BPFflow - Preventing information leaks from eBPF

Chinecherem Dimobi
chinedimobi@vt.edu
Virginia Tech
Blacksburg, VA, USA

Rahul Tiwari
rahult@vt.edu
Virginia Tech
Blacksburg, VA, USA

Zhengjie Ji
zhengjie@vt.edu
Virginia Tech
Blacksburg, VA, USA

Dan Williams
djwillia@vt.edu
Virginia Tech
Blacksburg, VA, USA

ABSTRACT

eBPF has seen major industry adoption by enterprises to enhance observability, tracing, and monitoring by hooking at different points in the kernel. However, since the kernel is a critical resource, eBPF can also pose as a threat if misused, potentially leading to privilege escalation, information leaks and more. While effective to some extent, existing mitigation strategies like interface filtering are coarse-grained and often over-restrictive. We propose BPFflow, a flexible framework for the system administrator to define policies that specify sensitive data sources, trusted sinks and permitted flows between them. These policies are enforced by an Information Flow Control (IFC) system within the eBPF verifier to track the propagation of sensitive data to prevent unauthorized leakage to userspace or any other untrusted sinks without any runtime overhead.

CCS CONCEPTS

• **Security and privacy** → **Information flow control**;

KEYWORDS

eBPF, Information Flow Control

ACM Reference Format:
Chinecherem Dimobi, Rahul Tiwari, Zhengjie Ji, and Dan Williams. 2025. BPFflow - Preventing information leaks from eBPF. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3748355.3748374

1 INTRODUCTION

eBPF is a Linux kernel extension technology which operates by attaching user-defined programs to specific kernel hook points with minimal overhead, which makes them an ideal choice for observability, tracing, monitoring, and network security. Cilium, Falco, Pixie, Tetragon, and Pyroscope [2–6] are some of the many tools which leverage eBPF to deliver advanced observability and security capabilities. Furthermore, large enterprises and projects such as Netflix, Google, Android, and Microsoft rely on eBPF to optimize performance and enhance their system monitoring capabilities [10].

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 [11]. The severity of these threats is evident from both research and real-world incidents [15] and the key to these threats is the misuse of a set of trusted kernel interfaces like eBPF helper functions and kfuncs. `bpf_probe_write_user`, `bpf_override_return` and `bpf_send_signal` [35] are some examples of helper functions which can facilitate attacks such as privilege escalation by overwriting return addresses in userspace, hijacking kernel function behavior and triggering signal handlers to induce unintended behavior. Prior research [1] has looked into preventing the usage of helper functions by generating policies based on program semantics that allow or deny eBPF programs. However, functions such as `bpf_probe_read` that can be used maliciously for accessing and leaking sensitive kernel data are also essential for benign applications like profiling and performance monitoring. Therefore, a key challenge is to distinguish between benign and malicious use of eBPF programs while ensuring that legitimate use is not overly restricted. We propose BPFflow, an information flow control system leveraging labels to enforce fine-grained policies for eBPF programs. The main contributions of the paper are as follows:

- We observe that seemingly benign eBPF programs can pose risks to production systems, as slight modifications may enable them to leak data to untrusted sinks.
- We introduce BPFflow, a declarative policy language that assigns and propagates labels throughout an eBPF program to enforce information flow control.
- Our system provides a way to express fine-grained policies per eBPF program, allowing system administrators to follow least-privilege best practices without imposing unnecessary constraints on legitimate programs.

2 THREAT MODEL

While eBPF programs have been previously used for malicious purposes, we focus on scenarios where a system administrator employs

eBPF '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/25/09.
https://doi.org/10.1145/3748355.3748374

Helper Function	Purpose
bpf_probe_write_user	Write user space memory
bpf_probe_read_user	Read user space memory
bpf_override_return	Alter kernel function retval
bpf_send_signal	Send SIGKILL to any process

Table 1: Some commonly used helper functions in malicious eBPF programs

third-party eBPF programs. We assume that the system administrator is trusted not to write their own malicious eBPF programs, and all programs will pass the verifier, which we consider bug-free for the purposes of this discussion. We also assume that all kernel-side protections still exist and the adversary cannot modify the kernel (e.g., modules) except for eBPF programs. The eBPF programs may be used for monitoring and observability for performance insights, security features such as intrusion detection (IDS) or intrusion prevention (IPS), and networking enhancements to efficiently route and manage traffic. However, malicious eBPF programs could 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.

Under this threat model, numerous attacks have been demonstrated and researchers as well as industry experts have become interested in understanding potential malicious use cases of eBPF [12, 13, 26, 28, 30]. The Linux Foundation also conducted a comprehensive security threat modeling of eBPF [11], highlighting possible threats such as malicious use of certain eBPF helpers (outlined in Table 1), sensitive information leakage, supply chain attacks, denial-of-service (DoS), rootkits, and detection evasion. One notable example is BPFDoor [7], a stealthy backdoor identified by PwC that leveraged eBPF to bypass firewall rules and evade forensic detection. Symbiote [32], on the other hand was able to collect user data and exfiltrate it to DNS servers. Pamspy [27] is another known proof-of-concept credential dumper that targets the Pluggable Authentication Module (PAM) framework in Linux by hooking into PAM-related function calls and silently collecting and exfiltrating authentication credentials.

2.1 Example: Weaponizing Observability

To effectively steal sensitive information, two conditions must be met: (1) there must be a source of sensitive data, and (2) there must be a channel through which the data can be exfiltrated. The core functionality of eBPF inherently satisfies both requirements by allowing programs to hook into the kernel and exfiltrate sensitive information through a map (explicit flow) or influence program behavior through control based decisions which depend on sensitive data (implicit flow).

We take a sample eBPF program from Pixie [4], a legitimate open-source observability tool for Kubernetes applications that utilizes eBPF for monitoring. With slight modification, as shown in Figure 1 (indicated by lines marked with '+'), we show that the program can be extended to access sensitive information. For example, instead of accessing the `start_boottime` to use as an

```
SEC("tracepoint/sched/sched_process_exec")
int trace_exec(struct
    trace_event_raw_sched_process_exec *ctx) {
    struct task_struct *task;
-   struct task_struct *group;
-   u64 start_time;
+   struct cred *cred;
+   struct request_key_auth *auth;

    // Step 1: Get pointer to current task struct
    task = (struct task_struct *)
        bpf_get_current_task();

    // Step 2: Read sensitive pointers from task
        struct
-   bpf_probe_read(&leader, sizeof(leader), &task
    ->group_leader);
-   bpf_probe_read(&start_time, sizeof(start_time)
    , &leader->start_boottime);
+   bpf_probe_read(&cred, sizeof(cred), &task->
    cred);
+   bpf_probe_read(&auth, sizeof(auth), &cred->
    request_key_auth);

    // Step 3: Store data outside program
-   bpf_perf_event_output(ctx, &perf_buf,
    BPF_F_CURRENT_CPU, &start_time, sizeof(
    start_time));
+   bpf_perf_event_output(ctx, &perf_buf,
    BPF_F_CURRENT_CPU, &cred, sizeof(cred));
+   bpf_perf_event_output(ctx, &perf_buf,
    BPF_F_CURRENT_CPU, &auth, sizeof(auth));
    return 0;
}
```

Figure 1: Small changes to a benign Pixie program can leak potentially sensitive data

ID, the modified program now accesses the request_key_auth field within the credential structure of the task struct. Like the original, the modified program passes the verifier. This small change could easily happen during a supply-chain attack and 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.[1] The core issue here is that currently there is 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.

2.2 Present Security Mechanisms are not Enough

While the eBPF verifier, ensures safety by preventing unsafe operations such as invalid memory access or unbounded loops, it lacks awareness of the sensitivity or confidentiality of the data being accessed and semantics of the data flowing through the program.

[1] `cred` in Figure 1 is shown directly as a struct field name for illustrative purposes; an attacker could instead use offsets to sensitive struct fields, making such access patterns harder to detect.

Furthermore, 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. Other approaches that involve limiting programs solely based on helper functions are coarse-grained and often end up restricting more functionality than intended.

3 BPFFLOW

Figure 2 depicts the high-level design of BPFflow. As a verifier component, it operates at load time, which ensures early rejection of a potentially malicious eBPF program with no effect on runtime performance. There are two major components of our design: the policy specification, which defines what is allowed, and the IFC static analysis engine, which enforces the defined policies.

3.1 Policy Specification

In our design, the system administrator specifies policies for each eBPF program using a human-readable JSON format that defines the sources of information for an eBPF program. These sources fall into two categories: data access and kernel interfaces such as helper functions and kfuncs.

The policies use per-field 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 be exfiltrated.

Listing 1 shows an example of the policy for the Pixie program discussed in Section 2.1. The policy defines that the program is allowed to access the `group_leader` and `start_boottime` fields within the `task_struct` of the running task, while the `cred` field is marked as sensitive. Additionally, the policy assigns labels to helpers to define per-helper access levels, where *sensitive* label refers to helpers that return sensitive data. The policy specification follows a *deny-by-default* model, which means 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 introduce predefined policy groups that can be assigned to eBPF program types like tracing and networking. These group policies grant reasonable default access to commonly used interfaces (e.g., helpers for retrieving process IDs) and non-sensitive data structures relevant to the program type.

3.2 IFC Static Analysis Engine

IFC static analysis engine performs static analysis on the eBPF bytecode by iterating through each instruction in the eBPF bytecode to generate a Control Flow Graph (CFG). We track data flow through each register and propagate taints by modeling an in-state and out-state with each basic block and use a worklist-based algorithm to propagate states along the CFG by merging states. At each data access, we consult the defined policy and if an access is disallowed, the program is rejected. Otherwise, the register is labeled (e.g., `sensitive`) and tracked through the program. Label propagation

```
{
  "program": {
    "name": "pixie.kern",
    "type": "observability",
    "hookpoint":
    "tracepoint/sched/sched_process_exec"
  },
  "data_access": {
    "task_struct": {
      "allow": ["group_leader",
      ↪  "start_boottime"],
      "sensitive": ["cred"],
      "description": "Access to the
      ↪  task_struct pointer"
    }
  },
  "helper_access_profile": {
    "bpf_get_current_task": "allow",
    "bpf_get_stackid": "allow",
    "bpf_probe_read": "sensitive",
    "bpf_map_delete_elem": "deny",
    "bpf_get_current_pid_tgid": "allow",
    "bpf_perf_event_output": "allow"
  }
}
```

Listing 1: An example of the policy for `pixie`

handles both explicit and implicit flows. For explicit flows, register assignments carry over labels:

```
x = kernel_data;
y = x; // y inherits label from x
```

For implicit flows, we taint successor blocks of conditionals that depend on sensitive values:

```
if (secret)
    y = 1; // y is tainted
```

The unified taint propagation mechanism tracks all values—directly or indirectly—derived from sensitive data, and rejects programs at load time if any such value reaches an untrusted sink defined by policy.

We have developed a prototype userspace implementation in Python to audit untrusted eBPF programs to ensure they only access what is expected of them.

4 RESULTS & EVALUATION

In this section, we evaluate BPFflow to understand how well it achieves its goal of preventing sensitive data leakage through untrusted eBPF programs. We aim to answer the following questions:

(1) **Security and policy enforcement:** Can BPFflow accurately prevent unauthorized data access and helper functions? Is it useful for detecting sensitive leakage?
(2) **Real world applicability:** Can BPFflow identify and prevent loading rootkits?

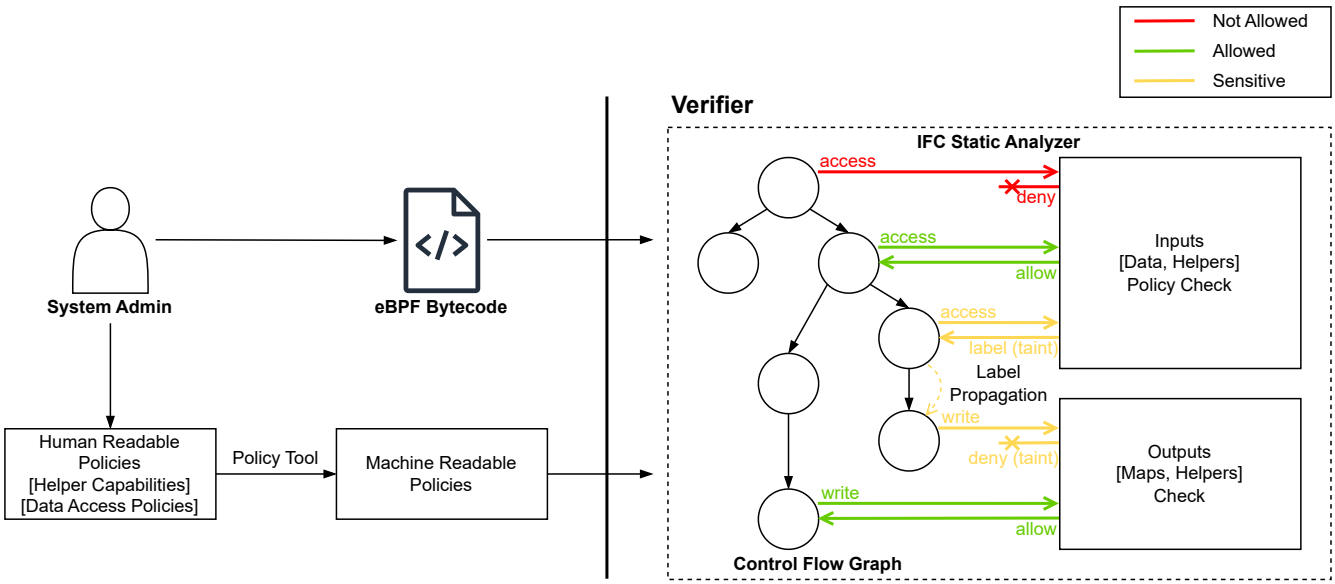

Figure 2: Information Flow Control System in eBPF

(3) **System impact:** What is the impact of BPFflow on the program's load-time performance?

4.1 Evaluation of BPFflow

For this evaluation, we assume the system administrator has a basic understanding of observability-type eBPF programs. To construct representative policies for observability programs, we gathered examples from open-source repositories including libbpf-tools [29], Beyla [18], and Pixie [4]. Some of these programs include *cachestat*, *cpufreq*, *llcstat*, *mountsnoop*, and *watcher* that monitor performance, events and cache activity within the kernel. Using these programs, we formulated an observability group policy specifying allowed hookpoints, allowed kernel data fields, and permissible helper functions. By extension, these programs pass the BPFflow check. In the policy, we explicitly tagged fields that may be sensitive, and in this case credentials or data read from user-space.

To evaluate policy enforcement by BPFflow, we compiled a set of eBPF programs from open-source repositories [28], shown in Table 2. These include known malicious programs tagged as "bad" due to their misuse of helper functions that could leak or alter kernel state. Additionally, we included programs not typically considered malicious but designed to leak sensitive kernel data beyond permitted observability scopes through map writes, debug outputs, or return values. All of these programs are run under the observability group policy described above, simulating the case in which the system administrator is tricked into running them.

We conducted two tests on each program. The first was a standard verifier check, compiling and loading the program into the kernel without IFC measures. Most programs passed the verifier except *textreplace*, which was rejected due to out-of-bounds memory access.

In the second test, programs were evaluated using BPFflow. BPFflow validates each hookpoint attachment (such as data read from the `ctx` argument in `R1`), kernel data access (from context or return from helper functions that return kernel data), and helper function usage against the defined observability policies. For instance, the *capturelsm* program violates policy by attaching to the `lsmfilemkdir` hookpoint, which was not permitted and outside the scope for observability. Known malicious helper functions are explicitly blocked, except `bpf_probe_read`, which is widely used for legitimate observability purposes. Hence, programs like *alterfilepath* that uses `bpf_probe_write` to alter an access file path violates the policy. In the case of our modified *pixie* from Figure 1, the program accesses the credential field of the task struct and tries to store it in a map which violates sensitive data leaving the program.

As we can see from Table 2, all the evaluated programs fail the BPFflow check from at least one of the violations discussed above. These findings demonstrate BPFflow's capability to detect a wide range of violations that are not detected by the verifier alone and shows how BPFflow is able to effectively place constraints on what data is accessed by the eBPF programs by enforcing policies. They also highlight the importance of tracking sensitive data flow explicitly, as relying only on helper blacklisting, and data access would miss several realistic attacks. Note that while the policies here are tailored to observability programs, these policies can be as flexible or as strict as possible for other eBPF program types. Policies can also be tied to a particular eBPF program as opposed to eBPF program type.

4.2 Evaluation on Real-World Rootkits

To evaluate whether BPFflow correctly detects and blocks real-world threats, we tested it against known malicious eBPF rootkits. Specifically, we evaluated three rootkits: `boopkit`, `bpfdoor`, and `pamspy`, of which the latter two were described earlier in Section 2.

Program	Description	Verifier	BPFflow	IFC Violation			
				Hookpoint	Helper	Data Access	Data Leakage
exechijack	Hijacks process and sends a message to a map	✓	✗			✗	
pidhide	Hides process from directory listing	✓	✗		✗	✗	
sudoadd	Elevates user privileges by editing sudoers	✓	✗		✗		✗
textreplace	Finds and replaces a target string	✗	✗		✗		✗
alterfilepath	Alters file access path via context manipulation	✓	✗		✗		
writeblocker	Blocks write syscalls for specific processes	✓	✗	✗			
logflags	Logs syscall arguments and trace flags	✓	✗			✗	
filtersock	Filter sockets and leaks sensitive data through return path	✓	✗	✗			
logswitch	Logs sensitive context values	✓	✗			✗	
capturelsm	Monitors LSM mkdir events	✓	✗	✗			
leaktaskaddrs	Leaks task struct address to map	✓	✗				✗
loghardids	Leaks hardcoded sensitive fields	✓	✗	✗			
xdpOOB	Performs out-of-bounds access on XDP data	✓	✗	✗			
pixiemal	Accessing and leaking credentials fields outside of its scope	✓	✗				✗

Table 2: Evaluation of security and policy enforcement of BPFflow

During our evaluation, each rootkit underwent auditing through BPFflow's IFC engine. All three rootkits were successfully prevented from attaching to the kernel due to two primary enforcement mechanisms:

First, since BPFflow employs a "deny-by-default" policy model, these malicious programs lack explicit policies defined by the system administrator, causing immediate rejection upon attempted loading. Secondly, in the case where these programs attempt to impersonate a legitimate program type (i.e observability) using the group policies, they were rejected because of hookpoint context access not specified in the policies. Furthermore, even if these hookpoints are specified in the policies, the rootkits still trigger data access or data leakage violations. This occurs because these malicious programs inevitably attempt to access kernel data fields not explicitly permitted in legitimate policies or attempt to exfiltrate sensitive kernel data via helper functions such as `bpf_map_update_elem` and `bpf_trace_printk`.

Table 3 summarizes the results of our evaluation against these rootkits, demonstrating BPFflow's robustness in detecting and blocking realistic eBPF-based threats.

Rootkit	Hookpoint	Data Leak	IFC Check
`bpfdoor`	Socket Filter (TCP)	Backdoor payload	✗
`pamspy`	Uretprobe (PAM)	User credentials	✗
`boopkit`	Tracepoint (TCP)	Packet data	✗

Table 3: Detection of Real-World eBPF Rootkits by IFC Enforcement

These results clearly demonstrate BPFflow's practical effectiveness in blocking real-world rootkits by enforcing policies on eBPF programs.

4.3 Performance Overhead

To assess the performance impact of integrating Information Flow Control (IFC) static analysis into the eBPF pipeline, we benchmark our Python prototype implementation using hyperfine [16] to simulate warmup runs. We observed that BPFflow introduces a one-time overhead of 55–62 ms at load time, with no runtime overhead. However, we expect significant performance gains if our implementation is ported to the in-kernel verifier, since we can piggyback on its existing instruction walk and CFG logic. Additionally, a C-based implementation would be substantially faster than our current Python-based approach.

5 DISCUSSION & FUTURE WORK

Policy specification helps prevent eBPF programs from leaking sensitive data from the kernel. However, it 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 break legitimate programs. We believe that policy groups, combined with metadata and provenance provided by eBPF developers, can reduce the burden on system administrators and improve scalability.

A promising direction for future research is to extend the framework to support runtime classification and declassification of data. For example, an eBPF program could classify packet data as sensitive based on its contents (e.g., detecting a password field) and subsequently declassify it after applying some form of hash/encryption. The proposed approach can also be extended to support dynamic context-dependent flow restrictions. For instance, a kernel pointer may be logged only if the process belongs to a trusted cgroup. This would allow enforcing fine-grained, conditional policies beyond static label checking. Additionally, it is important to note 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.

6 RELATED WORK

Information flow control or IFC, as a concept started with some early work based on data confinement around the mid-1970s [19]. Later Denning and Denning [8, 9] formalized IFC policies using a lattice structure combined with static analysis. Building on these principles, Myers and Liskov [25] introduced a decentralized IFC model, enabling applications to classify and declassify their own data rather than relying on a centralized authority. Several prior systems, have implemented dynamic taint tracking to enforce IFC at the kernel level [24, 36] and leveraged labels to limit information flow [17, 33, 37].

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 [23] leverages Intel Memory Protection Keys (MPK) to partition kernel memory, while SafeBPF [21] extends this approach by combining software-based fault isolation (SFI) with ARM's Memory Tagging Extension (MTE) to confine eBPF programs within sandboxed regions. SandBPF [20], in contrast, utilizes a software-based dynamic sandboxing mechanism to isolate unprivileged eBPF programs. 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.

Some previous work [14, 31, 34] have developed soundness specifications for the eBPF verifier and also improved its efficiency. Other efforts [22, 31] have focused on improving the functional correctness of the verifier. Tetragon[6] uses policies operating at the syscall and LSM hook granularity to passively audit BPF-related activity in a Kubernetes cluster. Despite these advancements, the approaches fail to address support for information flow control.

7 CONCLUSION

While the eBPF verifier ensures that programs do not crash, hang, or perform invalid memory accesses, it does not prevent them from accessing or misusing sensitive kernel data. Our work, BPFflow, addresses potential malicious usage of BPF programs. We supplement the current verification process with additional security guarantees by leveraging fine-grained policies enforced by an information flow control framework. The framework tracks the propagation of sensitive data through registers, stack, and helper calls in eBPF programs to ensure confidential kernel data cannot be leaked to any untrusted sinks.

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.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback and insightful comments. This work was funded in part by NSF grant CNS-2236966 and by a grant from 4-VA, a collaborative partnership for advancing the Commonwealth of Virginia.

9 ETHICAL CONSIDERATIONS

This work does not raise any ethical issues.

REFERENCES

[1] Neha Chowdhary, Utkalika Satapathy, Theophilus A. Benson, Subhrendu Chattopadhyay, Palani Kodeswaran, Sayandeep Sen, and Sandip Chakraborty. 2025. 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. https://subhrendu1987.github.io/pub/papers/2024.COMSNETS.Beeguard.pdf Accessed: 2025-02-03.

[2] Cilium Project Contributors. 2025. Cilium: eBPF-based Networking, Observability, and Security. (2025). https://cilium.io/ Accessed: 2025-01-09.

[3] Falco Project Contributors. 2025. Falco: Open Source Runtime Security. (2025). https://falco.org/ Accessed: 2025-01-09.

[4] Pixie Contributors. 2025. Pixie: Open-source observability for Kubernetes. (2025). https://github.com/pixie-io/pixie Accessed: 2025-01-09.

[5] Pyroscope Project Contributors. 2025. Pyroscope: Continuous Profiling for Developers. (2025). https://pyroscope.io/ Accessed: 2025-01-09.

[6] Tetragon Project Contributors. 2025. Tetragon: eBPF-based Security Observability and Enforcement. (2025). https://tetragon.io/ Accessed: 2025-01-09.

[7] Deep Instinct Threat Research. 2024. BPFdoor Malware Evolves: Stealthy Sniffing Backdoor Ups Its Game. (2024). https://www.deepinstinct.com/blog/bpfdoor-malware-evolves-stealthy-sniffing-backdoor-ups-its-game Accessed: 2025-03-30.

[8] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

[9] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712

[10] eBPF Community. 2025. eBPF Case Studies: Real-world Use Cases for eBPF Technology. (2025). https://ebpf.io/case-studies/ Accessed: 2025-01-09.

[11] Linux Foundation. 2025. ControlPlane — eBPF Security Threat Model. (2025). https://www.linuxfoundation.org/hubfs/eBPF/ControlPlane%20%E2%80%94%20eBPF%20Security%20Threat%20Model.pdf Accessed: 2025-01-09.

[12] Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau. 2021. eBPF, I Thought We Were Friends!. In *DEF CON 29*. https://defcon.org/html/defcon-29/dc-29-speakers.html#fournier Accessed: 2025-03-26.

[13] Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau. 2021. With Friends Like eBPF, Who Needs Enemies?. In *Black Hat USA 2021*. https://www.blackhat.com/us-21/briefings/schedule/#with-friends-like-ebpf-who-needs-enemies-23619 Accessed: 2025-03-26.

[14] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. 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)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. https://doi.org/10.1145/3314221.3314590

[15] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. 2023. Cross Container Attacks: The Bewildered eBPF on Clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5971–5988. https://www.usenix.org/conference/usenixsecurity23/presentation/he

[16] hyperfine [n. d.]. ([n. d.]). https://github.com/sharkdp/hyperfine

[17] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 321–334. https://doi.org/10.1145/1323293.1294293

[18] Grafana Labs. 2024. Beyla - eBPF-based auto-instrumentation agent for observability. https://github.com/grafana/beyla. (2024). Accessed: May 20, 2025.

[19] Butler W. Lampson. 1973. A note on the confinement problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615. https://doi.org/10.1145/362375.362389

[20] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2023. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions (eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 42–48. https://doi.org/10.1145/3609021.3609301

[21] Soo Yee Lim, Tanya Prasad, Xueyuan Han, and Thomas Pasquier. 2024. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. In *Proceedings of the 2024 on Cloud Computing Security Workshop (CCSW '24)*. Association for Computing Machinery, New York, NY, USA, 80–94. https://doi.org/10.1145/3689938.3694781

[22] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. 2024. Towards Functional Verification of eBPF Programs. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions (eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/3672197.3673435

[23] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2024. {MOAT}: Towards Safe {BPF} Kernel Extension. 1153–1170. https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi

[24] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy*. 415–429. https://doi.org/10.1109/SP.2013.35 ISSN: 1081-6011.

[25] Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/268998.266669

[26] Kris Nóva. 2022. Boopkit: Linux eBPF backdoor over TCP. https://github.com/krisnova/boopkit/tree/b8dc4ee0c9a7eeb042e20835f26591776f7a6cff. (2022). https://github.com/krisnova/boopkit/tree/b8dc4ee0c9a7eeb042e20835f26591776f7a6cff Accessed: 2025-03-30.

[27] pamspy [n. d.]. pampsy - Credentials Dumper for Linux using eBPF. ([n. d.]). https://github.com/citronneur/pamspy

[28] Pathtofile. [n. d.]. Bad BPF: A Collection of Malicious eBPF Programs. https://github.com/pathtofile/bad-bpf. ([n. d.]). Accessed: 2025-03-26.

[29] IOVisor Project. 2024. libbpf-tools - Collection of eBPF tools using libbpf. https://github.com/iovisor/bcc/tree/f2d3803272fcd39888b75bb508df8f095ad02411/libbpf-tools. (2024). Accessed: May 20, 2025.

[30] Embrace The Red. [n. d.]. eBPF Blog Posts. https://embracethered.com/blog/tags/ebpf/. ([n. d.]). Accessed: 2025-03-26.

[31] Hao Sun and Zhendong Su. 2024. Validating the {eBPF} Verifier via State Embedding. 615–628. https://www.usenix.org/conference/osdi24/presentation/sun-hao

[32] Acronis Cyber Protection Team. 2022. Symbiote: A new stealthy malware for Linux. (2022). https://www.acronis.com/en-us/cyber-protection-center/posts/symbiote-a-new-stealthy-malware-for-linux/ Accessed: 2025-03-30.

[33] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. 2007. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.* 25, 4 (Dec. 2007), 11–es. https://doi.org/10.1145/1314299.1314302

[34] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 226–251. https://doi.org/10.1007/978-3-031-37709-9_12

[35] Nezer Zaidenberg, Michael Kiperberg, Eliav Menachi, and Asaf Eitani. 2024. Detecting eBPF Rootkits Using Virtualization and Memory Forensics. In *Proceedings of the 10th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*. INSTICC, SciTePress, 254–261. https://doi.org/10.5220/0012470800003648

[36] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2011. Making information flow explicit in HiStar. *Commun. ACM* 54, 11 (Nov. 2011), 93–101. https://doi.org/10.1145/2018396.2018419

[37] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, USA, 293–308.