BBArmor: a Dynamic BPF-to-BPF LSM-Based Enforcement Tool

Fabio Piras
Dept. of Information Engineering
University of Pisa
Pisa, Italy
fabio.piras@ing.unipi.it

Giuseppe Lettieri
Dept. of Information Engineering
University of Pisa
Pisa, Italy
giuseppe.lettieri@unipi.it

Gregorio Procissi
Dept. of Information Engineering
University of Pisa
Pisa, Italy
gregorio.procissi@unipi.it

***Abstract*—In modern day applications, eBPF has emerged as a powerful mechanism for extensible networking, observability, and security. Yet its elevated in-kernel privileges also create new attack avenues, since third-party tooling and supply-chain compromises can introduce malicious BPF loaders. A stealthy attacker may embed trojaned eBPF programs in legitimate tools and application or exploit vulnerable plugins to gain `CAP_BPF` rights, then probe syscalls, trace kernel events and exfiltrate sensitive data; often without raising traditional alarms. In this paper we propose a threat model to encompass these attack vectors for infrastructure administrator and semi-trusted cloud environment where BPF itself becomes both a tool and a target. We introduce BPF-to-BPF Armor (BBArmor), a prototype solution that enforces stricter controls over BPF syscall usage, isolates BPF programs based on provenance and trust levels and blocks anomalous BPF interactions indicative of compromise. Our evaluation demonstrates that BBArmor mitigates BPF syscall misuse with minimal performance overhead, strengthening security against evolving supply-chain and software-supply threats.**

***Index Terms*—eBPF, LSM, supply-chain-attacks, security, cloud-infrastructure**

I. Introduction

eBPF is a programmable in-kernel sandbox that offers unparalleled flexibility for observability, networking, and security, and many solutions such as Pixie and Kindling use this technology for observability and performance monitoring purposes. According to the 2024 State of eBPF report, "Many of the US hyperscalers Meta, Google, Netflix use eBPF in production. Every Android phone uses eBPF to monitor traffic. Every single packet that goes in and out of a Meta datacenter is touched by eBPF". Moreover, "eBPF catalyzes next-generation cloud native workloads because it expands a platform's capabilities, increases performance, and reduces complexity" [1]. This highlights eBPF's indispensable role in modern infrastructure, especially within cloud environments. Gartner further predicts that by 2025 more than 95% of new digital workloads will run on cloud-native platforms [2]. This and the widespread adoption of eBPF-driven solutions such as Cilium for Kubernetes networking and others projects reinforce the convergence of eBPF and cloud-native technologies, a trend that is poised to continue for the foreseeable future, thus increasing even further the usage of eBPF in modern day solutions [3] [4].

However, because eBPF programs execute with elevated privileges inside the kernel, they can pose significant security risks: a malicious or compromised BPF program can intercept syscalls, trace kernel events and exfiltrate sensitive data from the machine itself [5] [6] [7]. In early 2024, the kernel community introduced the "BPF Token" [8] a mechanism that allows a privileged process to issue tokens to designated "trusted" unprivileged processes, thereby granting them scoped and partially limited BPF capabilities. This demonstrates a growing recognition in the need for fine-grained BPF permission management in the kernel community.

To address this challenge, we first survey and evaluate existing kernel-level mitigation strategies, including Linux capabilities, seccomp filters, and Linux Security Module (LSM) based restrictions. We highlight their limitations in terms of granularity and enforceability. Building on these insights, we propose BPF-to-BPF Armor (BBArmor), an in-kernel security solution that enables dynamic policy enforcement using eBPF itself, thus without permanently modifying the kernel itself. Our approach integrates with a user-friendly dashboard, allowing administrators to define and manage eBPF execution policies in real time, ensuring both flexibility and security. In summary, the key contribution of the paper are:

- BBArmor: a proactive, syscall-boundary mechanism that blocks dangerous BPF programs from entering the kernel while preserving legitimate BPF functionality and requiring no kernel source modifications.
- A fine-grained, lightweight enforcement model: namespace-scoped, per-program type, helper, attach-point policies with live updates.

II. Background & Threat Model

eBPF enables rapid deployment of programs into a sandboxed, in-kernel context. It allows developers to write "probes" program that attaches to predefined kernel hooks, including tracepoints, kprobes, LSM hooks, etc. to observe or modify system-wide events. Although the BPF verifier enforces memory safety and termination guarantees, the default Linux security model imposes no restriction on the visibility or scope of an installed BPF program once it is loaded; it only limits who may load such programs, typically requiring sudo or

978-3-903176-75-1

the CAP_BPF/CAP_SYS_ADMIN capability, not where in the system they operate [5].

Cloud infrastructure administrators routinely deploy BPF frameworks for networking, performance monitoring, and security enforcement in Kubernetes clusters. However, this widespread reliance on third-party tooling also provides fertile ground for attackers to conceal malicious BPF code within otherwise legitimate software. In a supply-chain compromise, an adversary might inject a trojaned BPF loader into a seemingly benign application distributed through standard CI/CD pipelines so that when administrators update or redeploy their monitoring agents, the backdoored code automatically installs unauthorized eBPF programs. Similarly, vulnerabilities in third-party monitoring plugins can be exploited to escalate privileges, allowing an attacker to load custom BPF bytecode that silently intercepts traffic, probes syscall activity, or exfiltrates secrets without raising alarms [9] [10].

More importantly, these threats do not only concern the system administrators but also tenants operating in semi-trusted environments, where containers may run with elevated privileges for observability or debugging purposes. In such cases, a compromised container, even when granted permissions for legitimate reasons, can install instrumentation that silently bypasses isolation boundaries and poses a serious risks to both co-located workloads and the host system itself.

Because these malicious modifications are packaged alongside expected features, they can blend in with normal operations making detection extremely difficult and may remain unnoticed until demo builds, or worse, stable releases are affected, as seen in the XZ Utils [11] and SolarWinds [12] incidents. Within the proposed threat model, malicious BPF programs can be especially insidious: their behavior may closely resemble that of legitimate observability or diagnostics tools. This resemblance allows them to blend into expected system activity, making them difficult to distinguish and complicating both detection and response efforts.

III. Problem Analysis & goals

In light of the security challenges posed by unrestricted BPF instrumentation (see section II), we identify the following requirements for an effective mitigation mechanism:

- **Transparency and Enforcement:** The solution must operate transparently to application developers, automatically intercepting and validating BPF programs without requiring any modifications to user-level code, yet remain mandatory for all BPF loads.
- **Portability and Compatibility:** It should be hardware-agnostic and compatible with a wide range of Linux kernel versions, avoiding reliance on specialized architectures or recent kernel features unavailable in many deployments.
- **Robustness and Security:** The enforcement layer itself must be non-bypassable, introduce minimal additional attack surface, and resist both direct attacks and indirect attempts at subversion.
- **Selective Blocking:** It must distinguish between benign and potentially malicious BPF programs, blocking only those deemed dangerous while permitting legitimate and safe ones to proceed unimpeded.

Our target deployment is a multi-tenant semi-trusted cloud environment, in which numerous high-level applications continuously execute. As with hardware caches or network QoS mechanisms, our enforcement layer should be invisible to users: they should neither need to be aware of its existence nor make any changes to their tools or software to benefit from its protection. By avoiding extensive kernel modifications and instead leveraging existing native Linux infrastructure, we reduce maintenance burden and facilitate broader adoption across diverse distributions and kernel versions.

Achieving these goals requires designing an in-kernel enforcement mechanism capable of analyzing BPF load requests and system calls in real time. We must ensure that such a mechanism integrates seamlessly with the pre-existing BPF environment without disrupting it or compromise the overall system stability or performance.

IV. Limitations of Existing Kernel-Level Defenses

Below we analyze existing kernel mechanisms and state-of-the-art solutions to identify features useful for our task. While these approaches provide confinement, auditing, or provenance, they lack the flexibility and adaptability required here, motivating a novel solution.

A. Capabilities analysis

Following the principle of minimum privilege [13], we initially might consider restricting BPF operations by assigning to a process only the minimal necessary capabilities. For example the CAP_PERFMON governs the usage of performance monitoring tools based on the perf_event_open syscall and it is mostly associated with kprobe and fentry BPF programs, which can be exploited to inspect or interfere with the kernel [14].

Table I shows some BPF programs alongside the capabilities required as well as an alternative minimal set if possible (Programs 1-2-4-5 taken from libbpf-bootstrap [1], program 2 taken from xdp-tutorial[2]). We would expect to always find CAP_BPF in the minimal set, but instead CAP_SYS_ADMIN occupies that space, this is due to the historical retro-compatibility of the Linux kernel as before Linux 5.8 both CAP_BPF and CAP_PERFMON were part of CAP_SYS_ADMIN and later separated. During the verification process for BPF and PERFMON privileges, the kernel performs the check shown in listing 1 where, if CAP_BPF, or CAP_PERFMON, are not possessed by a process, CAP_SYS_ADMIN can be used instead [14]. Consequently, it is impossible to enforce a strictly minimal capability model without modifying kernel behavior.

[1]https://github.com/libbpf/libbpf-bootstrap/tree/master/examples/c
[2]https://github.com/xdp-project/xdp-tutoria

TABLE I
BPF PROGRAM CAPABILITY REQUIREMENTS

ID	BPF Program Type	Minimal Capability Set	Alternative Capability Set
1	KPROBE	CAP_SYS_ADMIN	–
2	KPROBE (uprobe)	CAP_SYS_ADMIN	–
3	XDP	CAP_SYS_ADMIN	CAP_BPF, CAP_PERFMON, CAP_NET_ADMIN
4	TRACING	CAP_SYS_ADMIN	CAP_BPF, CAP_PERFMON
5	SOCKET_FILTER	CAP_NET_RAW, CAP_SYS_ADMIN	CAP_NET_RAW, CAP_BPF

```
static inline bool perfmon_capable(void)
{
        return capable(CAP_PERFMON) ||
            capable(CAP_SYS_ADMIN);
}

static inline bool bpf_capable(void)
{
        return capable(CAP_BPF) ||
            capable(CAP_SYS_ADMIN);
}
```

Listing 1. Kernel capabilities check for `CAP_BPF` and `CAP_PERFMON`[3]

B. System Call Filtering

An alternative to a capability based approach is to intercept attempts to load BPF programs at the `bpf()` system call boundary. The `bpf()` syscall accepts three arguments:

- `cmd`: an integer from the `enum bpf_cmd` indicating the requested operation (e.g. `BPF_PROG_LOAD`, `BPF_MAP_CREATE`, etc.).
- `attr`: a pointer to a `union bpf_attr` containing the parameters for the specific `cmd` (e.g. `prog_type`, `insns`, `insn_cnt` for `BPF_PROG_LOAD`).
- `size`: the size in bytes of the `attr` structure.

By examining `attr.prog_type` and `attr.insns`, we could implement a decision logic that blocks certain program types and certain operations. Seccomp filters, while lightweight and straightforward to deploy, suffer from several fundamental limitations. First, they operate solely at the syscall boundary and cannot perform deep inspection of syscall arguments or metadata, which makes it impossible to distinguish benign from malicious uses of the same syscall, they also lack important information derived from the execution context such as namespace of origin. Second, seccomp policies are inherently static: changes require unloading and reloading the entire filter set, often necessitating application restarts or container reprovisioning [15]. A more powerful—yet still lightweight—alternative is to leverage Linux Security Modules. Since Linux 5.7, the kernel exposes LSM hooks on the `bpf()` syscall [16], enabling security modules to inspect syscall arguments and make per-operation decisions. An LSM-based solution could perform the following steps:

1) Extract `attr.prog_type`, `attr.insns`, and other metadata (e.g., PID, namespace, cgroup).
2) Compare these attributes against a policy database or a dynamically maintained whitelist.

[3]Code taken from: https://elixir.bootlin.com/linux/v6.11/source/include/linux/capability.h\#L200

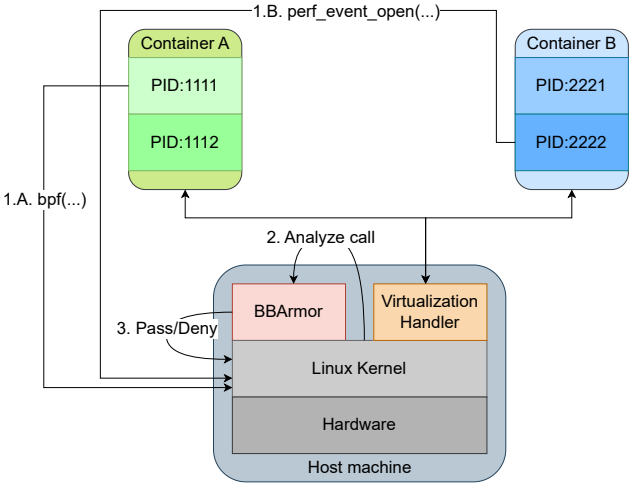

Fig. 1. High level architecture of BPF to BPF Armor

3) Permit, deny, or audit the load request according to the matching policy.

Because LSMs can also hook into other kernel subsystems such as file operations, network accesses, cgroup joins, etc. This approach can be used to create a unified policy enforcement mechanism for BPF and its usage. However, the current lack of full LSM stacking support means that integrating a custom BPF-filtering module alongside established frameworks like SELinux, AppArmor, or Landlock, often requires patching or reconfiguring the kernel build [17] [18]. Moreover, writing and maintaining policies in a domain-specific language can introduce its own operational overhead.

In summary, while seccomp's syscall interception lacks the granularity and contextual awareness needed for precise BPF filtering, a pure LSM-based solution provides the necessary depth at the cost of increased integration complexity and policy management burden. In the next section, we explore a hybrid design, BBArmor, that combines minimal in-kernel policy hooks with user-space policy evaluation to achieve both expressiveness and ease of deployment.

V. BPF TO BPF ARMOR

To enforce fine-grained BPF load policies without extensive kernel patches, we can leverage the existing LSM hooks already presented in §IV-B by implementing a small BPF program bound to the `lsm/bpf` and other critical hooks. Figure 1 illustrates the overall, yet simplistic, structure of

BPF to BPF Armor (BBArmor). Every time the `bpf()` or other BPF related syscalls are invoked, BBArmor examine it by applying container-aware whitelist or blacklist rules, inspect programs attributes and search for specific blacklisted bpf_helper calls. Given the target deployment illustrated in section III we associate each namespace with its own enforcement ruleset, defined through a JSON file.Whenever a new ruleset is added, BBArmor populates BPF maps with these rules, using a composite key of `<namespace_id, rule>`. When BBArmor intercept a call, to prevent intercepting its own calls recursively, the process skips any call originating from itself, then it retrieves the context, infers the key and queries the maps to determine whenever to allow or deny the operation. To achieve this, BBArmor attaches to the `lsm/bpf` hook to extract contextual information during BPF program loading, including:

- `BPF_PROG_TYPE`: The type of the BPF program being loaded.
- Attach Type: Indicates the target type (e.g., cgroup, tracepoint, network interface).
- ELF Section Name: Provides high-level semantic information about the probe target (e.g., `kprobe/*`).
- `attach_type`: Useful for retrieving metadata such as the network interface index the program attaches to.

Unfortunately, this setup does not allow filtering at the granularity of specific attach points for some hook types. For example, while intercepting `bpf()` enables BBArmor to allow or deny the loading of entire program categories such as `KPROBE` or `TRACEPOINT`, it cannot differentiate between individual probe targets, such as `kprobe/do_unlinkat` versus `kprobe/do_execve`. This limitation arises because this specif information is not always passed inside the `bpf_attr` field of the syscall.

To address the limitation described above, BBArmor also attaches to the `lsm/perf_event_open` hook. This allows it to intercept all `perf_event` data passed via the `struct perf_event_attr`, which is responsible for configuring performance events, including those used by BPF programs.

```
struct perf_event_attr {
    __u64 config;
    /* other fields */
    union {
        __u64 kprobe_func;
        __u64 config1;
        /* ... */
    };
    union {
        __u64 kprobe_addr;
        __u64 probe_offset;
        /* ... */
    };
};
```

Listing 2. Partial `perf_event_attr` structure

Listing 2 highlights selected members of the `perf_event_attr` structure that are useful for fine-grained enforcement. Specifically, the `kprobe_func` field contains the name of the kernel function to which a `KPROBE` or `KSYSCALL` program is attaching. Additionally, `probe_offset` stores the byte offset within the target function, indicating precisely where the probe is being inserted. For `TRACEPOINT` programs, the `config` field encodes the tracepoint ID, enabling similar filtering. This extended context allows BBArmor to implement more precise policy checks, including filtering based on the exact attach point of a BPF program, which is not possible via the `lsm/bpf` hook alone.

Since BBArmor heavily relies on BPF maps for its dynamic rule sets, we must prevent untrusted process from tampering with them. Although BPF map identifiers are normally process-private, tools like `bpftool`[4] can enumerate and access maps globally. We can, therefore, intercept map and program FD retrieval calls (cmd `BPF_MAP_GET_FD_BY_ID` and `BPF_PROG_GET_FD_BY_ID`) and return `-ENOENT` for any IDs corresponding to BBArmor's internal maps or programs, as shown in Listing 3.

Rulesets can be generated via sandbox-driven introspection: BPF programs and tooling are executed in a secure, isolated container or VM with extensive audit logging, record the program types, helpers, tracepoints, kprobes, attach points, etc. actually used, and, once the execution trace stabilizes, extract a minimal, namespace-scoped ruleset containing only the capabilities the program demonstrably requires. This profiling-based workflow is analogous to AppArmor's log-driven profile generation and helps enforce least privilege, reduce configuration effort, and shrink the attack surface; future work could fully automate the profiling-policy pipeline.

```
if(cmd == BPF_MAP_GET_FD_BY_ID &&
        check_map_id(attr)){
    return -ENOENT;
}
if(cmd==BPF_PROG_GET_FD_BY_ID &&
        check_program_id(attr)){
    return -ENOENT;
}
```

Listing 3. BBArmor self map protection

A. Additional framework

To streamline BBArmor policy management, we refactored the user-space loader into a lightweight shared library exposing lifecycle and rule-management functions such as initialization, cleanup, and map updates. This library is linked into a Python application via the `ctypes`foreign-function interface. On startup, the dashboard invokes the library's initialization routine to load and attach the BPF bytecode and prepare its internal maps. Subsequent RESTful API calls, served by a Flask application, translate user actions, such as adding or removing a ruleset thus updating permitted program types, PID, kprobe/fentry attach point or modifying helper-blacklists, into direct map operations. Each API handler marshals the appropriate parameters, calls the corresponding C function through `ctypes`, and returns status information to the user. Finally, a cleanup endpoint triggers the library's teardown function, detaching the BPF programs and freeing associated resources

[4]https://bpftool.dev/

in a graceful way. Furthermore, BBArmor includes logic to translate human-readable rulesets into kernel-compatible representations. For instance, when loading a `TRACING` program, the kernel requires the numeric `attach_btf_id` to identify the function being traced. Since expecting users to supply raw BTF IDs is impractical, BBArmor resolves symbolic names, like `do_execve`, to their corresponding BTF identifiers automatically. This abstraction enables more intuitive policy definitions without compromising enforcement precision and it is similarly reproduced for other rules. This architecture ensures that administrators can manage BBArmor policies dynamically and intuitively, without directly interacting with kernel interfaces or modifying container workloads.

VI. Performance Evaluation

To evaluate the performance overhead introduced by BBArmor, we conducted two primary tests; all tests were performed on Ubuntu 24.04 with Linux kernel version 6.11. The first test measured the delay incurred during the loading of BPF programs, while the second focused on the latency introduced in BPF map operations—specifically, update and delete operations.

The loading of BPF programs is the phase where BBArmor performs most of its policy enforcement checks. However, since the `BPF_PROG_LOAD` operation typically occurs infrequently, only once per program instance, and, considering that our results show the added latency being approximately 1 ms, we deem this impact negligible in practice.

The second test targeted the runtime overhead caused by BBArmor's interception of all `bpf()` syscalls, particularly map operations, which are frequently invoked in BPF applications. Although, each such syscall is routed through BBArmor's logic, optimization measures were introduced to bypass redundant checks for known-safe programs—those already verified during their load phase. This ensures that BBArmor does not re-evaluate security properties unnecessarily, thus preserving efficiency. As shown in Figure 2, a test sequence of 100 consecutive map update–delete operations incurs an average overhead of less than 0.5 ms. Similar tests were perfomed with the `BPF_MAP_[...]_BATCH` cmd family, but did not yield significantly different results. This result confirms that BBArmor maintains low latency even under high-frequency usage, demonstrating its suitability for deployment in performance-sensitive environments.

A. Evaluation: Blocking Malicious eBPF Examples

To validate BBArmor's effectiveness, we tested it against known malicious eBPF programs from the "bad-bpf" repository [19] a "de facto" standard for testing purposes. In each case, BBArmor prevented the malicious behavior by rejecting unsafe BPF loads early by simply forbidding dangerous bpf helpers such as `bpf_probe_write_user` or otherwise blocking specific attachment point. Advanced malware, such as BPFDoor, installs malicious BPF filters on sockets to trap or hide attacker traffic [20]. By default, such programs require `SOCKET_FILTER` program type and BBArmor policy can be used to restrict `SOCKET_FILTER` program types to forbid the load of unexpected port filters. Thus denying any attempt to attach a socket filter at load time.

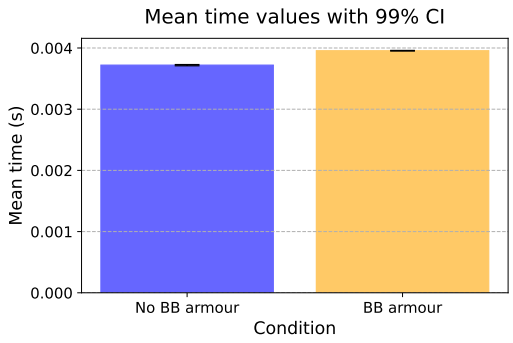

Fig. 2. Performance degradation introduced by BBArmor

B. Comparison with AppArmor, Reactive eBPF Frameworks and eBPF Token

AppArmor is a mature LSM that confines programs with per-executable profiles written in a domain-specific language and enforced at boot or on demand [21]. By contrast, BBArmor enforces BPF-specific policies via BPF/LSM hooks without kernel patches or a new profile language; Table II summarizes these differences and highlights BBArmor's fit for multi-tenant, cloud-native deployments. The usual AppArmor+seccomp combination is insufficient for our use case: as noted in §IV-B seccomp can only allow or deny whole syscalls and cannot filter on the arguments or metadata needed for fine-grained BPF control.

Reactive eBPF tools such as Falco and Tetragon monitor syscalls, kernel events, or BPF activity and raise alerts or take action after anomalies are observed. BBArmor is proactive: it intercepts and vets `bpf()` and related syscalls at load time, applying namespace-aware allow/deny rules so unauthorized BPF programs are prevented from entering the kernel, eliminating the window reactive systems must bridge

The recent "BPF Token" mechanism enables an unprivileged process to obtain a token from a privileged agent for performing BPF operations, offering provenance and centralized revocation [8]. However, the mechanism is still evolving and, once a token is issued, cannot selectively filter individual BPF actions from the same token holder. As currently specified, BPF Tokens do not satisfy the dynamic filtering requirements of our threat model.

VII. Related Works

Several recent efforts explore enhancing container security and restricting eBPF. In 2019 L. Deri *et al.* [22] showed how BPF can be used for container security. J. Jia *et al.* [23] advocate for a third seccomp mode powered by eBPF, enabling deep inspection of syscall arguments and fine-grained filtering beyond what legacy seccomp allows. More recently, in 2024, S. Yang, B.*et al.* propose Optimus to derive per-container

TABLE II
COMPARISON OF BBARMOR AND APPARMOR

Feature	AppArmor	BBArmor
Policy Domain	File system, network, capabilities, IPC	BPF syscalls, program types, helpers, tracepoints, etc.
Enforcement Hook	LSM hooks (exec, file, network)	LSM hooks (Syscall entry points)
Profile Language	Custom text profiles (`.conf`)	JSON-defined rulesets loaded via BPF maps
Granularity	Per-executable / per-path	Per-namespace, per-PID, per-program-type, etc.
Dynamic Updates	Reload entire profile or module	Update individual BPF maps at runtime without reload
Kernel Impact	Requires AppArmor module; boot-time configuration	No additional module; relies on existing support [5]
Container Awareness	Limited (profiles bound to binaries)	Native: rules keyed by namespace ID
Performance Overhead	Generally low, but broad syscall interception may incur occasional cost	Minimal: only intercepts BPF-related syscalls

syscall allowlists then apply seccomp denies [24], in 2025, H. Lu *et al.* describe a pod-side BPF "watcher" that enforces static syscall policies [25]. While these solutions demonstrate innovative uses of eBPF for confinement, syscall reduction, and distributed enforcement, none offer the proactive, namespace aware blocking of malicious BPF loads that BBArmor provides, but, though their policy engines and integration techniques, offer valuable directions for future extensions.

VIII. CONCLUSION AND FUTURE WORKS

In this paper, we presented BBArmor, a prototype that leverages BPF and LSM to provide fine-grained control over BPF program loading in containerized environments by intercepting and filtering `bpf()` and related syscalls via a BPF-based LSM hook. Administrators can rapidly and dynamically adjust allowlists and deny rules through a dashboard, enabling namespace-level isolation or per-process filtering without intrusive kernel changes or application rewrites.

Future work could focus on deeper helper-usage inspection, field-level redaction and control-flow integrity checks for BPF bytecode. Tighter orchestration integration with platforms such as Kubernetes and combining BBArmor's proactive blocking with reactive tools such as Falco or Tetragon; finally a lightweight classifier could be develop to trigger automated policy changes. All of these could yield a robust, adaptive, enterprise-grade eBPF security framework.

ACKNOWLEDGMENT

This work was partially supported by the Italian Ministry of Education and Research (MUR) through the ForeLab project (Departments of Excellence) and by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP C59J24000110004 (Project ASPIRE), partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART") and Mission 4, Component 1, CUP I53D23000410006 (PRIN 2022 Project NEWTON)".

REFERENCES

[1] eBPF Foundations, "The state of ebpf 2024," https://ebpf.foundation/report-the-state-of-ebpf/, 2024.
[2] Gartner, "Gartner says cloud will be the centerpiece of new digital experiences," https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences, 2021.
[3] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of ebpf for non-intrusive performance monitoring," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.
[4] G. Fournier, S. Afchain, and S. Baubeau, "Runtime security monitoring with ebpf," 2021.
[5] L. Rice, *Learning eBPF*. O'Reilly Media, Inc., 2023.
[6] B. Sharma and D. Nadig, "ebpf-enhanced complete observability solution for cloud-native microservices," in *ICC 2024 - IEEE International Conference on Communications*, 2024.
[7] L. Song and J. Li, "ebpf: Pioneering kernel programmability and system observability - past, present, and future insights," in *2024 3rd International Conference on Artificial Intelligence and Computer Information Technology (AICIT)*, 2024.
[8] Y. Hayakawa, "ebpf docs - bpf token," https://docs.ebpf.io/linux/concepts/token/, 2024.
[9] S. B. Guillaume Fournier, Sylvain Afchain, "With friends like ebpf, who needs enemies?" 2021.
[10] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li, "Cross container attacks: The bewildered eBPF on clouds," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023.
[11] NIST, "Cve-2024-3094," https://nvd.nist.gov/vuln/detail/cve-2024-3094, 2024.
[12] C. for Cyber Security, "Solarwinds: State-sponsored global software supply chain attack," Centre for Cyber Security, Tech. Rep., 2021.
[13] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, 1975.
[14] *capabilities(7) — Linux manual page*, 2025.
[15] J. Corbet, "ebpf seccomp() filters," https://lwn.net/Articles/857228/, 2021.
[16] D. Reimerink and jetlime, "ebpf docs - program type bpf_prog_type_lsm," https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_LSM/, 2025.
[17] J. Corbet, "Still waiting for stackable security modules," https://lwn.net/Articles/912775/, 2022.
[18] J. Edge, "Lsm stacking and the future," https://lwn.net/Articles/804906/, 2019.
[19] pat_h/to/file, "bad-bpf," https://github.com/pathtofile/bad-bpf, 2021.
[20] gwillgues, "Bpfdoor," https://github.com/gwillgues/BPFDoor, 2022.
[21] AppArmor, "Apparmor wiki," https://gitlab.com/apparmor/apparmor/-/wikis/home, 2025.
[22] L. Deri, S. Sabella, S. Mainardi, P. Degano, and R. Zunino, "Combining system visibility and security using ebpf." in *ITASEC*, vol. 2315, 2019.
[23] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable system call security with ebpf," 2023.
[24] S. Yang, B. B. Kang, and J. Nam, "Optimus: association-based dynamic system call filtering for container attack surface reduction," *J. Cloud Comput.*, vol. 13, no. 1, December 2024.
[25] H. Lu, X. Du, D. Hu, S. Su, and Z. Tian, "Bpfguard: Multi-granularity container runtime mandatory access control," *IEEE Transactions on Cloud Computing*, 2025.