Accepted in IEEE GLOBECOM 2026

# Vulnerability Analysis and Mitigation of eBPF-based Attacks in Containerized 5G Networks

Yash Deshpande and Samaresh Bera
Dept. of CSE, Indian Institute of Technology Jammu, India
Email: 2024pcs0044@iitjammu.ac.in, s.bera.1989@ieee.org

Luca Foschini
Dept. of CSE, University of Bologna, Italy
Email: luca.foschini@unibo.it

***Abstract*—The extended Berkeley Packet Filter (eBPF) is useful for faster packet processing and network monitoring in softwarized deployments. Similarly, softwarized deployments of 5G core network services adopted eBPF to meet the stringent latency and bandwidth requirements of underlying applications. While existing studies have focused on network performance, security concerns regarding eBPF-enabled platforms have been overlooked.**

**In this paper, we study the vulnerability analysis and mitigation for 5G core network deployments that use eBPF for packet processing and traffic monitoring. In particular, we consider the following aspects: a) tracing, b) denial-of-service, c) stealing information, and d) bash injection. We present the detailed attack scenarios with step-by-step implementation of containerized and eBPF-enabled 5G network functions using Open5GS. The experiment results show that the aforementioned vulnerabilities are present in eBPF-enabled 5G deployments and can be exploited by attackers. To mitigate the attacks, we implement and integrate a fine-grained eBPF helper permission model inside the Linux Kernel. The source code, implementation details, and Linux Kernel Patch are made publicly available on GitHub to ensure reproducibility.**

***Index Terms*—5G network security, Vulnerability analysis, eBPF helper functions, Exploitation, Bash injection, Information Theft, Attack mitigation**

## I. Introduction

The service-based architecture of 5G network allows the network operators to place 5G core network functions (NFs) on cloud platforms in a containerized or virtual machine form [1]. Consequently, telecom operators started deploying 5G network functions on public clouds, private clouds, or hybrid clouds. For example, Nokia and Telefónica deployed a 5G Standalone (SA) core on AWS for ultra-low latency services [2]. Similarly, Ericsson and Swisscom ran a 5G core proof-of-concept on AWS to explore hybrid-cloud use cases [3]. These deployments often use orchestration platforms (e.g., Kubernetes) to manage the containers.

The applications supported by 5G and beyond networks are broadly categorized as enhanced mobile broadband (eMBB), ultra-reliable and low-latency communications (uRLLC), and massive machine-type communications (mMTC). The quality-of-service (QoS) requirements of these applications range from high bandwidth to high-reliability and low-latency [4]. To meet these requirements, 5G networks must minimize processing overhead and quickly route user data. Thus, optimizing the data-plane performance of 5G NFs is critical. However, softwarized deployments of the network functions may not be suitable to meet stringent QoS requirements, unlike the hardware-based deployments. Thanks to the extended Berkeley Packet Filter (eBPF) [5] that allows custom bytecode programs to run directly inside the Linux kernel, enabling fast packet processing and in-kernel telemetry. In particular, eBPF programs, attached via the eXpress Data Path (XDP) mechanism, can process packets at the network driver level with minimal overhead. Thus, eBPF-based approaches help network administrators to meet the stringent QoS requirements in softwarized deployments. Recent works have utilized eBPF and XDP to accelerate 5G user-plane functions [6]–[8]. For example, Zhou *et al.* [6] showed that incorporating eBPF/XDP into a 5G UPF can reduce per-packet processing time by over 30%. On the other hand, Scheich *et al.* [7] proposed XDP extensions for high-capacity 5G UPFs. Xuan *et al.* [8] presented an eBPF/XDP-enhanced 5G Service-Based Architecture (SBA) platform and reported significant gains in throughput and latency reduction. These works illustrate that container-based 5G deployments can integrate eBPF/XDP for both performance and observability.

eBPF is also widely used for security and monitoring in cloud-native environments. For example, security tools like Falco [9] and network tools like Cilium [10] leverage eBPF tracing and maps to inspect network flows and system calls. eBPF can attach to kernel tracepoints, kprobes, or network interfaces to collect event logs without modifying the application code. Yang *et al.* [11] described an in-kernel tracing system (ZeroTracer) that uses eBPF to track HTTP requests across microservices with high accuracy, demonstrating eBPF's usefulness for observability. Similarly, Soldani *et al.* [12] demonstrated the feasibility of using eBPF for high performance monitoring and security in 5G and future 6G networks. They proposed a platform to unify observability, energy estimation, and policy enforcement in the cloud. Thus, eBPF's programmability and efficiency have led to its broad adoption for monitoring, load balancing, and security policies in containerized clusters.

The very features that make eBPF powerful can introduce new vulnerabilities. In a cloud environment where multiple containers share the host kernel, eBPF programs loaded by one container can potentially interact with or affect other containers. Recent studies have revealed that containerized

This work has been partially supported by the ISEA Phase-III Project (F. No. L-14017/1/2022-HRD) and Indian Institute of Technology Jammu, India.

eBPF programs can break out of their namespace isolation and bypass kernel protections [13], [14]. For instance, attackers can exploit specific helper functions to inspect or modify host processes, regardless of container boundaries. These capabilities enable cross-container attacks, ranging from data theft to denial-of-service, highlighting a gap in current security models where default permissions grant eBPF programs access to powerful helpers without adequate isolation.

Given these concerns, it is crucial to analyze the security of eBPF-enabled 5G deployments. There is a lack of prior studies examining how eBPF features might be abused in a 5G containerized environment. Our research objective is to identify vulnerabilities in such deployments, evaluate exploitability, and propose defenses. Specifically, we pose the following questions:

- *What vulnerabilities exist in containerized 5G networks that use eBPF/XDP?*
- *Can attackers exploit eBPF helpers or programs to attack 5G components?*
- *What measures can mitigate these threats?*

In this paper, we deploy a fully functional eBPF-enabled 5G core network, and then combine insights from existing works and experiments to address these questions. We draw on known eBPF attack primitives and adapt them to a 5G setting. The contributions of this work include a detailed threat analysis of eBPF in a containerized 5G system and a demonstration of attack procedures. We also propose mitigation technique against the vulnerabilities and exploits. The development and source code are made publicly available on GitHub to ensure reproducibility.

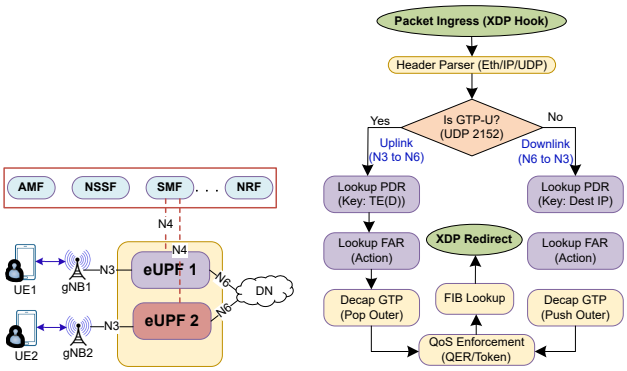
The remainder of this paper is organized as follows: Section II details the system architecture and the role of eBPF in 5G. Section III defines the threat model and experimental setup. Section IV presents the vulnerability analysis and attack algorithms. Section V discusses mitigations, and Section VI concludes the paper with future research directions.

## II. System Architecture

We consider a 5G Standalone (SA) network architecture as defined by 3GPP, consisting of virtualized network functions (NFs), as shown in Figure 1. The UPF handles all user plane data forwarding between the Radio Access Network (gNB) and external Data Networks. The 5G core separates the control plane from the data plane, allowing the UPF to be scaled and placed independently.

To simulate a multi-tenant 5G environment, our testbed deploys two distinct User Plane Function (UPF) instances. This dual-UPF configuration is intended to support network slicing, which allows network operators to create multiple virtual networks on a shared physical infrastructure. Our setup defines two distinct Network Slice Selection Assistance Information (S-NSSAI) values, with each UPF dedicated to a specific slice. This separation ensures that traffic from the User Equipment (UE) attached to different slices is routed through isolated data paths, catering to distinct use cases.

In our deployment, each 5G NF runs as a containerized network function managed via Docker Compose. One set of containers hosts the control plane (AMF, SMF, NRF, etc.), while separate containers host the two UPF instances. The UPFs in our system are based on EdgecomLLC's open-source eUPF implementation, which utilizes eBPF/XDP for fast packet processing [15]. The eUPF interacts with the 5G core via standard 3GPP interfaces: it exposes the N3 interface to receive GTP-U encapsulated packets from the Radio Access Network gNB, the N4 interface to receive Packet Forwarding Control Protocol (PFCP) rules from the SMF, and the N6 interface to route decapsulated user traffic to external Data Networks (DN), as depicted in Figure 1.




(a) Architecture (b) Packet processing inside eUPF

Fig. 1. 5G service based architecture with eUPFs and XDP-based packet processing

Internally, the eUPF attaches an XDP program to the host's network interface to handle these flows. Incoming N3 packets are processed by this XDP program in the kernel before reaching the main userspace logic. This allows critical packet handling—such as GTP header parsing, TEID lookup, and forwarding decisions—to be executed at a faster speed. Similarly, for outgoing traffic on the N6 interface, XDP can quickly redirect packets, bypassing the substantial overhead of the Linux kernel's standard network stack. eUPF container requires `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` privileges [5] to load and attach the necessary eBPF/XDP programs.

## III. Threat Model and Experimental Setup

5G networks are supposed to be cross-vendor compatible; operators often deploy components from diverse sources [1]. This heterogeneity introduces the risk that maliciousness may originate from any part of the supply chain. In our setup, we assume that one of the eUPF containers is malicious and serves as the launchpad for exploits. This compromise can manifest in two primary ways: the container may be pre-packaged with malware (a supply chain attack), or a legitimate container may be compromised and hijacked by an attacker post-deployment. Regardless of the entry point, our analysis focuses exclusively on the host kernel's eBPF functionality as the attack vector. All NFs are deployed on the same host. Security settings or kernel

configurations that are available by default are not modified in any system or container.

Our testbed consists of the following software specifications:

- **Host OS:** Ubuntu 24.04 LTS.
- **Kernel:** Linux 6.8 generic.
- **5G Software:** Open5GS [16] with UERANSIM [17].
- **User Plane:** EdgecomLLC eUPF (v0.6.4), an open-source high-performance UPF implementation using eBPF/XDP [15].
- **Container Runtime:** Docker with `docker-compose` orchestration.

The `docker-compose` configuration grants the eUPF container specific privileges required for its operation:

```
cap_add:
  - NET_ADMIN
  - SYS_ADMIN
volumes:
  - /sys/fs/bpf:/sys/fs/bpf
```

The complete source code for the experimental setup, attack implementations, and scripts is available at https://github.com/chimms1/5G-eBPF-exploits.

## IV. Vulnerability Analysis

We now analyze specific attacks enabled by eBPF in our 5G setup. Each attack exploits certain eBPF helper functions that, while useful for legitimate tasks, can be repurposed maliciously. We divide the analysis into four subsections—Tracing, Denial of Service, Information Theft, and Bash Injection—and discuss the implementation and working of each attack scenario in the subsequent sections. We note that all the attack scenarios are deployed and tested for user-plane traffic inside eBPF-enabled UPFs. However, it can be extended for 5G control-plane functionalities.

### A. Attack Scenario 1: Tracing

eBPF tracing enables low-overhead, programmable observability of kernel and user-space events. Few helpers, such as `bpf_get_current_comm()` and `bpf_get_current_uid_gid()` are essential for observability tools to map network traffic or `syscalls` to specific process names for debugging.

```
SEC("raw_tracepoint/sys_enter")
int trace_processes(struct bpf_raw_tracepoint_args
    *ctx)
{
    pid = bpf_get_current_pid_tgid() >> 32;
    unsigned long long uid_gid =
    bpf_get_current_uid_gid();
    bpf_get_current_comm(&comm, sizeof(comm));
    /* Print or filter process info */
}
```

Listing 1. Pseudocode for tracing attack

An attacker uses these to perform reconnaissance. As shown in Listing 1, by hooking into `sys_enter`, the attacker can enumerate all processes running on the host, identifying the PIDs of critical 5G NFs (e.g., `open5gs-amfd`, `open5gs-smfd`) or any other services that reside in separate containers. Figure 2 illustrates a segment of the output from the tracing program accessible to the attacker, displaying processes that are running on different containers. This breaks the process isolation guarantee of containerization.

```
open5gs-nrfd-134177 [004] ...2. 4723.101308:
     bpf_trace_printk: Matched! PID: 134176,
          COMM: open5gs-nrfd UID:999
open5gs-smfd-134656 [004] ...2. 4723.101321:
     bpf_trace_printk: Matched! PID: 134616,
          COMM: open5gs-smfd UID:999
falco-204133 [003] ...2. 4723.101338:
     bpf_trace_printk: Matched! PID: 204133,
          COMM: falco UID:0
```

Fig. 2. Output of the tracing program

### B. Attack Scenario 2: Denial of Service (DoS)

In general, `bpf_send_signal()` can be used by an eBPF program to send a signal to the current task. The attacker modifies the tracing program to send a `SIGKILL` (signal 9) to the identified security services or 5G processes. A sample attack using `bpf_send_signal()` is presented in Listing 2.

```
SEC("kprobe/__x64_sys_read")
int kill_target_process(struct pt_regs *ctx)
{
    bpf_get_current_comm(&comm, sizeof(comm));
    if (comm == TARGET_PROCESS) {
        bpf_send_signal(9); // SIGKILL
    }
    return 0;
}
```

Listing 2. Pseudocode for DoS attack

Since the eBPF program runs in the kernel, it bypasses container namespaces. Once the AMF or SMF process is detected, it is immediately killed. In a Kubernetes environment, the orchestrator would attempt to restart the pod, but the persistent eBPF hook would kill it again immediately, causing a *CrashLoopBackOff* and rendering the 5G core offline. This is a highly effective DoS attack.

### C. Attack Scenario 3: Information Theft

To intercept sensitive data, the attacker must bridge the gap between kernel space execution and user space memory. This is achieved using specific eBPF helper functions: a) `bpf_probe_read_user_str()` to capture the filename arguments from the `openat` syscall, allowing the program to identify when specific sensitive files are accessed; and b) `bpf_probe_read_user()` during the `read` syscall exit to extract the actual file content from the application's memory buffer. These helpers allow the eBPF program to bypass memory isolation boundaries and exfiltrate data being used by legitimate applications. The detailed implementation

of the exploit can be found in the aforementioned GitHub repository.

Attacker can monitor `openat` and `read` syscalls to intercept sensitive files opened by other containers or the host. For example, stealing SSH keys or 5G encryption keys (like the `K` and `OPc` stored in UDM configurations). The attack proceeds in two phases:

- **Open Detection:** When a target process (e.g., SSH or a config reader) calls `openat`, the program checks if the filename matches a sensitive target (e.g., `id_rsa`). If so, the PID is stored in a BPF map.
- **Read Interception:** When the saved PID exits a `read` syscall, the eBPF program reads the content of the user-space buffer populated by the kernel and exfiltrates it.

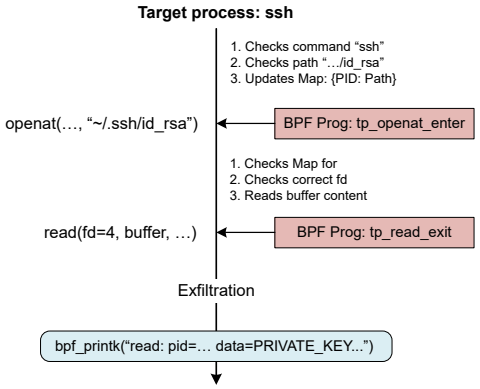



Fig. 3. Working principles of information theft

### D. Attack Scenario 4: Bash Injection

This section details a "Bash Injection" attack where a benign script execution is intercepted and replaced with malicious commands at runtime. While the previous attacks focused on passive monitoring, this attack causes active interference. `bpf_probe_write_user()` helper allows a privileged eBPF program to overwrite data in the user-space memory of a target process. Combined with `bpf_override_return()`, which allows the modification of a system call's return value, an attacker can silently alter the execution flow of admin scripts. Similar to information theft, detailed implementation of the exploit can be found in the aforementioned GitHub repository.

An attacker waits for a privileged process (like a root cron job or admin bash session) to execute a benign script. The eBPF program intercepts the `read` syscall where the script content is loaded and overwrites it with malicious commands. The attack monitors `execve` to identify when a target script is launched. When the interpreter reads the script file, the eBPF hook overwrites the buffer with a malicious payload and adjusts the return value to match the new payload length, ensuring the interpreter executes the attacker's code instead of the original script. This grants the attacker root-level execution on the host.

The mechanics of this injection rely on the sequence of events in the system call entry and exit, as illustrated in Figure 4. A standard `read(fd, buf, count)` syscall passes through two of these eBPF hook points:

- **Entry Hook (kprobe/sys_enter):** As shown in the timeline (refer Figure 5), when the kernel receives the syscall, the arguments (file descriptor and buffer address) are visible. However, the *User Buffer State* is currently **EMPTY** or stale because the kernel has not yet fetched data from the disk. The attack utilizes this stage solely to capture the target process ID and save the address of the user-space buffer (`buf`) into a BPF map.
- **Kernel Execution:** The kernel performs the internal work (e.g., reading the storage device) and populates the buffer with the legitimate script content.
- **Exit Hook (kretprobe/sys_exit):** This is the vulnerability window. As indicated in Figure 4, the *User Buffer State* is now **FILLED WITH DATA**, but control has not yet returned to the User space process (Bash).

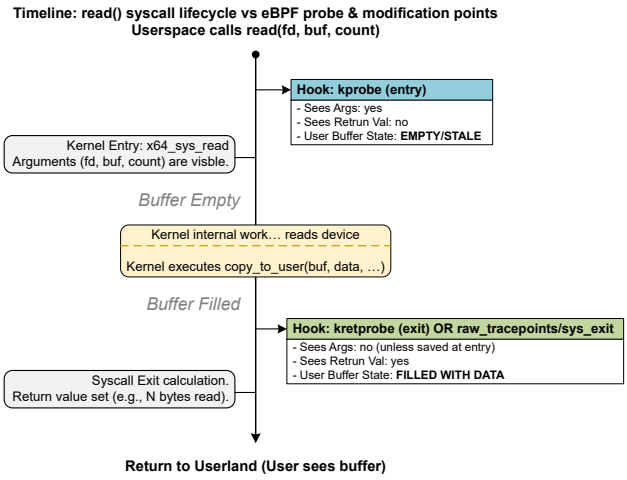



Fig. 4. Lifecycle of the `read()` syscall vs. eBPF modification points

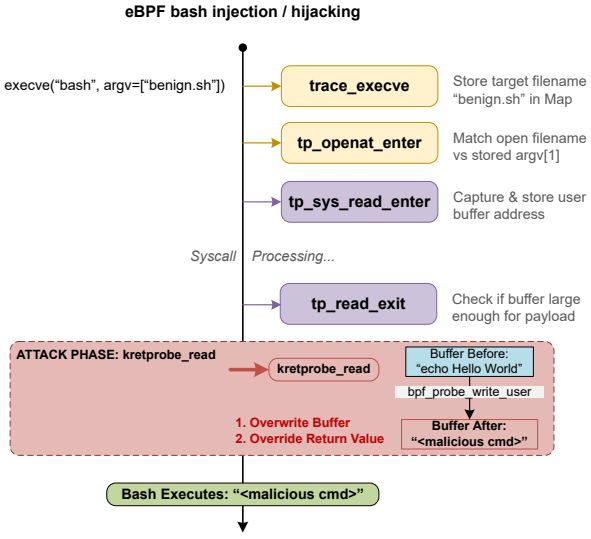



Fig. 5. Steps for bash injection

The exploit triggers at this specific Exit point. The eBPF program retrieves the saved buffer address from the map and invokes `bpf_probe_write_user`. Since the legitimate file content has already been written by the kernel, the malicious overwrite supersedes it. Finally, `bpf_override_return` is used to adjust the return value (byte count) to match the length of the injected malicious command, ensuring the Bash interpreter executes the payload cleanly without syntax errors.

## V. Mitigation Approaches

As demonstrated in our analysis, the shared kernel architecture of containerized deployments inherently exposes 5G Network Functions (NFs) to cross-container attacks via eBPF. The eBPF-based vulnerabilities represent a systemic threat rather than isolated implementation bugs. In this section, we first look into the existing defense models and their shortcomings. We then propose a new mitigation strategy.

### A. Existing Approaches

*1) Capability-based Control (CapBits):* Capbits model as proposed by He *et al.* [13] extends the Linux kernel's `task_struct` to include two bitmap fields: `cap_bits` and `allow_bits`. The `cap_bits` field specifies the permissible eBPF features for a process, including program types and helper functions, while the `allow_bits` field defines which external eBPF programs can interact with the process. At runtime, each helper invocation is checked against the capability bitmap, enabling fine-grained enforcement of permissions. Additionally, protection mechanisms that are incorporated in this solution work independent of the attacker's privilege level.

**Limitations Identified:** CapBits also presents several limitations. The enforcement occurs at runtime, introducing additional overhead (reported up to 5–30%) [13]. Furthermore, policy management becomes complex as it involves maintaining capability rules for both eBPF and non-eBPF processes. The model also does not fully address privilege separation for eBPF maps and may introduce inconsistencies due to misuse or incorrect configuration of bitfields.

*2) Linux Security Modules (LSM):* Linux Security Modules (LSM) provide a framework for enforcing security policies through kernel hooks. In the context of eBPF, LSM hooks such as `security_bpf()` and `security_bpf_prog_load()` can be used to control which users or processes are permitted to load eBPF programs. Security policies can be implemented without modifying the kernel, making it suitable for deployment in production systems. LSM can act as a perimeter defense mechanism by restricting access to eBPF functionality based on user identity, program type (e.g., XDP, tracing), or system-level policies.

**Limitations Identified:** LSM operates at a coarse granularity. It is primarily intended to control *who* can load eBPF programs rather than *what* those programs are allowed to do. Parsing and analyzing eBPF instructions within LSM hooks is non-trivial and not aligned with its design philosophy. Therefore, LSM is not well-suited for enforcing fine-grained helper-level permissions or preventing misuse of allowed programs.

### B. Proposed: Load-time Enforcement using eBPF Verifier

Given the limitations of existing approaches, we explore a load-time mitigation strategy by extending the functionality of the eBPF verifier. The verifier is a core component of the eBPF subsystem that analyzes programs before they are loaded into the kernel. Its primary purpose is to ensure safety properties such as bounded execution, memory safety, and correct access patterns. Since all user-space eBPF programs must pass through the verifier, it also makes for a strategic point to enforce security policies. We propose a modification

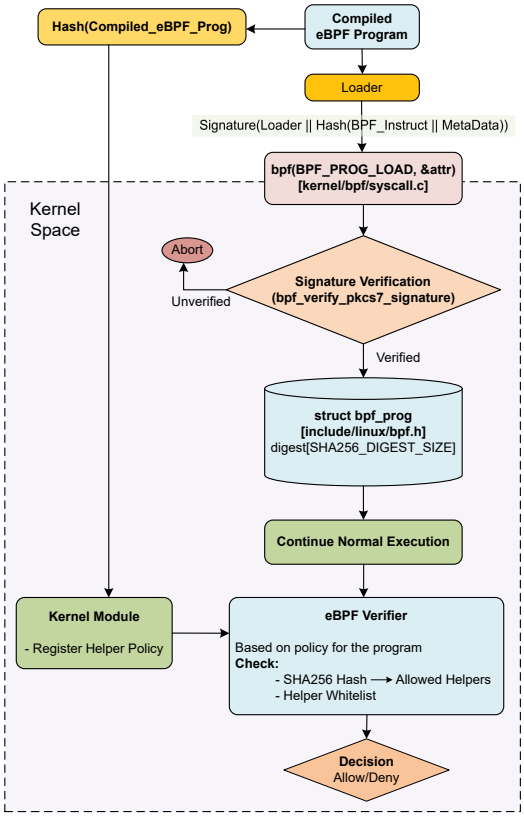



Fig. 6. Proposed model for load-time enforcement

to the verifier that introduces helper-level access control based on program identity. The key idea is to restrict the usage of sensitive helper functions by associating each program with a predefined set of permitted helpers. During the verification phase, each helper call is checked against this policy, and the program is rejected if it attempts to invoke unauthorized helpers. Figure 6 shows the steps involved in eBPF program load-time enforcement.

To implement this mechanism, our modified verifier integrates with the signed eBPF program framework introduced in the Linux kernel v6.18 [18]. This framework uses a trusted loader wherein a signature is generated over the loader instructions and the hash of the compiled eBPF program. The hash of the eBPF program is computed only on raw instructions to account for relocations. In the load phase, the call `bpf_prog_load` is made, and the PKCS#7 signature is received in the kernel for verification. Our solution uses the

digest from this secure pipeline, i.e., the SHA-256 hash of the eBPF program, as a stable cryptographic identity. This digest serves as the unique lookup key for the program's corresponding helper whitelist within the verifier. Furthermore, to enable flexibility and practical deployment, we introduce a kernel module that manages the helper access policy dynamically. It exposes interfaces to add and remove policies at runtime, allowing administrators to modify permissions without recompiling the kernel. The detailed implementation of our kernel patch can be found in the aforementioned GitHub repository.

Figure 7 shows the latency for loading the eBPF program incurred by the proposed mitigation approach compared to the default eBPF verifier. We use multiple eBPF programs that are having different number of eBPF helper invocations. It is evident that the proposed approach is competitive with the default eBPF verifier, while providing security against the aforementioned vulnerabilities.

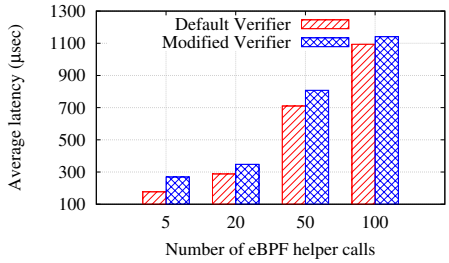



Fig. 7. Latency of eBPF verifier with Kernel modification compared to the default eBPF verifier.

This approach offers several advantages. First, enforcement occurs at load time, preventing malicious programs from being admitted into the kernel altogether. Second, it introduces fine-grained control over helper usage, directly addressing the root cause of the attacks demonstrated in Section IV. Third, it operates within the existing verification pipeline, avoiding additional runtime overhead. Furthermore, this design also follows a defense-in-depth strategy by combining multiple layers of protection. At the first layer, digital signature verification ensures that only trusted eBPF programs are loaded into the system. In the event of a container compromise, an attacker attempting to load a malicious eBPF program would fail this verification step. Even if the signing key itself is compromised, the second layer of defense remains effective. The verifier-based helper restriction enforces strict policies on the usage of helper functions, independent of program authenticity.

## VI. Conclusion

In this paper, we studied a vulnerability analysis of eBPF-enabled containerized 5G Core networks. By deploying a realistic testbed utilizing Open5GS and an eBPF-accelerated User Plane Function (UPF), we demonstrated that the shared kernel architecture inherent to cloud-native deployments exposes critical network functions to various threats. Our experimental results validated that a compromised container, possessing standard network privileges, can leverage eBPF to bypass namespace isolation. Furthermore, we presented mitigation approaches and implemented them inside the Linux Kernel.

In future work, we plan to expand this study to other prominent 5G software stacks [19], [20] to evaluate the universality of these vulnerabilities across different implementation choices. We also plan to investigate mitigation strategies based on reinforcement learning and online policy-driven dynamic permission models for runtime-generated eBPF programs. This will automatically infer fine-grained privilege boundaries for loaded eBPF programs based on observed program behavior, thereby enabling adaptive enforcement of least-privilege access control to reduce the attack surface.

## References


[1] "Study on management of cloud-native Virtualized Network Functions (VNF)," 3GPP, Tech. Rep. 28.834, 2022.

[2] "O2 Telefónica and Nokia roll out 5G standalone core on Amazon Web Services in the cloud," Available: https://www.nokia.com/newsroom/o2-telefonica-and-nokia-roll-out-5g-standalone-core-on-amazon-web-services-in-the-cloud/.

[3] "Swisscom, Ericsson and AWS collaborate on 5G SA Core for hybrid clouds – IEEE ComSoc Technology Blog."

[4] 3GPP, "5G; Study on Scenarios and Requirements for Next Generation Access Technologies," 3GPP, Tech. Rep. 3GPP TR 38.913, 2017.

[5] eBPF Foundation, "eBPF Documentation: What is eBPF?" eBPF Foundation, Tech. Rep., 2024.

[6] J. Zhou, Z. Ma, W. Tu, X. Qiu, J. Duan, Z. Li, Q. Li, X. Zhang, and W. Li, "Cable: A framework for accelerating 5G UPF based on eBPF," *Computer Networks*, vol. 222, p. 109535, Feb. 2023.

[7] C. Scheich, M. Corici, H. Buhr, and T. Magedanz, "eXpress Data Path Extensions for High-Capacity 5G User Plane Functions," in *Proc. of the 1st Workshop on eBPF and Kernel Extensions*, NY, USA, Sep. 2023, pp. 86–88.

[8] Y. X. Huang, K. C. Wang, and B. S. Ke, "Accelerating 5G Service-Based Architecture with Ebpf," in *Proc. of the International Conference on Networks, Communication and Computing*, NY, USA, Mar. 2024, pp. 200–209.

[9] "Falco: The Cloud-Native Runtime Security Project," The Falco Project, Tech. Rep., 2024.

[10] Isovalent, "Cilium: eBPF-based Networking, Observability, and Security," Isovalent, Tech. Rep., 2024.

[11] W. Yang, P. Chen, K. Liu, and H. Zhang, "ZeroTracer: In-Band eBPF-Based Trace Generator With Zero Instrumentation for Microservice Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 36, no. 7, pp. 1478–1494, Jul. 2025.

[12] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023.

[13] 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 *Proc. USENIX Conference on Security Symposium*, USA, Aug. 2023, pp. 5971–5988.

[14] D. Jin, V. Atlidakis, and V. P. Kemerlis, "EPF: Evil Packet Filter," in *USENIX Annual Technical Conference*, 2023, pp. 735–751.

[15] E. LLC, "Eupf: User plane function (upf) for 5g core network," Edgecom LLC, Tech. Rep., 2024.

[16] "Open Source implementation for 5G Core and EPC," Available: https://open5gs.org/.

[17] A. Gungor, "Ueransim: Open source 5g ue and ran simulator," Tech. Rep., Available: https://github.com/aligungr/UERANSIM.

[18] K. P. Singh, "[PATCH v5 00/12] Signed BPF programs," Available: https://lore.kernel.org/bpf/20250921133133.82062-1-kpsingh@kernel.org/.

[19] "free5GC: Open Source Core Network Implementation," Available: https://free5gc.org/.

[20] "Openairinterface: Open source 5g software for ran and core," Available: https://openairinterface.org/.