = kernel_data
y = x // y now holds the same label as x
```

For implicit flows, even if a register is not assigned directly, its value may still depend on sensitive data.

Implicit flow example:

```
if (sensitive_kernel_data):
    y = 1 // y is tainted due to the conditional dependency
```

In the second case of propagation, we apply implicit flow tracking for data labeled as sensitive.

If a conditional branch in a block depends on sensitive kernel data, we flag its successor block as a tainted block. Any value created or modified in this tainted block inherits the sensitive label as can be seen in 2a) and 2b) of Figure 5.6. This ensures that anything derived from sensitive data is tracked and prevented from leaving the program.

Policy Enforcement on Program Output:

Lastly, we enforce policies on outputs by checking all data that could leave the eBPF program. Data typically exits through function returns or helper functions, especially those used to write to maps, which can be read by user-space applications or other eBPF programs. For every argument passed into such a helper function, we check whether the data is tainted with a sensitive label. If it is, a violation is raised, and the program is prevented from loading into the kernel.

5.4 IFC in eBPF

Figure 5.7 gives a complete picture of our design of information flow control in eBPF. The system administrator provides the system policies alongside the eBPF program. We generate a control flow graph from the program, and each data access or helper function usage is vetted against the corresponding policy. If the program attempts unauthorized access or functionality, a program violation is raised. When sensitive data is accessed, or when a helper function that returns sensitive data is used, a taint label is propagated throughout the program. If at any point this taint labeled data attempts to leave the program, a violation occurs. In the absence of any violations, the program passes the IFC check and is then verified by the verifier.

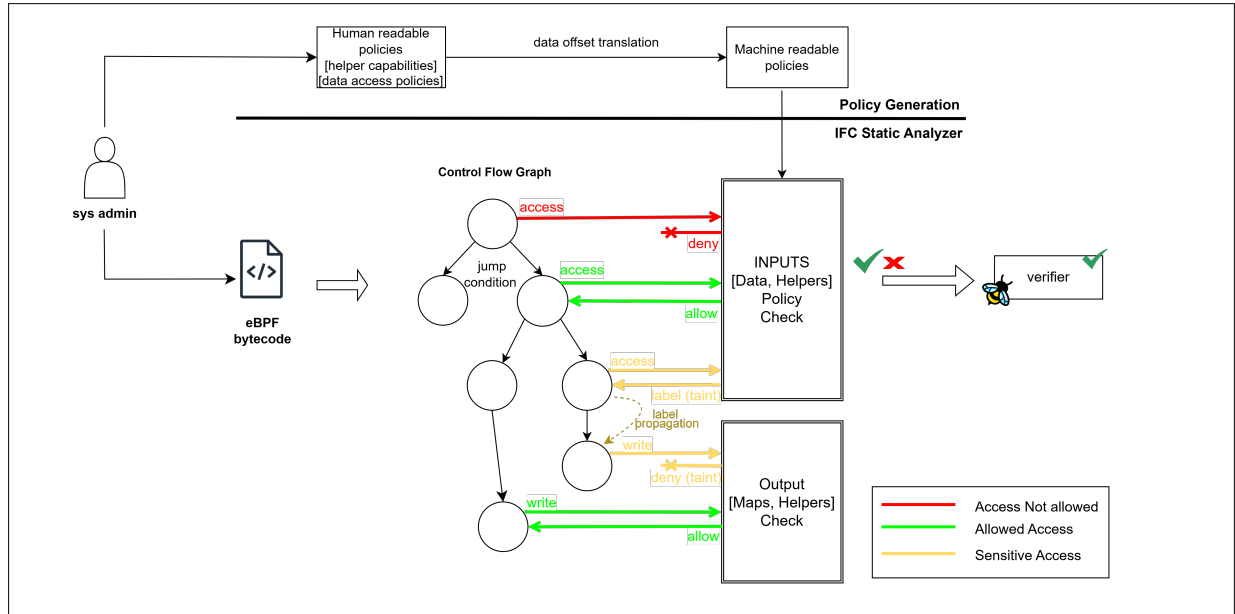


Figure 5.7: Detailed Information Flow Control System in eBPF

5.5 Implementation

The IFC system was implemented as a standalone Python script that runs before the verifier. The script performs static analysis on eBPF bytecode to enforce fine-grained policies on data access and helper function usage. We used Capstone, a Python disassembly library, to parse the bytecode before analysis.

5.6 Conclusion

Our work builds on the existing eBPF verification process by introducing an Information Flow Control (IFC) system specifically tailored for eBPF programs. The IFC component performs static analysis on eBPF bytecode to track all accesses to kernel data and verify that each access is explicitly permitted by a label-based policy specification. In addition, we define a mechanism that permits access to certain sensitive data but prevents that data

from leaving the program through shared data stores like maps.

We also enforce constraints on all helper functions used by the eBPF program, ensuring that only those permitted by both the verifier and the defined policies are allowed. Through this approach, we effectively block potentially malicious programs from gaining unauthorized access to kernel-level information.

By embedding our design before the verification stage, we are able to prevent unauthorized data flow, restrict sensitive information leakage, and provide a structured mechanism for enforcing security policies within the eBPF ecosystem. Overall, our methodology introduces a fine-grained and enforceable security model that complements existing kernel protections.

Chapter 6

Results & Evaluation

In this chapter, we evaluate our system to understand how well it achieves its goal of preventing sensitive data leakage through untrusted eBPF programs. We evaluated the system across three properties including security and policy enforcement, real-world applicability, and performance overhead.

6.1 Evaluation setup

All experiments were conducted inside a Docker container running Ubuntu 24.04.2 LTS with Linux kernel version 6.11.0-rc5. All programs were compiled and tested against the kernel. We used Clang version 18.1.3 to compile the eBPF programs, and verification was done using `bpftool`.

6.2 Evaluation objectives

In our evaluation, we aim to answer the following questions:

1. **Security and enforcement:** Can our information flow control accurately prevent all unauthorized data access and helper functions? And is it useful for detecting sensitive leakage?
2. **Real-world applicability:** Can our implementation prevent rootkits in the wild?
3. **System impact:** What is the impact of the information flow control static analysis on system performance?

6.3 Program Evaluation with Verifier and IFC Checks

To evaluate whether the IFC system correctly enforces the defined policies, we compiled a set of eBPF programs obtained from open-source repositories [33]. All these programs are known to be malicious and are tagged as bad because they use helpers in ways that could leak or overwrite kernel state. In addition to these known bad programs, we included a second group of programs that may not use traditionally disallowed helpers, but instead try to leak sensitive kernel data. This leakage happens either through map writes, debug output, or return values. Finally, we included one good program that performs valid monitoring without violating any policy. Table 6.1 shows the list of evaluated programs, including their type, behavior, and the helpers they attempt to use.

For this evaluation, we applied general policies that treat any data obtained from the kernel as sensitive. This approach was chosen due to the diversity of program types involved. So, we expect that kernel-derived data should not exit the program for this evaluation.

Table 6.1: Summary of Evaluated eBPF Programs: Type, Behavior, and Helpers

Program	Type	Behavior	Helpers Used
exec_jack.bpf.c	bad_helper	Hijacks process and sends a message to a map	bpf_probe_write_user
pid_hide.bpf.c	bad_helper	Hides process from directory listing	bpf_probe_write_user
sudo_add.bpf.c	bad_helper	Elevates user privileges by editing sudoers	bpf_probe_write_user, bpf_map_update_elem
text_replace.bpf.c	bad_helper	Finds and replaces a target string	bpf_probe_write_user, bpf_map_update_elem
text_replace2.bpf.c	bad_helper	Finds and replaces a target string	bpf_probe_write_user, bpf_map_delete_elem
alter_filepath.bpf.c	bad_helper	Alters file access path via context manipulation	bpf_probe_read, bpf_probe_write_user
write_blocker.bpf.c	bad_helper	Blocks write syscalls for specific processes	bpf_ringbuf_submit
log_flags.bpf.c	sensitive_leak	Logs syscall arguments and trace flags	bpf_map_update_elem, bpf_get_current_pid_tgid
log_switch.bpf.c	sensitive_leak	Logs sensitive context values	bpf_printk
capture_lsm_mkdir.bpf.c	sensitive_leak	Monitors LSM mkdir events	bpf_map_update_elem, bpf_get_current_uid_gid
leak_task_address.bpf.c	sensitive_leak	Leaks task struct address to map	bpf_map_update_elem, bpf_get_current_task
leak_hard_ids.bpf.c	sensitive_leak	Leaks hardcoded sensitive fields	bpf_map_update_elem
xdp_oob.bpf.c	sensitive_leak	Performs out-of-bounds access on XDP data	bpf_map_update_elem
monitor_tcp.bpf.c	good	Monitors TCP connection attempts	bpf_perf_event_output
filter_sock.bpf.c	sensitive_leak	Leaks sensitive data through return path	bpf_skb_load_bytes, bpf_printk

These policies serve to validate our implementation’s effectiveness in preventing sensitive information leakage. Additionally, we enforce policies that automatically block the use of known malicious helper functions, as detailed in Table 3.1 except for `bpf_probe_read` which is also used for legitimate purposes.

We ran three tests on each program. The first was a verifier-only check, where the program is compiled and passed to the kernel without our IFC system in place. As shown in Figure 6.1, most programs passed the verifier check, apart from two that attempted out-of-

bounds pointer access and were rejected due to structural issues.

In the second test, we turned on only one part of our IFC component—helper function enforcement—while leaving sensitive tracking disabled. This means no registers were labeled sensitive, and thus taint propagation did not take place, but only prevented the usage of malicious helpers from Table 3.1. As seen in Figure 6.1, this mode of the IFC system caught programs that used explicitly disallowed helpers, such as `bpf_probe_write_user`, or `bpf_override_signal`.

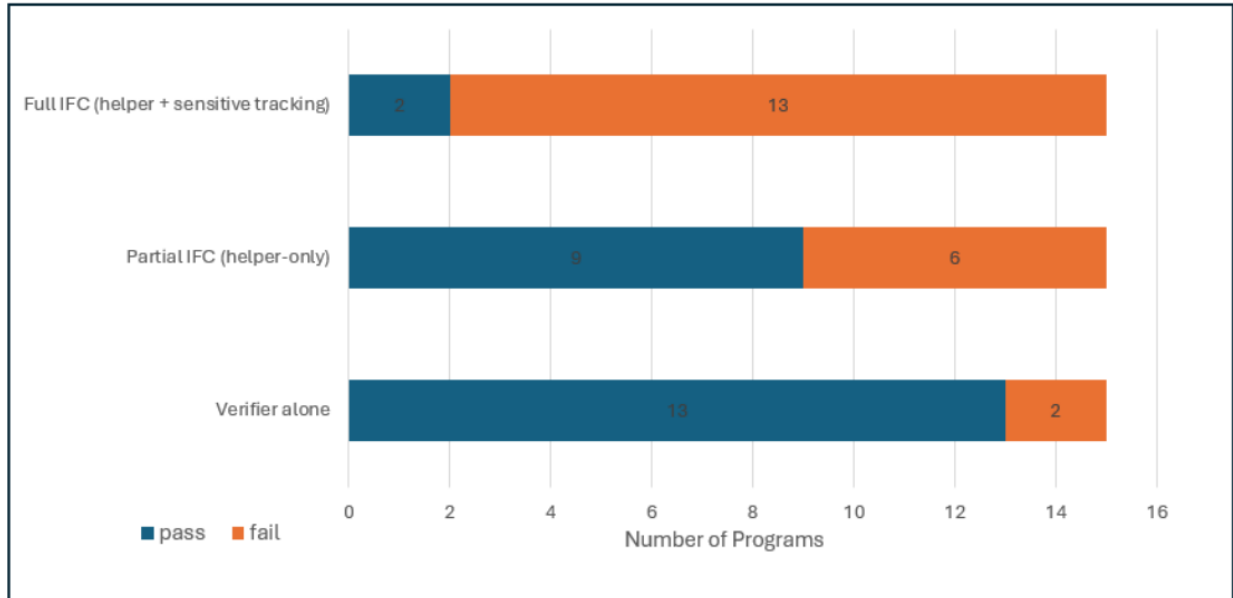


Figure 6.1: Detection outcomes under verifier, helper-only IFC, and full IFC

In the third test, we enabled the full functionality of the IFC system. In this mode, any kernel data read from context (e.g., `ctx` passed into `r1`) or returned from helpers is marked sensitive and tracked throughout the program. We observed that the full IFC check successfully blocked all the bad programs leaking sensitive data except for one program. This program `capture_lsm_mkdir` does not receive a generic `ctx` pointer from the kernel as you would in tracepoints. Instead, the hook’s arguments are passed directly into the eBPF program.

In the future, we plan to update the implementation to handle such cases. Otherwise, as expected, the one good program passed both the verifier and all IFC checks without issues.

These results confirm that our system catches a broad range of violations that are not detected by the verifier alone or by prior approaches that rely solely on helper restrictions. They also highlight the importance of tracking sensitive data flow explicitly, as relying only on helper blacklisting would miss several realistic attacks.

It is worth noting that `bpftool`, which is commonly used to load and inspect eBPF programs, often only raises a warning when programs use dangerous helpers like `bpf_probe_write_user`, rather than blocking them outright. This further motivates the need for a stronger pre-verification enforcement mechanism like ours.

6.4 Evaluation on Real-World Rootkits

To evaluate whether our IFC system correctly detects and blocks real-world threats, we tested it against known malicious eBPF rootkits. Specifically, we evaluated three prominent rootkits: `boopkit`, `bpfdoor`, and `pamspy`. Each of these rootkits aligns directly with the threat scenarios described earlier in Section 4.3, covering networking, system security, and monitoring use cases.

`Bpfdoor` utilizes BPF socket filters to stealthily inspect network packets, looking for specific trigger patterns to open a hidden backdoor shell. `pamspy` employs uretprobes on the Pluggable Authentication Module (PAM) to silently collect sensitive user authentication credentials. Lastly, `boopkit` attaches eBPF programs via XDP hooks to intercept and manipulate packets at an early stage, potentially exposing or redirecting sensitive network data.

In our evaluation, we audited each rootkit through our IFC enforcement mechanism.

All three programs triggered IFC policy violations due to sensitive kernel data attempting to leave the kernel boundary through helpers such as `bpf_map_update_elem` and `bpf_trace_printk`. Table 6.2 summarizes the results of this evaluation.

Table 6.2: Detection of Real-World eBPF Rootkits by IFC Enforcement

Rootkit	Hook Type	Sensitive Data Leakage	IFC Check
<code>bpfdoor</code>	Socket Filter (TCP)	Trigger packet payloads	X
<code>pamspy</code>	Uretprobe (PAM)	User authentication credentials	X
<code>boopkit</code>	Tracepoint (TCP)	Packet metadata and payloads	X

These results clearly demonstrate our IFC system’s practical effectiveness in blocking real-world rootkits by enforcing data-flow policies. Importantly, the evaluation highlights that our IFC component successfully identifies and prevents sensitive kernel information from leaving via allowed helpers, providing security assurances beyond the kernel verifier’s capabilities.

6.5 Performance Overhead

To get a clearer sense of the performance cost of including IFC static analysis in the eBPF pipeline, we ran benchmark tests comparing the load time of eBPF programs using two setups: the regular kernel verifier (using `bpftool`) and the IFC static analysis in Python. For the IFC tests, we used a script with `hyperfine`, set to `RUNS=5` and `WARMUP=3` to get more stable averages. All parts of the IFC static analysis were on including the helper checks, sensitive data tracking, and taint propagation.

Figure 6.2 clearly shows the performance difference between the verifier load time and the IFC check. The kernel verifier consistently shows very low load times (mostly under 0.2 ms), while our IFC component exhibits substantially higher audit times (ranging roughly between 55 and 62 ms).

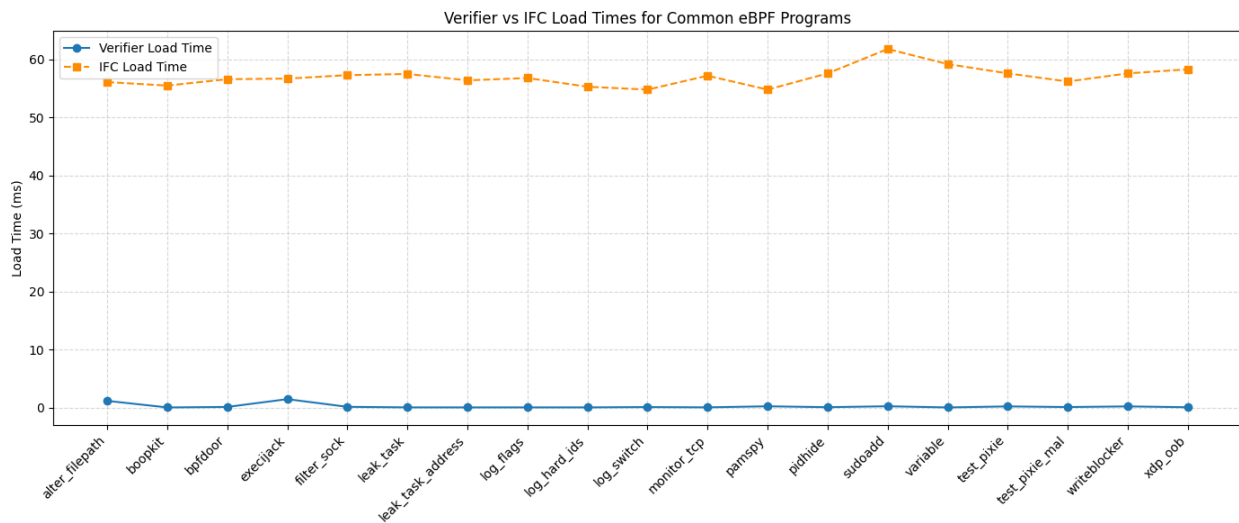


Figure 6.2: Comparison of kernel verifier and IFC load time

Several factors likely contribute to the higher overhead observed with the IFC loader compared to the native verifier:

First, our IFC static analysis is implemented in Python, inherently adding overhead compared to the kernel verifier implemented in highly optimized C. Furthermore, the IFC analysis heavily relies on external libraries, particularly Capstone for disassembly, which introduces additional processing delays not present in the native verification.

Another significant contributor to the overhead is the policy check mechanism. Each kernel data access within the program is individually checked against defined policies, adding cumulative processing time. Additionally, the IFC component employs Control Flow Graph (CFG) analysis involving basic blocks and taint state propagation, which further increases the complexity and execution time, especially for programs with a higher number of blocks or multiple program sections. Programs such as `sudoadd`, with five separate eBPF sections (`SEC()`), and `pidhide`, with three sections, naturally incur greater overhead due to the increased number of state merges across their multiple blocks.

These benchmarks highlight that while our IFC approach introduces non-negligible overhead at program load time, the security benefit of comprehensive sensitive data tracking and enforcement greatly outweighs the additional performance cost, especially considering that these analyses occur once at load time and impose no runtime penalties.

6.6 Summary

The IFC system successfully enforced fine-grained policies and blocked sensitive data leaks that the kernel verifier missed, including in real-world rootkits like boopkit, bpfdoor, and pampsy. It flagged both disallowed helper usage and data flows through allowed paths, showing the importance of tracking how data moves, not just what functions are used. The added overhead at load time is reasonable, and could be significantly reduced if the analysis were moved into the kernel.

Chapter 7

Related Work

Information flow control or IFC, as a concept started with some early work based on data confinement around the mid-1970s. Lampson [23] initially discussed confining an arbitrary program to prevent data leakage and later Rotenberg [35] proposed a mechanism to confine data in systems using a tree structure to enforce information flow and restrict secrets from flowing to untrusted entities, which he called "domains". Subsequently, Denning [13] formalized IFC policies using a lattice structure and Denning [14] later leveraged this and enforced IFC policies using static analysis. Building on these principles, Myers and Liskov [30] introduced a decentralized IFC model, enabling applications to classify and declassify their own data rather than relying on a centralized authority.

7.0.1 IFC in Systems

Several prior systems, including seL4 [28] and HiStar [44], have implemented dynamic taint tracking to enforce IFC at the kernel level. Asbestos [38] and Flume [22] have leveraged labels to implement decentralized information flow control(DIFC) to track and limit information

flow between unprivileged processes. Similarly, DStar [43] demonstrated the use of labels to enforce DIFC in a distributed environment.

7.0.2 Enhancing eBPF Security through Isolation Mechanisms

In the context of eBPF, recent research has explored both software and hardware features to enhance eBPF security through spatial isolation and also dynamic sandboxing. MOAT [27] leverages Intel Memory Protection Keys (MPK) to partition kernel memory, assigning distinct protection keys to eBPF programs and critical kernel data structures, thereby preventing unauthorized memory access even if verifier checks are bypassed. SafeBPF [26] extends this approach by combining software-based fault isolation (SFI) with ARM’s Memory Tagging Extension (MTE) to confine eBPF programs within sandboxed regions. While these techniques mitigate memory corruption risks, they primarily focus on enforcing spatial isolation rather than controlling information flow between sensitivity domains. Furthermore, they rely on hardware-specific features which restricts their applicability across different architectures. SandBPF [25], in contrast, utilizes a software-based dynamic sandboxing mechanism to isolate unprivileged eBPF programs.

7.0.3 Improving the verifier’s static analysis capabilities

Other works have been more focused on improving the eBPF verifier’s static analysis capabilities. Agni [40] developed soundness specifications to verify the abstract interpretation logic used by the verifier for value tracking and similarly PREVAIL’s [20] abstract interpretation handles complex loops and path explosions more efficiently than the default verifier. Sun et al. [36], on the other hand, used state embedding to detect logic bugs in the verifier by embedding concrete states as correctness checks. Despite these advancements, the approaches

discussed above primarily focus on enhancing the verifier's soundness and efficiency rather than addressing support for information flow control.

Chapter 8

Discussion & Future Work

In this chapter, we start by discussing the strengths of our IFC system. Then we discuss some of the current limitations, like cases where IFC is not enough on its own. We conclude with ideas for improving the system going forward, including how we can safely allow certain sensitive data to be used or stored when needed.

8.1 Discussion

Our static information flow analysis helps prevent eBPF programs from leaking sensitive data from the kernel. It also keeps programs from using helper functions that could let them do things outside their intended purpose. Even in cases when a program uses allowed helper functions, we can still limit what data they are able to access through the use of policies.

However, this all depends on how well the system administrator understands the eBPF subsystem to be able to define these policies correctly. Without this understanding, system administrators may write policies that grant excessive access, or ones that are too strict and

breaks legitimate programs.

One way to make this easier is by grouping policies based on the type and purpose of eBPF program which we implement. For example, XDP programs might have one set of rules, while tracepoints programs for scheduling have another. This helps reduce the cognitive load and gives administrators a good starting point, which they can tweak based on how the program is supposed to behave. It also encourages developers to provide some kind of provenance metadata or documentation with their programs, so that admins have a better idea of what the program is doing and how to enforce their preconceived behavior of the programs. This could scale our work in preventing supply chain attacks involving malicious or unexpected data leaks.

It is important to note, however, that the static IFC analysis does not detect or prevent all forms of malicious eBPF attacks. For instance, it cannot detect side-channel attacks that leak information by measuring execution time or CPU behavior. It also will not catch programs designed to exhaust kernel resources and cause denial-of-service (DoS). These types of attacks go beyond what static data tracking can detect and would need additional solutions.

8.2 Future Work

One thing our system focuses on is blocking sensitive data from being sent out or exposed. But there are real cases where you might want to use some form of the sensitive data, like a transformed or anonymized version. For example, you might want to store encrypted data in a map for future analysis. In future work, we aim to extend our framework to support safe *declassification* operations. These would allow certain transformations, such as encryption or other privacy-preserving techniques, to occur before sensitive data is stored in maps for

long-term aggregation or analysis.

Additionally, our system currently treats all eBPF maps as potentially malicious sinks for sensitive data. But there are cases where we might want to store some sensitive data in a map that we trust to not be accessible to malicious actors. We plan to introduce map-level granularity, allowing specific maps to be marked as trusted and safe for storing declassified information, while continuing to restrict access to others.

Finally, our IFC enforcement is currently implemented as a userspace module that runs before the eBPF verifier. In future iterations, we aim to integrate our system directly into the eBPF verifier itself. That way, policy enforcement for eBPF programs would be more seamless and not rely on an external preprocessing step.

Chapter 9

Conclusion

This work set out to explore how to better balance the need for access to sensitive data in kernel-level monitoring tools, like eBPF programs, with the need to prevent that data from leaking to malicious actors. We focused on improving the verification process that currently exists for eBPF programs, because while the verifier ensures programs do not crash the system, it does not stop them from reading or misusing sensitive data.

To address this, we designed a static information flow control (IFC) system that tracks what data a program accesses and ensures those accesses follow well-defined policies. We provide constraints on what functions the programs can use as well as, to ensure that even if sensitive data is accessed, it cannot leave the program unless it is explicitly allowed.

We implemented this as a lightweight step that runs before the verifier and complements what the kernel already does. Using a real-world example like Pixie, we showed that the system can effectively catch programs that try to leak data while still allowing safe, legitimate programs to run without issues.

Through this thesis, we introduced a new way to define and enforce fine-grained sensitivity

policies for eBPF programs. We also developed a mechanism for tracking and propagating sensitivity labels through program execution, and built a working system that can distinguish between safe and unsafe programs, even when both access sensitive data.

Overall, this work moves us closer to building safer and more trustworthy systems by giving system operators better tools for controlling how kernel data is accessed and handled by eBPF programs.

Bibliography

- [1] pwru (packet, where are you?). <https://github.com/cilium/pwru>.
- [2] Lsm bpf programs. https://docs.kernel.org/bpf/prog_lsm.html, May 2024.
- [3] Tetragon. <https://github.com/cilium/tetragon>, May 2024.
- [4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2 edition, 2006. ISBN 978-0321486813.
- [5] Eunomia bpf Project Contributors. eunomia-bpf: Open-source tools for building and managing ebpf programs, 2025. URL <https://github.com/eunomia-bpf/eunomia-bpf>. Accessed: 2025-01-09.
- [6] Neha Chowdhary, Utkalika Satapathy, Theophilus A. Benson, Subhrendu Chattopadhyay, Palani Kodeswaran, Sayandeep Sen, and Sandip Chakraborty. BeeGuard: Explainability-based policy enforcement of ebpf codes for cloud-native environments. In *Proceedings of the 17th International Conference on Communication Systems & Networks (COMSNETS)*, Bengaluru, India, 2025. URL <https://subhrendu1987.github.io/pub/papers/2024.COMSNETS.Beeguard.pdf>. Accessed: 2025-02-03.
- [7] Cilium Project Contributors. Cilium: ebpf-based networking, observability, and security, 2025. URL <https://cilium.io/>. Accessed: 2025-01-09.

- [8] Falco Project Contributors. Falco: Open source runtime security, 2025. URL <https://falco.org/>. Accessed: 2025-01-09.
- [9] IOVisor Project Contributors. Bcc: Tools for bpf-based linux io analysis, networking, and monitoring, 2025. URL <https://github.com/iovisor/bcc>. Accessed: 2025-01-09.
- [10] Pixie Contributors. Pixie: Open-source observability for kubernetes, 2025. URL <https://github.com/pixie-io/pixie>. Accessed: 2025-01-09.
- [11] Pyroscope Project Contributors. Pyroscope: Continuous profiling for developers, 2025. URL <https://pyroscope.io/>. Accessed: 2025-01-09.
- [12] Deep Instinct Threat Research. Bpfdoor malware evolves: Stealthy sniffing backdoor ups its game, 2024. URL <https://www.deepinstinct.com/blog/bpfdoor-malware-evolves-stealthy-sniffing-backdoor-ups-its-game>. Accessed: 2025-03-30.
- [13] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360056. URL <https://dl.acm.org/doi/10.1145/360051.360056>.
- [14] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359712. URL <https://dl.acm.org/doi/10.1145/359636.359712>.
- [15] Linux Kernel Developers. Bpf verifier documentation, 2025. URL <https://docs.kernel.org/bpf/verifier.html>. Accessed: 2025-01-09.
- [16] eBPF Community. ebpf case studies: Real-world use cases for ebpf technology, 2025. URL <https://ebpf.io/case-studies/>. Accessed: 2025-01-09.

- [17] Linux Foundation. Controlplane — eBPF security threat model, 2025. URL <https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20%E2%80%94%20eBPF%20Security%20Threat%20Model.pdf>. Accessed: 2025-01-09.
- [18] Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau. With friends like eBPF, who needs enemies? In *Black Hat USA 2021*, 2021. URL <https://www.blackhat.com/us-21/briefings/schedule/#with-friends-like-ebpf-who-needs-enemies-23619>. Accessed: 2025-03-26.
- [19] Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau. eBPF, I thought we were friends! In *DEF CON 29*, 2021. URL <https://defcon.org/html/defcon-29/dc-29-speakers.html#fournier>. Accessed: 2025-03-26.
- [20] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, pages 1069–1084, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314590. URL <https://dl.acm.org/doi/10.1145/3314221.3314590>.
- [21] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered eBPF on clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/he>.
- [22] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions.

- SIGOPS Oper. Syst. Rev.*, 41(6):321–334, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294293. URL <https://dl.acm.org/doi/10.1145/1323293.1294293>.
- [23] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10): 613–615, October 1973. ISSN 0001-0782. doi: 10.1145/362375.362389. URL <https://dl.acm.org/doi/10.1145/362375.362389>.
- [24] libbpf Community Contributors. bpftool: Bpf inspection and debugging tools, 2025. URL <https://github.com/libbpf/bpftool>. Accessed: 2025-01-09.
- [25] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, eBPF '23, pages 42–48, New York, NY, USA, September 2023. Association for Computing Machinery. ISBN 979-8-4007-0293-8. doi: 10.1145/3609021.3609301. URL <https://dl.acm.org/doi/10.1145/3609021.3609301>.
- [26] Soo Yee Lim, Tanya Prasad, Xueyuan Han, and Thomas Pasquier. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. In *Proceedings of the 2024 on Cloud Computing Security Workshop*, CCSW '24, pages 80–94, New York, NY, USA, November 2024. Association for Computing Machinery. ISBN 979-8-4007-1234-0. doi: 10.1145/3689938.3694781. URL <https://dl.acm.org/doi/10.1145/3689938.3694781>.
- [27] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. {MOAT}: Towards Safe {BPF} Kernel Extension. pages 1153–1170, 2024. ISBN 978-1-939133-44-1. URL <https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi>.
- [28] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From General Purpose

- to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429, May 2013. doi: 10.1109/SP.2013.35. URL <https://ieeexplore.ieee.org/document/6547124>. ISSN: 1081-6011.
- [29] Andrew C. Myers. Jflow: Practical mostly-static information flow control. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.
- [30] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 129–142, New York, NY, USA, October 1997. Association for Computing Machinery. ISBN 978-0-89791-916-6. doi: 10.1145/268998.266669. URL <https://dl.acm.org/doi/10.1145/268998.266669>.
- [31] Kris Nóva. Boopkit: Linux ebpf backdoor over tcp. <https://github.com/krisnova/boopkit/tree/b8dc4ee0c9a7eeb042e20835f26591776f7a6cff>, 2022. URL <https://github.com/krisnova/boopkit/tree/b8dc4ee0c9a7eeb042e20835f26591776f7a6cff>. Accessed: 2025-03-30.
- [32] Pathtofile. Bad bpf: A collection of malicious ebpf programs. <https://github.com/pathtofile/bad-bpf>. Accessed: 2025-03-26.
- [33] pathtofile. bad-bpf. <https://github.com/pathtofile/bad-bpf>, 2025. Accessed: 2025-04-23.
- [34] Embrace The Red. ebpf blog posts. <https://embracethered.com/blog/tags/ebpf/>. Accessed: 2025-03-26.
- [35] Leo Joseph Rotenberg. *Making computers keep secrets*. Thesis, Massachusetts Insti-

- tute of Technology, 1973. URL <https://dspace.mit.edu/handle/1721.1/33481>. Accepted: 2006-11-15T12:47:43Z.
- [36] Hao Sun and Zhendong Su. Validating the {eBPF} Verifier via State Embedding. pages 615–628, 2024. ISBN 978-1-939133-40-3. URL <https://www.usenix.org/conference/osdi24/presentation/sun-hao>.
- [37] Acronis Cyber Protection Team. Symbiote: A new stealthy malware for linux. <https://www.acronis.com/en-us/cyber-protection-center/posts/symbiote-a-new-stealthy-malware-for-linux/>, 2022. URL <https://www.acronis.com/en-us/cyber-protection-center/posts/symbiote-a-new-stealthy-malware-for-linux/>. Accessed: 2025-03-30.
- [38] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11–es, December 2007. ISSN 0734-2071. doi: 10.1145/1314299.1314302. URL <https://dl.acm.org/doi/10.1145/1314299.1314302>.
- [39] David Vernet. More features and use-cases for sched_ext. In *LSM/MM/BPF Summit 2024*, May 2024.
- [40] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the Verifier: eBPF Range Analysis Verification. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 226–251, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-37709-9. doi: 10.1007/978-3-031-37709-9_12.
- [41] Nezer Zaidenberg, Michael Kiperberg, Eliav Menachi, and Asaf Eitani. Detecting ebpf rootkits using virtualization and memory forensics. In *Proceedings of the 10th*

- International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 254–261. INSTICC, SciTePress, 2024. ISBN 978-989-758-683-5. doi: 10.5220/0012470800003648.
- [42] Nikolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Secure information flow by tracking storage in operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 129–142. ACM, 2006.
- [43] Nikolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI’08, pages 293–308, USA, April 2008. USENIX Association.
- [44] Nikolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011. ISSN 0001-0782, 1557-7317. doi: 10.1145/2018396.2018419. URL <https://dl.acm.org/doi/10.1145/2018396.2018419>.