








Flexible and Secure Process Confinement with eBPF

Carlo Mazzocca^{}, Andrea Garbugli^{}, Michele Armillotta^{},
Rebecca Montanari^{}, and Paolo Bellavista^{}

University of Bologna, Bologna, Italy

{carlo.mazzocca, andrea.garbugli, rebecca.montanari,
paolo.bellavista}@unibo.it, michele.armillotta2@studio.unibo.it

Abstract. To avoid potential bugs and vulnerabilities, it is crucial to confine process execution within well-defined boundaries, specifying which resources are accessible and what operations are allowed. Numerous technologies have emerged in Linux environments to address process confinement or isolation. However, these solutions often lacked tailored support, leading to a fragmented landscape of complex implementations. Given the need to support different security abstractions, the Extended Berkeley Packet Filter (eBPF) has emerged as a promising technology for extending the capabilities of the Linux kernel functionalities, offering a simple and flexible approach for process confinement. This paper introduces a framework that leverages eBPF to achieve flexible and secure process confinement. We developed a prototype implementation and evaluated its overhead in limiting filesystem capabilities. Experimental findings underscore the effectiveness of our framework, demonstrating that it can seamlessly integrate into Linux systems without incurring remarkable overhead.

Keywords: eBPF · Operating System Security · Application Confinement · Process Isolation · Sandboxing · Linux Kernel Security

1 Introduction

Process confinement or isolation is a key security technique to prevent untrusted components from directly accessing resources and executing operations. This is particularly critical in contexts like cloud computing [18], where the integrity of virtualized platforms and the software systems running on them can be compromised by malicious actors exploiting insufficient security isolation [2]. Thus, confining processes within well-defined boundaries and restricting their access to only authorized resources and operations reduces the potential damage that bugs or vulnerabilities could cause [19], strengthening the overall system security.

Over the years, a wide variety of technologies for process confinement have emerged on Linux (e.g., SELinux [4] and seccomp [15]). Despite their popularity, most were not originally tailored for this task, and relying solely on individual

mechanisms does not provide comprehensive security coverage. On the other hand, working solutions combine existing technologies, which can increase system complexity and introduce vulnerabilities [9]. Given the need to support different security abstractions without complex implementations, the Linux community introduced the Linux Security Modules (LSM) framework [20]. LSM offers a generic interface that allows the creation of flexible access control policies in a modular way, without modifying the kernel. Most of the confinement and isolation mechanics leverage kernel functionalities rather than developing kernel modules from scratch as it is a costly operation and small errors can result in serious failures.

In this direction, the Extended Berkeley Packet Filter (eBPF) [7] is envisioned as a valuable technology for implementing process confinement by running programs in the privileged context of the kernel without directly modifying it [6]. An eBPF program is loaded in the kernel as a bytecode and is activated when the kernel or an application triggers a certain hook point, e.g., the `write()` system call. Before being attached to the requested hook, an eBPF program is validated and the bytecode is translated into the machine-specific instruction set, achieving the same performance as natively compiled kernel code. Additionally, eBPF supports a data structure called *map* that allows efficient data exchange between userspace and kernelspace.

This paper presents a framework for implementing flexible and secure process confinement using eBPF. Our solution employs a modular architecture composed of various eBPF programs, each responsible for regulating access to different subsystems such as filesystem and networking. These programs enforce access control policies, provided through maps, which specify how processes are restricted. The combination of eBPF and access control policies allows dynamically managing Linux resources based on specified conditions.

We developed a prototype of the proposed framework and evaluated its performance in limiting filesystem capabilities, i.e., accessing protected resources. Experimental results demonstrate that eBPF can effectively confine processes attempting to access protected files in a Linux system, without introducing significant overhead. The primary contributions of this paper are as follows: (i) presenting a framework for implementing flexible and secure process confinement with eBPF, and (ii) developing a prototype and evaluating the overhead involved in restricting filesystem capabilities.

The remainder of this paper is structured as follows. Section 2 presents eBPF and offers an overview of existing security mechanisms in Linux systems. Section 3 reviews the related work in the field. In Sect. 4, we describe our framework, whose evaluation is reported in Sect. 5. Section 6 discusses experimental results and design choices. Finally, Sect. 7 concludes and presents future research directions.

2 Background

This section provides the background for understanding eBPF and the main mechanisms for implementing process confinement in Linux systems.

2.1 eBPF

As clearly outlined by the full form (i.e., Berkeley Packet Filter), the original BPF also referred to as classic BPF (cBPF) is a technology employed for packet filtering. However, the extended version can do much more and eBPF is considered a standalone term, often used interchangeably with BPF. The success of eBPF owes to its capacity to extend the kernel capabilities without modifying the kernel source code or loading kernel modules. This is achieved through eBPF programs, which run at the kernel level and add further capabilities to the operating system. These programs are event-driven as they are executed when the kernel or an application triggers a specific hook point, such as system calls or network events.

Thus, developers are only required to implement eBPF programs. To simplify this process, they can also write their eBPF programs using a subset of higher-level programming languages such as C by leveraging available development toolchains. For instance, our eBPF programs have been implemented in a C-like language and compiled through libbpf-bootstrap. These eBPF programs are associated with hooks and loaded into the kernel through `bpf(2)` system call. As eBPF programs are run in the kernel, they are restricted in what they can do and undergo a verification process that ensures they will not cause damage to the system. For instance, all programs must run to completion, hence, they cannot sit in a loop forever. To guarantee execution efficiency, the generic bytecode of the program is translated into the machine-specific instruction set through a Just-in-Time (JIT) compilation step. This has led to many novel use cases that leverage eBPF, including high-performance networking in cloud environments, and security functionalities as done in this paper.

2.2 Linux Process Confinement

Process confinement is a crucial concern for operating systems as it prevents unauthorized access to sensitive resources, ensures that processes operate within their designated boundaries, and provides better control and management of system resources (e.g., CPU, memory, and filesystem). In the following, we present the main mechanisms for process confinement in Linux systems and discuss their limitations.

Linux DAC: The most basic solution to confine processes is Discretionary Access Control (DAC) [20], which enforces security by ownership. A user owning a file can set the read, write, and execute permissions for that file. The system owner does not have full control over the system, while users control data at their discretion. In this access control model, the main concern is given by the root user who can do almost anything to a system, demanding additional and more secure mechanisms.

Linux MAC: When using Mandatory Access Control (MAC), the operating system constrains the capability of a process or user (including root) to access and perform operations on resources according to predefined security policies. Among existing LSMs implementing MAC, Security-Enhanced Linux (SELinux) and AppArmor are widespread solutions [4]. In SELinux, each process and resource is assigned a security context, used by security policies to grant or deny access. Policies define rules that establish how processes with security contexts can interact with resources having different security contexts. AppArmor allows administrators to specify the resources that are accessible from a process. Each process is bound to a profile that defines the set of permitted operations to resources. AppArmor profiles are simpler to write and manage, being more suitable for scenarios where simple MAC mechanisms are required.

Namespaces and Cgroups: Namespaces and groups are Linux features that limit the resources and how much of those resources a process can use [17], enabling a simple case of process confinement. Namespaces offer a mechanism to partition resources, allowing one group of processes to access a specific set of resources, while another group sees a distinct set. Partitioned resources span a spectrum: process IDs, file systems, network sockets, and user IDs. On the other hand, cgroups is another Linux feature for constraining non-enumerable resources like memory, CPU, and I/O bandwidth.

Ptrace: While primarily designed for debugging, the `ptrace` system call can be somehow employed for process confinement with certain limitations [5]. By leveraging this function, a process gains the ability to monitor and control its child processes, intercepting and manipulating system calls, modifying memory contents, or terminating the child process when required. However, employing `ptrace` for process confinement has significant drawbacks. These include notable performance overhead due to the need for system call interception, increased complexity in implementation, and increased security risks, as `ptrace`-based solutions may be vulnerable to privilege escalation or bypass.

Seccomp: Seccomp [15] is a security feature of Linux that restricts processes to a limited set of system calls, creating a secure and controlled execution environment. A process that tries to invoke a system call not included in the predefined set is terminated by the kernel. `seccomp-bpf` [13] is an extension that overcomes the low flexibility of seccomp by adding the capability to filter system calls through eBPF rules.

3 Related Work

Although eBPF has a history of focusing on performance, recent literature underscores a growing body of research that proposes it as a valuable technology to

enhance system security. This section reviews key works utilizing eBPF for process confinement in Linux systems. Findlay et al. [9] proposed `bpffbox`, a tool that combines a policy language with eBPF to implement process confinement for userspace applications. While recognizing that this framework was among the first to adopt eBPF for process confinement, it does not support the instrumentation of processes at the system call level, a key feature of our framework. Additionally, from the publication date, `bpffbox` has not been maintained and new eBPF functionalities introduced since then are unsupported by this tool. Hung et al. [11] presented `Sfiter`, a framework for protecting security-critical kernel modules in Android devices. `Sfiter` uses predefined policies to filter system calls from untrusted processes to Qualcomm KGSL GPU device driver and Binder IPC modules. Although this tool can be potentially deployed on any device running Linux, it only focuses on these modules, resulting in limited flexibility compared to our solution. Jia et al. [12] created a programmable tool for filtering system calls called `Seccomp-eBPF`, which overcomes the restricted programmability of traditional `seccomp` where security policies are limited to static allow lists. However, `seccomp-eBPF` is restricted to system calls and does not consider hook points.

Moreover, using eBPF for process isolation has also found applications in specialized domains such as web applications. Abbadini et al. [1] introduced `Cage4Deno`, a Deno JavaScript and TypeScript runtime enhancement. This extension facilitates the creation of fine-grained sandboxes for executing subprocesses, safeguarding against compromised utilities that could threaten filesystem confidentiality and integrity. In containerization, eBPF has been adopted as a valuable technology to strengthen the host kernel against privilege escalation attacks mounted from within containers [8] and to allow containers to safely enforce fine-grained policies in the kernel [3].

4 Framework Architecture

As illustrated in Fig. 1, our framework’s architecture consists of a policy manager, a loader, a policy enforcer, and several maps. The policy manager receives access control policies to enforce, parses them, and loads them into maps. Maps are particular eBPF data structures that connect the userspace applications and kernelspace eBPF programs. Once the loader compiles and loads the eBPF code into the kernel, the policy enforcer retrieves the access control policies from the eBPF maps. The policy manager and the loader are implemented in C, while eBPF programs are written in a restricted C-like language.

4.1 eBPF Maps

eBPF offers data structures called maps to facilitate information exchange between userspace and kernelspace [14]. These maps come in various types, such as hash, array, and bloom filters. Data shared using maps can be accessed

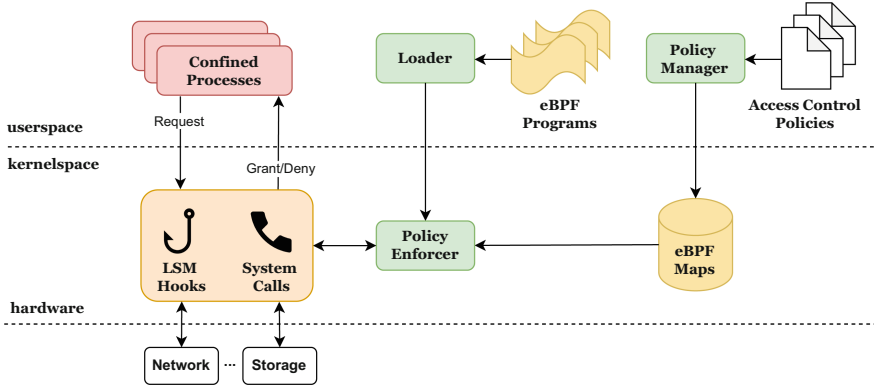


Fig. 1. Overview of our framework’s architecture.

through the `bpf(2)` system call, which allows developers to create maps and efficiently manage their elements, i.e., read, update, and delete operations.

We use maps to share access control policies from userspace to eBPF programs running within the kernel. Maps are crucial to our architecture as they empower dynamic modification of access control policies without directly impacting eBPF programs. This flexibility allows for real-time adjustments to policy enforcement, enhancing the overall adaptability and security of the system.

4.2 Policy Manager and Loader

Access control policies are structured as key-value pairs, following the format **key:value**. To ensure high flexibility, each subsystem has policies that can only be defined and modified by authorized users (e.g., system administrators), as these policies are stored in a directory controlled by the root. We design policies following a *whitelisting* approach, where all operations and resources that are not specified are denied. By forcing authorized users to explicitly determine the admitted operations on resources, we prevent undesired circumstances where access to resources is unintentionally granted, e.g., creating a new file and forgetting to restrict its access.

The policy manager plays a key role in our architecture. It parses access control policies and stores them in maps, which are then referenced by the corresponding eBPF programs to enforce access control efficiently and flexibly. Listing 1.1 reports the snippet of an access control policy that restricts access to the file `protected.txt` exclusively to the `cat` program using the `open()` system call. Thus, any other operation will be rejected.

```

1 open.protectedFiles:/home/ex/Desktop/protected.txt;
2 open.allowedProc:cat;

```

Listing 1.1. Example of an access control policy.

It is worth noting that the proposed architecture decouples policy definition from their enforcement, allowing any changes without requiring modification the eBPF program source code.

Similarly to the policy manager that uploads policies into maps, the loader provides eBPF programs to the policy enforcer. The loader receives eBPF programs responsible for enforcing access control policies and compiles them into a loadable object file, which is then uploaded into the kernel.

4.3 Policy Enforcer

The policy enforcer consists of various eBPF programs, each responsible for regulating access to specific kernel functionalities and resources, such as the filesystem, networking, and process management. These eBPF programs are linked to a range of hooks within the Linux operating system, allowing them to intercept critical operations or events. Predefined hooks include system calls, kernel tracepoints, and network events, among others. When a process triggers a hook point, the corresponding eBPF program is activated and, based on the access control policies stored in maps, either grants or denies access to the resource or functionality.

Furthermore, our framework extends beyond simple authorization by implementing *controlled execution* of unauthorized processes. While existing solutions enforce access control policies by granting or denying access to certain resources, we allow unauthorized processes to perform mock executions, monitoring and logging their intended actions. This feature can be potentially useful for evaluating unknown process operations, effectively emulating a sandbox-like environment before letting them operate on the real system. For instance, regarding the `write()` system call, the policy enforcer can redirect it to a mock file, which can be analyzed later to assess potential threats.

It is worth noting that eBPF programs can be seamlessly attached to any predefined hook. If a predefined hook does not meet a particular need, developers can create custom kernel probes (`kprobe`) or user probes (`uprobe`) to intercept actions within kernel or user space applications. These capabilities empower users with great flexibility and fine-grained control over their operating systems.

In Listing 1.2, we report a snippet of an eBPF program that enforces the policy previously introduced. Specifically, this eBPF program restricts access to the file `protected.txt` according to the policy reported in Listing 1.1. Thus, only the `cat` program is allowed to execute the `open()` system call, while other operations are denied. For example, suppose a process attempts to perform a `write()` operation. In that case, the policy enforcer verifies whether the `write()` belongs to the permitted operations (lines 8–18) by comparing the system call invoked and the file path used by the process with the content of the corresponding map loaded by the policy manager. Since the write operation is not allowed on `protected.txt`, the `value_proc` is set to 0 and the access is denied. Additionally, to monitor intended operations, the `write()` system call is redirected to an error file (lines 24 and 25).

```

1 // Get file name
2 bpf_probe_read_user_str(filename, sizeof(filename), (char*)ctx->
   args[1]);
3
4 // Get process name
5 bpf_get_current_comm(comm, sizeof(comm));
6
7 // Look up elements in eBPF maps
8 value_file = bpf_map_lookup_elem(&OPEN_FILES_MAP, &filename);
9 value_proc = bpf_map_lookup_elem(&OPEN_PROC_MAP, &comm);
10
11 // Check flag status
12 if (value_file && (*value_file > 0)) {
13     flag_path = 1;
14 }
15
16 if (value_proc && (*value_proc > 0)) {
17     flag_process = 0;
18 }
19
20 if (flag_path == 1 && flag_process == 1) {
21     bpf_printk("Access denied\n");
22
23     // Redirect to error file
24     char new_path[MAX_DIM_PATH] = "/home/ex/Desktop/error.txt";
25     ret = bpf_probe_write_user((void *)ctx->args[1], new_path,
   sizeof(new_path));
26     return ret;
27 } else {
28     return 0;
29 }

```

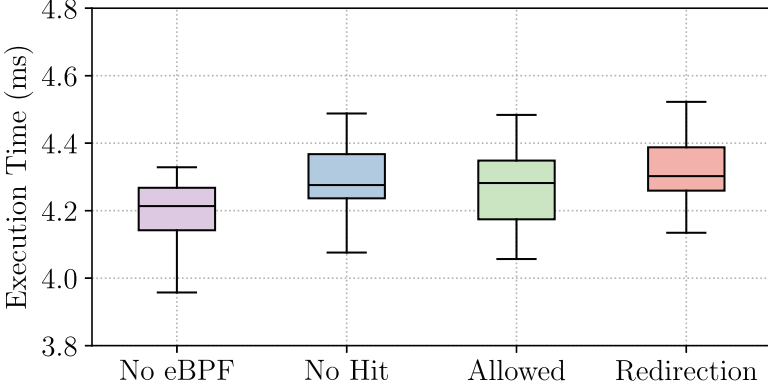
Listing 1.2. Example of an eBPF program for process confinement.

5 Evaluation

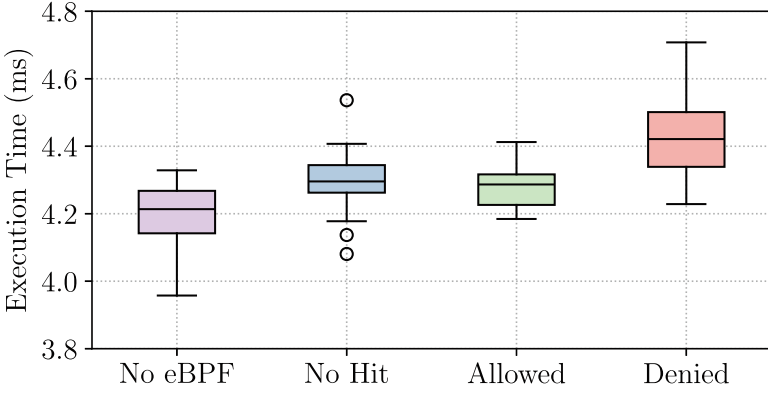
In this section, we conduct experiments to evaluate how our framework impacts filesystem access latency. We developed a benchmark suite to measure the overhead imposed on both confined applications and the system when opening files. Our experimental findings demonstrate that our framework can enforce fine-grained access control with minimal overhead.

5.1 Experimental Setup

All experiments were performed on a local setup that matches a typical cloud environment. The machine runs Ubuntu 22.04 and is equipped with an 18-core



(a) `open()` system call - Execution times
1



(b) `file_open()` LSM hook - Execution times
1

Fig. 2. Performance evaluation of the proposed solution for different eBPF hooks (system call and LSM).

Intel i9-10980XE @ 3.00 GHz and 64 GB of RAM. We implemented a prototype of our framework in C and evaluated it using the eBPF program shown in Listing 1.2, which restricts access to a protected file.

5.2 Experiments

We conducted a set of experiments to evaluate the impact of our framework on the filesystem access latency. Each experiment was executed 50 times, and results have been averaged.

We evaluated the overhead imposed by our framework for authorized access to the file `protected.txt` (**Allowed**), access to an unprotected file (**No Hit**), and

redirection of unauthorized access to the error file `error.txt` (**Redirection**). We also run the experiments without deploying our framework, using it as a baseline operation of the system (**No eBPF**) to compare the introduced overhead. Furthermore, to show the flexibility of the proposed solution, we performed experiments varying the event that triggers the activation of the eBPF program, summarized below.

System Call: The eBPF program is activated when the benchmark invokes the `open()` system call.

LSM Hook: The eBPF program is activated when the benchmark invokes the `file_open` hook.

In Fig. 2a and Fig. 2b, we report the system’s overhead associated with the `open()` system call and `file_open` LSM hook, respectively. Under both configurations, the operations do not show significant differences, with an average execution time slightly above 4 ms. Additionally, there are no notable differences among operations when using eBPF for enforcing access control policies. The figure shows that systems not using eBPF for confining processes do not achieve enhanced performance.

It is worth noting that when using the LSM hook, we cannot redirect the request to an unprotected file; the operation is simply denied (**Denied**). Despite this difference in handling unauthorized requests, the execution time remains practically the same.

Figure 3 offers a comparison among the execution time for kernelspace eBPF programs attached to the `open()` system call and `file_open` LSM hook. The **Hit** column corresponds to authorized access to the resource, while the **Miss** column corresponds to unauthorized requests. It is worth noting that, in this case, attaching eBPF programs at the system call or LSM hook level does not show remarkable differences as the execution time is a few microseconds.

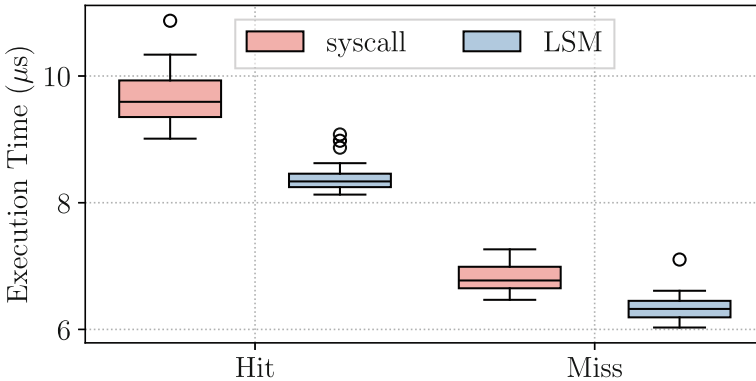


Fig. 3. Comparison of the execution times for the kernelspace eBPF programs attached to the `open()` system call and `file_open` LSM hook.

6 Discussion

Experimental results highlight that our eBPF-based framework can restrict filesystem access for opening files while introducing negligible overhead, regardless of whether the eBPF program is triggered by a system call or an LSM hook. The main difference observed between using the system calls and the LSM hooks levels lies in the flexibility they offer. Attaching the eBPF program to the `open()` system call enables controlled execution by redirecting the request to an error file. However, this method does not support returning an error code, which is possible when the eBPF program is triggered by the `file_open` hook. Moreover, system call redirection seems not always feasible, e.g., redirecting networking packets to a mock socket. In future work, we plan to deeply explore these mechanisms for the various Linux subsystems.

On the other hand, using the LSM hook allows the eBPF program to access more detailed information like the struct file, enabling the design of more fine-grained access control policies. For example, the `file_open` LSM hook can read the file's content and scan for forbidden keywords before allowing opening the file. However, this approach does not allow for redirecting the request to an error file as the primary role of this LSM hook is to allow or deny access, not to modify the behavior of the file operation to open a different file. This functionality may be achieved by creating a wrapper function that attempts to open the desired file and, upon receiving a denial, automatically redirects the request to an error file instead.

Given these considerations, it is evident that there cannot be a one-size-fits-all approach for selecting the level at which eBPF programs are triggered; the choice depends on the specific scenarios and requirements. For instance, LSM hooks are preferable when real-time information about the context of a potential threat is needed, and a simple yes-or-no decision suffices to determine whether the execution of the observed process is allowed. On the other hand, interception at the system call level is preferable when it is necessary to activate a redirect trigger to a mock and isolated "object" that can be analyzed later.

7 Conclusions and Future Work

This paper presents a framework for confining applications within Linux systems, leveraging the high flexibility of eBPF to enable fine-grained control over process behavior, enhancing security and system integrity. We implemented a prototype and evaluated its efficiency by measuring its impact on filesystem access latency. Experimental results demonstrate that using eBPF to confine processes introduces negligible overhead.

Future work includes expanding our framework to encompass additional critical Linux subsystems and comprehensively evaluating its impact across these areas. Furthermore, we plan to integrate our framework with network acceleration platforms to address security issues often overlooked in such contexts [10, 16]. Such frameworks typically leverage shared memory mechanisms to achieve zero-copy communications and enhanced performance, though this comes at the cost

of removing isolation between applications. Integrating our eBPF solution for process confinement could address this issue.

Acknowledgments. This work was partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan program funded by the European Union - NextGenerationEU.

References

1. Abbadini, M., Facchinetti, D., Oldani, G., Rossi, M., Paraboschi, S.: Cage4deno: a fine-grained sandbox for deno subprocesses. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, pp. 149–162. Association for Computing Machinery, New York (2023). <https://doi.org/10.1145/3579856.3595799>
2. Bazm, M.M., Lacoste, M., Südholt, M., Menaud, J.M.: Isolation in cloud computing infrastructures: new security challenges. *Ann. Telecommun.* **74**(3), 197–209 (2019)
3. Bélair, M., Laniepce, S., Menaud, J.M.: Snappy: programmable kernel-level policies for containers. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC 221, pp. 1636–1645. Association for Computing Machinery, New York (2021). <https://doi.org/10.1145/3412841.3442037>
4. Brimhall, B., Garrard, J., De La Garza, C., Coffman, J.: A comparative analysis of linux mandatory access control policy enforcement mechanisms. In: Proceedings of the 16th European Workshop on System Security, EUROSEC 2023, pp. 1–7. Association for Computing Machinery, New York (2023). <https://doi.org/10.1145/3578357.3589454>
5. Connor, R.J., McDaniel, T., Smith, J.M., Schuchard, M.: PKU pitfalls: attacks on pku-based memory isolation systems. In: 29th USENIX Security Symposium (USENIX Security 2020), pp. 1409–1426. USENIX Association (2020). <https://www.usenix.org/conference/usenixsecurity20/presentation/connor>
6. Dejaeghere, J., Gbadamosi, B., Pulls, T., Rochet, F.: Comparing security in eBPF and WebAssembly. In: Proceedings of the 1st Workshop on EBPF and Kernel Extensions, eBPF 2023, pp. 35–41. Association for Computing Machinery, New York (2023). <https://doi.org/10.1145/3609021.3609306>
7. eBPF Documentation: eBPF (2024). <https://ebpf.io/>. Accessed 2 May 2024
8. Findlay, W., Barrera, D., Somayaji, A.: Bpfcontain: fixing the soft underbelly of container security. arXiv preprint [arXiv:2102.06972](https://arxiv.org/abs/2102.06972) (2021)
9. Findlay, W., Somayaji, A., Barrera, D.: bpfbox: simple precise process confinement with eBPF. In: Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW 2020, pp. 91–103. Association for Computing Machinery, New York (2020). <https://doi.org/10.1145/3411495.3421358>
10. Fried, J., et al.: Making Kernel bypass practical for the cloud with junction. In: 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 2024), pp. 55–73 (2024)
11. Hung, H.W., Liu, Y., Sani, A.A.: Sifter: protecting security-critical kernel modules in Android through attack surface reduction. In: Proceedings of the 28th Annual International Conference on Mobile Computing And Networking, MobiCom 2022, pp. 623–635. Association for Computing Machinery, New York (2022). <https://doi.org/10.1145/3495243.3560548>

12. Jia, J., et al.: Programmable system call security with ebpf. arXiv preprint [arXiv:2302.10366](https://arxiv.org/abs/2302.10366) (2023)
13. Kernel, T.L.: Seccomp BPF (SECure COMPUting with filters) (2024). https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html. Accessed 2 May 2024
14. Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M.V.: Creating complex network services with eBPF: experience and lessons learned. In: 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), pp. 1–8 (2018). <https://doi.org/10.1109/HPSR.2018.8850758>
15. manual page, L.: seccomp(2) (2024). <https://man7.org/linux/man-pages/man2/seccomp.2.html>. Accessed 2 May 2024
16. Rosa, L., Garbugli, A., Corradi, A., Bellavista, P.: INSANE: a unified middleware for QoS-aware network acceleration in edge cloud computing. In: Proceedings of the 24th International Middleware Conference, pp. 57–70 (2023)
17. Rosen, R.: Resource management: linux kernel namespaces and cgroups. *Haifux* **186**, 70 (2013)
18. Shu, R., et al.: A study of security isolation techniques. *ACM Comput. Surv.* **49**(3) (2016). <https://doi.org/10.1145/2988545>
19. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: ERIM: secure, efficient in-process isolation with protection keys (MPK). In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1221–1238. USENIX Association, Santa Clara (2019). <https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner>
20. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux security modules: general security support for the linux kernel. In: 11th USENIX Security Symposium (USENIX Security 2002) (2002)