UNIVERSITY OF BERGAMO

School of Doctoral Studies

Doctoral Degree in Engineering and Applied Sciences

XXXVII Cycle

SSD: ING-INF/05

Sandboxing and Data Protection in Cloud Computing Environments

Advisor

Prof. Stefano Paraboschi

Doctoral Thesis by

Marco Abbadini

Student ID 1048650

Academic Year 2023/2024

To my mom
to my dad and brother
to my family

* * *

A mia mamma
a mio papà e mio fratello
alla mia famiglia

Abstract

Modern cloud applications often rely on cloud environments like NodeJs for their execution. As the size of applications grows, so does the number of dependencies and the risks deriving from the usage of native code libraries written in an unsafe language. Even if the cloud app is written in a high level safe language and the runtime is considered safe, a vulnerability inside a binary dependency (i.e., a bug in its implementation) can lead to a system compromise. Sandboxing represents a robust way to limit applications' access to unauthorized portions of the system and protect sensitive information from data leakage. The goal of this thesis is to provide cloud environments with a robust, secure and flexible protection by proposing mechanisms to implement sandboxes with strong security guarantees without sacrificing applications' performance. In the results reported in this thesis we have investigated the security model of modern JavaScript, TypeScript and WebAssembly runtimes (e.g., NodeJs, Deno, Bun and Wasmtime, WasmEdge, Wasmer respectively) and propose a way to enforce a fine-grained security protection through policy definition and enforcement. In particular, we point out how critical are the risks caused by unsafe third party dependencies in JavaScript runtimes and how the WebAssembly System Interface (WASI) of modern WebAssembly runtimes does not provide sufficient security guarantees while interacting with the underlying filesystem. After investigating these critical aspects of these runtimes we have proposed a security model and an implementation that could provide sufficient protection against a great number of vulnerabilities, and we have conducted an extensive experimental analysis to evaluate the effectiveness, efficiency and performance of our proposal. Subsequently, after having explored the aspects of data protection in Cloud environments, we investigated how it was possible to extend data protection and data control guarantees in a decentralized context, where decentralized filesystems can be deployed on and complement cloud infrastructures, by proposing cryptographic techniques to protect users' data. The results of the research presented in this document have been published on different international conference proceedings and are accompanied by open-source implementations.

Table of Contents

List of Figures

List of Tables

1. Introduction

The last decades have been characterized by a great growth of digital services, from applications for video streaming, ticket booking, text editing to more complex ones. The need of efficient, portable and scalable applications led to growth of cloud-based services that nowadays represent a great majority. The advantages of developing cloud-based applications (e.g, flexibility and scalability, cost saving and quick deployments) led to make a transition from old standalone applications to this modern paradigm. From public administrations to private services, cloud computing represents the *de-facto* standard for application development.

Due to the important role that this type of applications cover, the need arises to provide secure environments and protect users' data. Indeed, a compromise of the system running a cloud application could led to data leakages, denial of services or even worse consequences. Modern frameworks provide a good level of protection but since services become more and more complex and since technology keep changing rapidly, it is not easy to provide complete, secure, efficient and flexible protections.

1.1 Research topic and objectives

The goal of this thesis is to propose advancements in cloud environments protection by developing sandboxing solutions that provide access control mechanisms on the underlying system. The idea is to enable developers to define access control policies for their applications that then are enforced by sandboxing mechanisms. Restricting the access of the system resources allows to reduce the attacking surface that a buggy software could expose. The core idea of the access control policies and their enforcement is to propose a *least privilege* compliant solution that can reduce as much as possible the vulnerable surface of the system. Focusing our attention on cloud environments, JavaScript and TypeScript are widely used languages for the implementation of cloud applications and their usage on the server-side is enabled by JavaScript runtimes like NodeJs. Runtimes like NodeJs support the execution of native code software to facilitate the development of the application backend by the developers. However, the security concerns of breaking the isolation between the JS application and the host OS due to the execution of memory unsafe libraries led industry and researchers to provide more secure alternatives [60, 133, 41, 191]. Deno [60] has been introduced as a security-oriented JavaScript runtime that proposes a capability-based interaction with the system, but it still lacks of a robust

protection against native code dependencies. WebAssembly [133] (Wasm) represents a great step forward for its security guarantees, but modern Wasm runtimes (e.g., Wasmtime [12] and Wasmer [195]) lack in granularity to access the system through the WebAssembly System Interface (WASI). Considering industry's need to provide increasingly stronger protection as the complexity and size of modern systems increases, and the continued increase in cloud application development, the need arises to provide a robust and efficient way to protect against vulnerabilities in cloud runtimes and to provide users' data with more protection and control. Data protection of storage providers represents a critical aspect of today's modern systems that heavily relies on these technologies. A goal of my research work is also to provide data owners with novel data protection techniques in order to provide greater control over the resources uploaded on these type of services.

1.2 Dissertation structure

In this section the outline of the document is described.

Chapter 2 describes the research approach adopted during the PhD research period explaining the processes and methods applied.

Chapter 3 presents the work done in Cage4Deno. The goal of this work is to extend the security guarantees of the Deno runtime providing a way to sandbox processes of native code software executed as a third party dependency of an application. The proposal describes the threat model, the policy and the code implementation, as well as the experimental analysis.

- *Section 3.1* introduces the scenario, the problem and the goal of the work, pointing out our contribution;
- *Section 3.2* describes the technologies background, focusing on the Deno runtime and Linux Security Modules (LSMs) involved;
- *Section 3.3* introduces and describes the threat model and our contribution, highlighting the design objectives of our proposal;
- *Section 3.4* describes the design choices and the implementation of Cage4Deno. The section introduces the policies supported and how they are enforced through Landlock LSM and eBPF LSM;
- *Section 3.5* describes how the policies for Cage4Deno can be generated

- *Section 3.6* reports the experiments' results and shows how our solution is effective in mitigating CVEs. A performance analysis is reported and the overhead of our solution is discussed.

- *Section 3.7* discusses the related work of our proposal

- *Section 3.8* concludes the chapter discussing possible future improvements.

Chapter 4 reports the results obtained in [4] where we proposed a solution to enhance WebAssembly runtimes' security guarantees by using kernel-level mechanisms to enforce access control policies. The goal is to enhance the protection offered by the WASI interface while interacting with the resources of the underlying system. The proposal describes the critical aspects and risks of the current implementation and provide a model improvement.

- *Section 4.1* introduces the problem and the state-of-the-art WebAssembly runtimes;

- *Section 4.2* describes the thread model considered;

- *Section 4.3* introduces to the architecture of WebAssembly runtimes and their interaction with the underlying system, and describes the improvements and the new interaction proposed by our solution;

- *Section 4.4* reports the experiment results of our approach;

- *Section 4.5* describes the related works of our proposal;

- *Section 4.6* concludes the chapter and discuss future works.

Chapter 5 describes advances in cloud application sandobxing proposing an extension of the work done with Cage4Deno by introducing NatiSand, a new, finer-grain and more powerful sandboxer that guarantees a better level of protection over the system. Here we extended the protection guarantees of Cage4Deno by restricting accesses to the filesystem, Inter Process Communication (IPC) mechanisms and network resources. The work is supported by an extensive experimental analysis to evaluate the performance impact of our proposal and the level of protection that NatiSand can provide using our policy-based implementation.

- *Section 5.1* introduces the problem, state-of-the-art runtimes, the scenario and our contribution;

- *Section 5.2* describes the background of JavaScript runtimes and resource protection mechanisms used by our proposal, giving an insight of Seccomp, eBPF and and Landlock LSMs;
- *Section 5.3* discusses the motivation of our work, giving examples of security risks and compromises. The threat model is introduced and explained.
- *Section 5.4* describes the design choices and the implementation effort to provide our protection. The section points out the security, usability, compatibility and performance objectives that our solutions satisfies, and gives an high level description of NatiSand's architecture. This section also highlights the isolation features provided and the possibilities of integration with JS runtimes.
- *Section 5.5* introduces the policies used by NatiSand giving an insight into the policy structure and generation;
- *Section 5.6* describes the case study of the Deno runtime, where our solution has been implemented;
- *Section 5.7* discussed the experiments and reports the performance evaluation analysis and the exploit mitigation results;
- *Section 5.8* discusses the related work of NatiSand;
- *Section 5.9* concludes the chapter.

Chapter 6 describes the work published at IEEE CLOUDCOM [3] focused on sandboxing cloud applications with a lightweight impact on performances. The work discusses a way to track and detect processes' dependencies and enforce policy-based sandboxes with a resource-based granularity. Experiments showcase the mitigation of vulnerabilities using kernel security modules based on policies generated through `dmng`, the tool we propose that takes advantage of the methodology described in this work and that we provided with an open-source implementation.

- *Section 6.1* gives an overview and introduces the scenario of cloud computing applications;
- *Section 6.2* describes the motivation of our proposal, the threat model and the requirements of modern cloud applications;
- *Section 6.3* describes the approach followed while developing our solution and the way our policies are created;

- *Section 6.4* introduces the instrumentation methods adopted by Dmng describing Ptrace-based solutions and eBPF-based ones;
- *Section 6.5* introduces the policy structure and the policy customization possibilities, discussing the coverage and effectiveness of the generated templates;
- *Section 6.6* discusses how applications can be sandboxed taking advantage of the fine granularity of our policies, and proposing LSM-based sanboxes;
- *Section 6.7* describes the experiments conducted and the effectiveness of our proposal. An overhead analysis is conducted and reported;
- *Section 6.8* presents the related work of our proposal;
- *Section 6.9* concludes the chapter and discusses future works.

Chapter 7 extends our research to decentralized storage systems. Specifically, we analyzed the security guarantees of the InterPlanetary FileSystem (IPFS) and proposed *Mix-IPFS* an adaptation of IPFS to support the *Mix&Slice* [18] primitive in order to provide data owners with the ability to maintain the control over resources uploaded on IPFS.

- *Section 7.1* introduces to decentralized storage technologies and the problem of data owner control;
- *Section 7.2* discusses the preliminary concepts and technologies;
- *Section 7.3* describes the high level idea of our proposal;
- *Section 7.4* introduces and describes the architecture of Mix-IPFS, the architectural design and the workflow of resources on the network;
- *Section 7.5* reports the experimental evaluation highlighting the impact on latency and storage;
- *Section 7.6* discusses the related works of Mix-IPFS;
- *Section 7.7* concludes the work.

Chapter 8 concludes the document and discusses future works.

1.3 Publications

This research resulted into different papers published in international conferences proceedings and that are following listed in chronological order:

1. **"Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses"** [5], Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, Stefano Paraboschi, *in Proc. of the 18th ACM ASIA Conference on Computer and Communications Security (ASIACCS), Melbourne, Australia, July 10-14, 2023*

2. **"Leveraging eBPF to enhance sandboxing of WebAssembly runtimes"** [4], Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, Stefano Paraboschi, *in Proc. of the 18th ACM ASIA Conference on Computer and Communications Security (ASIACCS), Melbourne, Australia, July 10-14, 2023*

3. **"NatiSand: Native Code Sandboxing for JavaScript Runtimes"** [6], Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, Stefano Paraboschi, *in Proc. of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Hong Kong, China, October 16-18, 2023*

4. **"Lightweight Cloud Application Sandboxing"** [3], Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, Stefano Paraboschi, *in Proc. of the 14th IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM), Naples, Italy, December 4-6, 2023*

5. **"Supporting Data Owner Control in IPFS Networks**" [2], Marco Abbadini, Michele Beretta, Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Pierangela Samarati, Matthew Rossi, *in Proc. of the IEEE International Conference on Communications (ICC), Denver, CO, June 9-13, 2024*

2. Research approach and core aspects

2.1 Introduction

Taking into account the scenario presented in Chapter 1, that points out the important and critical role played by cloud applications, we have investigated the main critical issues that characterize the most common JavaScript, TypeScript and WebAssembly runtimes. For each of them we considered the vulnerabilities that in recent years involved native code libraries used in cloud computing environments, or other critical aspects about the interaction between the runtime and the underlying system. The research has been conducted firstly analyzing both state-of-the-art industry sandboxers and proposals, proposing then a way to overcome some common critical aspects we have identified (e.g., performance and overhead impact on the system). We proposed sandboxes that are well integrated with the most used runtimes and with the security mechanisms available in the system, without the need of modifications or interventions by the developer. The sandbox must be easy to use without any specific understanding of the complex and low level security features leveraged by our solution, and it must provide a fine-grained access control over the resources of the system. To support the cloud developer in defining an easy-to-use sandbox, we decided to enforce the protection based on policies that describe which resources of the system can be accessed by the allowed processes. For this reason, our approaches are supported by a semi-automatic policy generation tool and open-source implementations that can be used by the developers to construct the access rules of the application and enforce the protection. Our proposal must provide strong guarantees of vulnerability mitigation, for this reason we leveraged kernel level Linux Security Modules (LSMs) to protect the resources to be accessed by an unauthorized process. Lastly, we paid attention to provide a sandbox that did not have excessive performance overhead by conducting extensive experimental tests comparing our solution with the most known alternatives and testing the effectiveness of our approach through different tests of CVEs mitigation.

2.2 Policy

In order to provide solutions that are as flexible and usable as possible, we decided to provide developers with tools that can automatically produce an access policy for a process. To keep the policy readable and understandable, the rules are defined within a JSON file following the *read-write-exec* convention (`RWX`) for the file system resources. The structure of the policy must be flexible and adopts a *default-deny* model, so that anything that does not appear in the list cannot be accessed. This choice permits to easily define sandboxes compliant with the *least privilege* principle. To provide a more comprehensive system protection, the policy limits also other resources in addition to the file system; indeed we have identified more than 30 recent Common Vulnerabilities and Exposures (CVEs) involving native code software that could be mitigated by restricting access to three main system resources: file system, Inter Process Communication mechanisms (IPC) and Network resources. For this reason, the research led to policies that specify also which IPC mechanisms must be granted to the application, as well as which network domains and ports can be accessed. By limiting the application access only to the necessary resources, we can mitigate the effect of a good portion of vulnerabilities involving unsafe third party dependencies.

2.2.1 Policy enforcement

To enforce the rules specified by the policy file, we take advantage of the kernel security mechanisms available in modern systems. This permits to provide a i) portable solution, since kernel-level modules are shared by most of the Linux systems (i.e. the majority of cloud system providers), ii) a performance-aware protection, since a lower lever interaction enables a more efficient control, and iii) a robust security, obtained by adopting well-known and tested solutions. In providing such protection, we paid attention that the proposed solution does not conflict with pre-existing ones (e.g., AppArmor, or SELinux LSMs). As a result, we selected three main mechanisms: Landlock LSM, Seccomp-bpf and eBPF LSM. All of these tools are *stackable* with existing ones (i.e., are fully compatible with other security measures).

Landlock LSM

Landlock LSM [122] enables unprivileged applications to restrict their right to access file system resources complying with the *least privilege* principle. Landlock supports the definition of `RWX` rulesets over a file system path. Since the rights granted by Landlock are thread-based and inherited by all the children, it can be used to

define path-based sandboxes with a file-based granularity over a specific process. Many rulesets can be combined to obtain more precise restrictions, allowing for a finer-grain policy definition. Moreover, Landlock is stackable, meaning that is fully compatible with other major Linux Security Modules.

Seccomp-bpf

Seccomp-bpf [37] is a Linux kernel mechanism that enables system calls filtering of a userspace application. Seccomp permits to reduce the kernel surface exposed to an application by reducing the potentially vulnerable system calls. Every syscall invoked by an application can be intercepted and thus allowed or denied.

eBPF

eBPF [104] is and extension of Berkeley Packet Filter (BPF). Originally BPF was introduced to provide a kernel facility to filter network packets. Extended BPF (eBPF) provides a new framework to execute programs inside the kernel, offering a large number of opportunities to monitor and control how an application interacts with the system. Specifically, eBPF allows to extend the kernel without the necessity to change its source code or load new modules. This aspect provide eBPF with great observability and tracing capabilities over the system, as well as finer grain security enforcements mechanisms and low performance overhead.

2.2.2 Main Results

The main results of this approach are described in Chapters 3, 4, 5 and 6 where we propose policy-based sandboxers for JavaScript and TypeScript-based runtimes, as well as WebAssembly runtimes. Our research is accompanied by extensive experiments to evaluate the effectiveness of our approach and the performance overhead. All our proposals prove to have a low impact on system performance after an evaluation conducted with the most common benchmarking tools. Moreover, we tested our sandboxes against different CVEs by running different proof of concepts while our protection is enabled. The results, which are better described in each chapter, prove how our protection is effective against recent CVEs affecting cloud systems and common native code dependencies. Specifically, we proved our protection against Arbitrary Code Execution (ACE), Arbitrary File Overwrite (AFI) and Local File Inclusion (LFI) CVEs.

2.3 Data protection in decentralized filesystems for cloud environments

One last aspect investigated concerns decentralized filesystems. Specifically, we have considered the InterPlanetary FileSystem (IPFS) and how it could be deployed on and complement a cloud infrastructure, so we moved our attention on how to provide data protection guarantees on resources uploaded on it. To guarantee control over the data uploaded we leveraged the cryptographic properties of the All-Or-Nothing-Transform encryption mode to provide a secure and efficient deletion of the content and an efficient way to encrypt data. The results of our research are presented in Chapter 7 that describes our Mix-IPFS architecture and functionalities.

3. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses

3.1 Introduction

JavaScript is currently one of the most popular programming languages [168]. One of its strengths is versatility, indeed, it can be used both in the front-end and in the back-end to write fully fledged web applications. JavaScript was originally meant for the browser, and its porting on the back-end by *Node.js* is not without security risks. As highlighted in previous studies [190, 189, 169, 202, 73], vulnerabilities affecting the language or the toolchain can lead to severe breaches.

A recent initiative aiming to reduce the risk coming from the execution of vulnerable or untrusted JavaScript code on the back-end is *Deno* [60], a modern, secure, open-source, cross-platform JavaScript runtime. Contrary to its well-known predecessor (i.e., Node.js), Deno was designed with security as one of its primary goals [61]. This aspect partly stems from the programming language used to implement it (i.e., Rust instead of C++), but mostly originates from its default behavior of executing JavaScript code in a completely isolated sandbox. Unless otherwise stated by the developer, Deno prevents any program from accessing the filesystem, network, environment variables, and even high-resolution time measurements. Unfortunately, although its permission model allows developers to grant fine-grained authorizations (read/write/execute privilege on a single file, permissions to connect to a known hostname, etc.), dynamic libraries and subprocesses can access system resources regardless of the permissions granted to the Deno program that spawned them, essentially invalidating the security sandbox [63].

Research has shown that third-party code accounts for a great portion of a JavaScript application codebase [189]. The 2022 State of Open Source Security [166] shows that on average open-source JavaScript projects rely on 174 third-party dependencies. The practice of reusing third-party libraries is so widespread that it led the authors of Deno to implement the *Node compatibility mode* [65], which enables the execution of Node packages in Deno. Third-party modules often depend on subprocesses [169, 89, 72]. There is a broad spectrum of programs that fall into this category, ranging from Linux utilities to handle files [139], to programs that process media (e.g., image/video conversions [138], metadata removal [137]), and many more. Since these programs *(i)* are executed outside of a sandbox, *(ii)* are

potentially exposed to unsanitized input, and *(iii)* are often written with unsafe languages, the risk of security violations is concrete. The 2022 State of Open Source Security reports that, on average, JavaScript projects are affected by 40 vulnerabilities when dependencies are taken into account. We focused on the records of the last 5 years, and have identified a sample of 15 high-severity CVEs that affect third-party software used by popular packages (averaging more than 1M downloads/week), and allow an attacker to perform privilege escalation on the host, open reverse shells, perform local file inclusion, and corrupt the filesystem.

Our contribution In this work we propose *Cage4Deno*: an extension of the Deno security functions aimed at mitigating vulnerabilities that may be introduced when subprocesses are used. Specifically, we design and implement[1] a solution targeting Linux systems, which leverages the recent *Landlock* Linux Security Module and the *Extended Berkeley Packet Filter* (eBPF), to transparently sandbox processes currently executed outside of the sandbox by Deno. The primary goal of the proposed protection mechanism is to preserve the integrity of the filesystem, and preventing access to confidential resources. While designing Cage4Deno, we paid great attention to the usability by developers: *(i)* no understanding of the kernel security features leveraged by our solution is required to use it, and *(ii)* although developers should be knowledgeable about the functionalities of the program they want to use, we do not expect them to be fully aware of its functioning under the hood. To this end, we first design a flexible and compact policy model. It consists of rules granting `read` (`R`), `write` (`W`) or `exec` (`X`) permissions on a filesystem node, which propagate towards its descendants, and `deny` rules (`D`) to block the propagation, thus reducing the number of rules and the effort of the developer. Then, we implement an auxiliary open-source tool named `dmng` to generate the least-privileged `RWX` rules required by each program. The tool can be invoked interactively by the developer via CLI, or integrated into CI/CD pipelines and run against a set of use cases, as best practice suggests. We also highlight that Cage4Deno preserves backward compatibility, meaning sandboxing of existing code, including direct or transient dependencies, can be achieved without code changes.

We demonstrate the security benefits of our solution showing how it mitigates a number of real-world vulnerabilities, which have been exploited against popular services (e.g., GitLab [51] and TikTok [95]). Finally, we perform an experimental evaluation comparing our solution to scenarios with no sandboxing, and sandboxing with state-of-the-art proposals, showing substantial performance improvements with respect to the available alternatives.

[1] `https://github.com/unibg-seclab/cage4deno`

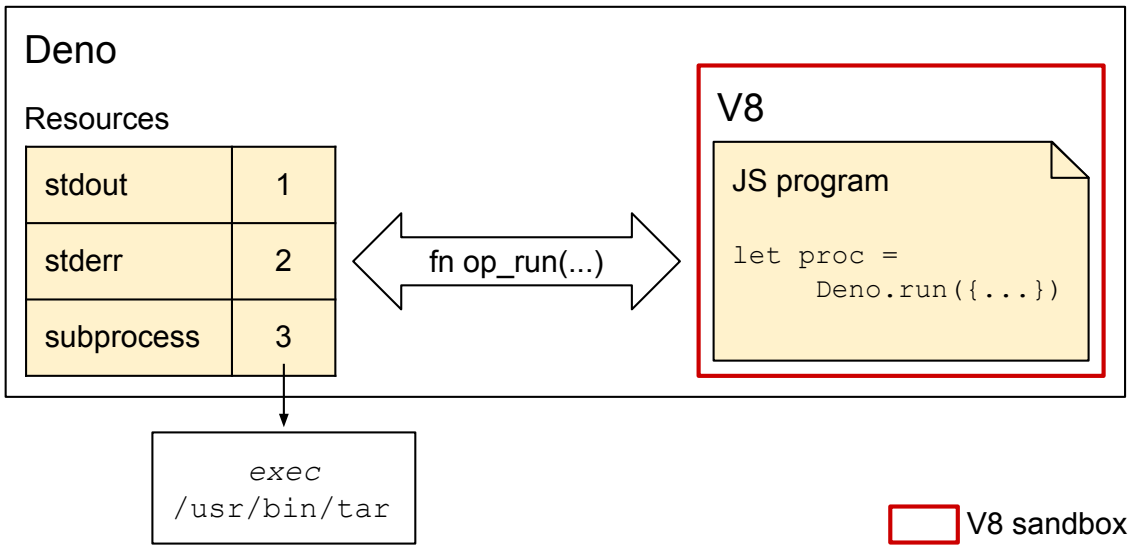


Figure 3.1: High level view of the Deno architecture

3.2 Background

This section overviews the frameworks used in our proposal.

3.2.1 Deno

Deno [60] is a runtime for JavaScript and TypeScript embedding *V8* [185], an open-source high-performance JavaScript and WebAssembly engine by Google. As already mentioned, Deno can be seen as an alternative to Node.js, with some key security features that derive from its design. In detail, it has a modular architecture organized into three main components: *(i) rusty_v8*, which defines the set of bindings to the V8's API; *(ii) deno_core*, a package built on top of *rusty_v8* implementing the abstractions required to run JavaScript (i.e., the *JsRuntime*); and *(iii) deno*, a package providing the Deno executable and the user-facing API.

Among the abstractions implemented in *deno_core* there are *ops*, native functions directly called from JavaScript that expose services not directly available in V8. These include primitives to open files, create network sockets, spawn child processes, access environment variables, etc. From a security perspective, each of the requests issued by the program running inside the V8 sandbox (i.e., the *op* calls) can be monitored by Deno and explicitly blocked or authorized based on a set of permissions. By default apps run without any permission, that is why Deno claims to be secure by default. The permission system categorizes the resources based on their type (filesystem, environment variables, network, etc.). There are two granularity levels: access to the whole resource category (e.g., all the files on the filesystem), and access to a single resource in the category (e.g., a single file). The resources

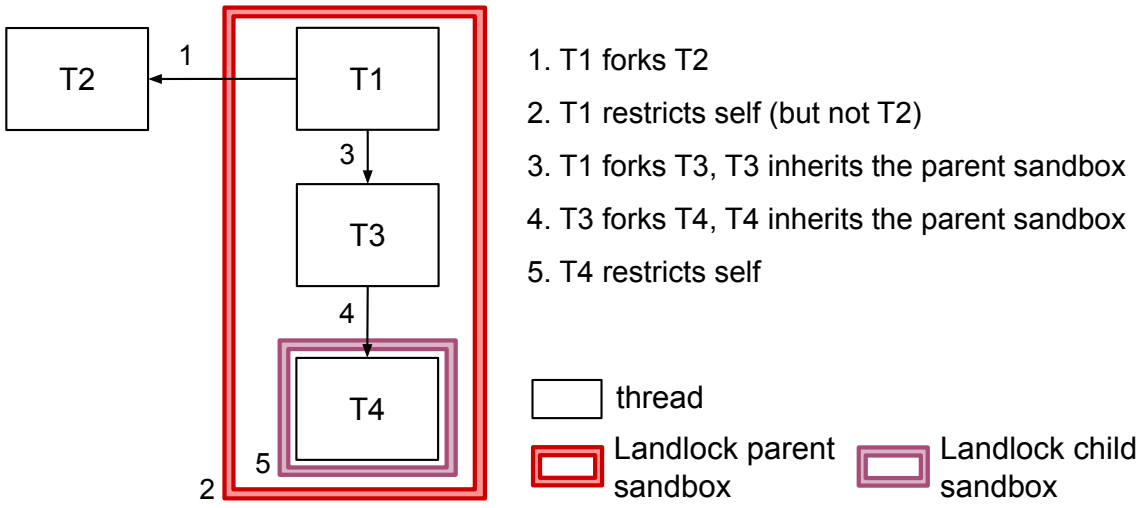


Figure 3.2: Creation and inheritance of a Landlock sandbox

accessible through *ops* are stored by *deno_core* in a resource table, and are uniquely identified by integers, similarly to the Unix concept of file descriptor.

The *ops* and resources interface abstraction gives Deno the ability to control the information flow between the system and the sandboxed app. This is a big step forward compared to Node.js in terms of security. Unfortunately, this level of protection applies only to the components written in JavaScript. When the app executes a program on the host leveraging for example the `Deno.run()` function, the request is handled by Deno spawning a new subprocess. As depicted in the high level view of the Deno architecture shown in Figure 3.1, this process runs unconstrained on the host (without sandboxing). This is a limitation of the current security model as, in case of a vulnerability, the host can be compromised [61].

3.2.2 Landlock LSM

The *Landlock* Linux Security Module (LSM) aims to provide developers with an unprivileged solution to implement application sandboxing. The availability of Landlock is expected to help mitigate the security impact of bugs and unintended or malicious behavior of user-space applications. Also, it is fully composable with other LSMs like SELinux, AppArmor and Yama.

Currently, Landlock focuses on the protection of the filesystem. In detail, Landlock classifies kernel resources as *objects*, permitting *actions* over them based on *rulesets*. As an example, a ruleset can grant the thread `read` and `exec` access to objects stored under `/tmp`. Rulesets are created using the `landlock_create_ruleset()` and `landlock_add_rule()` system calls, and subsequently activated with `landlock_restrict_self()`. Rules are *inherited* by the chil-

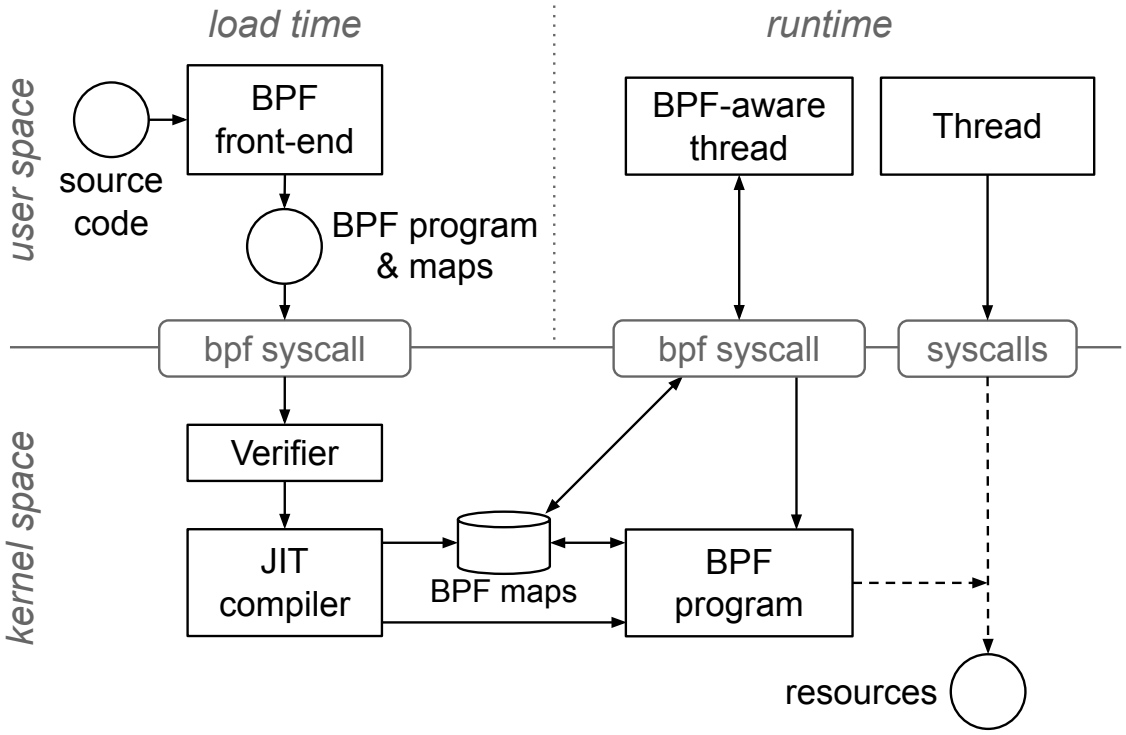


Figure 3.3: Overview of the BPF architecture

dren created after sandbox activation, and the actions available can only be further restricted. The process is detailed in Figure 3.2.

Although Landlock permits to sandbox a thread ensuring low performance overhead, there are a few limitations. The most relevant to this paper are: *(i)* it is not possible to perform any filesystem topology modification (i.e., arbitrary mounts), and *(ii)* there is no support for a deny listing approach during policy creation.

3.2.3 eBPF

The *Extended Berkeley Packet Filter* (eBPF, henceforth referred to as BPF) [91, 104] enables the execution of programs within the operating system kernel. The programs, which are loaded at runtime, extend the kernel capabilities, without requiring the developer to change the kernel source code, nor loading new kernel modules.

BPF programs are non-preemptable event-driven programs that are run when a certain user- or kernel-space hook point is reached. Pre-defined hooks include system calls, tracepoints, network events, function entry/exit, etc. BPF programs are usually written using a BPF front-end, which provides an abstraction to write programs in a high-level language, specify attachment points, declare data structures, and compile the source code into BPF bytecode. A few BPF front-ends exist; we use *libbpf* [111, 14] as it elegantly addresses portability following a Compile Once – Run Everywhere approach [14]. After a BPF program is compiled to bytecode, it

can be loaded into the Linux kernel via the `bpf()` system call. This is a privileged operation that requires the `CAP_BPF` Linux capability, and optionally `CAP_PERFMON` (to load tracing-related programs) and `CAP_NET_ADMIN` (to load networking-related programs) [1]. As the program is loaded into the kernel, it is subject to the *verification* and *JIT compilation* phases. The verification phase ensures the program is safe and does not introduce reliability issues (e.g., termination is guaranteed, memory requirements are satisfied), while the Just-In-Time (JIT) compilation translates the generic bytecode to architecture-specific optimized code. Once completed, the program is loaded into the kernel and attached to the selected hook. The architecture of BPF and the loading process are shown in Figure 3.3.

BPF programs cannot call arbitrary kernel functions and cannot freely share the information collected with user space. Instead, they rely on *helper functions*, a stable API implemented by the kernel that is used for tasks like manipulating network packets, inspecting kernel data structures, etc. Among the most frequently used helpers, there are functions to read and write *maps*. Basically, maps are data structures that permit to keep a state between different invocations of BPF programs, and share data with user-space applications.

BPF's capabilities have been further extended in 2020 with the addition of the Kernel Runtime Security Instrumentation (KRSI) [50], also known as BPF LSM. This feature permits to attach BPF programs to LSM hooks, and thus enforce access control.

3.3 Cage4Deno

This Section gives an overview of *Cage4Deno*, clarifying the threat model. We also introduce our design objectives (Section 3.3.3).

3.3.1 Overview

Cage4Deno aims to provide a set of sandboxing functions to strengthen the Deno security model. The proposal arises from the need to provide isolation for subprocesses spawned with the `Deno.run()` and `Deno.Command.spawn()` functions. As mentioned in Section 3.1, the practice of executing subprocesses is heavily used by developers. Unfortunately, any subprocess that executes this way falls outside of the Deno security model, essentially invalidating the sandbox (see Section 3.2.1). Indeed, an attacker that successfully exploits a security flaw affecting the utility run by a subprocess can perform privilege escalation on the host, open reverse shells, perform local file inclusion, corrupt the filesystem, etc.

Cage4Deno extends Deno giving developers the ability to constrain the execution of subprocesses. To do that, the developer associates each subprocess with a policy file, listing a set of rules. Rules are straightforward, each of them granting `read` (`R`), `write` (`W`) or `exec` (`X`) permissions on a filesystem node. To reduce the number of rules in the policy, the set of `RWX` permissions is extended with `deny` (`D`). This enables permissions granted on a filesystem node to propagate towards its descendants, and to block the propagation with the use of deny rules. The permissions granted by the developer are then automatically assigned to the subprocess at runtime. This is achieved by the *sandboxer*, a new module we added to Deno that leverages the Landlock LSM and the BPF framework (operating in stacking mode [203]) to implement the sandbox at kernel level. This ensures security checks are not bypassable, however it implies that our solution only works on Linux-based systems with these features enabled. Nowadays, Landlock LSM is already available in most systems, BPF LSM not yet. However, bleeding edge distributions, like Arch Linux, are starting to release with it enabled by default.[2] Compared to other well-known general-purpose sandboxing solutions like Minijail [87], and Google Sandbox2 [88], Cage4Deno does not require the developer to know anything about the advanced security features offered by the kernel to confine a process. All the developer has to understand is the straightforward `RWX+D` permission model. Moreover, Cage4Deno does not require the developer to understand the internal logic of the utility executed by the subprocess, all she needs to know are its input and output (provided by either absolute or relative path). An overview of our proposal is shown in Figure 3.4.

3.3.2 Threat model

Similarly to other research proposals [170, 190, 57], Cage4Deno focuses on the runtime compromise of possibly buggy or vulnerable utilities (i.e., binaries or libraries), and does not target actively malicious ones. Therefore, we do not regard the developers of the utilities as malicious actors, but nevertheless, they may inadvertently introduce vulnerabilities into their code. In this setting, attackers may control arguments of the utilities by passing malicious payloads through web interfaces or programmatic APIs. Prominent examples include utilities that offer manipulation capabilities of multimedia files (e.g., ExifTool, FFmpeg, GraphicsMagick, ImageMagick), or object de-compression (e.g., GNU Tar). These programs are usually written in memory-unsafe languages, such as C/C++, and due to their size and complexity are often the source of vulnerabilities [132]. The constant and extensive exposure to untrusted inputs in web applications may enable the attacker to trigger these vul-

[2]Enabling BPF LSM on systems not supporting it by default requires replacing the system kernel with the same release of the kernel, but with BPF LSM available.

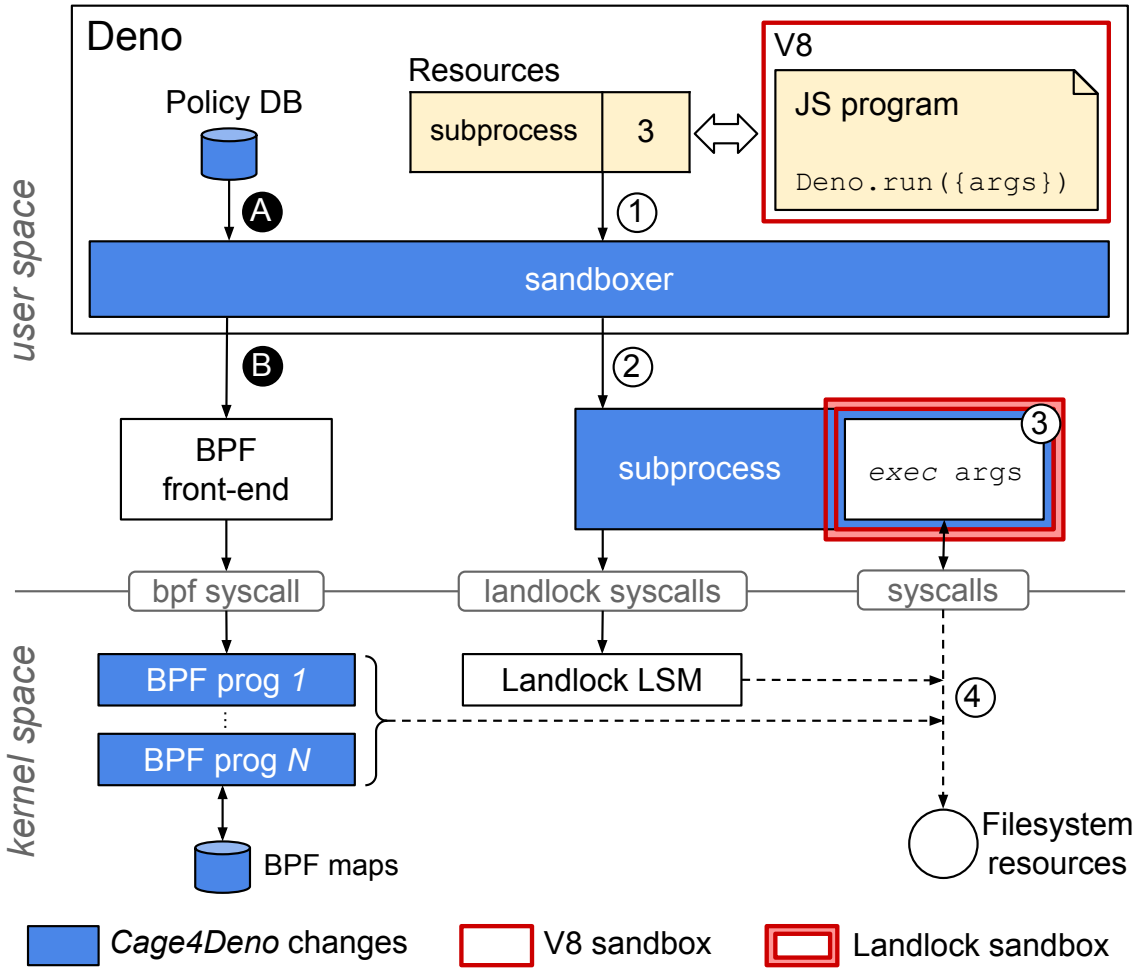


Figure 3.4: Overview of Cage4Deno implementation. Initial setup: *(A)* read the policy and *(B)* load BPF programs and write maps according to it. On subprocess request: *(1)* intercept subprocess creation, *(2)* create a Landlock-sandboxed subprocess with BPF deny, and *(3)* execute the utility. All the requests issued by the subprocess are restricted according to the policy *(4)*

nerabilities, leading to memory corruption issues. Arbitrary file reading or writing, local file inclusion and remote code execution are only a few of the possible risks associated with this kind of attack vector. In addition to that, package managers for languages commonly used in web development (e.g., `npm`, `pip`) do not enforce any kind of permission system. Thus, attackers can exploit vulnerabilities introduced by direct or indirect dependencies installed when 3rd party modules are imported by the application. Previous studies [123, 204, 56] have reported the risks associated with the propagation of vulnerabilities coming from dependencies in various software ecosystems. Usually, the exploitation of this kind of vulnerability leads to breaches that undermine confidentiality and integrity of data and code that reside outside the utility under attack. The goal of Cage4Deno is preventing a compro-

mised utility running inside a Deno subprocess to violate access confidentiality and integrity of the filesystem. Notice that preventing integrity violation of the filesystem means, first and foremost, preserving the integrity of the web application code, configurations and data.

3.3.3 Design objectives

Several objectives need to be fulfilled by our proposal to ensure security, efficiency and ease of use.

O1. **Integration with existing solutions (Compatibility)**: The proposal must be compatible with the current Deno architecture, and should be considered as an opt-in feature by the developer (i.e., backward compatible). Also, it has to be stackable with other security mechanisms already active on the host, like SELinux, AppArmor and Yama.

O2. **Ease of use (Transparency)**: To be aligned with the needs of the web development scenario, the solution must be simple and easy to use. No specific understanding of the advanced security features leveraged by our solution, and of the underlying operating system, must be required to use it.

O3. **Fine-grained access control (Flexibility)**: Unlike other security features that create a separated view of global resources (i.e., *Linux namespaces*), the solution must enable the developer to access the whole filesystem, granting access with file-level granularity rather than volume-level. Also, the developer must be able to flexibly switch between different policies, without requiring a host reboot when a new policy needs to be loaded.

O4. **Automatic generation of policies (Usability)**: The proposal must provide tools to support zero-effort policy creation. With the exception of its input and output, no specific understanding of each utility should be required to run it successfully.

O5. **Mitigation of vulnerabilities (Security)**: The proposal must prove effective in addressing the threat model described in Section 3.3.2 and mitigate CVEs affecting common Linux utilities.

O6. **Low overhead (Performance)**: The overhead introduced with the additional security features must be compatible with the requirements of Web applications, which tend to prioritize short response time. It should be lower than the one introduced by state-of-the-art general-purpose sandboxing solutions, such as Minijail and Google Sandbox2.

3.4 Design and Implementation

This Section illustrates the design and implementation of Cage4Deno. We initially describe the interface used by the developer to input the policy, then we detail the changes introduced to support `read`, `write`, `exec` and `deny` permissions. A minimal program executing the `tar` utility in a subprocess is used as running example. The approach used to automatically generate the policy to confine `tar` is explained in Section 3.5, while additional BPF implementation details that focus on performance improvement are discussed in Appendix A.

3.4.1 Policy and interface

In the current Deno architecture, attributes and permissions associated with a program are always specified prior to execution. The developer provides this information through the Deno command line interface (CLI) using *runtime flags*. Runtime flags of the `deno run`, which precede the program name in the argument list, are immediately parsed and added to the global state before the JS program is executed. This design philosophy permits to directly address access requests issued by the application at runtime and, in case no permission is available, the program is promptly terminated. As an example, consider the program shown in Listing 3.1. Its purpose is simply to execute the `tar` utility to extract the compressed archive `input.tgz` to the `output` directory. Since to execute `tar` it leverages a subprocess, the argument `--allow-run=tar` must be provided, as the lack of such a permission would lead to the operation being prohibited.

As explained in the overview (Section 3.3.1), Cage4Deno relies on the ability to attach a policy to the program. To be compliant with the current design of Deno, we extended the global state adding the `--policy-file` runtime flag. As the name suggests, it receives as input a policy file. The policy file uses a JSON format, which is easy to edit and parse, and well-known by web developers. The file contains an array of policies identified by a `policy_name`. Each policy includes four arrays, `read`, `write`, `exec` and `deny`, listing a set of filesystem entries. As shown in Figure 3.4, the life cycle of Cage4Deno can be divided into two phases: load time and enforcing time. When Cage4Deno starts, the content of the policy file is read by the *sandboxer* (Ⓐ); to avoid repeating this step each time a new subprocess is spawned, the rule sets are stored in the permission state of Deno. Once all policies have been parsed, if any of them contains negative rules, the *sandboxer* loads the set of BPF programs needed to enforce deny, and a map for each policy containing the corresponding prohibited paths (Ⓑ). Due to this additional step, this is the only part of the execution in which Cage4Deno requires additional privileges compared to Deno. These privileges

Listing 3.1: An example of child process in Deno

```
1  let a=Deno.run({cmd: ["tar", "xzf", "input.tgz", "-C", "
       output"]});
2  await a.status();
```

Listing 3.2: Restricting a child process using `policyId`

```
1  let a=Deno.run({cmd: ["tar", "xzf","input.tgz", "-C", "
       output"], policyId:"tarPolicy"});
2  await a.status();
```

consist of the set of Linux capabilities necessary to load BPF components. To avoid any additional interaction the capabilities are configured as file capabilities. Once every BPF component has been loaded, we drop the capabilities, so to avoid running with higher privileges with respect to Deno at runtime.

After the initial setup, Cage4Deno is responsible of enforcing the policies provided by the developer. The *sandboxer* intercepts subprocess creation requests issued by the JS application at runtime (①) and, according to the policies available, it restricts the permissions associated with the subprocess (②) before the *exec* of the command provided to the `Deno.run()` is performed (③). During the subprocess (or any of its children) lifetime, file interactions are controlled by Landlock and our BPF programs to make sure access to the requested path is granted according to the policy definition provided by the developer (④). Two options are available to retrieve proper policy from the list: *(i)* automatically select the one with policy name matching the utility to be run in the subprocess (e.g., `tar` in Listing 3.1), and *(ii)* using the policy directly specified by the developer in the JS code using the newly introduced `policyId` option as shown in Listing 3.2. In both cases, if no policy is found, we fall back to the default Deno permission model. Option *(i)* allows the developer to run the subprocess without modifications to the JS program, while option *(ii)* gives more flexibility when multiple policy profiles for the same utility are available. This interface is straightforward and is perfectly aligned with the current Deno security model and architecture (Objective **O1**). Moreover, it does not require the developer to understand how the *sandboxer* leverages the kernel security features to add the restrictions, while granting the developer direct observability of the security boundaries put in place (Objective **O2**).

3.4.2 Support to `RWX` rules

Deno permits to set filesystem-related permissions through the `allow-read`, `allow-write` and `allow-run` runtime flags. As explained in the background (Sec-

Listing 3.3: `RWX+D` permissions associated with `tar`

```
{
  "policies": [
    {
      "policy_name": "tarPolicy",
      "read": [
        "/usr/bin/tar",
        "/home/user/input.tgz"
      ],
      "write": ["/home/user/output"],
      "exec":  ["/usr/bin/tar"],
      "deny":  ["/home/user/output/misc"]
    }
  ]
}
```

tion 3.2.1), the flags can be used to configure access to the whole filesystem, or can be refined specifying a list of comma separated filesystem entries. Similarly, we expect the developer to detail the `read`, `write` and `exec` permissions inside the JSON policy file. Listing 3.3 exemplifies the `RWX` permissions associated with the `tar` example.

The support for `RWX` permissions was introduced to Deno changing the implementation of *deno_core*. Internally, when a program performs a call to the `Deno.run()` and `Deno.Command.spawn()` functions, the bindings defined in *rusty_v8* are used to translate the request into an *op_run*. The operation is subsequently sent to the JS runtime, which is responsible to manage the request. In the case of a subprocess creation, this is done leveraging the asynchronous Tokio [71] runtime to configure an instance of `Command`, a process builder providing fine-grained control over how the process should be spawned. To apply the correct policy, we modified process creation, scheduling a closure (Rust equivalent of a C++ lambda expression) to be run just before the `exec` function is invoked. The closure leverages our *sandboxer* to create a new Landlock ruleset consisting of the `RWX` permissions found in the policy, and then invokes the `landlock_restrict_self()` syscall to apply it. Thanks to the policy inheritance property of Landlock, the policy attached to the process also restricts its children (as explained in Section 3.2.2). Should any of them try to access a filesystem location not listed in the policy, the request is denied.

From a software development perspective, `RWX` permission rules are easy to understand and immediate to write. Contrary to the use of other frameworks such as the combination of Linux Namespaces [29] and Control Groups [85], `RWX` rules also permit to grant access with file-level granularity on the whole filesystem, and not only on single volumes (Objective **O3**).

3.4.3 Support to `deny` rules

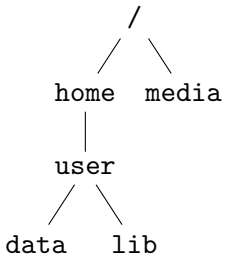
Deny rules ensure a process has no access privileges over a file or a directory. Its introduction significantly reduces the overhead of the developer writing the policy. Unfortunately, the current *inode-based* design of Landlock does not provide support for it [158]. Given the low overhead associated with BPF (Objective **O6**), its fine-grained access control capabilities (Objective **O3**) and its compatibility with other LSM security mechanisms including Landlock (Objective **O1**), we identified BPF as the ideal candidate to implement the support to deny rules. However, loading BPF programs and maps is a privileged operation, so Cage4Deno needs to execute a preliminary initialization phase with additional permissions (as highlighted in Section 3.4.1).

This Section details the work we did to support deny rules. First, we present the problem of efficiently supporting access control decisions given a sequence of deny rules, then we explain how to translate the approach into BPF programs, making clear how the *sandboxer* enforces access control at runtime.

Deny listing approach

Similarly to what happens for `RWX` rules, we expect the developer to list the `deny` rules into the JSON policy file (see Listing 3.3). Based on the list provided, the *sandboxer* instruments the kernel so that, upon receiving an access request from a sandboxed process, the kernel is able to allow or deny it. For instance, assume the developer lists in the policy the `deny` rules `/home/user/data`, `/home/user/lib` and `/media`. When the sandboxed process issues an `open` to `/home/user/file` the request is allowed, but when the process tries to access `/home/user/data` (or any of its content) the request is denied. So, given a path request issued by the user, whether it matches exactly one of the rules in the deny list, or a deny rule is a prefix of the requested path, the access request is blocked. A classic solution to solve this problem efficiently can be implemented using a prefix tree (or trie). The idea is to split each path into a sequence of substrings using `/` as delimiter, treating the substrings as single character prefixes. The prefixes are then used to build a prefix tree as shown in Table 3.1a. At runtime, access requests are simply evaluated traversing the trie. If a leaf node is visited, we can conclude that a deny rule was hit, and the access request must be denied. The time complexity of a trie traversal depends on its maximum height. Given N the length of the longest deny rule D_{r_i} in the policy,[3] the maximum depth of the trie is $N/2$ (in the case of a path structured as $[/c]^+$). When a hashmap is used to implement the trie, each lookup (used to

[3]The maximum path length is bounded to 4096 chars in `linux/limits.h`



(a)

		BPF Map	
r	**Trie prefix p_i**	**Hash**	**isLeaf**
1	`/`	$hash(p_1)$	0
2	`/home`	$hash(p_2)$	0
3	`/home/user`	$hash(p_3)$	0
4	`/home/user/data`	$hash(p_4)$	1
5	`/home/user/lib`	$hash(p_5)$	1
6	`/media`	$hash(p_6)$	1

(b)

Table 3.1: Deny prefix tree (a) and BPF Map_{policy} (b)

jump from the parent to the child) takes $O(1)$ time, thus traversing the trie takes $O(N)$ time (worst case upper bound).

Implementation using BPF maps

The trie-based approach presented above cannot be translated directly to a BPF program. Indeed, there are some constraints a BPF program must satisfy to be executed by the kernel (see Section 3.2.3). The most important, in our case, is related to the use of memory. The current implementation of BPF does not provide support for multi-level map-in-map [120] structures required to implement the trie. Hence, we decided to translate the trie into a single-level hashmap Map_{policy} storing all the prefixes represented in the trie, as shown in Table 3.1b. The hash of each prefix is used as lookup key, while the value is a boolean to indicate whether the prefix is a leaf node in the trie. This representation permits to answer access queries with the same efficiency of the high-level approach. In fact, the trie traversal is easily replaced by a sequence of $O(N)$ lookups, each taking $O(1)$ time when a hash function is used to process the string representing the access path. Storing in the map also the prefixes that are not leaves in the trie increases the size of the map up to $O(M \cdot N)$ given a list of M deny rules, yet, it permits to reduce the query time for all the path prefixes that are not contained in the trie, as the lookup sequence terminates when a key error (i.e., a *miss*) occurs.

This design strategy relies on the ability to convert strings to integers using a cryptographic hash function. Unfortunately, such function is not currently available in BPF. In Appendix A we present this aspect in detail, explaining how we adapted the design to achieve the best trade-off between query response time and map size using an incremental hash function. Other details, such as explicit collision handling, are also discussed there, as they complement the design of our proposal.

Thread lifecycle hooks
uprobe/attach_policy
lsm/task_alloc
tp_btf/sched_process_fork
tp_btf/sched_process_exit

(a)

Access control hooks
lsm/path_mknod
lsm/path_mkdir
lsm/path_link
lsm/path_symlink
lsm/file_open
lsm/path_rename
lsm/path_rmdir
lsm/path_unlink

(b)

Table 3.2: Hooks and tracepoints monitored by Cage4Deno

BPF policy attachment

To enforce access control at runtime, the *sandboxer* must be able to retrieve the Map_{policy} according to the argument provided by the developer to the `Deno.run()`, and then to *attach* it to the sandboxed process. The attachment of the proper policy to the sandboxed process is delicate, and requires a set of programs to be loaded by Cage4Deno into the kernel alongside the maps. To reduce the runtime overhead, BPF programs, together with the BPF policy maps associated with all the policies listed in the JSON policy file, are loaded from user space to kernel space by Cage4Deno at startup time. These operations are carried out using *libbpf* [111]. The programs are separated in two categories: *(i)* programs that are needed to trace the lifecycle of a sandboxed process, and *(ii)* programs to evaluate the access queries it performs (according to the strategy described in Section 3.4.3).

The first group of BPF programs, namely the ones used to trace the process lifecycle, are responsible for maintaining in memory the map Map_{task} of processes running on the system that are subject to the restrictions imposed by Cage4Deno. Specifically, Map_{task} associates each thread identifier to the proper Map_{policy}. The entries in Map_{task} are updated when one of the hooks listed in Table 3.2a is reached. For instance, when a traced process issues the `clone` syscall, the tracepoint `tp_btf/sched_process_fork` is reached, the related BPF tracing program executed, and the new child process, inheriting the policy of the parent, is added to Map_{task}.

The second group of BPF programs, the ones that are associated with the evaluation of the access query, are executed when any of the hooks listed in Table 3.2b is reached. The role of these programs is to check whether the current process (i.e., the process issuing the access request) belongs to the Map_{task}, and accordingly to perform the sequence of lookups in Map_{policy} to allow or deny the request.

3.5 Policy generation

A key aspect of our proposal is usability by developers. The straightforward `RWX+D` permission model is functional to achieve ease of use (Objective **O2**), however, we believe that it is also important to offer to the developers tools to support the generation of policies (Objective **O4**). Indeed, each policy can be seen as a template comprising the dependencies to run a program that can be further extended with information related to the input, the output, and the restrictions the program may be subject to. This section details the work we did to support the process, explaining how our solution can be leveraged to generate the `RWX` rules shown in Listing 3.3.

The retrieval of the dependencies used by a program is a recurring problem. Just to mention a few, it occurs in the Google Sandboxed API project, where the Sandbox2 sandboxing utility [88] leverages `ldd` (acronym for *List Dynamic Dependencies*) to retrieve the dependencies used by programs available on the host as *Executable and Linkable Format* (ELF) files; or in SlimToolkit [164], where a containerized service is run against a test suite to automatically create Seccomp [37] and AppArmor [38] security profiles matching the least-privilege principle. In Cage4Deno, we adopt a similar approach, which retrieves the files a program should be able to read, write, or execute to work as intended. Moreover, we need to provide the developer with functions to manage multiple policies simultaneously (updating the lists of dependencies and denials), and to serialize them into the JSON format as shown in Listing 3.3.

The solution we adopted was to develop `dmng`, a CLI tool written in Go (∼3.5k lines of code), which ships within Cage4Deno, allowing the developer to automatically generate `RWX rules`, and interactively refine, the policy. The utility supports the retrieval of program dependencies that are available on the host as ELF files, and those that are spawned by dedicated POSIX or shell wrappers. It can also be used with programs loading dynamic modules at runtime (e.g., media processors loading custom encoders), or programs executing several binaries. To support these use cases, `dmng` relies on both `ldd` and `strace`. The former was preferred to `objdump` and `readelf` since it can be used to recover the so called *transient* dependencies of programs available as ELF files; while the latter is a diagnostic, debugging and instructional user-space utility for Linux that is used to trace programs at runtime.

Similarly to the approach presented in SandTrap [8], the developer can use `dmng` to run a test suite against a program (or a script). This apparoach allows the developer to reuse existing tests to generate `RWX` rules that satisfy multiple execution paths of the binary. By using `strace`, `dmng` monitors the interactions between the threads spawned by the tested utility and the kernel for a certain amount of time (usually less than $2s$). In this time frame, the file-related syscalls issued by the set

Listing 3.4: Semi-automatic generation of `tar` policy (Listing 3.3)

```
1  # Set the active policy_name for a program
2  dmng -c tar --setcontext tarPolicy
3  # Add the dependencies of tar to the template
4  dmng -c tar -t -s
5  # Add the input to the template
6  dmng -c tar -a input.tgz -p r--
7  # Add the output to the template
8  dmng -c tar -a output -p -w-
9  # Add the denials
10 dmng -c tar -a output/misc -d
11 # Serialize the entries into a JSON file
12 dmng --serialize tar_policy_file
```

of threads, along with their arguments, are captured and saved to a log file. The log is then used by `dmng` to generate the `RWX` rules accordingly. The list of syscalls monitored comprises the ones wrapped by the standard system functions `execve()`, `open()`, `create()`, `link()`, `mkdir()`. The `dmng` tool exposes to the developer many functions to inspect, add, update or remove the entries associated with the policy template. Specifically, it allows to add deny rules, which generation cannot be automated. It also implements heuristics to reduce the number of `RWX` rules, thus improving their readability, and facilitating policy auditing, when explicitly requested by the developer. This inherently comes with the downside of producing coarser permissions. The process is called *permission pruning* and it is based on common security practices, like the identification of write-or-exec (`W^X`) memory regions, but also contextual information about the specific region of the filesystem being considered (e.g., there are no practical advantages in assigning fine-grained permissions under `/usr/share/fonts/`, while it is certainly useful to do so under `/home/user/Documents`). For this, the paths are arranged into a Trie, and then the above heuristics are used to incrementally prune leaf nodes untill the developer's target number of rules is met.

For each of the programs to be restricted, a sequence of commands may be input by the developer to synthesize the final policy (state is persisted between tests using a SQLite database). Since the developer can work with distinct programs and many policies simultaneously, a cache is used to store the active context (i.e., the current policy) of each program (line 2 in Listing 3.4). Then, each context is customized adding the dependencies as previously described. For instance, in line 4 `dmng` collects all the dependencies required by `tar` using dynamic tracing, and in line 10 the developer manually adds the deny rule. After all the policy contexts have been configured, the policy is serialized into the JSON format as expected

by Cage4Deno (line 12). Appendix B reports the complete `tar` policy produced following the instructions in Listing 3.4.

In this paper we employ dynamic analysis for the automatic generation of policies, however this is widely known for producing results whose completeness depends on the test suite coverage [193]. To address this limitation, the current approach could be complemented by either source or binary code analysis. Relevant works (e.g., [57]) use static analysis to pinpoint the system calls, however they do not keep track of parameter values. On the other hand, *AutoArmor* [109] uses static taint analysis to perform semantics-aided program slicing from network API invocations, and then uses these slices to extract access control attributes. The approach also looks promising for the generation of file system policies.

3.6 Experiments

In this Section we present the experimental evaluation of Cage4Deno. The evaluation considers two aspects of our solution: exploit mitigation (Section 3.6.1), and performance overhead (Section 3.6.2). To perform our evaluation we used the following test environment: an Ubuntu 22.04 LTS server powered by an AMD Ryzen 3900X CPU with 12 cores (24 threads), 64 GB RAM, 2 TB SSD. Then, we replaced the pre-installed Linux Kernel v5.15 with the same release of the kernel, but with BPF LSM available (Landlock LSM is enabled by default).

3.6.1 Exploit mitigation

The primary goal of this Section is to demonstrate the capability of Cage4Deno to mitigate real CVEs (Objective **O5**). Focusing on the records of the last 5 years, we selected a sample of 15 vulnerabilities affecting common web application utilities, such as ExifTool, FFmpeg, Git, GNU Tar, GraphicsMagick, Ghostscript, ImageMagick, OpenSSL, Pip, UnRAR, and Unzip. The CVEs we considered are classified into Remote Code Execution (RCE), Local File Read (LFR), and Arbitrary File Overwrite (AFO). Their details are reported in Table 3.3. These vulnerabilities have been selected as they represent examples of 0-click exploits, they can lead to severe security breaches, and they target utilities extensively used by web applications. In general, similar considerations extend to a broader set of CVEs.

In our analysis we reproduced each of the vulnerabilities adapting public Proofs of Concepts. For each of them, we provide a single-click test showcasing the sandboxing capabilities added to Deno. In particular, we show that: *(i)* the attacker can successfully exploit the vulnerability when Deno is used (despite its permission model being in place), and *(ii)* the attack is unsuccessful when the sandboxing func-

CVE ID	Utility	Use case
	Local File Read (LFR)	
CVE-2016-1897	FFmpeg v3.2.5	Video processing
CVE-2016-1898	FFmpeg v3.2.5	Video processing
CVE-2019-12921	GraphicsMagick v1.3.31	Image processing
	Arbitrary File Overwrite (AFO)	
CVE-2016-6321	GNU Tar v1.29	Archive decompression
CVE-2019-20916	Pip v19.0.3	Dependency fetch
CVE-2022-30333	UnRAR v6.11	Archive decompression
	Remote Code Execution (RCE)	
CVE-2016–3714	ImageMagick v6.9.2-10	Image processing
CVE-2020-29599	ImageMagick v7.0.10-36	Image processing
CVE-2021-3781	Ghostscript v9.54.0	PDF processing
CVE-2021-21300	Git v2.30.0	Clone repository
CVE-2021-22204	ExifTool v12.23	Image processing
CVE-2022-0529	Unzip v6.0-25	Archive decompression
CVE-2022-0530	Unzip v6.0-25	Archive decompression
CVE-2022-1292	OpenSSL v3.0.2	Certificate verification
CVE-2022-2566	FFmpeg v5.1	Image processing

Table 3.3: Sample of CVEs mitigated by Cage4Deno. Despite being discovered in 2016, CVEs 1897, 1898, and 6321 have seen recent exploitation in 2018 [167] and 2021 [95]

tions offered by Cage4Deno are used. The only difference between case *(i)* and *(ii)* is that a policy is provided with the `--policy-file` argument. This aspect testifies how simple it is to benefit from the sandboxing functions we have introduced (Objective **O2**). The policy files used to restrict each utility have been automatically generated with the `dmng` tool described in Section 3.5. Table 3.4 reports the number of rules generated for the selected list of utilities without applying any form of policy minimization (i.e., pruning). As shown in the table, the utilities require less than 115 permissions to work as intended, with `pip` requiring the largest set.

From a security standpoint, it is worth noting that Cage4Deno can block the attacks at multiple levels. For instance, in CVE-2020-29599, in which a crafted picture is sent to ImageMagick to execute a target command (e.g., `id`), Cage4Deno does not allow `RX` access to `/usr/bin`, blocking `/usr/bin/echo` first (which is used to inject `id` in a subshell), and then the target command itself (i.e., `usr/bin/id`). In this case, the denial is due to the Landlock sandbox allowing access solely to `/usr/bin/convert` (i.e., the ImageMagick binary). Instead, when we consider CVE-2016-6321, in which a decompression of an archive permits the attacker to overwrite

Utility	**#rules**	**Deno [ms]**	**Cage4Deno [ms]**
`cat`	9	3.05±0.23	3.81±0.25
GraphicsMagick	81	10.16±1.02	12.16±1.12
UnRAR	25	13.86±1.97	15.84±2.71
ImageMagick	17	17.49±2.14	18.74±2.26
Unzip	15	20.90±3.95	22.66±3.62
OpenSSL	17	27.80±4.93	30.10±7.50
Git	26	66.52±4.75	72.46±5.22
ExifTool	38	109.20±6.67	112.88±4.25
GNU Tar	14	114.52±7.21	125.48±6.89
FFmpeg	12	321.50±9.55	336.70±9.78
Ghostscript	20	449.96±18.19	455.66±21.37
Pip	115	3022.52±20.55	3203.32±20.84

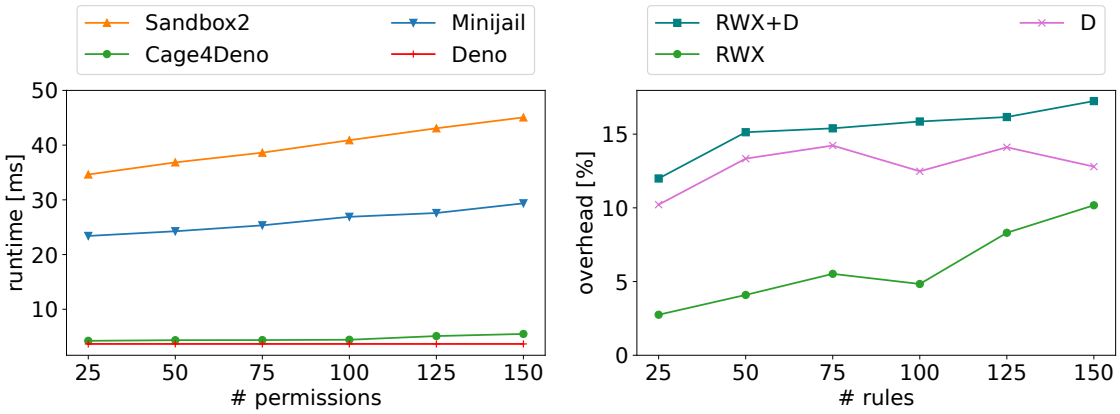
Table 3.4: Preliminary test showing the execution time of utilities reported in Table 3.3 over 500 runs (as $\mu \pm \sigma$)

a target file, we show how deny rules can be used to prevent filesystem corruption. As a final note, we highlight the ability to simultaneously use distinct policies for a single utility. To give an example, in CVE-2021-21300 several distinct policies can be assigned to `pip`. By doing so, the developer can select a dedicated policy based on the Python virtual environment currently in use.

Popular packages affected by the aforementioned vulnerabilities can be found in both `deno.land/x` and `npm` package archives. Among them, we selected `astrodon` and `fast_forward` from `deno.land/x`, `fluent-ffmpeg` and `gm` from `npm` (executed in Node compatibility mode). We confirmed that the vulnerable versions of the binaries are still exploitable through the mediation of third-party modules, and Cage4Deno proves to be effective in their mitigation without code modification to the application and its dependencies.

3.6.2 Performance evaluation

A requirement of our proposal is that it must not introduce a large runtime overhead compared to the scenario in which the developer relies solely on the basic functions offered by Deno (Objective **O6**). This is fundamental, as any additional delay could negatively affect the end-user experience or increase the cost to host the application. To investigate this aspect, we performed tests comparing the execution time of an application using vanilla Deno with respect to the same application subject to access restrictions with Cage4Deno. A second interesting aspect to consider in the evaluation is the overhead introduced by our proposal compared to general-purpose sandboxing solutions proposed by the industry. Finally, we evaluate how our solution



(a) Server execution time **(b) Client perceived overhead**

Figure 3.5: Deterioration of overhead varying the policy size

scales with the numer of rules in the policy.

Runtime overhead

In our analysis we expected the runtime overhead to vary according to the size of the policy and the number of filesystem requests performed by the sandboxed utility. We therefore conducted a preliminary test involving the utilities affected by the CVEs detailed in Section 3.6.1. For each of them, we measured the execution time of vanilla Deno and Cage4Deno. The results of the test are reported in Table 3.4. The data clearly show that the largest overhead is associated with the larger policies and the shorter execution time. To further highlight this aspect, we imagined the worst case scenario in which a simple binary is used to perform a single access to the filesystem. As shown in row 1 of Table 3.4, printing the content of an empty file using the `cat` binary is associated with the greatest overhead, roughly 25% of the execution time. `cat` is characterized by a minimal execution time, and requires only 9 rules in the policy to work as intended, thus proving to be the best candidate to highlight the overhead introduced by Cage4Deno.

To evaluate how our solution scales with the increase of rules in the policy, we set up a second test. We again used `cat`, but this time we added to its minimum set of permissions a varying number of `RWX` rules, ranging from 25 to 150. To measure the execution time we relied on the benchmarking module provided by the Deno standard library [62], and to ensure the statistical value of the results we repeated the test 500 times. Initially, we compared the execution time of vanilla Deno and Cage4Deno. While Cage4Deno inevitably introduces an overhead, its performance degradation (compared to Deno's) ranges between 0.8 and 2.5 ms, which is a reasonable price for the additional security guarantees it provides.

To further inquire, we compared Cage4Deno with two general-purpose sandbox-

ing solutions: *(i) Minijail* [87], and *(ii) Sandbox2*, a key component of the Sandboxed API [88] framework. As shown in Figure 3.5a, the execution time associated with the use of Minijail is 5.52 to 5.63 times slower compared to the one of Cage4Deno, and Sandbox2 exhibits even a worse degradation, which is never less than 8.15 times. To investigate the principal components causing the overhead, we profiled the execution of each tool with `perf`. Each of them presented two distinct phases: *(i)* sandbox setup, and *(ii)* restricted execution. For Minijail and Sandbox2 the first phase is dominated by creating the mount namespace, changing root directory, and performing a `bind-mount` for all the files needed to run the utility. For Cage4Deno, equivalent protection can be achieved with only the creation and enforcement of the Landlock rulesets. As for the execution phase, Minijail and Sandbox2 suffer slowdowns whenever performing filesystem operations due to the execution of kernel code paths related to the resolution of namespaces and bind mounts. In addition, Sandbox2 reference monitor architecture requires inter-process communication between the tracee and its tracer, which further degrades performance. On the other hand, Cage4Deno only pays the overhead of evaluating the Landlock security checks.[4] This validates our design choices, proving that our solution can provide significantly better performance with respect to industrial general-purpose sandboxing solutions. Moreover, Figure 3.5a depicts an interesting trend. Among the sandboxing alternatives, not only Cage4Deno delivers the best performance, but it also exhibits the slowest linear growth with respect to the increase of the number of rules, making it the best option when a large number of policy rules is necessary.

Overhead associated with deny rules

The previous tests do not take into account the deny rules. In fact, both Minijail and Sandbox2 do not support the presence of negative permissions. It is then important to measure the overhead introduced by them, as well as its degradation when the policy size increases. To better investigate the overhead experienced by end-users when Cage4Deno is used on the back-end, we evaluated the response time of a single HTTP endpoint that responds to incoming request with the content printed by the `cat` command. To generate the HTTP requests we relied on *JMeter* [15], a tool written in Java designed to load test and measure the performance of web applications. Specifically, we configured it to generate 100 warm up requests and measured the latency of the following 5000. To simulate the overhead experienced by end users of a web application, we also set up a network delay of 10±5 ms (with a normal distribution) using `tc` [13], a Linux utility to configure and shape the network traffic. The test is executed four times with the following policy configurations: *(i)*

[4]The flame graphs used for the analysis are available in the repository of the project.

no policy (i.e., vanilla Deno), *(ii)* only positive rules (i.e., `RWX`), *(iii)* only deny rules, and finally *(iv)* a policy characterized by the same number of `RWX` and `D` rules. Again the number of rules in the policy ranges between 25 and 150.

In general, the results reported in Figure 3.5b confirm the trend shown in Figure 3.5a, with an overhead increasing almost linearly with the number of rules in the policy. This is a positive aspect, since it applies to both `RWX` and `D` rules. The relative overhead perceived by clients, with respect to the default implementation of Deno, ranges from 2.74% to 10.21% for policies listing only `RWX` rules. This attests that the cost associated with Landlock is low, but it increases linearly with the number of rules (in accordance with the current inode-based implementation of the LSM). On the other hand, the use of BPF is distinguished by a higher initial cost, but the performance degradation introduced when the number of deny rules increases is smaller compared to the one measured with pure `RWX` policies, since it ranges between 10.17% and 12.79%. Moreover, policies composed of both rule types still exhibit a linear trend, with a client perceived overhead ranging from 11.99% to 17.24%. It is important to note that the presence of deny rules not only improves the readability and maintainability of the policy, but can also improve the performance as it permits to reduce the overall number of rules in the policy.

3.7 Related Work

Several approaches to strengthen the security guarantees provided by JavaScript execution environments have been proposed by both industry and academia. While developing Cage4Deno we focused on solutions that target runtimes and are therefore applicable to the server-side scenario. Our proposal complements them providing effective isolation of subprocesses using recent technologies. In the following, we separate the related works into four main categories.

Secure JavaScript sandboxes

Several works have been proposed to strengthen the security guarantees offered by JavaScript runtimes before the advent of Deno [121, 172, 149, 189, 190, 8, 202, 142, 143, 64].

Secure EcmaScript (SES) [121] is a runtime library to execute third-party code safely in lightweight compartments. Essentially, SES implements a *frozen* execution environment in which scripts have no abilities to interfere with each other. An alternative for the safe execution of untrusted third-party JavaScript code is *vm2* [149]. Its peculiarity is that it overrides the built-in `require`, enabling the developer to restrict access to a pre-defined set of modules. SES and vm2 have been effective

in mitigating the vulnerabilities coming from unverified or untrusted third-party JavaScript code, since they allow the developer to practically limit the APIs exposed to the sandboxed JavaScript program. However, when access to a dangerous API such as `Deno.run()` or `child_process.spawn()` is granted, no restriction on the subprocess is enforced.

Another interesting approach to reduce the security risks coming from the use of third-party modules is *BreakApp* [189]. The authors use module boundaries to automate compartmentalization of systems and enforce security policies. To this end, BreakApp spawns and executes modules in protected compartments, while preserving their original behavior. Compartments are characterized by three levels of isolation: sandbox, process, and container level. While the approach is powerful and ensures strong security properties when the container level is used, it is not straightforward to setup fine-grained filesystem permissions when the process level is selected.

Mir [190] is a relevant proposal to mitigate the security risks coming from third-party modules through the use of library-specialized contexts. Mir applies a fine-grained `RWX` permission model to every field of every free variable name in the context of an imported library, with permissions inferred through static and dynamic analysis. *SandTrap* [8] shares with Mir the idea of protecting read, write and call on entities (primitive values, functions, objects) and construct policies on cross-domain interaction. The goal again is to mitigate the security risk coming from the use of third-party modules, but in the Trigger-Action Platforms (TAPs) framework. While both the approaches can mitigate several JavaScript vulnerabilities, no restriction is applied on code executed outside of the runtime.

Wolf at the Door [202] is a recent proposal to reduce the risk coming from the installation of third-party modules. The authors propose to use the Apparmor LSM to detect install time anomalies such as connection to unknown hosts or read of confidential files. The security checks are enforced based on a policy typically written from the package maintainer. The approach protects the host against undesired behavior of third-party packages installed through `npm`, but the protection is enforced only at install time.

Isolation and sandboxing of unsafe libraries

The problem of isolating subprocesses addressed by Cage4Deno shares strong similarities with the isolation and sandboxing of third-party libraries. This area has received significant attention recently, and many proposals have been published [86, 159, 192, 154, 187, 96, 150, 106, 132, 157]. *Galeed* [154], *PKRU-Safe* [106], and *NoJITsu* [150] guarantee strong isolation of unsafe components with the use of

Memory Protection Keys (MPK). Galeed and PKRU-Safe preserve the memory safety of Rust code when used in conjunction with unsafe code (e.g., C/C++). No-JITsu brings hardware-backed, fine-grained memory access protection to JavaScript engines, thus successfully hindering a wide range of memory corruption attacks. Differently from previous solutions, *RLBox* [132] achieves sandboxing of third-party libraries through software-based-fault isolation. RLBox facilitates the retrofitting of existing applications using a type-driven approach that significantly ease the burden of securely sandboxing libraries in existing code. These solutions are complementary to Cage4Deno, as they can be used in conjunction with it to enforce trust boundaries in the interaction between Deno and its subprocesses.

General-purpose sandboxers

To reduce the impact of potentially vulnerable processes, several approaches leveraging system call interposition have been proposed [28, 105, 57, 87, 88, 134, 47].

TRON [28] is a discretionary access control system. The authors designed a Unix based capability system to limit process access to files, directories and directory trees. Although the approach allows fine-granularity access control, it suffers from two major drawbacks: *(i)* it requires modifications on the kernel, and *(ii)* it requires programs changes to make them capability-aware.

Another interesting initiative to implement unprivileged sandboxes is *MBOX* [105]. MBOX executes a program in a sandbox, and prevents it from modifying the host filesystem by layering the sandbox filesystem on top of it. Only at the end of the execution the user can examine changes in the layered filesystem and commit them back to the host. To do so, MBOX interposes system calls using Seccomp filters, and relies on `ptrace` to enforce permissions. However, the use of `ptrace` is prone to TOCTOU attacks [102], and as experimented by the authors, it can lead to non-negligible overhead.

In practice, system call interposition is often used to limit the set of system calls available to a program [57, 164, 193]. This effectively limits the capabilities of an attacker expoiting the program, and reduces the attack surface of the kernel [110]. Nowadays, most solutions rely on Seccomp filtering, a Linux kernel feature that allow to filter system calls based on their identifiers and parameter *values*. While valid, Seccomp filters can neither dereference pointers to user memory, nor actionably use file descriptor numbers, thus they are not suitable to perform access control of filesytem resources. These solutions can be used in conjunction with Cage4Deno to reduce the attacker capabilities and the attack surface of the kernel.

In Section 3.6 we have already compared Cage4Deno to other general-purpose sandboxing solutions such as *Minijail* [87], and *Sandbox2* [88]. These tools support

multiple types of containment techniques such as the introduction of new user ids, restriction of capabilities, policy-based Seccomp filtering, and namespace isolation. Both the tools are powerful and flexible, but they are not specifically aimed towards protecting web applications. Thus, they are not optimized for the execution of short-lived programs, and they target security experts, making them of difficult use to a wider audience. Specifically, to achieve comparable protection to our `RWX` rules, the developer needs to configure them to: *(i)* create a new mount namespace, *(ii)* change the root directory of the binary, and *(iii)* remount every part of the file hierarchy necessary for the functioning of the binary under the new root. On the other hand, Minijail and Sandbox2 use well enstablished kernel features available in every modern Linux kernel version, and can be used without additional privileges provided there is no need to bind-mount privileged resources. Similar considerations can be made about *Firejail* [134] and *Bubblewrap* [47]; sandboxing tools functionally comparable to Minijail and Sandbox2, but less mature.

BPF-based sandboxers

Other proposals have used BPF as the primary means to enforce access control policies [81, 80, 25, 4, 99, 175, 77].

BPFBox [81] and *BPFContain* [80] are runtime security frameworks focusing on the containment of processes and containers, respectively. Comparing Cage4Deno to both the proposals, *(i)* we do not require a privileged runtime daemon to keep track of the traced processes (single point of failure), *(ii)* we rely on Landlock to enforce `RWX` permissions so to guarantee low runtime overhead, and *(iii)* we provide a tool for the automatic generation of policies.

Snappy [25] strengthens the security of containers using namespaces and BPF policies. To support the programmable policies described, the authors introduced in the kernel a set of new *dynamic helpers*. Jia et al. [99] present a mechanism to define advanced syscall filtering policies with the *extended* BPF. This is currently achieved by hooking syscall tracepoints that cause system-wide performance degradation and are still subject to TOCTOU attacks. On the other hand, Jia et al. successfully address these limitations proposing changes to the Linux kernel. These proposals require to recompile the kernel with the addition of ad hoc functions not part of the kernel codebase, thus limiting their usability and portability.

There are also industrial solutions using BPF, for instance: *Cilium* [175] and *Falco* [77]. The former provides BPF-based networking, observability and security between container workloads, the latter is a threat detection engine for clusters. Both operate at the container granularity; hence, developers may find it difficult to set up fine-grained permissions with these frameworks.

These proposals demonstrate the value of BPF in securing different types of system resources. Extending our proposal to the protection of non-filesystem resources is promising for future work.

3.8 Conclusions

Web development is a fast-paced environment characterized by tight time constraints. In this vast ecosystem, we specifically considered the role of Deno, a modern and secure runtime for JavaScript and TypeScript. Our proposal presents a way to improve the Deno security model by sandboxing the invocation of subprocesses, which represent an important attack vector of web applications. As shown in the experimental evaluation, Cage4Deno effectively mitigates real CVEs affecting widely-used utilities in web applications. Moreover, it exhibits significantly better performance with respect to other general-purpose sandboxing solutions. We paid great attention to reduce the overhead of the developer willing to strengthen the security of its application. This is achieved not only by exposing a simple and clear interface to the developer (requiring no mandatory code changes), but also providing a command line tool to generate least-privileged, yet human-readable, access control policies. We believe the proposed approach is general enough to be applied to other runtimes, since they share the same design as Deno regarding the unconstrained execution of subprocesses (e.g., Node.js and Bun). This is a promising line of research we plan to explore in the future.

4. Leveraging eBPF to enhance sandboxing of WebAssembly runtimes

4.1 Introduction

The work done with Cage4Deno highlighted the importance of having secure and isolated environments for software and code execution. Considering the cloud scenario, a technology that has seen a great growth is represented by WebAssemby (Wasm), and its runtimes that allow interaction with the underlying system. Their increasingly widespread usage led us to focus our attention on how the interaction between these runtimes and the system occurs to propose enhancements of their security guarantees.

WebAssembly [94] is a popular binary instruction format that enables the execution of untrusted code in a safe, isolated environment. Moreover, it is a portable compilation target for different languages, and can be executed efficiently on a wide range of platforms without the need of dedicated hardware. Wasm was originally meant to be run inside web browsers, but given the considerable advantages it brings, many runtimes that allow execution in standalone mode have been developed recently. Popular examples are Wasmtime, WasmEdge, Wasmer, and WAMR.

To answer the developers' need to access resources of the host system from within the runtime, a standardization effort called WebAssembly System Interface (WASI) [198] is undergoing. Its goal is to provide a stable and multi-platform system interface. To be WASI-compliant, each runtime must implement all the calls defined in the interface with dedicated functions, which are named hostcalls. However, implementing these functions is non-trivial, since (i) the code must not introduce violations to the Wasm memory model, and (ii) it is possible to break the separation between the system and the isolated environment in which the Wasm module is executed. The solution adopted by current runtimes leverages WASI Libc [196], a library providing POSIX-compatible APIs built on top of hostcalls.

Currently, every WASI-compliant runtime implements the proposed file system interface with a libpreopen-like layer [130]. Whenever the runtime receives a request to open a file, it first checks whether the path belongs to the authorized list of directories, then it opens the file on behalf of the Wasm program, redirecting the

content to the caller. Previous work [101, 30, 108] proved the approach to be error-prone, leaving the system unprotected when a vulnerability was introduced in a hostcall wrapper (Figure 4.1). Moreover, this approach provides limited flexibility, as it is associated with directory-based granularity instead of file-based. Lastly, in order to audit the policy regulating resource access, one must find the permissions by looking at the code. We claim that there is no practical advantage in having several implementations of the same access control checks for different runtimes. Our idea is to replace the user-space runtime-specific security checks with a single in-kernel implementation that leverages eBPF [179]. There are considerable advantages in doing so: (i) it permits to decouple the implementation of hostcall wrappers and the access control details, minimizing the risk of bugs [104, 157, 5], (ii) it enables the introduction of per-module policies with file-based granularity, and (iii) it fulfills Wasm's promise of portability as eBPF programs are portable across different kernel versions [131] and also operating systems, thanks to Microsoft's undergoing effort to port eBPF to Windows [127].

4.2 Threat model

Our assumptions reflect the threat model employed by Wasm runtimes. We assume that the code executed by the runtime is either untrusted or it is trusted but potentially affected by security vulnerabilities due to bugs. The goal of the attacker providing the code is to bypass the security checks enforced by the runtime to get access to the host file system. To fulfill this objective, the attacker can leverage the interface provided by WASI and send any argument. Runtime escapes caused by memory corruption or alteration of the program flow are out of scope of our work, since protection can be provided by other existing solutions (e.g., [30]).

4.3 Architecture

Our analysis starts from the scenario illustrated in Figure 4.1. Currently, WASI-compliant runtimes implement dedicated user-space wrappers to enforce the security boundaries of hostcalls. File system access is granted by the user on a set of pre-opened directories that are specified via CLI before the Wasm module is run (e.g., with the `--dir` option). We follow a similar approach, asking the user to state the permissions of each Wasm module in a JSON policy file. Contrary to existing runtimes, permissions can be granted with file-based granularity. Three permissions are available: (i) `read` to open and read a file, (ii) `write` to modify, truncate and append content to a file, and (iii) `delete` to remove the file. When permissions are

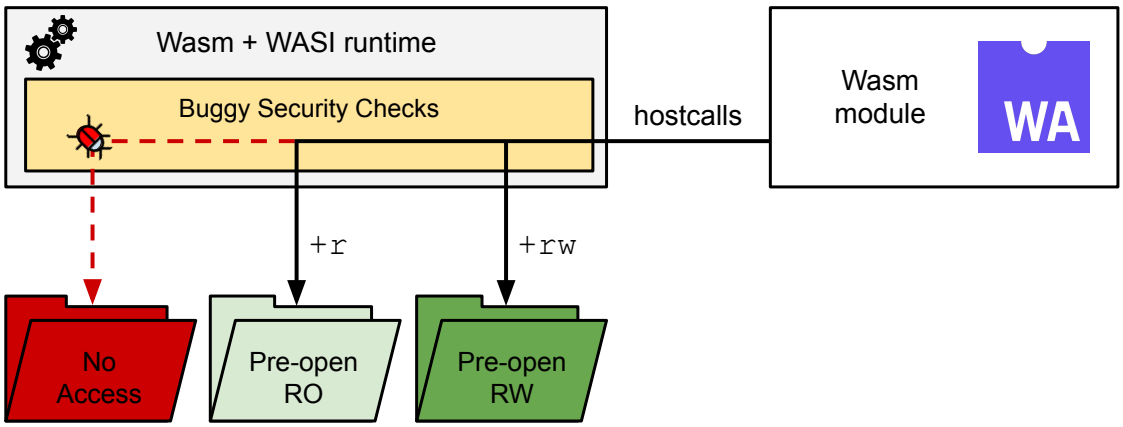


Figure 4.1: Current implementation of WASI by runtimes. A bug present in a hostcall wrapper permits the module to read the unauthorized directory on the left (red dotted arrow)

related to a directory, `read` translates to listing its content, `write` allows to create and delete files within it. We extended the Wasmtime and WasmEdge runtimes to load the policy at startup, and, instead of pre-opening the directories available to the Wasm module, we enforce the policy with eBPF. eBPF code is split into programs attached to a kernel- or user-space function called *hook point* and executed whenever the hook is reached. Programs have visibility of function parameters, they can persist state and share it with user space using *maps*, and most of all they can enforce security checks based on this information. Once the policy is encoded inside the map and the eBPF programs are loaded, the runtime instantiates the Wasm module selected by the user (arrow Ⓐ in Figure 4.2). At this stage, the modified runtime invokes a dedicated user probe specifying as a parameter the policy that confines the loaded Wasm module (Ⓑ). The argument is captured by a dedicated eBPF program that also annotates the identifier of the thread running the Wasm interpreter in a tracing map. We highlight that the policy is activated before the runtime executes the module (i.e., before untrusted code is interpreted). The consequence is that, from this point on, all the hostcalls performed by the Wasm module are restricted by our eBPF programs (arrows ①, ②). The eBPF programs that make the security decisions are evaluated every time a file-related kernel security hook is reached (e.g., `security_file_open`), and any access decision is enforced at kernel level. When an unauthorized request is performed by the Wasm code (③), the related eBPF program detects the violation and denies the request, returning to the caller a permission denied error. When the execution of untrusted Wasm code terminates, another eBPF program is responsible for removing the access restriction from the thread executing the Wasm runtime. No further intervention from the runtime is required, as the maps and the eBPF programs are automatically removed from the kernel

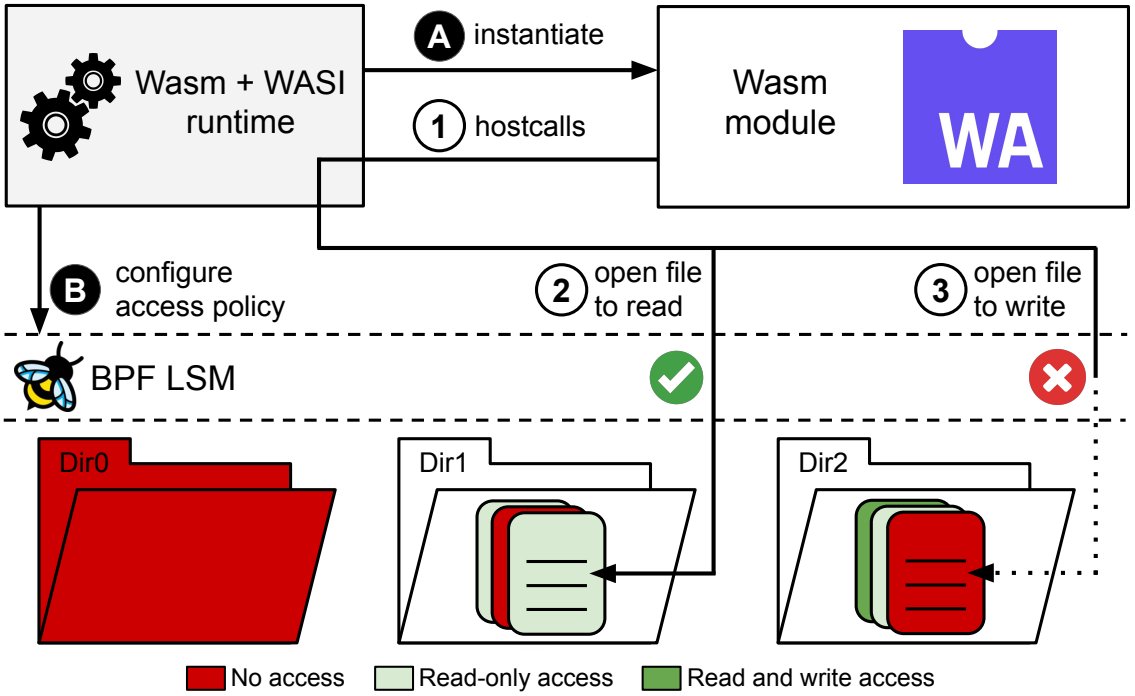


Figure 4.2: eBPF-based restriction of Wasm modules. The runtime instantiates the Wasm module (Ⓐ), and configures the associated policy calling the traced user probe (Ⓑ). After the Wasm module is run, all the hostcalls issued by the program (①) are restricted by eBPF (②, ③)

immediately after the process running the runtime terminates.

This architecture offers several advantages. First, it eliminates the risks coming from buggy user-space security checks (e.g., wrong filepath resolution [126], wrong directory removal [44]). Then, by leveraging kernel hook points [179], our approach allows the runtime developer to focus on the interaction between Wasm code and the memory unsafe system call, leaving aside authorizations and policy-related issues. Lastly, access constraints can be audited by simply looking at the JSON policy, instead of inspecting the code.

4.4 Experiments

To investigate the overhead introduced by our solution we implemented it in WasmEdge and Wasmtime, two industrial state-of-the-art Wasm runtimes. The evaluation has been performed in the following test environment: an Ubuntu 22.04 LTS server powered by an AMD Ryzen 2950X CPU with 16 cores, 128 GB RAM, and 2 TB SSD. In order to assess the performance, we tested one of the most popular binaries that can be compiled to Wasm with support to WASI: `uutils coreutils`, the porting of the coreutils in Rust [176]. First, we compiled the core-

Utility	WasmEdge	WasmEdge*	Wasmtime	Wasmtime*
`head`	32	34 (+6.25%)	14	16 (+14.29%)
`sum`	134	137 (+2.24%)	130	136 (+4.62%)
`tac`	149	150 (+0.67%)	152	155 (+1.97%)
`wc`	285	287 (+0.70%)	309	310 (+0.32%)
`shuf`	298	300 (+0.67%)	356	358 (+0.56%)
`ls`	512	526 (+2.73%)	1077	1113 (+3.34%)
`seq`	1155	1157 (+0.17%)	1526	1533 (+0.46%)
`cut`	1403	1411 (+0.57%)	359	360 (+0.28%)
`join`	1601	1603 (+0.12%)	2054	2065 (+0.54%)
`split`	4416	4694 (+6.30%)	4933	4998 (+1.32%)

Table 4.1: Average execution time in *ms* of the coreutils without and with* our approach (% overhead in parenthesis)

utils with the `wasm32-wasi` target, and applied runtime-specific optimization (with `wasmedgec` [194] for WasmEdge and with `wasmtime compile` [11] for Wasmtime) to further speed up the code. Then, we reproduced the benchmarks reported in the coreutils repository, with the exception of those that are not portable to WASI due to temporary lack of support (e.g., the `dd` utility needs to spawn threads, a feature that is yet to be implemented [76]). Finally, we repeated the experiments with our protection in place. The Hyperfine benchmarking tool [52] was used to log measures, and 1000 runs were performed (with 100 warmups). As shown from the results in Table 4.1, our approach introduces a limited overhead, ranging from an additional 0.12% to 6.30% for WasmEdge, and from 0.28% to 14.29% for Wasmtime. As expected, the highest overhead is experienced by short-living utilities (e.g., `head`). We also observe that there are notable differences between the WasmEdge and Wasmtime test execution time for some utilities (e.g., `ls` and `cut`); from our analysis these differences are mostly caused by the specific post-compilation optimizations.

4.5 Related work

There are several successful solutions that leverage Wasm to sandbox untrusted code [132, 152, 84]. RLBox [132] is a framework that facilitates the isolation of third-party libraries in pre-existing software. eWASM [152] optimizes the execution of Wasm in embedded systems with constrained resources. Sledge [84] enables efficient Wasm-based serverless execution on the edge. The use of our approach for restricting access to the file system within these frameworks can strengthen their security assurance.

The memory safety guarantees of Wasm depend on the runtime implementa-

tion [108]. Hence, Bosamiya et al. [30] explore the problem of producing provably safe sandboxes. WaVe [101] explains that any interaction with the unsafe interfaces exposed by WASI can introduce security and safety violations. Thus, the authors proposed a verified secure runtime system implementing WASI. However, both works require to redesign the runtime toolchain, while our solution can be directly integrated into existing runtimes.

The academic and industrial communities have investigated the use of eBPF for the isolation of software [81, 80, 175, 77]. *BPFBox* [81] and *BPFContain* [80] use an eBPF daemon to confine processes and services. *Cilium* [175] provides eBPF-based networking, observability and security for container workloads. *Falco* [77] enables lightweight threat detection in the cluster. These solutions highlight the potential of eBPF, and provide a simple and flexible confinement of system resources. However, they focus on containers or services, while our solution aims at enforcing fine-grained per-sandbox policies.

4.6 Conclusions

The results achieved by our approach are promising: not only it permits to introduce fine-grained policies to restrict file system access, it is also associated with a limited overhead which is aligned with the needs of a modern sandbox. The protection is currently applied only to the file system, but our approach has the potential to be extended also to network sockets, which are in the first stage of the standardization process [23]. We believe this could be an interesting line of research for future work.

5. NatiSand: Native Code Sandboxing for JavaScript Runtimes

5.1 Introduction

Based on the research conducted in Chapter 3 we have continued on this line of research by extending our proposal with a more fine-grained access control. In this regard, we proposed a way to further restrict the access that modern JavaScript and TypeScript runtimes have on the system resources. In fact, JavaScript (JS) and TypeScript (TS) are popular choices for the implementation of web applications. This success is motivated by their flexibility, since both are simple to use for the development of frontend and backend services, and by the vast ecosystem of open source packages that are available. For instance, the sole *npm registry* collects more than 1.3 million packages [136].

The execution of JS code on the server-side is enabled by a *JS runtime*. Since its introduction in 2009, Node.js [147] has been the de facto solution selected by developers, but recently Deno [60] and Bun [36] have received considerable attention by the community. While the three platforms provide distinctive features, they all depend on a key external component, namely the *JS engine*, V8 [185] in the case of Node.js and Deno, JavaScriptCore [16] in the case of Bun. The engine is a sophisticated software that securely renders the JS code in an isolated sandbox. Runtimes extend the engine providing components to access resources and functions that are not directly available to the web application from within the sandbox [144, 66]. Prominent examples are the functions to access the network and to read/write the filesystem. Runtimes also provide support for the execution of native code – i.e., running binary programs installed on the host operating system and calling functions from the available shared libraries.

The support provided by the runtime for the execution of native code greatly simplifies the work of the developer building the backend of a web application. However, the APIs enabling access to system resources and the execution of native code also raise security concerns, since they effectively break the isolation between the JS application and the host OS. The ability to control the resources accessible to a JS program was indeed one of the reasons that led to the creation of Deno in 2018 [153], and the solution identified by the community was to configure the resources available to an application with simple permission flags [67]. This change also influenced the

design of Node.js, which introduced a similar flag-based control model[1] two years later [145]. Unfortunately, while permissions are effective in restricting access to the JS application, they do not provide isolation guarantees when native code is executed, leaving the host exposed to security breaches [67].

Previous research [170, 78] has already shown that frequently JS modules depend on components written in native languages such as C or C++. The reuse of existing utilities permits to take advantage of popular high performance libraries and, in addition to performance, it minimizes the cost of development. Notable examples are: the node-sqlite3 [183] and deno-sqlite3 [69] database drivers; modules to perform image/video conversions, such as sharp [141], fluent-ffmpeg [137] and gm [138]; OCR engines like Tesseract [173]; and the cryptography modules relying on bcrypt [140]. The 2022 State of Open Source Security [166] claims that each open source JS project relies on an average of 174 third-party dependencies; also, each project is estimated to be affected by 40 vulnerabilities when its dependencies are taken into account. Taking into consideration that web applications in most cases process untrusted input, the risk of security incidents is high. For instance, we identified a sample of 32 high severity CVEs[2] that affect native code used by popular packages (with 2.6M downloads/week), and allow an adversary to corrupt the filesystem, perform privilege escalation, execute arbitrary code, open network connections to exfiltrate data, etc.

Our contribution We see a security gap in the way modern JS runtimes execute native code, as neither Node.js, nor Deno, nor Bun sandbox it. In this chapter we propose NatiSand, a framework to provide strong isolation guarantees against the execution of native code. In detail, NatiSand allows the developer to control on a native-component basis, access to filesystem, Inter-Process Communication, and network, effectively reducing the risks coming from the execution of binary programs and shared libraries. Our solution is characterized by a compact, generic architecture that fits nicely with modern runtimes. Internally, it leverages *Seccomp* [37] and Linux Security Modules (LSMs), such as *Landlock* [122] and *eBPF* [49] to restrict access to protected resources. In the design of our solution we paid attention to usability by developers; it is not necessary to have a full understanding of its advanced security features to use it. The developer is only required to provide a concise and readable JSON-formatted policy file, detailing the *ambient rights* – i.e., the access privileges available to the components of the web application that rely on native code. To this end, we provide the developer a comprehensive and interactive CLI tool to support policy generation, which, as best practice suggests, can also be

[1] Node.js support for creating security policies is still experimental as of 2023.
[2] The list is reported in Table 5.3.

integrated into CI/CD pipelines and run against a set of test cases [8, 164]. Another key advantage of our approach is that it permits to sandbox native code preserving backward compatibility, namely it does not require to change existing modules (including third-party dependencies) to leverage the new security features.

We implemented NatiSand and integrated it into Deno. We demonstrate the security benefits showing how our solution mitigates a number of recent, high-severity vulnerabilities. We performed an extensive experimental evaluation to assess its performance. We compare the overhead introduced by our solution to scenarios in which no native code sandboxing is performed, and when sandboxing is achieved through other general purpose, state of the art solutions. The experiments show that, compared to the alternatives, NatiSand exhibits substantial performance improvements. In addition it also provides an easier interface that does not require any specific security expertise to be correctly configured.

5.2 Background

This section overviews the structure of a modern JS runtime. It also provides a concise description of the components that are used by NatiSand to build the sandbox.

5.2.1 JS runtimes

JS code rendering is a complex process, involving tasks such as code compilation, code optimization, memory allocation, runtime garbage collection of objects no longer needed, and many others. To perform these critical tasks, modern JS runtimes rely on *engines*, dedicated components implementing the ECMAScript specification that were originally developed for web browsers. As already mentioned in Section 5.1, Node.js and Deno embed Google's V8 [185], while Bun relies on JavaScriptCore [16]. The interoperability between runtime and engine is achieved with specialized bindings, which are defined in the *node:v8* [146] module in the case of Node.js, in the *rusty_v8* [68] library for Deno, and by *webcore* [148] in Bun.

While an engine provides all the tools to securely execute JS code in an isolated context (we call it the *JS context*), a development platform requires complementary features to be fully functional. For instance, a backend web application may need to open network connections, handle several concurrent HTTP requests, or access the filesystem to read configuration files. To address these requirements, the architecture of a modern runtime extends the engine with various runtime-specific components dedicated to the interaction with the host system. A few well-known functions implemented following this design pattern are: interaction with the filesystem (e.g.,

fs, Deno.FsFile), creation of UNIX sockets (e.g., net), and exposure of HTTP servers (e.g., http, Deno.serveHttp).

From the web application perspective there is no difference between the functions defined by the ECMAScript standard, and the ones provided by the runtime (and its extensions) [32, 70]. However, non-standard APIs are not served directly by the engine, but are redirected to the runtime leveraging the aforementioned bindings. Since these APIs deliberately permit to break the isolation between the environment controlled by the engine and the underlying system, JS runtimes allow developers to restrict them through the definition of permissions [145, 67]. Based on the runtime, permissions work with different granularities (e.g., single API vs set of APIs) and different default behavior. For example, Deno uses a default-deny model requiring the developer's explicit consent to access system resources, with effect on multiple APIs [67].

Permissions are intuitive and effective, but they do not offer significant security guarantees when a module needs to run binary programs, or import shared libraries to leverage cross-language function calls [67]. To do so, the web application must be granted the permissions to call APIs like *command* or *dlopen* (e.g., with `allow-run` and `allow-ffi` flags in Deno). In the case of *dlopen*, native code is directly copied into one of the processes owned by the runtime itself before being executed, while with *command*, the runtime first delegates to the OS the creation of a process with the `clone` system call, then performs an `exec` to replace the process image and run the desired program. Independently of the runtime used, both APIs require to execute code outside of the isolated context managed by the engine, as shown in Figure 5.1, which means that this code runs with the same privileges of the user executing the entire JS application.

5.2.2 Components for resource protection

Landlock

Landlock [122] is a Linux Security Module (LSM) introduced in the kernel starting from release 5.13. The goal of Landlock is to enable unprivileged applications to restrict their ambient rights in accordance with the least privilege principle. Ambient rights are specified by *rulesets* – i.e., simple structures that associate a set of permissible actions with a filesystem path (e.g., `read` and `exec` over the resources stored in `/tmp`). Several rulesets can be combined to determine the final set of actions available to an application. To make them effective, a call to the `landlock_restrict_self()` function is performed.

The ambient rights granted by Landlock are thread-based, and are automatically

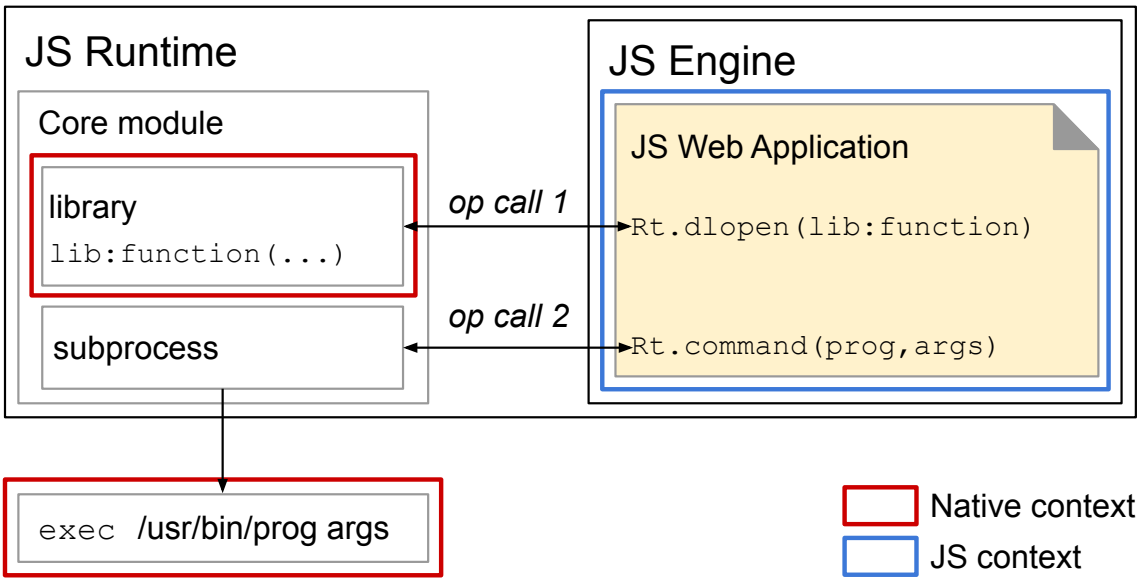


Figure 5.1: Execution of binary programs and shared library functions by the JS runtime

inherited by all the children subsequently created via `clone`. After a Landlock sandbox is enforced (either by self restriction or inheritance), it is only possible to further narrow it. It is also important to mention that Landlock is stackable, hence it is fully composable with other LSMs already available on the host, such as SELinux, AppArmor and SMACK. Although Landlock offers a simple, yet powerful, sandboxing API, currently, the protection offered is only limited to the filesystem.

BPF

Berkeley Packet Filter (BPF) was originally devised in 1992 [125]. The goal was to provide an in-kernel facility to filter and multiplex network packets, similarly to what was proposed by Mogul et al. [128]. This version of BPF, which is now commonly referred to as *classic BPF* (cBPF), was greatly revised in 2014 resulting in *extended BPF* (eBPF) [49]. The new framework provides an environment to execute programs inside the kernel [91, 104]. This permits to extend the kernel safely, without changing its source code nor loading new modules. eBPF has a wide variety of use cases, ranging from low overhead observability and tracing, to load-balancing, and container runtime security enforcement.

eBPF code is organized into compact units called programs. Each program is attached to a specific function named *hook point*, and is executed in a non-preemptable fashion every time the hook is reached. There are several types of hook point both in kernel space and in user space. Valid examples are [179]: system calls, kernel tracepoints, network events, function entry/exit points, and LSM hooks. Specifically, LSM hooks correspond to the functions used by LSMs (e.g. SELinux) to perform

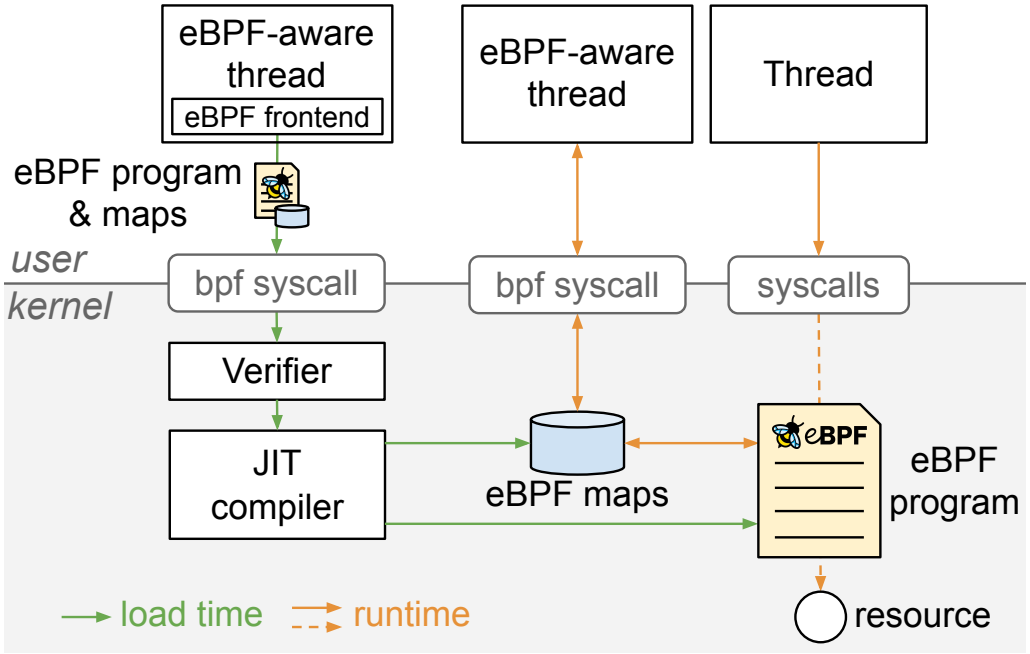


Figure 5.2: Overview of the eBPF architecture

security decisions and are characterized by operating entirely on arguments in kernel memory. To persist information between distinct invocations of the same program, data structures named *maps* are used. Maps also permit to share data among eBPF programs and user space applications.

eBPF programs are written in bytecode and are loaded into the kernel using the `bpf` syscall [112]. This is a privileged operation that requires a few capabilities, which vary with the nature of the program [1]. Briefly, `CAP_BPF` is always required, `CAP_PERFMON` is necessary to load tracing-related programs, while `CAP_NET_ADMIN` is used to load networking-related ones. After being loaded, each program undergoes a two-phase process comprising *program verification* and *JIT compilation*. The former is required to guarantee that the program is safe to execute by the kernel. The second phase instead ensures the bytecode is optimized, hence it can be run as efficiently as compiled kernel code on the underlying architecture. In case no errors are raised, the eBPF program is attached to the proper hook and it is ready to be executed.

Modern eBPF development is facilitated by the presence of *frontends*. These frameworks permit to write eBPF programs in a C dialect, and also assist the developer in automatically performing the steps needed to load and attach the programs to the intended hooks (see Figure 5.2). In the work presented in this chapter we rely on *libbpf* [111], a modern library leveraging the *Compile Once-Run Everywhere* (CO-RE) approach [14], which ensures that the bytecode produced at compile time works correctly across different kernel versions.

Seccomp

Seccomp [37] is a mechanism provided by the Linux kernel to restrict the system calls available to a userspace application. The rationale is that the implementation of system calls may be affected by bugs or errors, therefore reducing the kernel surface exposed to an unprivileged application narrows the attack surface.

The initial implementation of Seccomp restricted the set of allowed system calls only to `exit`, `sigreturn`, `read` and `write` (on previously opened file descriptors) [98]. The implementation was greatly extended in 2012, and it now permits to intercept system calls and determine whether each of them is safe to execute. To this purpose a *filter* program written in the cBPF dialect must be provided. Unfortunately, a classic program has only access to the values of the arguments passed to the system calls (e.g., configuration *flags*), and pointers cannot be dereferenced to avoid TOCTOU issues [74].

5.3 Security motivation

JS runtimes let developers specify the set of access privileges given to a web application [67, 145]. This is possible through a set of configuration flags that, when specified, allow to constrain how JS code can access system resources. This is a significant security improvement compared to the past, where applications were able to access any underlying system resource [169, 205]. However, these constraints only apply to JS code; any function written in other languages is executed unconstrained, either through a subprocess or the use of Foreign Function Interfaces (FFI). Indeed, native code does not access system resources using the APIs provided by the JS runtime and the reference monitor of the JS runtime is bypassed [67].

There is a broad variety of applications that rely on the use of native code. One well-known example is the use of database drivers; low latency of queries is crucial to satisfy the constraints on response time of a web application and a pure JS implementation may not be able to match them. This led to the development of third-party modules that depend on the code of shared libraries corresponding to the required database driver (e.g., libsqlite3.so and libmysqlclient.so). To testify the wide adoption of this practice, popular modules for both Node.js (e.g., node-sqlite3) and Deno (e.g., deno-sqlite3) report more than 600 thousand downloads/week. Notably, the deno-sqlite3 module was part of the official showcase of the performance of Deno when invoking the native code of a shared library [59]. Previous work [170] demonstrated how this module can be exploited with harmful effects for the web application and the underlying system.

Database drivers are just one example of how web application development relies on native code. Other popular use cases are audio encoding (e.g., libopus), image processing (e.g., ImageMagick, libvips), video manipulation (e.g., FFmpeg), optical character recognition (e.g., Tesseract), and many others. The native code may contain vulnerabilities, which may be exploited and lead to a variety of security violations.

Filesystem compromise Guaranteeing the integrity and confidentiality of the application host filesystem is crucial to mitigate risks of data corruption and exfiltration [28]. As a whole a web application often has access to many critical resources: databases, executables, private keys, user confidential files, etc. When native code is executed, it can use the same privileges of the web application. In line with the least privilege principle, the potentially vulnerable components should be able to access only the files needed to perform their duties. Authorizing access only to the needed portions of the filesystem restricts what can be read, written or run by an attacker, highly limiting the security risk associated with the presence of vulnerabilities.

Escalation of privileges Another relevant attack surface is the privilege of using the IPC channels provided by the operating system (e.g., pipes, message queues, unix sockets). By leveraging IPC, a compromised binary can establish a communication channel with system components and attempt a confused deputy attack to achieve privilege escalation on the host [161, 33]. Given the potential of this attack vector, it is important to limit the scope and set of IPC channels available to a specific native component only to those strictly necessary for its benign behavior.

Malicious network channels Network access is a precious resource that a malicious actor can leverage during an attack. A significant portion of malicious payloads open reverse shells to gain control of the victim system over the network [34]. In addition, attackers may open network channels to remotely recover data obtained on the vulnerable host [160]. Restrictions on how a single native component of the web application can access the network can greatly improve the overall security of the application. Network access should be forbidden or restricted only to domains defined by the developer, thus restricting the ability of adversaries to perform data exfiltration or fetch malicious payloads.

Notice that the JavaScript application may require a significant number of privileges to ensure all of its components operate as intended. Therefore, the application of sandboxing at runtime-level rather than native component-level not only is in contrast with the least privilege principle, it also increases the attack surface, so the chances of an attack to be successful.

5.3.1 Threat model

We consider the operating system trusted, although binary utilities may be malicious due to supply chain attacks, or affected by vulnerabilities due to incorrect memory management, improper data validation, etc. Protection against attacks targeting JS code is out of the scope of our proposal, since we consider JS engines and the permissions system enforced by JS runtimes able to securely render JS code. NatiSand aims to constrain the execution of potentially malicious, or vulnerable, binary utilities and functions used by JS applications. This native code accesses system resources unconstrained by the security mechanisms offered by the JS runtime, and its actions may cause severe security breaches. Moreover, the input processed by the web application is often untrusted and can be unsanitized, due to errors in the sanitization process, misconfiguration or lack of awareness by the developer. Therefore, a malicious actor can exploit this attack vector by submitting specifically crafted requests targeting the unconstrained native dependencies of the web application, compromising the host system. The attack vectors may take multiple forms, e.g., strings, videos, images, and audio files, depending on the input provided by the JS application to the vulnerable components. The goal of our proposal is to mitigate the security risk by empowering developers with a way to establish clear security boundaries for the execution of binary utilities and components depending on them with a per-native-component granularity.

5.4 Design and implementation

In this section we present NatiSand, our proposal to enable the isolation of native code for JS runtimes.

5.4.1 Objectives

We start with the definition of the design objectives.

Security As a secure sandbox, NatiSand must provide protection against recent, high-severity vulnerabilities affecting native components used by web applications. Furthermore, the additional protection must not result in a loss of functionality. The goal is to enable the developer to follow the least privilege principle when designing its application, reducing the attack surface in the presence of vulnerabilities. To do so, NatiSand must be able to execute distinct native code in separate lightweight compartments isolated from the rest of the application, and characterized by policy-based ambient rights. The security restrictions must be enforced independently of

the method leveraged by the application to execute native code, giving the developer the power to confine executables, shared libraries, and functions. Lastly, no root permission should be used at runtime to configure and activate the isolated compartments.

Usability An important requirement to consider is usability by developers. We cannot force them to rewrite their application (or large parts of it) just to use the sandbox. At the same time, we cannot expect them to be aware of the internal structure of the third-party native code used by the application, nor to fully understand the advanced security mechanisms that can be leveraged to securely sandbox a program. The effort required to take advantage of NatiSand should be extremely low. Ideally, a single configuration file specifying the ambient rights associated with each compartment should be enough to successfully configure it. To facilitate the transition from no sandboxing to complete isolation, a valuable solution should permit to start by sandboxing the components associated with the highest risk, and then gradually extend the protection to the remainder of the application.

Compatibility A valuable solution should be generic enough to be integrated into different JS runtimes without requiring substantial changes to the internal architecture. This also means that it must be aligned with the current permission-based model implemented by the most widely used platforms. Moreover, it must be compatible with other access control mechanisms already enabled by the underlying OS. This refers to the potential of stacking the sandbox on top of security mechanism adopted by other software.

Performance Latency and throughput are critical metrics for web applications, therefore it is important to reduce their degradation to a minimum. NatiSand aims to introduce lower overhead compared to current state of the art sandboxing and isolation frameworks.

5.4.2 High level architecture

NatiSand permits to transparently execute code in ad hoc *contexts*, isolated compartments that are characterized by policy-based ambient rights. This allows the developer to configure fine-grained access to confidential or privileged system resources, such as files, message queues, shared memory areas, sockets, and other resources.

NatiSand separates system resources into three categories: filesystem, IPC, and network. By default, native code sandboxed by our solution cannot access any

privileged resource in each category. Indeed, the developer must explicitly grant access to resources using a JSON-formatted policy file. JSON is a popular format among the web community and the ability to configure fine-grained permissions using a single, easy-to-read text file greatly simplifies the development activity. No specific knowledge is required to configure the policy, and no effort needs to be spent by the developer to understand how permissions are enforced.

Internally, NatiSand leverages dedicated Linux Security Modules to restrict access to each resource category. Filesystem-related permissions are enforced using Landlock, while the availability of IPC channels to interact with other processes or services already running on the host is controlled with Seccomp and eBPF. Finally, eBPF constrains the ability to open new connections and limits the devices reachable by a context. Three important characteristics are shared by the selected LSMs: (i) they are lightweight, (ii) they do not require to leverage root permissions while the application is running, and (iii) they operate in stacking mode [165], hence they are compatible with other LSMs already running on the host, such as AppArmor, SELinux, and SMACK. The stacking behavior also means that whenever the access decisions of two LSMs do not match, deny takes precedence. To give an example, Seccomp can deny the application to create a fifo file, even when Landlock grants the permission to write in the target directory.

The architecture of our solution is shown in Figure 5.3. Shortly after the JS runtime is executed, NatiSand parses the policy file input by the developer. Based on the policy, a set of sandboxing and tracing programs along with maps are initialized and loaded into the kernel. A pool of isolated contexts is also prepared by the sandbox. At runtime, NatiSand intercepts all the calls to native code performed by the application and executes them safely in the proper isolated context. A technical description of how our proposal is integrated into a modern JS runtime is given in Section 5.4.3, while details about its isolation features are reported in Section 5.4.4. The policy syntax used by NatiSand to configure permissions, along with the support to policy generation, are described in Section 5.5.

5.4.3 Integration with JS runtimes

NatiSand complements the architecture of the JS runtime with the addition of the *sandboxer*, a component that parses the policy and enforces the isolation of native code accordingly. In the following we explain the operations performed by the runtime to use it, referring to Figure 5.3.

Bootstrap The JS runtime boot procedure is modified to read the JSON policy file, which is then parsed by the sandboxer Ⓐ to retrieve the information associated

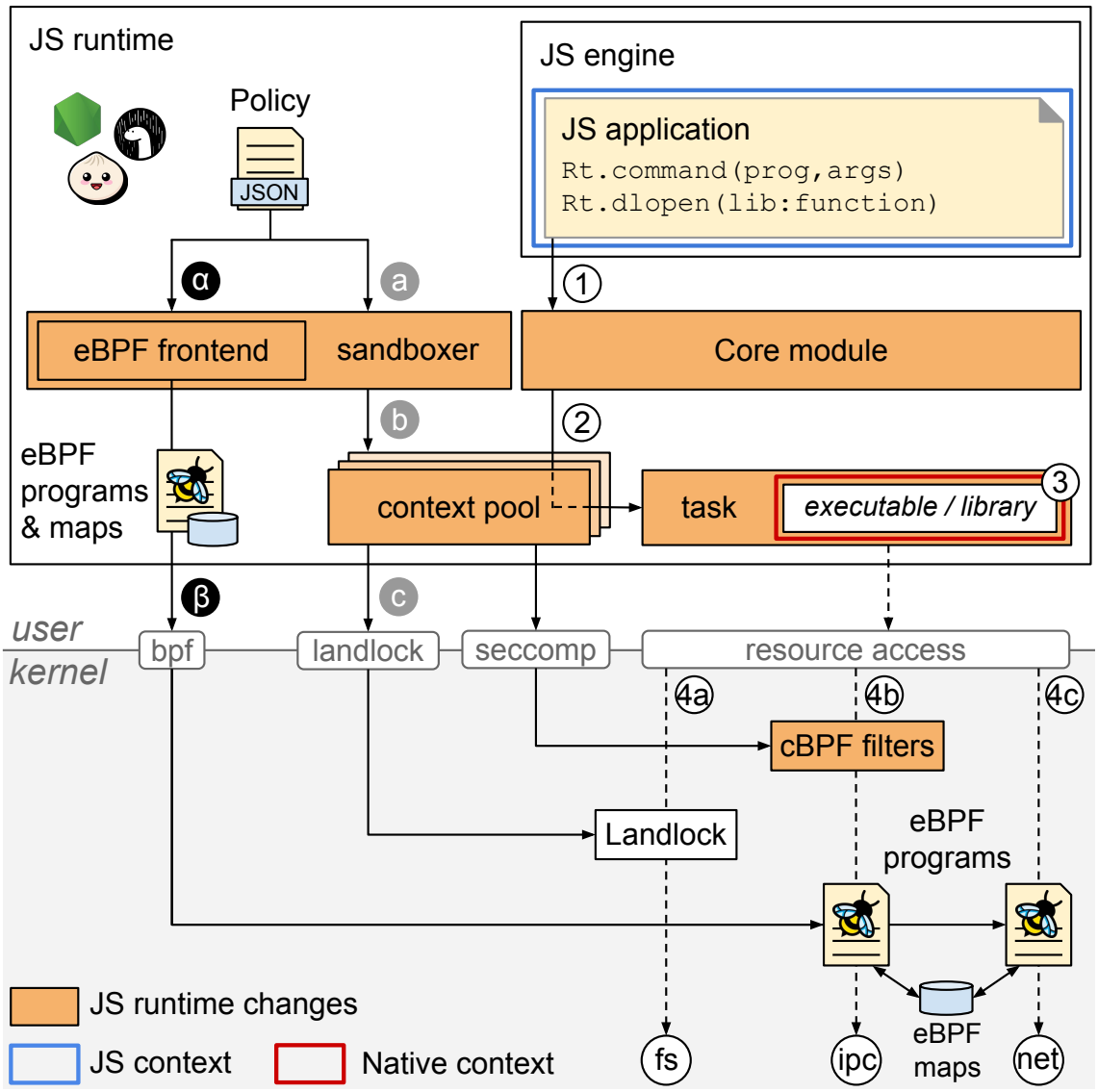


Figure 5.3: Integration of NatiSand in the JS runtime. Bootstrap: import security contexts (α, β), creation of the context pool (a, b, c). Application runtime: isolated execution of binary programs and shared library functions (1, 2, 3, $4a$, $4b$, $4c$)

with each security context. Based on the policy, the sandboxer (i) configures the required Seccomp filters, and (ii) prepares and loads into the kernel the necessary eBPF programs and maps (β). The eBPF programs are used to track the security contexts and enforce network-related and (part of) IPC-related restrictions (more details in Section 5.4.4). Maps instead associate each isolated compartment with a security policy, and store the ambient rights granted to them. To determine which policy is associated with a given isolated compartment we leverage its kernel `task_struct` identifier, which is used as the key in an eBPF map of type `TASK_STORAGE` to lookup the policy identifier. This information is used as an address within an eBPF map

of type `ARRAY_OF_MAPS`, and permits to retrieve an inner `HASH` map containing the ambient rights. Loading eBPF maps and programs is a privileged operation that requires the `CAP_BPF`, `CAP_PERFMON`, `CAP_NET_ADMIN` capabilities to be performed, thus we grant the JS runtime executable the corresponding Linux file capabilities [48]. After the completion of these steps, the capabilities are no longer necessary, hence they are dropped. This satisfies the requirement that root permissions at runtime are not needed for the activation of security contexts.

The sandboxer is also responsible for the creation of the security contexts where native code will be executed at runtime. Each context is an OS thread with permissions restricted by Landlock, Seccomp, and eBPF. To avoid paying the performance cost to instantiate each security context during the invocation of executables and shared libraries, we modified the JS runtime to allocate a thread per security context defined by the policy, restrict their permissions, and then, park them in a context pool (ⓐ, ⓑ, ⓒ). With Landlock and Seccomp, restriction of privileges is performed calling the corresponding syscalls from the specific context, while restriction of permissions based on eBPF is simply performed invoking the uprobe `attach_policy` reported in Table 5.1a. As a result, the `task_struct` identifier of a given context is annotated in the dedicated eBPF map along with the associated policy identifier. This design choice allows to reuse security contexts, thus minimizing latency, which, as highlighted in the objectives (Section 5.4.1), is a critical metric for web applications.

Application runtime After the web application is started, two operations can lead to the execution of native code: (i) the execution of a binary program in a subprocess, and (ii) the invocation of a shared library. NatiSand intercepts all the requests originating from the web application that require to execute native code ①, and leverages the sandboxer to assign them to the proper pre-allocated isolated context ②. Based on the type of request, a dedicated task inheriting the selected security context is launched and used to execute the native code ③. Specifically, when there is a request to run an executable, the JS runtime forks a process. On the other hand, when a shared library should be loaded, a thread is spawned. The consequence is that any request to access filesystem, IPC, and network resources will be subject to the restrictions imposed by the LSMs (④ⓐ, ④ⓑ, ④ⓒ). The approach implemented by NatiSand ensures that native code is never loaded nor executed in a task running unconstrained, thus strengthening the boundary between the web application and the OS.

Context lifecycle
uprobe/attach_policy
tp_btf/sched_process_fork
tp_btf/sched_process_exit

(a)

	Access control
IPC	fentry/fifo_open
	lsm/socket_bind
	lsm/socket_connect
Network	lsm/socket_bind
	lsm/socket_create
	lsm/socket_connect

(b)

Table 5.1: Hooks and tracepoints monitored by NatiSand

5.4.4 Isolation features

Native code executed in isolated contexts can vary from library functions to entire programs. In the following we detail how NatiSand enforces isolation and summarize the sandboxing features.

Policy inheritance While Landlock and Seccomp guarantee policy inheritance after a `clone` syscall is performed, the eBPF map that tracks the restricted contexts must be updated explicitly. To this end, NatiSand relies on the eBPF tracing programs that are loaded into the kernel during the bootstrap phase and are attached to the `fork` and `exit` tracepoints reported in Table 5.1a. Whenever a security context allocates a new task with a fork operation, the tracing program registers a new entry into the map of restricted contexts. The entry maps the task identifier of the child to the policy identifier associated with the parent. When instead a context terminates its duties and issues an exit, its task identifier is deleted from the map. No intervention by the developer is required, as policy inheritance is transparently handled by our solution.

Filesystem NatiSand restricts access to the filesystem using Landlock. The sandbox enforces a straightforward `read`, `write`, `exec` (RWX) permission model, specified with three allow-list vectors (e.g., lines 5, 6, and 7 in Listing 5.1). After the security context has been activated, the available permissions can only be further restricted.

IPC To explain the isolation features NatiSand provides, we start with a description of how programs and libraries generally use IPC. Native programs often rely on parallelism and concurrency to achieve high resource utilization. Parallel execution typically requires to handle synchronization and communication between a parent and a group of child tasks. In this setting, best practice suggests to provide the

children with the necessary communication channels through the inheritance properties of the `clone` syscall [118]. For instance, when two programs are piped in the Bash shell, an IPC mechanism, in the form of a pipe, is created by the shell process and is inherited by the two child programs, so that the latter can read the output from the former. Similarly, a parent and a child task can leverage an unnamed UNIX socket pair to share messages [119]. These use cases do not pose a significant security risk, since (i) the communication happens between tasks associated with the same security context, and (ii) the IPC channels used to communicate are not visible to other services running on the host OS. Conversely, CVE-2020-16125 and CVE-2021-3560 demonstrate that uncontrolled interaction with globally available IPC channels used by other services can lead to concrete security problems.

To block the communication between components associated with incompatible security contexts, NatiSand by default denies IPC over globally visible communication mechanisms. In this category there are fifo (i.e., named pipes), message queues, named semaphores, non-private shared memory, signals, and UNIX named sockets. Many of these mechanisms can be fully blocked by denying access to the related system calls, but in some cases the evaluation of syscall configuration flags is necessary. For instance, the creation of shared memory maps is permitted by the sandbox only when the `mmap` syscall is invoked with `MAP_ANONYMOUS` or `MAP_PRIVATE`. Similarly, the creation of named special files is allowed only when the `mknod` and `mknodat` syscalls are not invoked with `S_IFIFO` and `S_IFSOCK`. Syscall filtering based on configuration flags is performed efficiently by NatiSand using Seccomp. However, the information available to Seccomp is not always sufficient to make the access decision. This is the case for the `bind`, `connect`, `open`, and `openat` syscalls. Indeed, information about the type of socket referenced for the `bind` and `connect` syscalls resides in user memory, and unfortunately Seccomp cannot safely dereference it (due to TOCTOU risks [74]). Likewise, the `open` and `openat` syscalls do not represent the type of file to be opened through configuration flags, so Seccomp cannot handle the specific case properly. To solve these problems NatiSand relies on the eBPF programs attached to the hooks reported in Table 5.1b (which are not affected by TOCTOU issues, since they operate on arguments that were previously deep copied by the kernel). In particular, in the case of UNIX sockets the programs are attached to the `lsm/socket_bind` and `lsm/socket_connect` hooks, while for fifo files the kernel function `fifo_open` is used. A summary of the IPC mechanisms controlled by NatiSand, along with the LSMs leveraged to perform the security checks, is reported in Table 5.2.

Network NatiSand permits to control how each isolated context connects to network resources. In detail, it permits to completely revoke access to the network, to

connect only to a restricted list of hosts, and when needed, to use the network without restrictions. The sandboxer relies on eBPF programs to enforce permissions. The programs restrict the ability to `create`, `connect`, and `bind` sockets, and are thereby attached to the LSM hooks reported in Table 5.1b.

The creation of a socket opens a communication channel and returns a file descriptor as a result. By default, the sandboxer restricts the available communication domains to Internet Protocol (IP), denying applications the use of protocol families such as Bluetooth, Radio, VSOCK, and many more (this information is directly available from the arguments input to the `socket_create` interface). No restrictions are instead applied to UNIX domain sockets and the type of socket to be opened (e.g., stream, datagram). Opening a connection to a host is permitted only when the developer grants the isolated context to do so. The eBPF program that checks the opening of a connection first recovers the policy restricting the current security context, then uses it as a key to lookup the map of ambient rights from the `ARRAY_OF_MAPS` described in Section 5.4.3. The ambient rights map is an allow-list that stores the reachable (i.e., policy allowed) hosts, hence the security check is carried out with a lookup. Each network resource is uniquely identified by its IP address and port. Internally, we use the value zero for the port to represent the permission of opening a connection to a given host on every port.

Up to now, we have discussed the restrictions when the application connects as a client to a service. However, web applications frequently need to serve incoming requests. To do so, it is necessary to assign a "name" to a socket – i.e., configuring its address. This operation is done with the `bind` syscall, and we decided to permit it only when the policy gives the current security context access to the corresponding address and port pair. Again, the value zero for the port is used as a placeholder to allow binding on every port. On the other hand, no restrictions are applied to the `listen` and `accept` syscalls. Listen only marks a socket as passive, meaning that it will be used to accept incoming requests. However, no connection to a socket can happen if an address was not previously assigned to it [117]. The same applies to `accept`, which is used to extract the first connection request from the queue of pending connections [115].

Limitations While NatiSand significantly restricts the set of permissions and system resources associated with subprocesses and shared libraries, it provides strong memory isolation guarantees only when executables are run, as each subprocess is executed within its own address space. On the other hand, shared libraries are loaded within the hosting thread address space, hence a native library bug can impact the web application memory. Several research works have studied this problem and have proposed countermeasures [132, 106, 201, 24, 189, 41]. In general, these works are

IPC	Subclass	Linux system call	Seccomp	eBPF
Message queue	POSIX	mq_open, mq_getsetattr, mq_notify, mq_timedreceive, mq_timedsend, mq_unlink	✓	
	System V	msgctl, msgget, msgrcv, msgsnd	✓	
Pipe	Named	mknod, mknodat, open, openat	✓*	✓
Semaphore	POSIX	futex, mmap	✓*	
	System V	semctl, semget, semop, semtimedop	✓	
Shared memory	POSIX	mmap	✓*	
	System V	shmat, shmctl, shmdt, shmget	✓	
Signal	Standard	kill, pidfd_send_signal, tgkill, tkill	✓	
	Real-time	rt_sigqueueinfo, rt_tgsigqueueinfo	✓	
UNIX socket	Named	bind, connect, mknod, mknodat	✓*	✓

Table 5.2: LSMs used by NatiSand to restrict Linux IPC. The checkmark ✓* indicates when Seccomp needs to evaluate the syscall configuration flags to make the access decision

compatible with the design of our solution, therefore they could be used in conjuction with NatiSand to improve isolation of shared libraries. Among them, BreakApp [189] and BinWrap [41] propose approaches tailored for interpreted languages, but they require either the introduction of wrappers or the execution of remote procedure calls. RLBox [132] provides strong guarantees against memory corruption, but it demands the developer to manually retrofit existing code, a process that can take up to "few days" for each library according to the authors. Improved intra-process isolation can also be achieved leveraging dedicated hardware features like Intel Protection Keys for Userspace (PKU), but as demonstrated by PKU Pitfalls [46] these solutions can be bypassed using kernel functions that are agnostic of intra-process isolation (i.e., the attacks use the kernel as a confused deputy). Since NatiSand is completely aligned with the memory isolation assumptions made by the kernel, our solution is not affected by this issue.

5.5 Policy

In this section we present the structure of the policy file, then we explain how to generate the permission rules.

5.5.1 Policy structure

JS applications executed by NatiSand are associated with a policy file. The policy must be provided by the developer before the application is run, and to this end, a CLI flag (e.g., `native-sandbox`) needs to be added to the JS runtime. The policy file is formatted in JSON, with the following structure: a policy defines an array

of objects and each object details the permissions available to a security context. Within each object, a *name* is used to identify the context, a *type* indicates whether the context applies to an executable, a library, or a function of a library; the sections *fs*, *ipc* and *net* are used to configure the corresponding permissions. The structure of the objects is flexible, and only a *name* is required to configure a valid context. As the policy follows a *default-deny* model, a context that specifies only its name has no permissions at runtime. An excerpt from a policy file is shown in Listing 5.1 (the complete example is reported in Appendix C), while a summary of the most relevant policy features is described next.

Name, Type

The name and type elements are used by NatiSand to determine which policy context must be enforced. The type element can be set to `executable` (the default value), `library` or `function`. At runtime NatiSand extracts the absolute path of the native program and function name, and based on the information available, it identifies the most selective entry in the policy. This gives the developer the flexibility to use different policies in case binaries and libraries have the same basename, or when different functions from the same library are invoked. Moreover, since absolute paths are used, this approach ensures symlinks cannot trick the lookup of the security context. Listing 5.1 shows a policy that is enforced every time the application runs the curl program.

Fs

The fs element is used to configure filesystem-related permissions. Fs stores three optional arrays: *read*, *write* and *exec*. Filesystem paths are used as array values. As an example, the context detailed in Listing 5.1 can read and execute the curl binary, and write to `response.json` in the current working directory of the web application. In case the developer wants to operate with a coarser granularity, the value `true` can be used to replace any of the *fs*, *read*, *write* and *exec* arrays to grant access to the whole filesystem.

Ipc

To restrict IPC access we decided to expose developers a simple interface where flags can be turned on and off based on their needs. Six optional flags are available in the policy: *fifo*, *message*, *semaphore*, *shmem*, *signal*, and *socket*. For example, in Listing 5.1, curl is allowed to use abstract, named, and unnamed Unix sockets. It is up to our sandboxer to abstract away the complexity of the underlying architecture

Listing 5.1: Example of JSON policy file with single context

```
[{
   "name": "/usr/bin/curl",
   "type": "executable",
   "fs": {
            "read":   ["/usr/bin/curl", ...],
            "write":  ["response.json"],
            "exec":   ["/usr/bin/curl", ...]
   },
   "ipc": {
            "socket": true,
   },
   "net": [{
            "name":   "https://www.example.com",
            "ports":  [443]
   }]
}]
```

and enforce the policy when IPC is performed between groups of threads associated with separate contexts. No understanding of the standards available (e.g., System V, POSIX) is required by developers to restrict the permissions associated with their application. Similarly to the filesystem case, the developer can use a coarser granularity by setting the *ipc* element to `true`, enabling all communication mechanisms. Notice that globally available IPC channels are often bound to filesystem resources, so, while the granularity of the six flags described above may seem coarse, finer-grained permissions can be specified leveraging the path associated with the IPC resource. For example, the developer can restrict the use of a specific named pipe (pinned to the filesystem) by using its fully qualified path.

Net

Web application developers are often interested in restricting the hosts an application can connect to. The policy permits to specify an array of reachable hosts. Each host is fully qualified by its URL/IP, and the sequence of permitted ports. As in the case of the filesystem, the policy permits to grant access to the network without limitations (setting *net* to `true`), enable all the ports for a specific host (setting *ports* to `true`), or completely remove access to the network (leveraging the default-deny behavior). In Listing 5.1, the process executing curl is only allowed to connect to `https://www.example.com` on port 443.

5.5.2 Policy generation

While designing NatiSand we opted for a minimal and easy to understand policy syntax to target a broad spectrum of users. However, writing a policy for large components may be a tedious and tricky task, since we do not expect all the developers to be aware of how binaries and external libraries used by their web application work internally. To assist the developer, we follow an approach similar to Slim-Toolkit [164], where a service is run against a test suite to generate the security profile. Specifically, we developed a CLI utility written in Go that provides generation of policy templates the developer can understand, modify, and audit. The utility persists policy-relevant information in a SQLite database, and exposes to the user many functions that permit to configure multiple contexts, merge the results collected from multiple tests, and refine policies interactively. In the following we provide details on the work we performed for each of the protected subsystems.

Filesystem

The automatic retrieval of the dependencies of a binary is a well-known problem. Our utility is capable to discover the dependencies of programs installed as ELF files, and programs that are spawned by dedicated POSIX or shell wrappers. The utility first uses *ldd* [113] to discover the direct and transient dependencies, then, it relies on *strace* [114] to monitor the interaction between the kernel and a traced binary, so to complement the information previously found with additional filesystem permissions.

IPC

NatiSand adopts a policy language that abstracts away IPC complexity. Our utility supports policy generation by analyzing the results of multiple test cases where the Seccomp filter and eBPF programs of the sandboxer are set to *auditing mode*. These programs return the flags to be enabled.

Network

Network rules are relatively easy to write. However, we do not assume developers to be necessarily aware of every network connection needed by the native code. So, we automate the generation of the policy by observing the execution of the binary with eBPF programs. In fact, these provide a list of the domain names resolved, their IPs, and those hardcoded IPs the utility connects to without performing name resolution. To track domain name resolutions we attach a `uprobe` to the `getaddrinfo` function of the libc library, for IPs we observe network socket connections using `kprobes` on

`socket_bind` and `socket_connect` LSM hooks. To handle IP address migrations that may occur at runtime, we similarly propose to capture the list of IPs returned by the `getaddrinfo` with a dedicated eBPF program, which also updates the eBPF map of allowed hosts accordingly. By doing so we make sure that the security checks reflect the policy. This approach can also be extended monitorng DNS traffic on port 53, hence providing support for native components that do not rely on libc functions.

Policy generation is subject to limitations: (i) policy generation for malicious code produces overly permissive policy and obviously cannot be trusted, (ii) test suite with limited coverage might provide overly strict policies not allowing the execution of legitimate code. Overall, the correctness of the policy depends on the test suite used to collect the permissions. The closer the tests align with the use of the native utility in production, the more the developer can consider the policy generated effective. Nonetheless, to have complete assurance that the policy generated is correct, an auditing process may be required.

5.6 Case study: Deno runtime

There are three well-known alternatives for the execution of JS code on the backend, namely Node.js, Deno, and Bun. As highlighted in Section 5.2.1, their architectures have strong similarities, and NatiSand is designed to be compatible with all of them (since no assumption is made on specific runtime components). Nevertheless the integration is not trivial, and it requires significant engineering effort, therefore we integrated NatiSand into only one of them to demonstrate the achievement of the set objectives (Section 5.4.1). In this section we explain our decision, then we highlight the main architectural changes.

5.6.1 Runtime selection

Considering (i) popularity among web developers, (ii) availability and support of third-party modules, and (iii) security-oriented features provided by the runtime, we selected Deno. Bun is the latest runtime released, and thus currently the least used by developers. Node.js is nowadays the most widely used platform, and it offers developers the largest collection of open source packages. However, Node.js by default does not prevent JS applications to access system resources, and although it recently introduced a module-based permission model, the feature is experimental [145]. Deno was instead designed with the protection of the host as one of its main goals [153], thus no access to privileged system resources is given to JS

applications unless the developer explicitly grants it. Deno provides the *Node Compatibility Mode* [65], a feature enabling the reuse of code and libraries originally built for Node.js. The availability of this function permits to import packages hosted by Deno on `deno.land/x`, as well as modules published to npm. To conclude, Node.js and Deno prevail over Bun on popularity and security-oriented features. Node.js wins over Deno on popularity (but Deno is quickly growing), they are comparable in terms of third-party modules, and Deno significantly outperforms Node.js on security oriented-features, leading us to choose Deno.

5.6.2 Deno integration

Deno has a modular architecture organized into components. Three of them are particularly important for NatiSand: (i) *rusty_v8*, the package that bridges Deno and the V8 engine implementing the set of bindings to the V8's C++ API, (ii) *deno_core*, which leverages *rusty_v8* to expose the interfaces provided by Deno to the JS application, and (iii) *deno*, which defines the runtime executable together with the Command Line Interface.

Bootstrap Shortly after the Deno executable is run, the *deno* component is used to read the permissions granted by the developer via CLI. We extended this stage to read and parse the policy file specified with the new `native-sandbox` flag, then we added the permissions associated with each security context to the *global state* stored by the runtime. To complete the bootstrap phase, we also integrated the steps to load the necessary eBPF programs and to initialize the pool of isolated contexts, as explained in Section 5.4.3.

Application runtime After the JS application is started, the function calls that cannot be directly handled by V8 are routed to the Deno runtime through the bindings defined by *rusty_v8*. Each of them is associated with the *op_code*, a unique code identifying the operation to be performed. The *deno_core* component receives such requests, it checks the permissions available from the global state, and serves them accordingly. We identified requests that require to execute native code (e.g., `command`, `dlopen`, `run`), and modified *deno_core* so that they are restricted by NatiSand. The *op_code* along with the arguments are used to select the proper security context.

5.6.3 Support to fast JS calls

In October 2020 V8 announced the support to fast JS calls [184]. The function allows V8 to directly invoke optimized native functions without leveraging the bindings that

Listing 5.2: Code sandboxing with `nativeCall()`

```
function query(db, stmt) {
    const sqliteDB = new sqlite3.Database(db);
    const query = sqliteDB.prepare(stmt);
    const tuples = query.all();
    sqlite_db.close();
    return tuples;
}

const db = "database.db";
const stmt = "SELECT * FROM table";
const ts = Deno.nativeCall(query, [db, stmt]);
console.log(ts); // print tuples
```

connect V8 and the embedder (e.g., JS runtime). This permits to obtain substantial performance gains, since native function calls can be resolved in nanoseconds.

Deno has introduced unstable support to fast JS calls in July 2022 [58]. The change affected the implementation of the *dlopen* API, which is now able to generate an optimized and a fallback (i.e., standard) execution path for native functions. The optimized path is triggered only when V8 is actually able to optimize a symbol, and it entails the execution of code leveraging the fast call interface. While the optimized path is associated with minimum overhead, from a security perspective it permits the web application to execute native code without the mediation of the JS runtime, invalidating the security reference monitor of Deno. In our prototype we address this Deno security issue offering developers two alternatives: (i) turn off the fast call support and safely rely on the execution of sandboxed native functions with NatiSand without any code change, and (ii) enable insecure fast calls but allow to select the JS functions that need to be isolated with minimal code changes. The second option permits to take advantage of fast calls when performance is critical and risks are limited (e.g., arithmetic operations), and at the same time benefit from the security features NatiSand provides. To this end, we introduced a new API named `Deno.nativeCall()`. The API receives, as first argument, the name of the function to be sandboxed, along with the list of its arguments. Listing 5.2 shows how to sandbox the functions from the native database driver *sqlite3*.

5.7 Experiments

NatiSand must satisfy two properties to be practical: (i) it must mitigate real-world vulnerabilities by blocking the associated exploits, and (ii) it must introduce a limited overhead compared to a scenario where no protection is applied. In the

experimental evaluation, we first show our solution is able to protect web applications relying on binary programs and shared libraries affected by high severity vulnerabilities (Section 5.7.1), then we investigate the performance of our approach (Section 5.7.2). Both tests use a server with Ubuntu 22.04 LTS, an AMD Ryzen 3900X CPU, 64 GB RAM, and 2 TB SSD.

5.7.1 Exploit mitigation

To conduct our analysis we built a representative sample of vulnerabilities targeting executables and libraries widely used in web applications. We identified the 32 CVEs reported in Table 5.3. The entries are separated into three classes: Arbitrary Code Execution (ACE), Arbitrary File Overwrite (AFO), and Local File Inclusion (LFI). The list of vulnerable utilities includes programs used to compress files (e.g., GNU Tar, RAR, Zip), to process multimedia (e.g., FFmpeg, GraphicsMagick, ImageMagick), database drivers (e.g., SQLite), and also Machine Learning libraries (e.g., Lightning, Sockeye, TensorFlow). We highlight that the vulnerabilities affect popular open source modules with 2.6M downloads/week available from the npm and `deno.land/x` archives. Concrete examples are sharp and fluent-ffmpeg from npm, or flat and sqlite from `deno.land/x`.

First, we checked that public Proofs of Concept of the CVEs in Table 5.3 successfully exploit the vulnerable version of the utilities. Then, we analyzed whether the vulnerabilities were exploitable sending the malicious payload through the JS module interface, and confirmed the feasibility of the attack. The *Node compatibility mode* was leveraged to execute in Deno the modules downloaded from npm. We finally repeated the experiment activating the security functions provided by NatiSand, and verified that the attack was no longer successful, while the application was still able to serve benign requests (i.e., no functionality loss). The only change we introduced in the experiment was the specification of a security policy through the `native-sandbox` CLI argument. The policy was generated using the tool described in Section 5.5.2. No modification to the web application, nor its dependencies, was required to benefit from the new sandboxing capabilities.

From a security perspective it is worth mentioning that NatiSand can mitigate attacks at multiple levels. For instance, in CVE-2022-2566 a heap out-of-bound memory bug exists in FFmpeg. The goal of the attacker is to achieve Arbitrary Code Execution sending to the web application a malicious `MP4` payload. NatiSand denies the compromised component attempts to access confidential files, open reverse shells, interact with privileged services through IPC, and transfer data to unauthorized network hosts. We point out that, while sandboxing limits the privileges an attacker can gain from exploiting a vulnerable program, it cannot eliminate vulnerabilities,

Class	CVE Id	Utility	Type	Use case
ACE	CVE-2016–3714	ImageMagick	bin	Image processing
	CVE-2019-5063	OpenCV	lib	Computer Vision
	CVE-2019-5064	OpenCV	lib	Computer Vision
	CVE-2020-6016	GNSockets	lib	P2P networking
	CVE-2020-6017	GNSockets	lib	P2P networking
	CVE-2020-6018	GNSockets	lib	P2P networking
	CVE-2020-17541	libjpeg-turbo	lib	Compress image
	CVE-2020-24020	FFmpeg	lib	Video processing
	CVE-2020-24995	FFmpeg	lib	Video processing
	CVE-2020-29599	ImageMagick	bin	Image processing
	CVE-2021-3246	libsndfile	lib	Audio encoding
	CVE-2021-3781	Ghostscript	bin	PDF processing
	CVE-2021-4118	Lightning	lib	Machine learning
	CVE-2021-20227	SQLite	lib	Query database
	CVE-2021-21300	Git	bin	Clone repository
	CVE-2021-22204	ExifTool	bin	Extract metadata
	CVE-2021-37678	TensorFlow	lib	Machine learning
	CVE-2021-43811	Sockeye	lib	Translation
	CVE-2022-0529	Unzip	bin	Decompress archive
	CVE-2022-0530	Unzip	bin	Decompress archive
	CVE-2022-0845	Lightning	lib	Machine learning
	CVE-2022-1292	OpenSSL	bin	Verify certificate
	CVE-2022-2068	OpenSSL	bin	Verify certificate
	CVE-2022-2274	OpenSSL	lib	Cryptography
	CVE-2022-2566	FFmpeg	bin	Video processing
AFO	CVE-2016-6321	GNU Tar	bin	Decompress archive
	CVE-2017-1000472	POCO	lib	Common libraries
	CVE-2019-20916	Pip	bin	Dependency fetch
	CVE-2022-30333	UnRAR	bin	Decompress archive
LFI	CVE-2016-1897	FFmpeg	bin	Video processing
	CVE-2016-1898	FFmpeg	bin	Video processing
	CVE-2019-12921	GraphicsMagick	bin	Image processing

Table 5.3: Sample of CVEs mitigated by NatiSand

nor it can make infeasible to use them in an exploit chain.

5.7.2 Performance evaluation

To assess the performance of NatiSand we considered a broad set of programs, including several GNU Core Utilities, executables to process multimedia, database drivers, and Object Character Recognition engines. The goal is twofold: (i) evaluate the slowdown compared to a scenario where no protection is available (i.e.,

regular Deno), and (ii) compare NatiSand with well known sandboxing and isolation frameworks. In the following we first investigate the impact on executables, then we analyze libraries.

Executables

In the first batch of experiments we analyze the overhead associated with executables. Compared to the default scenario where no protection is available, NatiSand spawns each program in a dedicated subprocess with its own set of constrained ambient rights. A handful of general purpose sandboxers can be adopted to achieve a comparable degree of protection by wrapping the execution of each subprocess with the chosen sandboxing utility. In our evaluation, we considered *Minijail* [87] and *Sandbox2* [88]. Minijail is a tool used in ChromeOS and Android to launch and sandbox other programs based on the set of arguments specified, while Sandbox2 is a C++ library written by Google that can be used to sandbox entire programs or portions of them. Both Minijail and Sandbox2 support multiple containment techniques, such as the introduction of dedicated user ids, restriction of the Linux capabilities, introduction of policy-based Seccomp filters, and isolation based on Linux namespaces.

Benchmark I In the first benchmark we implemented a JS application to test the execution of 17 common Linux utilities with four configurations: Deno, NatiSand, Minijail, and Sandbox2. The application uses `Deno.run()` to spawn each utility in a subprocess, and it leverages `Deno.bench()` to determine the duration of each request. The function ensures that each measure is statistically robust, as it automatically performs a dynamic number of rounds based on the duration of the test (i.e., the shorter the test duration, the higher the number of repetitions). The results are shown in Table 5.4 (tests are ordered by increasing execution time). As expected, the cost of activating the sandbox is amortized with the increase in the test duration. The tests also show that NatiSand suffers from a smaller performance degradation compared to Minijail and Sandbox2. This aspect is particularly evident for short-lived utilities. The reason is that our approach is integrated by design and, contrary to the other solutions, leverages lightweight technologies that introduce a smaller performance footprint.

Benchmark II While the experiments part of Benchmark I focus on the server side scenario, with Benchmark II we wanted to show the overhead experienced by a remote client. To this end, we used three microservices, each representing a real use case scenario of high performance native programs. Two microservices rely on

Utility	Deno [ms]	Minijail	Sandbox2	NatiSand
b2sum	2.37	7.19x	9.37x	**2.88x**
cut	2.52	7.11x	8.97x	**2.86x**
sum	2.61	7.00x	8.25x	**2.87x**
tac	2.76	6.51x	8.21x	**2.34x**
wc	2.97	6.25x	7.69x	**2.44x**
dd	3.60	5.29x	6.26x	**2.23x**
seq	3.80	5.02x	5.96x	**2.13x**
shuf	4.29	4.68x	5.55x	**2.17x**
ls	4.75	3.72x	4.68x	**1.76x**
factor	5.03	4.06x	5.03x	**1.86x**
join	5.20	4.08x	5.18x	**2.05x**
head	6.73	3.16x	3.85x	**1.56x**
ping	12.20	2.27x	2.79x	**1.47x**
sort	14.37	1.44x	1.77x	**1.43x**
dig	22.14	1.71x	2.15x	**1.17x**
wget	53.24	1.18x	1.42x	**1.13x**
curl	81.27	1.23x	1.24x	**1.16x**

Table 5.4: Average execution time for common Linux utilities

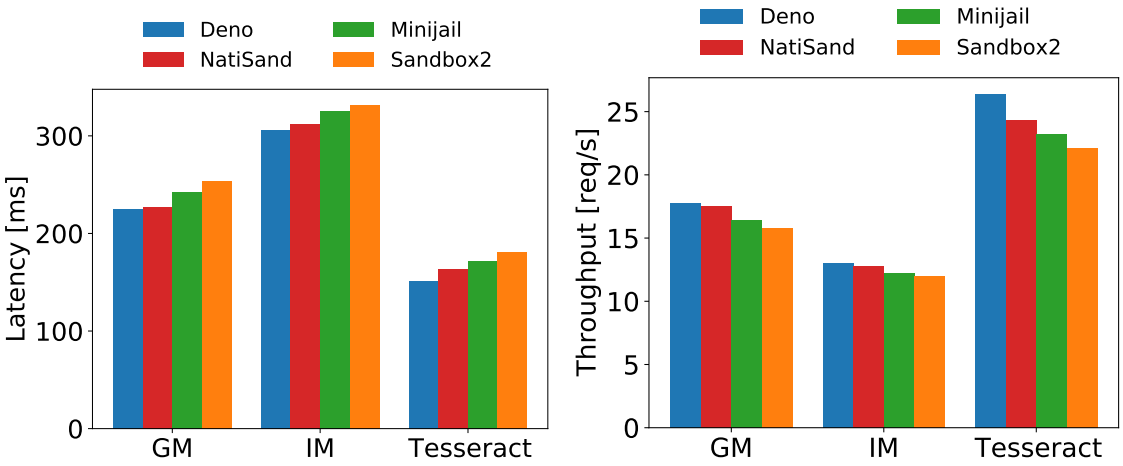


Figure 5.4: Average latency and throughput for microservices that execute subprocesses

GraphicsMagick and ImageMagick, to perform a *sharpen* operation on images input by the client, while the third microservice relies on Tesseract to perform Optical Character Recognition on a second sequence of images input by the client. Similarly to the previous case, the test was repeated for each of the four configurations: Deno, NatiSand, Minijail, and Sandbox2. This time the HTTP benchmarking tool *wrk* was used to measure the performance of each microservice. Network bandwidth and latency are 1 Gbps and 10 ms, respectively, while 100 warmup requests were carried out. Figure 5.4 shows the average latency and the throughput observed

over a period of 30 seconds. The results once again confirm the previous analysis, as longer durations make the cost to setup the native sandbox less relevant. It is worth to mention that NatiSand exhibits lower overhead compared to Minijail and Sandbox2, with approximately 5 to 10 ms less latency for each microservice.

Usability Although general purpose sandboxers can be used to restrict the permissions associated with executables, to provide a protection comparable to NatiSand: (i) they force the developer to introduce changes in the web application, and (ii) they require to understand in depth the techniques used by the kernel to restrict ambient rights (e.g., capabilities, namespaces, Seccomp filters). Another problem is that to restrict IPC and network with Minijail and Sandbox2 it is necessary to leverage namespaces, which are characterized by coarser granularity than NatiSand policies.

Libraries

In the second batch of experiments we analyze the overhead associated with libraries. Contrary to the default JS runtime behavior, NatiSand transparently executes native library functions in dedicated contexts with limited ambient rights. A modern, valuable alternative approach to isolate libraries is to compile them to WebAssembly (Wasm), a standardized, portable binary instruction format executed in a memory safe, sandboxed environment. This approach has gained considerable attention recently, as browsers such as Firefox have used it to retrofit some of their components to safely interface with native libraries [132].

Benchmark III Similarly to Benchmark I, we implemented a JS application to highlight the overhead experienced on the server when native libraries are executed. In this case three configurations are evaluated: Deno, NatiSand, and Wasm. The application tests the operations provided by four popular libraries: (i) libxml2, to open and query XML data, (ii) libpng, to read metadata information and verify the signature of a png image, (iii) opus to encode and create an audio trace, and (iv) sqlite3, to open and query the Northwind database. Test durations were again measured with `Deno.bench()`, and the results are reported in Table 5.5. Deno exhibits a consistent performance advantage for operations that require up to 30 microseconds. However, NatiSand proves to be more efficient than Wasm, which in turn is affected by a substantial overhead in almost every test. This difference is due to the nature of Wasm; while there have been improvements, the just-in-time compiled language [186] remains slower than its native counterpart. Remarkable are the cases of opus and sqlite3, which used `nativeCall` and demonstrate its efficiency.

Test	Deno [μs]	Wasm	NatiSand
libxml2 (open)	9.33	8.96x	**2.51x**
libxml2 (query)	11.53	4.35x	**1.63x**
libpng (verify)	11.58	13.34x	**9.61x**
libpng (info)	28.33	12.63x	**9.39x**
opus (encode)	58.67	2.03x	**1.55x**
opus (create)	203.72	1.70x	**1.64x**
sqlite3 (open)	63.62	5.68x	**1.54x**
sqlite3* (query)	143.98	2.43x	**1.51x**

Table 5.5: Average execution time for common native libraries (* marks the use of `nativeCall`)

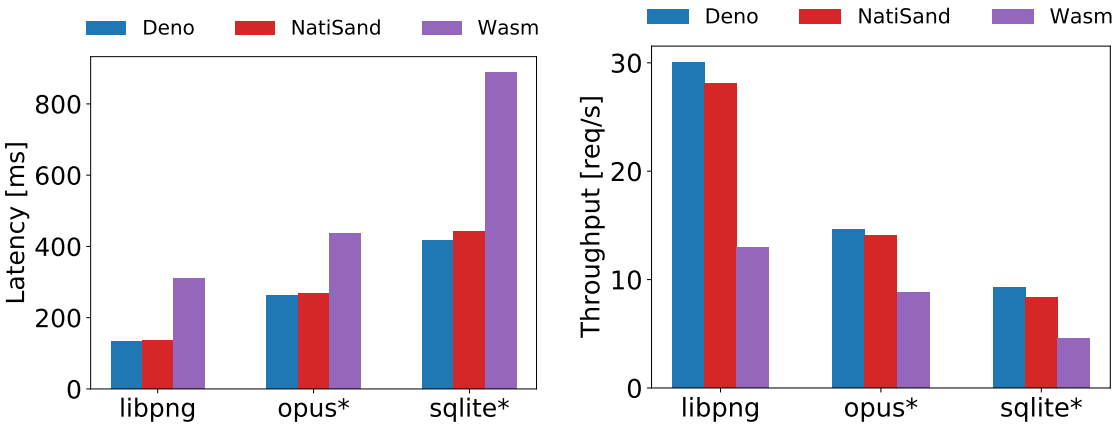


Figure 5.5: Average latency and throughput for microservices that execute native functions (* marks the use of `nativeCall`)

Benchmark IV To understand the slowdown perceived by a remote client, we exposed the functions of the libpng, opus, and sqlite3 libraries with microservices. For each of them, we configured the client to send the input to the server, and measured the latency and throughput using *wrk* (as explained in Benchmark II setup). The results are visualized in Figure 5.5. Once again the client observes a small degradation of latency and throughput when using NatiSand instead of Deno, but the overhead is far less noticeable compared to the results discussed in Benchmark III. Conversely, Wasm is affected by a significant degradation of latency. This is due to the just-in-time compilation of Wasm, and the additional memory management required to exchange data between the JS application and Wasm.

Usability While Wasm offers strong isolation guarantees, it also comes with drawbacks compared to NatiSand. First of all it requires the developer to use a Wasm-compatible version of the library. In our evaluation we used a precompiled version of sqlite3, but we had to manually compile opus and libpng using the Emscripten

toolchain [75] and the WASI Sdk [197], respectively. Moreover, current implementations of the WebAssembly System Interface (WASI) can only restrict ambient rights programmatically, and filesystem privileges work at directory granularity. Lastly, Wasm requires the developer to explicitly allocate, write, and read bytes from the Wasm module linear memory.

5.8 Related work

Isolation of software has been widely investigated by both the academic and industrial communities [28, 105, 57, 87, 88, 134, 47, 157, 4]. *MBOX* [105] features an unprivileged sandboxing mechanism that prevents a process from modifying the host filesystem by layering the sandbox filesystem on top of it. The solution is implemented by interposing syscalls using Seccomp and *ptrace*. The use of *ptrace* required the authors careful attention to avoid the risk of TOCTOU attacks, moreover it suffers from non-negligible performance degradation. DeMarinis et al. [57] propose *Sysfilter*, a static analysis framework to reduce the attack surface of the kernel, by restricting with Seccomp the system call set available to processes. This approach proves to be effective in limiting the kernel APIs that can be abused by attackers, but whenever a system call is necessary for the benign behavior of a program, there is no way to control with Seccomp the specific instance of the resource used. *BPF-Box* [81], *BPFContain* [80], and *Snappy* [25] are security frameworks that provide confinement of processes and containers with the use of eBPF. These solutions highlight the benefit of eBPF by providing simple, efficient, and flexible confinement of system resources, however these solutions either require a privileged daemon or require to load kernel modules to introduce a set of *dynamic helpers*. Often, industrial sandboxing solutions take advantage of multiple protection techniques to support process containment. This is the case for *Minijail* [87], and *Sandbox2* [88]. Some of the security mechanisms used are: introduction of new user ids, capabilities restriction, namespace isolation and policy-based Seccomp filtering. These tools expose a powerful interface meant to be used by security experts. Similar considerations are shared with other less mature tools such as *Firejail* [134] and *Bubblewrap* [47].

Multiple research efforts have studied the use of third-party components in software products [132, 106, 201, 24]. *PKRU-Safe* [106] proposes an automated method for preventing the memory corruption of memory-safe languages due to the interaction with unsafe code. It leverages the compiler infrastructure to provide hardware-backed memory protection requiring changes to build files, dependencies, and code (in the form of code annotations). *Codejail* [201] provides partial isolation of libraries by spawning a new process and configuring the necessary communication

channels to support tight memory interactions with the main program. To support the change in the interactions without modifications to the library code it is necessary to write a wrapper library. *Cali* [24] is a compiler-assisted library isolation system that compartmentalizes libraries into their own process, and automates the configuration of the necessary communication channels by tracking data flow between the program and the library at link time. *RLBox* [132] is a framework to isolate libraries in lightweight sandboxes – i.e., process, Wasm. It facilitates the retrofitting of applications employing static information flow enforcement and dynamic checks expressed in the C++ type system. These solutions were designed for compiled languages, so while some of the concepts are portable to JS runtimes, the solutions are not easily adapted to this domain.

A recent study by Staicu et al. [170] highlights how the possibility to invoke native code from scripting languages undermines the security assumption of applications. They discuss a methodology to detect misuses of the native extension API and show how the exploit of these vulnerabilities in npm packages can lead to web applications compromise. Previous proposals [189, 202, 41] tackle this problem by providing solutions to isolate the execution of third-party modules. *Wolf at the Door* [202] reduces the risk associated with the installation of npm packages by mediating their install-time capabilities. It enforces complex user-defined policies by leveraging AppArmor, hence prohibiting unauthorized access to confidential files and connections using an LSM that currently cannot coexist with SELinux and SMACK. *BreakApp* [189] takes advantage of module boundaries to compartmentalize npm modules in accordance with a set of code annotations. Modules are isolated with software, process, or container isolation, and it is possible to configure the visibility of the application context available to external modules. Process and container isolation enable the protection of native code, however the specification of their permissions are beyond the scope of the proposal. *Cage4deno* [5] protects filesystem resources from subprocesses executed by JavaScript runtimes. *BinWrap* [41] separates the execution of third-party components from the rest of the application using distinct execution threads for different domains of trust. The main focus of the proposal is prohibiting arbitrary accesses to sensitive data stored in the memory of the JS runtime by leveraging Intel's MPK/PKU. NatiSand is complementary to the above solutions since our goal is to specify and enforce permissions on native code dependencies of web applications, rather than providing memory isolation for untrusted components.

Protecting JS code from being compromised is out of the scope of NatiSand, nonetheless, since proposals in this domain and ours both target the web development audience, our proposal shares some ideas with previous works in this domain [190, 8, 172, 142]. For instance, Ferreira et al. [78] propose a lightweight per-

mission system providing per-package on/off switches that limit access to Node.js core modules (e.g., child_process, fs, http). By doing so, it can prohibit access to subprocess, filesystem, and network resources for the JS code. Similarly, NatiSand takes care of protecting filesystem, IPC, and network resources, targeting native code. *Mir* [190] is a system preventing the compromise of the application by third-party modules with the enforcement of fine-grained RWX permissions on every field of every variable in the JS context. NatiSand adopts an equivalent permission model to contain native code when accessing filesystem resources. Another research work enforcing security boundaries stated in a policy is *SandTrap* [8]. The approach enforces fine-grained access control policies on cross-domain interactions between application code and the third-party modules. The creation of policy files described by the authors consists in running test suites to create a policy with acceptable static cross-domain interaction coverage. We adopt a similar approach in the policy generation of NatiSand. Note that, differently from our proposal, solutions protecting JS code can run in user space, thus they do not limit the portability of the JS runtime to Linux systems.

5.9 Conclusions

The increase in scale and complexity of modern web applications has led to the introduction of new security mechanisms in JS runtimes. Unfortunately, native code execution still represents a clear risk, since no isolation is provided by all the major platforms. NatiSand solves this problem, introducing new measures to confine the execution of binaries and shared libraries. The proposal is not dependent on a particular JS runtime, and was designed to be integrated into different architectures. Considerable attention was dedicated to usability; little effort is required by developers to sandbox their applications. Indeed, no specific security expertise is necessary to benefit from the protection, nor are changes to the application.

We believe that the approach proposed in this chapter can contribute to improve the state of the art in this domain and support the evolution toward more secure software platforms.

Availability

The source code and the artifacts produced to support the proposal are available open source at `https://github.com/unibg-seclab/natisand`

6. Lightweight Cloud Application Sandboxing

6.1 Introduction

The work done with Cage4Deno (Chapter 3), NatiSand (Chapter 5) and Wasm (Chapter 4) pointed out the necessity to support developers with the definition of correct and robust policies. For this reason we have further investigated this aspect and proposed a flexible way to automatically generate policies' templates and to enforce them with a lightweight performance impact. Indeed, this is a critical aspect since cloud applications are often built of many components, each implementing a specific set of business requirements. Components naturally evolve, their requirements may change, and as the overall scale of the application grows it can be challenging to ensure a high security standard. Many factors contribute to making this objective hard to achieve; the most common ones are: the presence of buggy components, the reliance on potentially vulnerable native libraries written with memory unsafe languages, and broad access to system resources.

Several research works have investigated the scenario (e.g., [170, 41, 6, 205]). For instance, Staicu et al. [170] explain the security problems that arise when web applications interact with native extensions. BinWrap [41] and NatiSand [6] analyze recent vulnerabilities and integrate new security measures in the Node.js and Deno runtimes to improve isolation and limit access to confidential resources. Lastly, Zimmermann et al. [205] present an extensive study of third-party dependencies, finding that large parts of the entire web ecosystem can be impacted by security issues even when individual packages are vulnerable or they include malicious code on purpose.

Recently, industry standards have also emerged to encourage organizations to proactively include new security measures. A notable example is the NIST SP 800-190 [135], which focuses on environments that adopt microservices, containers and Kubernetes. The production environment is given particular attention, with the goal of finding and stopping malicious threats in real time. The directive clearly indicates that there is a need for policies to defend against vulnerabilities that could lead to disruptions, such as modification of important files. The same regulation also provides instructions to prevent the tampering of the file system, prompting that applications and containers must be run with a set of permissions as minimal

as possible, namely following the *least privilege* principle.

Powered by the recent eBPF kernel technology, frameworks like Tetragon [181] and Falco [77] have been proposed to monitor cloud applications, identify unexpected security-relevant events, and act on them (e.g., denying access). While effective, they tend to introduce non-negligible overhead when fine-grained policy rules are used [177]. We argue that the combined use of classical operating system access control and sandboxing mechanisms may lead to a more resource efficient solution to isolate application components. For instance, the recent Landlock LSM [122] permits a process to restrict itself ensuring strong security guarantees with minimum performance footprint. Furthermore, the integration of Landlock, or an equivalent security mechanism, in cloud applications significantly mitigates the risk associated with the exploitation of a vulnerability, as the amount of resources available to a potential attacker is greatly reduced.

Unfortunately, to benefit from this protection developers must obtain a policy that clearly states the resources an application component must be granted access to, and the related permissions. This is far from trivial in the case of complex applications. Indeed, the list of resources can vary based on the production environment, or be subject to changes when different inputs are provided. All these reasons hold back the potential of sandboxing, and cloud applications may solely rely on the coarse isolation provided by the virtual machine or the container in which the application is executed.

Our contribution In this chapter we propose a new approach to systematically integrate fine-grained sandboxing in cloud applications that is aligned with the current regulations and best practices. In detail, we provide an intuitive, open-source solution to retrieve all the file system resources required by an application component, to build and customize least privilege policies. We then showcase how policies are used to sandbox programs with Landlock, mitigating the impact of severe CVEs. The approach we propose is flexible and does not depend on the toolchain leveraged to build each application component. The experiments showcase the minimal performance footprint at runtime, and highlight the ability to monitor the application without affecting its execution state.

6.2 Motivation

This Section explains the threat model and the motivation.

6.2.1 Threat model

We assume that the code part of the cloud application (including native code executed by it) is trusted and not malicious, but potentially affected by vulnerabilities due to bugs. In this scenario, an attacker may leverage web interfaces or programmatic APIs to send the application malicious payloads with the goal of exploiting such vulnerabilities. This attack vector may leave the application exposed to, e.g., arbitrary file read and write, file system compromise, and execution of arbitrary programs; all of which can lead to an inconsistent state of the application and its volumes. We aim at the creation of fine-grained, component-specific policies that are used by developers to gradually introduce sandboxing, mitigating the impact of vulnerabilities. This approach is complementary to the use of containers, as a compromised process running in a container can still damage all the resources available to it.

6.2.2 Dependency identification

An important use case for cloud applications is represented by services that handle media resources such as videos, photos, and audio. These applications typically rely on an extensive set of editing libraries and codecs to perform a number of operations like crop, scale, introduction of effects, format conversion, and compression. This software is usually available as dynamic libraries that are loaded and executed depending on the type of operation to be performed, on the input source, and on the hardware support available. A solution able to automatically collect all the resources used by a component for a set of test cases can significantly help the developer detecting and isolating the dependencies of the application.

6.2.3 Mitigation of bugs

Open source libraries and programs are used extensively while developing cloud applications. Examples include database drivers, media processing libraries, and encoding utilities. This software is often trusted, but it may be subject to vulnerabilities as explained in Section 6.2.1. When vulnerable third-party code is executed by the application, it can be targeted and compromised by an attacker who sends malicious payloads, as described in [10, 5, 78]. Popular cases are the Server-Side Request Forgery and Arbitrary File Read vulnerabilities found in FFmpeg and exploited against TikTok [95], and the Remote Command Execution vulnerability found in ExifTool and exploited against GitLab [51]. Other examples include: 1) CVE-2020-24020, CVE-2022-2566, CVE-2020-2499 associated with video processing software, 2) CVE-2022-1292, CVE-2022-2068, CVE-2022-2274 related to cryptographic soft-

ware, and 3) CVE-2021-4118, CVE-2021-37678, CVE-2022-0845 targeting machine learning software.

With our approach we aim to support the progressive introduction of fine-grained policies that are used to sandbox the application or any of its components, making harder for an attacker to tamper the file system, corrupt data, or exfiltrate sensitive information such as private keys or database entries.

6.2.4 Performance and usability

Modern cloud applications are often deployed as Kubernetes Pod instances and executed in one or more containers. Kubernetes provides several tools to improve the security and isolation of Pods. Relevant to this scenario are the support for RBAC policies and the availability of restricted Seccomp profiles. RBAC policies by default permit to define access of human users, however, they can also be used to govern the behavior of software resources through *service accounts*. Unfortunately, these policies can only restrict access to Kubernetes APIs, therefore they are not suitable for fine-grained restriction of permissions at an application component level. The same limitation is shared with Seccomp profiles, which can limit the kernel interface available to the cloud application, but are only applied at container level [180] and cannot operate depending on the specific requested resource [100].

Recent solutions based on eBPF, like Tetragon and Falco, enable the introduction of fine-grained policy rules to overcome the previous limitations. To this end, they load into the kernel dedicated filters which are run system-wide every time a security event like the opening of a file or the execution of a program occurs. Based on the content of the policy, and therefore the filters, these frameworks can grant or deny a particular action, effectively restricting the privileges available to a cloud application. However, as mentioned in Falco's documentation [177], the main problem associated with this approach is that performance overhead can have a large variability. This is mainly due to the fact that filters are evaluated every time a certain hook point (e.g., a syscall) is triggered, and that many hooks may need to be controlled to enforce a given security policy. Furthermore, these frameworks do not assist the developer in the generation of application-specific policies.

With our proposal we aim at giving the developer a complementary approach to secure applications and limit the file system resources accessible to them. Our idea is that the developer can benefit from the advantages brought by technologies like Landlock (i.e., strong security guarantees, low overhead), while at the same time rely on eBPF-based technologies, but only to monitor a restricted subset of important security events.

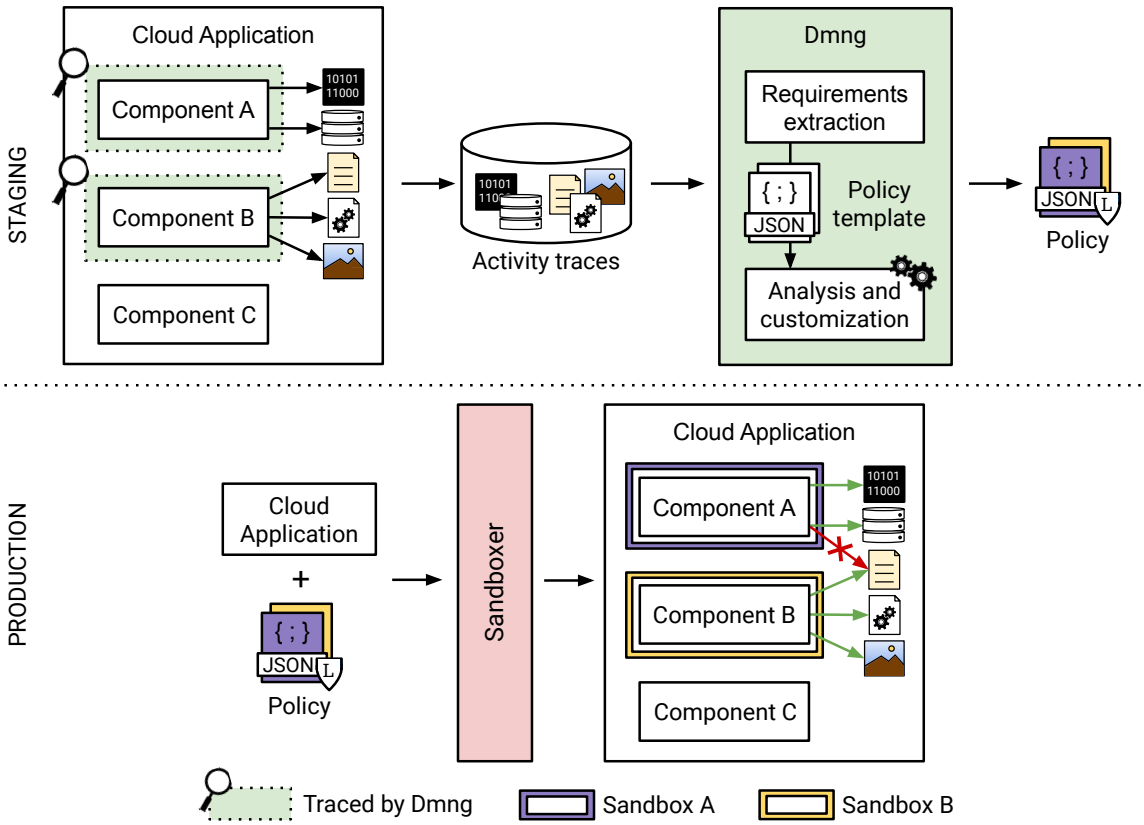


Figure 6.1: Approach overview: 1) Dmng uses probes to trace the application components A and B, 2) activity traces are saved into a SQLite DB, 3) requirements are extracted from the traces and used to build security policies, finally 4) policies are leveraged by the sandboxer to secure the application in production

6.3 Approach overview

Frequently, developers start building cloud applications from base container images, which are subsequently customized and extended with third-party software (e.g., web frameworks, database drivers, etc.). Once the application has been developed, it is released to the staging area, a replica of the production environment, not accessible from the outside, where developers and cloud architects can test new features so to detect design flaws and prevent unexpected errors to hit the production environment. The similarity to the production area makes it the best candidate to generate accurate least privilege policies to restrict the permissions available to the application.

To operate as intended, each application component requires access to system resources like programs, scripts, dynamic libraries, shared memory, and files. We simply call these resources *requirements*. In order to collect them, we provide an intuitive open source tool called Dmng that performs service instrumentation as

shown in Figure 6.1. In detail, the tool leverages ptrace and eBPF to setup and activate temporary probes that register the actions performed by an application component, making it possible to track file-opening requests, reading and writing of data, use of shared memory and execution of native code via subprocesses and shared libraries. All these requests are automatically registered into a database and subsequently used to generate policy templates. Together with the file system path, each requirement is associated with permissions. Compatibly with Unix-like systems, three permissions are available: *read*, *write*, and *exec*. The policy templates generated by Dmng can then be interactively modified by the developer with the addition, modification or removal of policy rules. After the changes are committed, the policy is serialized into a JSON file and it is ready to be used to sandbox the application.

Sandboxing can be introduced with several technologies and frameworks; given that we aim to restrict file system resources, we provide a sandboxing utility based on Landlock that parses the set of requirements needed by the application (i.e., path and permission pairs) from the JSON policy file generated by Dmng, and it restricts the permissions accordingly. We selected Landlock due to its outstanding performance and stackability property. Indeed, the use of Landlock allows us to be compatible with systems that already rely on other LSMs (e.g., AppArmor, SELinux). We highlight that, whenever two or more LSMs are available on the host, a single denial prevents the access to a resource (i.e., deny takes precedence).

After the cloud application has been deployed to the production environment it is important to ensure that services are running as expected, there are no anomalies, and proactive measures are taken to identify potential threats. By default, access to any file system resource not listed in the policy is blocked by Landlock. However, there may be cases in which the developer would like to generate reports on the set of requested resources by the application. To enable this, Dmng allows to temporarily observe and record the activity traces generated by any application component without changes to the application itself nor its execution state. These checks are not bypassable, which is a considerable advantage compared to alternative techniques that either rely on LD_PRELOAD or perform instrumentation through code dependency injection.

The following sections are organized as follows. Section 6.4 details the technologies used to support policy generation and implement monitoring. Section 6.5 clarifies the structure of the policy. Section 6.6 describes sandboxing through Landlock. Lastly, Section 6.7 showcases the mitigation capabilities and investigates the performance overhead.

6.4 Cloud application instrumentation

Our solution implements two methods to collect the requirements and generate the policy. The first uses ptrace and is based on syscall argument inspection, the second leverages eBPF, which dynamically extends the kernel attaching dedicated probes on relevant in-kernel file system-related events. Both the approaches are used to instrument the application, however, using ptrace significantly affects performance and thus is meant to be used only during staging. On the other hand, eBPF has lighter impact on performance, hence it can be used also in production.

6.4.1 Ptrace-based instrumentation

Ptrace is a functionality implemented by the kernel aimed at debuggers and code analysis tools that permits a process, called the *tracer*, to control and observe the activity performed by another process, the *tracee*. In our implementation, Dmng acts as the tracer for any application component. To this end, it prepares a parent and a child process as shown in Figure 6.2, and then uses the ptrace system call to instruct the kernel that the child will be traced by the parent (through the PTRACE_TRACEME request). After this step is completed, Dmng injects into the child process the component to be run, and then starts it. While being traced, every time an event occurs, the tracee is stopped by the kernel and a notification is sent to the tracer, which has the possibility to inspect and perform changes before the execution of the tracee is resumed. Dmng leverages ptrace to capture all file system-related syscalls, effectively monitoring the requests issued to the kernel by the component. The syscalls and their arguments are recorded by the tracer and saved to the SQLite database mentioned in Section 6.3. The set of monitored syscalls includes interfaces such as open, openat, creat, execve, link, linkat, mkdir, and the related permission flags (e.g., O_APPEND, O_CREAT, O_RDONLY). It is important to point out that Dmng automatically captures and monitors the possible children spawned by the tracee, and it is also capable to identify the set of dynamic libraries they depend upon.

When the developer wants to stop the tracing process, Dmng detaches itself from the tracing of the child process with the PTRACE_DETACH request and it terminates the child process by sending a SIGKILL signal.

6.4.2 eBPF-based intrumentation

Similarly to other recent security and observability frameworks, like Cilium [175], and Tetragon [181], Dmng relies on the eBPF technology to implement continuous

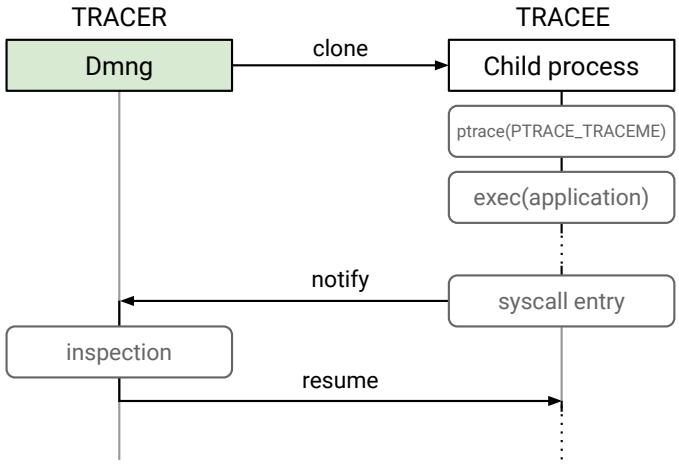


Figure 6.2: Dmng acts as a tracer for the application, inspecting the arguments of every syscall

monitoring. The eBPF subsytem allows to change at runtime the behavior of the kernel without changing its implementation nor adding new modules. Briefly, it permits to do so by loading compact programs within the kernel, which are evaluated (without preemption) by a virtual machine-like component every time a certain hook point is reached. There are many types of hook within the kernel, examples are network events, tracepoints, and LSM functions. To store data persistently between different eBPF program invocations and to share data between kernel and user space, data structures called *maps* are used. They provide abstractions such as arrays and hashmaps. It is important to mention that eBPF programs must be safe to run within the kernel and must not introduce bugs. To ensure these conditions are met, the eBPF susbsystem automatically performs the two stages of Program Verification and Just-In-Time Compilation at load time; only if both terminate without exceptions then the loading of the program is successful.

Dmng activates eBPF-based tracing on a given application component using its thread identifier. To this end, it leverages the *libbpf* [111] frontend to load into the kernel the eBPF programs and maps needed to perform tracing, and then starts collecting data. The process is shown in Figure 6.3. The set of eBPF programs comprises of: 1) dedicated programs to trace the application component lifetime, and 2) programs to monitor the file system-related events generated by the component. The former group of programs ensure that monitoring extends to tasks spawned through the clone system call by the component. Hence, they are attached to the *sched_process_fork* and *sched_process_exit* kernel tracepoints. Instead, the programs that record file system-related events are attached to hooks reported in Table 6.1. Whenever one of these hooks is triggered, the attached program writes the requirement path and the related permission to a ring buffer shared with the Dmng user

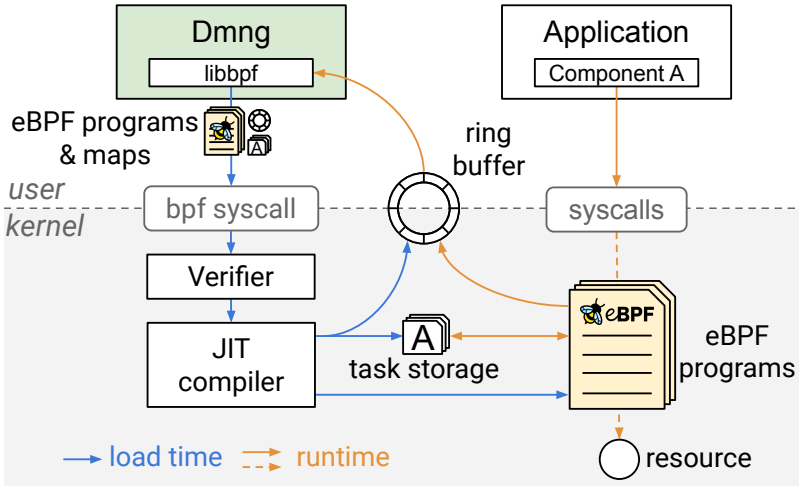


Figure 6.3: Dmng uses libbpf to load the eBPF tracing programs and maps, then it polls data from the shared ring buffer

space process. This permits Dmng to poll data regularly, and then to save it in the already mentioned SQLite database (Section 6.3).

We highlight that the collection of data using this method has minimal invasiveness: no changes must be introduced in the code of the application, nor it is necessary to restart it to setup the process. Indeed, when the developer wants to stop tracing, the eBPF programs and maps loaded by Dmng are automatically removed, leaving the system unmodified.

6.5 Policy

In this section we present the structure of the policy, explaining how it can be customized. We also discuss aspects such as coverage and effectiveness.

Policy structure The policy obtained from Dmng is a JSON file structured as a list of objects as shown in Listing 6.1. For each of them, a field *policy_name* identifies the application component the policy applies to, while the sections *read*, *write* and *exec* are used to configure the related permissions. The structure of the policy is flexible, and for each object only the field *policy_name* is required. Since the policy implements a *default-deny* model, an object that does not list any section in its body has no runtime permission, hence the corresponding component cannot access any file system resource. Listing 6.1 shows an example in which a component called *filter* is granted execution access to the Awk program (together with its shared libraries) to process the read-only *users.csv* dataset.

Table 6.1: List of file system traced hook points

Hook name
fentry/security_file_fcntl
fentry/security_file_ioctl
fentry/security_file_lock
fentry/security_file_mprotect
fentry/security_file_open
fentry/security_file_receive
fentry/security_file_set_owner
fentry/security_inode_getattr
fentry/security_path_chmod
fentry/security_path_chown
fentry/security_path_chroot
fentry/security_path_link
fentry/security_path_mkdir
fentry/security_path_mknod
fentry/security_path_rename
fentry/security_path_rmdir
fentry/security_path_symlink
fentry/security_path_truncate
fentry/security_path_unlink
lsm/bprm_check_security
lsm/mmap_file

Policy customization Dmng provides a CLI interface that work simultaneously with multiple data sources to produce the list of requirements. By default, it implements the logic to automatically merge the requirements collected with ptrace and the eBPF programs. Moreover, it permits to customize the policy interactively, adding, changing and removing requirements. For instance, it allows to delete requirements based on the permission mask (e.g., `r_x`, `r--`), or change the permission associated with all the requirements that match a given path regex (e.g., `/usr/bin/libnet.*`).

A useful feature implemented by Dmng is *permission pruning*. This function takes advantage of the structure of the Directory Tree [116] to help the developer lower the number of requirements in the policy by reducing the granularity of permissions. The reduction in granularity is based on a *pruning goal* set by the developer that represents the desirable maximum number of policy rules associated with an application component. To implement this feature, Dmng first uses the policy template to build a trie (or prefix tree), then starts pruning its branches iteratively following a best effort approach, until the pruning goal is achieved. The rationale is that there are areas of the file system in which fine granularity brings strong security guarantees (e.g., `/lib`), but there are also many other areas where fewer rules make

Listing 6.1: Example of JSON file with single policy

```
{
  "policies": [{
      "policy_name": "filter",
      "read": [
        "/lib/x86_64-linux-gnu/libsigsegv.so.2",
        "/lib/x86_64-linux-gnu/libreadline.so.8",
        "/lib/x86_64-linux-gnu/libmpfr.so.6",
        "/lib/x86_64-linux-gnu/libgmp.so.10",
        "/lib/x86_64-linux-gnu/libm.so.6",
        "/lib/x86_64-linux-gnu/libc.so.6",
        "/lib/x86_64-linux-gnu/libtinfo.so.6"',
        "/lib64/ld-linux-x86-64.so.2",
        "/usr/bin/awk",
        "users.csv"
      ],
      "exec": [
        "/lib/x86_64-linux-gnu/libsigsegv.so.2",
        "/lib/x86_64-linux-gnu/libreadline.so.8",
        "/lib/x86_64-linux-gnu/libmpfr.so.6",
        "/lib/x86_64-linux-gnu/libgmp.so.10",
        "/lib/x86_64-linux-gnu/libm.so.6",
        "/lib/x86_64-linux-gnu/libc.so.6",
        "/lib/x86_64-linux-gnu/libtinfo.so.6"',
        "/lib64/ld-linux-x86-64.so.2",
        "/usr/bin/awk"
      ]
  }]
}
```

the policy more concise without affecting security (e.g., `/share`). So, it is important to consider contextual information about the current prefix path to guide the pruning process. Moreover, we want to comply with the *write xor execute* memory protection policy whereby every file may be either writable or executable, but not both. Thus, limiting the propagation of potentially insecure configurations (e.g., no dynamic library stored in `/lib` must be writable and executable by the application). After the pruing process terminates, the changes to the template are audited by the developer, and can be committed or discarded.

Coverage and effectiveness To generate the policy templates, Dmng registers the activity performed by the application while a set of test cases is executed. This approach is similar to the one proposed by the Slim toolkit [164] to identify the dependencies of a container and minify its image, and to the one followed by the Google Sandbox2 utility [88] to retrieve the requirements of programs distributed as ELF files. However, test-based policy generation can be subject to coverage issues if the set of test cases is not exhaustive. Another aspect worth mentioning is that applications poorly structured may benefit less from the isolation properties pro-

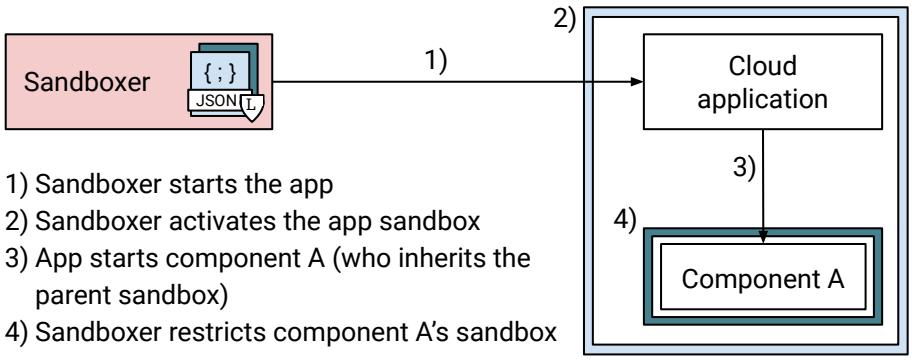


Figure 6.4: Landlock sandbox setup and inheritance

vided by sandboxing. Indeed, on a traditional Unix operating system, components executed within the same thread will inevitably share the same policy. Since Dmng supports the sandboxing of components with a per-thread policy, we recommend to leverage this function and execute potentially vulnerable components in dedicated compartments.

6.6 Application sandboxing

In this work we implement the sandbox leveraging Landlock, an unprivileged sandboxing mechanism officially merged into the Linux kernel in 2021 (version 5.13), with the goal of mitigating the security impact of bugs and unintended or malicious behavior in user-space application. The main reasons why Landlock was preferred to alternative sandboxing solutions such as Google Sandbox2 [88] are: 1) it does not rely on a proxy to implement the restrictions, hence it ensures low overhead at runtime, and 2) it is directly implemented within the kernel, thus it provides strong security guarantees. This section clarifies how policies are enforced with Landlock. Furthermore, it explains how `rwx` policy rules are translated into the Landlock permission model, and how restrictions are inherited by new components dynamically spawned at runtime.

The sandboxer is an extension of Dmng written in Rust that receives the JSON policy as input and modifies the application start procedure setting the permissions available to components before they are executed. The first task performed by the sandboxer is then to translate the `rwx` policy rules into the action-based permission model implemented by Landlock. In detail, Landlock groups permissions into *rulesets*, which collect the *actions* (e.g., FS_EXECUTE, FS_READ_FILE) permitted on each *object* (e.g., file, directory). The sandboxer separates the available actions to match the `rwx` categories, and then leverages the *landlock_create_ruleset()* and *landlock_add_rule()* interfaces to populate the rulesets accordingly. To activate the

restrictions, a call to *landlock_restrict_self()* is perfomed. The process is illustrated in Figure 6.4.

An important property defined by Landlock is *policy inheritance*. Whenever a new component is dynamically spawned by the application in a child process, it automatically inherits the restrictions set on the parent. Moreover, after a ruleset has been activated, no new permissions can be granted to a component, as the ruleset can only be further restricted. Figure 6.4 shows the policy inheritance process for a generic application. The figure also shows how the inherited ruleset is further narrowed with a subsequent call to *landlock_restrict_self()* by the component.

6.7 Experiments

This section presents our experimental evaluation. In the first part (Section 6.7.1), we show the benefits coming from the introduction of sandboxing in cloud applications. In detail, we reproduce a sample of CVEs affecting open source software, and showcase how the sandbox mitigates the exploits. In the second part (Section 6.7.2), we analyze the performance overhead. Specifically, we implement an application that performs various operations on media resources, and then evaluate the degradation of latency when sandboxing and eBPF-based monitoring are activated. The tests have been executed on a workstation with Arch Linux, kernel version 6.4, an AMD Ryzen 5 7600X CPU, 32 GB RAM, and 1 TB SSD.

6.7.1 Mitigation of vulnerabilities

To demonstrate the importance of the introduction of sandboxing, we selected the sample of high severity vulnerabilities reported in Table 6.2. These vulnerabilities affect software extensively used in cloud application development such as FFmpeg, ImageMagick, OpenSSL and Exiftool. For each of them, we installed on the system a vulnerable version of the program or library, and then verified it was exploitable using public Proof of Concepts when the input is sent through programmatic APIs or web interfaces. Subsequently, we leveraged Dmng to generate least privilege policies as explained in Section 6.4. Finally, we repeated the execution of the previous tests starting each vulnerable component with our sandboxer. When benign input was submitted to the application, we were successfully able to conduct the tests without loss of functionality. When instead malicious input was used, Landlock correctly blocked the exploit, limiting access to only resources listed in the policy. We highlight that, in general, similar protections extend to a broader set of CVEs.

Table 6.2: Sample of CVEs reproduced in our evaluation

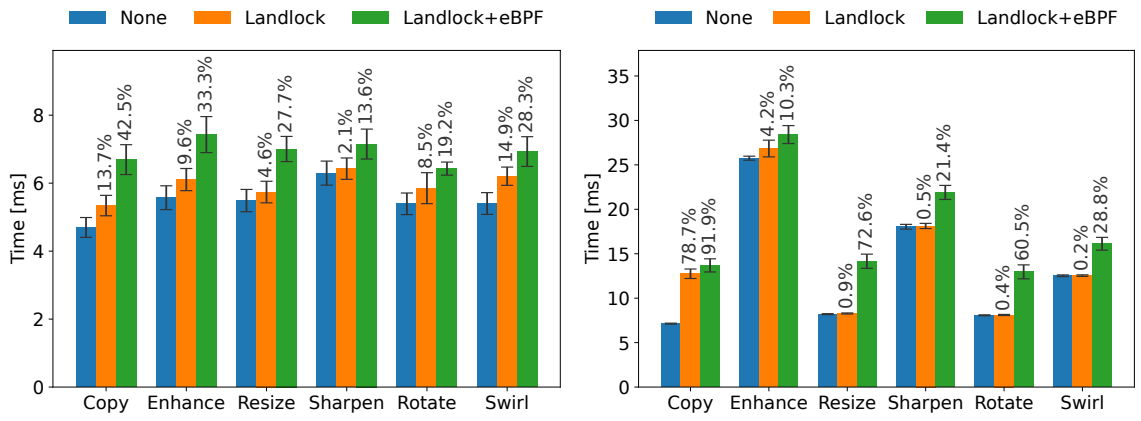
CVE	Software	Version	Description
CVE-2016-1897 CVE-2016-1898	FFmpeg	v2.x	A crafted AVI video is used to read arbitrary files
CVE-2020-29599	ImageMagick	v7.0.10-36	A bug in the PDF codec enables arbitrary code execution
CVE-2022-1292	OpenSSL	v3.0.2	Improper sanitisation allows command injection
CVE-2021-22204	ExifTool	v12.23	Improper neutralization of user data in the DjVu file format is used to run arbitrary executables

6.7.2 Overhead

As mentioned in Section 6.2, an important use case for cloud applications is represented by services that handle media resources such as videos, photos and audio. Therefore, to evaluate the overhead associated with our approach we implemented a Rust application that, upon receiving a request, leverages third-party software to apply several transformations on a media resource. Our goal is to measure the latency, or rather the time taken by the application to perform a given operation. Three test configurations are used: 1) no sandboxing is applied, hence no protection, 2) sandboxing is enabled leveraging our Landlock-based sandboxer, and 3) sandboxing is enabled plus eBPF-based continuous monitoring provided by Dmng is activated. Each operation is repeated 1000 times, and the measures are reported with 95% confidence intervals. Moreover, we focus on the server-side execution time, hence we do not consider the delay introduced by the network, which may make harder to visualize latency degradation for short-lived operations.

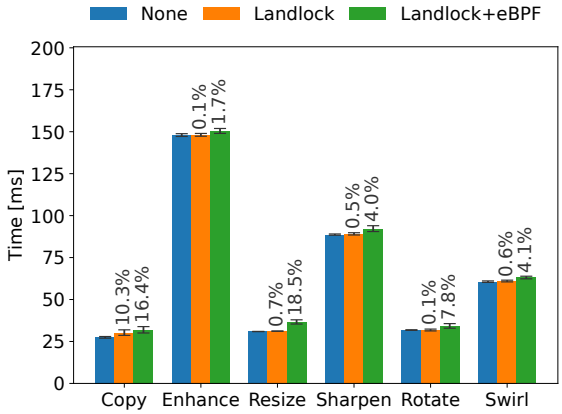
The first set of experiments focuses on image processing. In detail, the application leverages *convert* to copy, enhance, resize, sharpen, rotate and swirl images with 32x32, 640x480 and 1920x1080 resolutions. The results are shown in Figure 6.5. Inevitably, sandboxing introduces a slight degradation of latency compared to a scenario without protection. However, the overhead is non-negligible only for short-lived operations that last less than 10 ms, a duration that is considerably less than the average network delay. When instead eBPF-based monitoring is enabled, the data show worse latency degradation, especially for operations that last less than 50 ms. Remarkable is the case 640x480, in which the average operation overhead associated with eBPF-based monitoring is 47.6%.

The second set of experiments focuses on video processing. In detail, the application leverages *ffmpeg* to decode, copy, cut, loop and extract audio from videos with 480p resolution and with 6 seconds, 1 minute, and 10 minutes duration respectively.



(a) 32x32 image resolution

(b) 640x480 image resolution



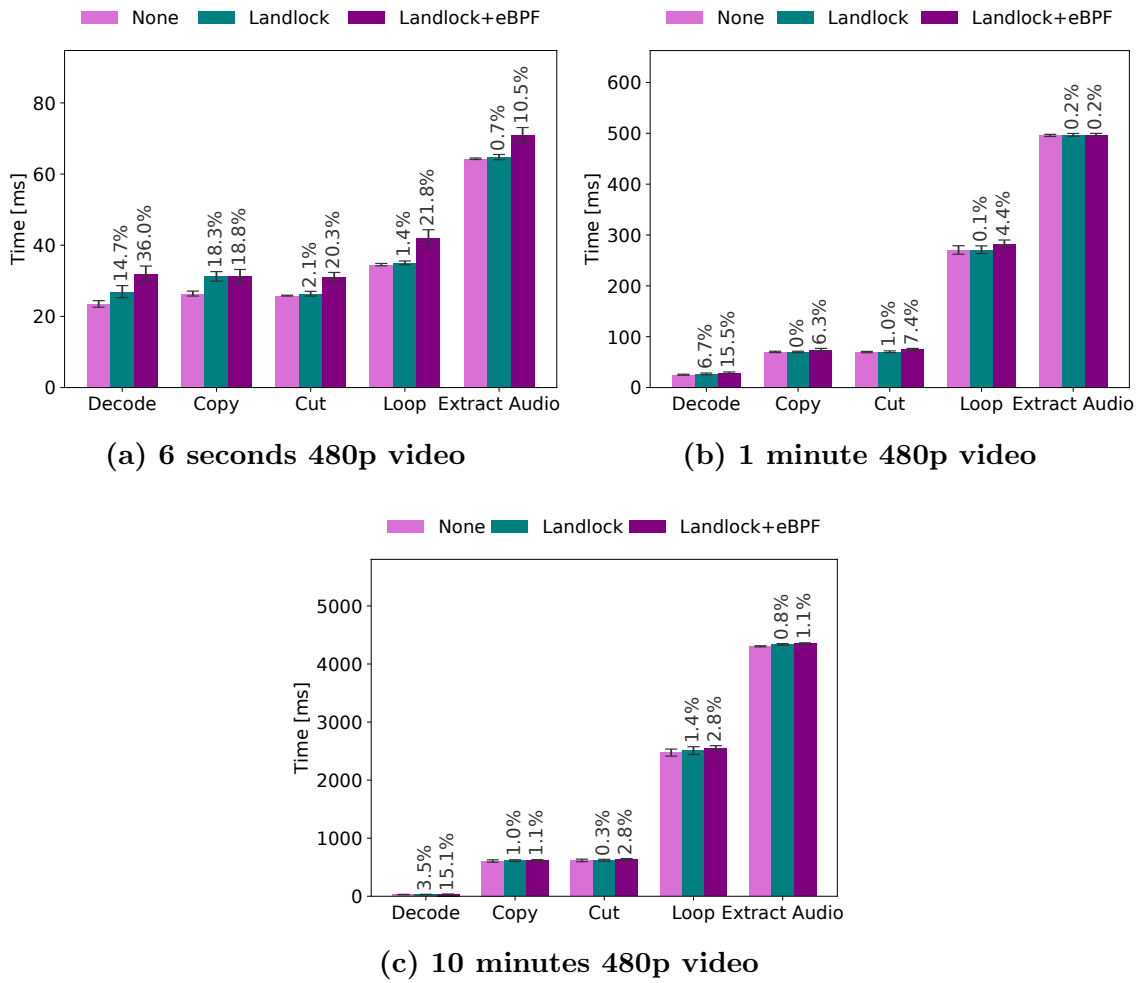
(c) 1920x1080 image resolution

Figure 6.5: Latency associated with various operations for an image processing application

The results are shown in Figure 6.6. The overhead introduced by sandboxing is perceptible only for the decode and copy operations on the shortest video (6 seconds). However, it never exceeds 18.3%. As expected, the overhead becomes practically negligible for operations that take longer than 100 ms. The same considerations extend to the eBPF-based monitoring, which again confirms to be associated with more degradation compared to the Landlock-only solution.

6.8 Related work

Several research works have highlighted the importance of sandboxing and isolation techniques in modern software [105, 200, 28, 132, 157, 9]. Indeed, sandboxing plays a key role in many plaftorms (e.g., Linux, Windows, iOS, Android), and is integrated in widely used software such as browsers (e.g., Chrome [42], Firefox [129]), service



(a) 6 seconds 480p video

(b) 1 minute 480p video

(c) 10 minutes 480p video

Figure 6.6: Latency associated with various operations for a video processing application

managers (e.g., Systemd [55]) and document viewers (e.g., Acrobat [7]).

With specific reference to the cloud scenario, many recent proposals have investigated the use of sandboxing to mitigate vulnerabilities [6, 5, 4, 10, 170, 41, 205, 78]. In *NatiSand* [6] and *Cage4Deno* [5] the authors modify the Deno runtime to control the permissions available to applications running native code. *BinWrap* [41] proposes similar measures to restrict the permissions available to Node.js native addons. *SandDriller* [10] describes an approach based on dynamic analysis for detecting sandbox escape vulnerabilities for Node.js applications. Zimmermann et al. [205] and Ferreira et al. [78] study the risks associated with vulnerable or malicious third-party dependencies and propose possible install (and update) time countermeausures. In general, all the previous proposals address the issues associated with a specific runtime ecosystem. Conversely, we aim to secure applications independently of their build toolchain or runtime.

Virtual machines and containers are two fundamental technologies in modern cloud architectures. Both permit to virtualize resources and execute applications in an isolated environment. Virtual machines ensure stronger security guarantees at the cost of higher resource utilization with respect to containers. The main reason is that applications executed in separate virtual machines have a distinct set of resources and do not share the same kernel [40, 39]. With specific reference to our scenario, both these technologies are associated with coarse granularity. Indeed, when working with them developers grant the application access to volumes rather than single resources. So, we provide a complementary approach to enable the introduction of fine-grained, per-resource access rules.

Modern industrial platforms like Cilium [174] and Falco [77] rely on eBPF as the primary means to enforce security policies in cloud applications. Cilium provides networking, observability, and security functions for container workloads, while Falco implements a threat detection engine for clusters. Both solutions are enterprise-oriented, hence the developer may find difficult to set up fine-grained policies leveraging them. Moreover, as already mentioned in the chapter, the performance of eBPF-based solutions is associated with large variability when fine-grained rules are used [177]. Therefore, we propose to complement these solutions by assisting the developer in the generation of least privilege security policies and using recent sandboxing technologies like Landlock, to reduce the overhead and strengthen the security boundary of the application.

6.9 Conclusions

The mitigation of security bugs and vulnerabilities that affect cloud applications is an important topic. In this chapter, we presented an approach to support the introduction of security policies to restrict the file system resources available to an application. To facilitate adoption, a central aspect in our proposal is the support to policy generation. We provide an open source tool to generate and customize least privilege policies leveraging the powerful, yet complex, ptrace and eBPF kernel technologies. Compared to virtualization technologies such as VMs and containers, which are associated with coarse granularity, we demonstrate that our proposal enables the introduction of fine-grained, per-resource access rules. The experiments showcase the capability of our approach to mitigate severe CVEs at the cost of limited, often negligible, overhead.

While currently the protection is limited to the file system, the isolation can be extended to other subsystems (e.g., the network). This is a promising line of research we aim to explore in the future.

Availability

The source code and artifacts produced for the evaluation of our proposal are available open source at `https://github.com/unibg-seclab/dmng`

7. Supporting Data Owner Control in IPFS Networks

7.1 Introduction

Based on the research we have conducted in the previous works, we investigated the possibility to extend our advances in system security and data protection to other types of systems. Specifically, a technology that has caught our interest is represented by decentralized filesystems that have points in common with traditional ones and that can be deployed on and complement modern cloud infrastructures. A widely used decentralized filesystem is represented by the InterPlanetary FileSystem (IPFS) that represents an alternative to the major cloud storage providers. Indeed, since their origins, Information and Communication Technologies have been characterized by a spectrum of options, with systems controlled by a single or a few powerful companies on one side, and open architectures that support the activities of a multitude of actors on the other side. In several domains, the evolution of technology has transformed scenarios that saw a single or dominant actor into a landscape where multiple companies and individuals can contribute. As technology evolves, components become more standardized, and the cost required to offer services decreases, an opportunity arises to build open services that replace what traditionally was offered by a relatively small number of companies. Networks, with their increasing efficiency and pervasiveness, have played a significant role in this evolution.

A domain where this evolution may occur is cloud storage. The role of cloud storage providers is significant and several services are offered today by major players in the IT world (e.g., Amazon S3 and Dropbox). Cloud storage services are well-engineered and highly performant, even if they sometimes exhibit critical failures [156, 45, 124]; the design of novel protocols can extend the opportunities available to users and facilitate the success of cloud storage solutions based on the participation of a large number of entities.

Decentralized storage architectures are emerging as a robust and scalable solution for storing resources. A solution that is particularly interesting is *InterPlanetary File System* (IPFS), a protocol that supports the dissemination of resources according to the P2P paradigm and represents the largest and most widely used *Decentralized Web* platform. In general, there is a continuous interest in the de-

sign and development of IPFS, as shown by the wide deployment and variety of applications built over it [182]. The IPFS infrastructure has been deployed in over 2700 Autonomous Systems, across 464k IP addresses, covering 152 countries. IPFS is seeing widespread uptake with more than 3M web client accesses and beyond 300k unique nodes serving content in the network every week. Although content retrieval in IPFS is slower than direct HTTP access, delays are still reasonable for a number of use cases. For instance, 75% of retrievals from Europe are under 2 s (including looking up the content host and fetching a 0.5 MB file). Moreover, the use of gateway caching can substantially reduce retrieval latency, with 76% of the requests being served in less than 250 ms.

A crucial aspect in the realization of any storage service, especially when it offers network access to resources, is the ability to protect resource confidentiality. By design, IPFS focuses on reliability and integrity, but does not provide confidentiality of resources, which should be protected by the owner before storing them on IPFS. Also, being distributed and replicated, IPFS does not guarantee permanent resource deletion, and hence owners lose control on their resources.

In this paper, we present an approach, called *Mix-IPFS*, that protects the confidentiality of resources and allows their owners to maintain the control on the resources uploaded to IPFS. Our proposal applies an *All-Or-Nothing-Transform* (AONT) encryption mode, which ensures that the whole encrypted resource is needed to reconstruct the corresponding plaintext. Our solution is transparent and fully integrated within the file system. We implement a virtual file system that automatically: splits the resource in IPFS chunks; encrypts each chunk; and uploads the encrypted resource chunks to IPFS. Our file system locally stores a small portion (fragment) of each encrypted resource, which is not stored in IPFS, to enable permanent deletion. When the data owner wants to delete a resource, the file system deletes the locally stored fragment, making the reconstruction of the plaintext resource impossible from the chunks stored in IPFS. Our solution offers the opportunity to reduce economic costs using spare storage space made available by third parties and, potentially, ease the permanent-storage requirement. Our experiments show that the overhead of Mix-IPFS on access times is limited (less than 2%) while not affecting performance.

Mix-IPFS's source code is available at `https://github.com/unibg-seclab/ipfs-owner-control`.

7.2 Basics concepts

The two building blocks of our solution are the adoption of IPFS for data storage and of AONT for data encryption.

IPFS. IPFS is an open-source distributed data storage network that relies on a content-based P2P network [27, 182]. Each IPFS node is identified by a *PeerID*. A resource stored in IPFS is split in chunks of fixed size (by default 256 KiB), each identified by a unique *Content Identifier* (CID). The CID of a chunk contains the result of a cryptographic hash function applied on the chunk, and other metadata. The chunks composing a resource are organized in a Merkle Directed Acyclic Graph (Merkle DAG), where the CID of each node is obtained by hashing the CIDs of its children. The CID of the root of the Merkle DAG is the resource CID. The use of a cryptographic hash function and of the Merkle DAG structure provides data immutability and self-certification. In fact, any IPFS user can autonomously verify whether a resource has been modified by checking the resource CID against the resource content (i.e., a resource cannot be modified without changing its CID). A Distributed Hash Table (DHT) maintains the association between a CID and the PeerID(s) where the corresponding resource is stored. To publish a new resource, the IPFS node of the owner splits the resource in chunks, builds the Merkle DAG, and pushes a new record for the resource CID in the DHT, in association with its PeerID. To retrieve a resource identified by a CID, an IPFS node retrieves from the DHT the PeerIDs of the IPFS nodes providing the CID, fetches the corresponding chunks, and verifies whether the resource matches its CID. Once a resource has been accessed, the IPFS node can make it accessible as a replica. While the publication of a resource involves the P2P network, the deletion of a resource is a local operation: if the resource has been replicated it remains available.

AONT. AONT is an encryption mode that transforms a plaintext resource into an encrypted resource guaranteeing complete interdependence among all the bits of the encrypted resource (i.e., each bit of the output depends on every bit of the input). Complete mixing implies that missing even a small portion of the ciphertext prevents reconstruction of the plaintext resource, even if the encryption key is known. Among the different proposals for implementing an AONT (e.g., [155, 26, 18, 21, 31]) we leverage the Mixing structure described in [18], which can benefit from the hardware accelerated implementation of AES available in most modern CPUs and ensures protection also in scenarios where users from which access should be prevented know the encryption key.

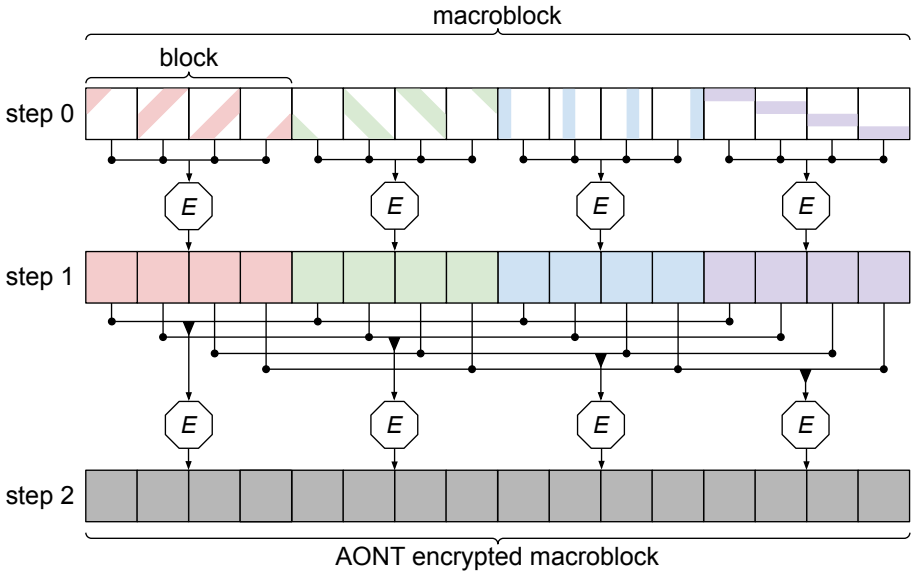


Figure 7.1: AONT mixing scheme

7.3 Mix-IPFS

While using IPFS for resource storage and management provides a number of advantages (e.g., data availability, limited economic costs), its wide adoption might be restricted by the loss of control by the owner, who cannot permanently delete resources. To support resource deletion while maintaining the advantages of IPFS, Mix-IPFS encrypts a resource with AONT. Intuitively, using AONT, the resource owner can force resource deletion by deleting a portion of the encrypted resource. To maintain full control, this portion, which we call *golden fragment*, is locally stored only at the owner side (i.e., not outsourced). Protection of cryptographic secrets is out of the scope of our work, however existing works address this problem (e.g., [3]). We now describe how Mix-IPFS uploads and access resources in IPFS.

Upload. A resource that needs to be stored in the IPFS network is first split into a set $\{M_1, \ldots, M_m\}$ of *macroblocks*, whose size must be a power-of-2 of the size of the block input to the symmetric block cipher at the basis of the working of Mixing. Mixing encrypts each macroblock through a sequence of symmetric encryption steps. At each step, the block input to the block cipher is obtained combining bits from different blocks resulting from the previous step. This guarantees, after a well defined number of steps, complete interdependence among the bits in the encrypted macroblock. Figure 7.1 illustrates an example of encryption of a macroblock composed of $4 = 2^2$ blocks, using a block cipher E (e.g., AES) that takes as input a block of size 16 bytes. In the first step, each block of the plaintext macroblock is separately encrypted. In the second step, the four blocks input to the four encryption operations E are obtained combining a different quarter (e.g., the first 4 bytes

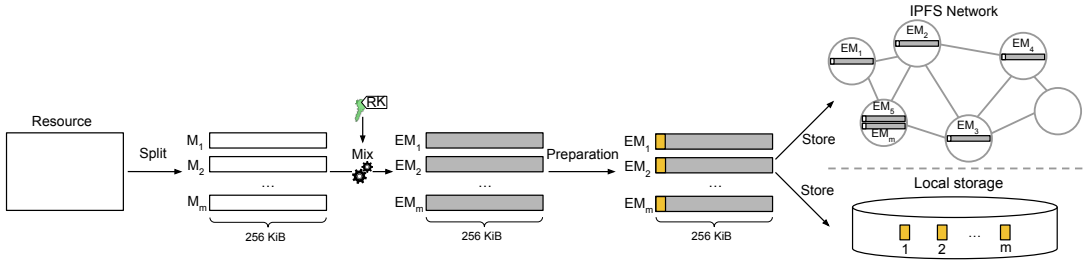


Figure 7.2: Resource upload with Mix-IPFS

of each block for the first E) of each of the four output blocks of the previous step. In this way, each bit in the four output blocks depends on each bit in the four input plaintext blocks, as visible from the color-coding in the figure. The number p of encryption steps necessary for mixing a macroblock depends on the size of the macroblock, the size b of the block, the number n of blocks mixed at each step (4 in our example). In particular, p steps guarantee the mixing of a macroblock of size $n^p \cdot \frac{b}{n}$ bytes. For instance, a macroblock of size 256 KiB requires 8 steps of encryption (i.e., 256 KiB $= 4^8 \cdot 16/4$ bytes).

For each macroblock M_i, $i = 1, \ldots, m$, the owner locally stores a small portion of arbitrary size of its encrypted representation. The collection of all these portions of the macroblocks forming a resource represents the golden fragment of the resource. For each encrypted macroblock, the bits locally stored are replaced in the macroblock with all 0s. While in principle the owner can locally store any sequence of bits from a macroblock, for simplicity, in our implementation we store in the golden fragment the initial sequence of bits of each macroblock. The resulting macroblocks are then uploaded to the IPFS network. Note that the size of macroblocks and IPFS chunks can be independently chosen by the resource owner. However, it is convenient, in terms of access time, to use macroblocks of the same size as IPFS chunks. Figure 7.2 illustrates the process of publishing a resource in an IPFS network using our approach, assuming macroblocks of 256 KiB and using key RK for encryption.

Access. To access a resource, the process illustrated in Figure 7.2 is reverted. A user first retrieves the chunks composing the resource identified by a given CID, and gets the locally stored golden fragment. When the download of a chunk has been completed, the user can replace the initial sequence of 0s with the corresponding bits in the golden fragment. The macroblock can then be decrypted (applying Mixing in decryption mode). The overhead of Mixing is negligible compared to the application of a simple direct AES encryption on the IPFS chunks (see Section 7.5). Also the size of the golden fragment is expected to be negligible compared to the size of the chunks.

7.4 Mix-IPFS Architecture

We now illustrate the architectural design and working of Mix-IPFS. In the design of Mix-IPFS, we aim at a system that allows for transparency, that is, hiding to owners the adoption of AONT encryption for protecting resources and of IPFS for their storage. In the following, we will use the term *user* to generically refer to the user of Mix-IPFS, which is the owner uploading resources, and the user accessing them.

7.4.1 Architectural design

Our solution relies on a virtual file system that represents the interface through which a user interacts with the IPFS network to upload and access resources. Transparency is guaranteed by providing the user with a directory in the file system, listing resources stored in IPFS using our protection technique. Mix-IPFS is then responsible for mediating store and access requests (i.e., to transparently apply encryption/decryption operations and to manage the golden fragment).

Mix-IPFS uses FUSE (*Filesystem in USErspace*) [82] as a virtual file system, since it is a well-known Linux kernel technology that allows unprivileged users to access virtual file systems without the need to modify the kernel code or load new modules. Also, FUSE has the advantage of being compatible with all Unix and Unix-like systems (e.g., FreeBSD, macOS) as well as with Windows [199]. FUSE consists of two modules: *i)* the FUSE Linux kernel module, and *ii)* the `libfuse` library [83], which operates in userspace and provides a reference set of APIs used by the Linux kernel module to serve file system-related requests. Mix-IPFS provides the APIs defined by `libfuse` necessary to handle upload and access operations, work with symbolic links, and modify resource metadata (i.e., file attributes including, for example, access privileges, last accessed and modified timestamps). Mix-IPFS has been designed with performance in mind, aiming at minimizing access times. To this purpose, Mix-IPFS organizes information about resources in a trie data structure, which enhances performance of prefix-based searches in the directory.

Before mounting the file system, Mix-IPFS authenticates the user through the user master password (MP), which is also used to derive a master key (MK) that is adopted for encrypting Mix-IPFS metadata (Figure 7.3, step A). Mix-IPFS metadata include the relevant structures needed for the functioning of the virtual file system, such as attributes and names of files, cryptographic keys used for resource encryption/decryption (i.e., RK), and identifiers of the data stored in IPFS (i.e., CIDs of resources and reference to the golden fragments). If authentication succeeds, Mix-IPFS loads (or initializes if it is not already created) the file system

metadata and, based on them, mounts the virtual file system in a dedicated directory (Figure 7.3, step B). The user can then interact with the directory in the same way as with any other directory in the file system. User interactions are however captured by the Linux kernel and redirected to the FUSE module via the VFS (*Virtual Filesystem Switch*) module. The FUSE module serves the user's requests through the `libfuse` library.

7.4.2 Management of upload and access operations

We now illustrate how the insertion of a resource into the local directory of the user file system and the access to a resource are managed by Mix-IPFS. Figures 7.3 and 7.4 show the working of the upload and access operations. The green (gray in black and white printout) modules in the figures are specific of Mix-IPFS.

Upload. When a user inserts a new resource into the local directory (step 1 in Figure 7.3), VFS intercepts the request and passes it to FUSE (step 2). The request is then forwarded to Mix-IPFS (step 3). Mix-IPFS first encrypts the resource as described in Section 7.3, keeping the macroblock size the same as IPFS chunks (256 KiB each) and keeping as golden fragment the first 1024 bytes of each macroblock (step 4). Mix-IPFS then asynchronously loads the encrypted macroblocks to IPFS nodes and annotates the CIDs (calculated by IPFS) of the encrypted macroblocks (step 5). Mix-IPFS finally maps the name of the resource to a tuple of the form ⟨CID, `out_loc`⟩, where CID is the CID of the resource and `out_loc` is a token (e.g., a path on the local file system) necessary to retrieve the golden fragment, and updates metadata (step 6). Note that the information locally stored at the user-side to support access to resources includes the golden fragment, the encryption key RK used by Mixing, tuple ⟨CID, `out_loc`⟩, and the size of the portion extracted from macroblocks to build the golden fragment.

Access. When the user wants to access their resource (step 1 in Figure 7.4), VFS intercepts the request and passes it to FUSE (step 2). The request is then forwarded to Mix-IPFS (step 3), which reverses the protection applied when the resource has been uploaded to IPFS. Mix-IPFS then first retrieves the (locally stored) tuple ⟨CID, `out_loc`⟩ for the resource of interest (steps 4 and 5) and downloads, from the IPFS network, the resource with the identified CID (steps 6 and 7). Mix-IPFS then operates in parallel on all the retrieved macroblocks: it replaces the initial sequence of 0s with the corresponding sequence of bits in the golden fragment and applies AONT Mixing in decryption mode (step 8). The plaintext resource is then returned to the user (steps 9–11).

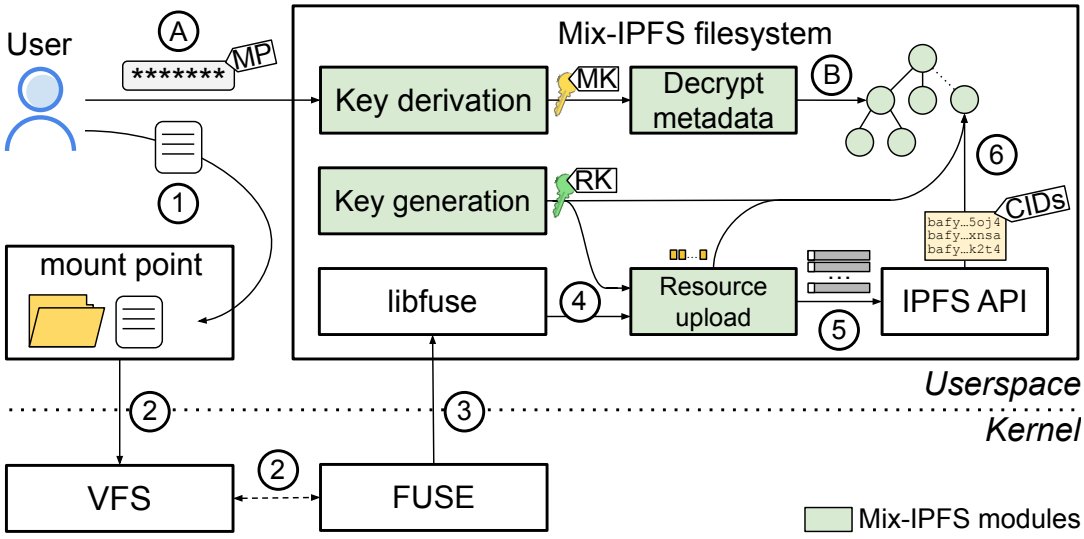


Figure 7.3: Mount Mix-IPFS and upload a resource

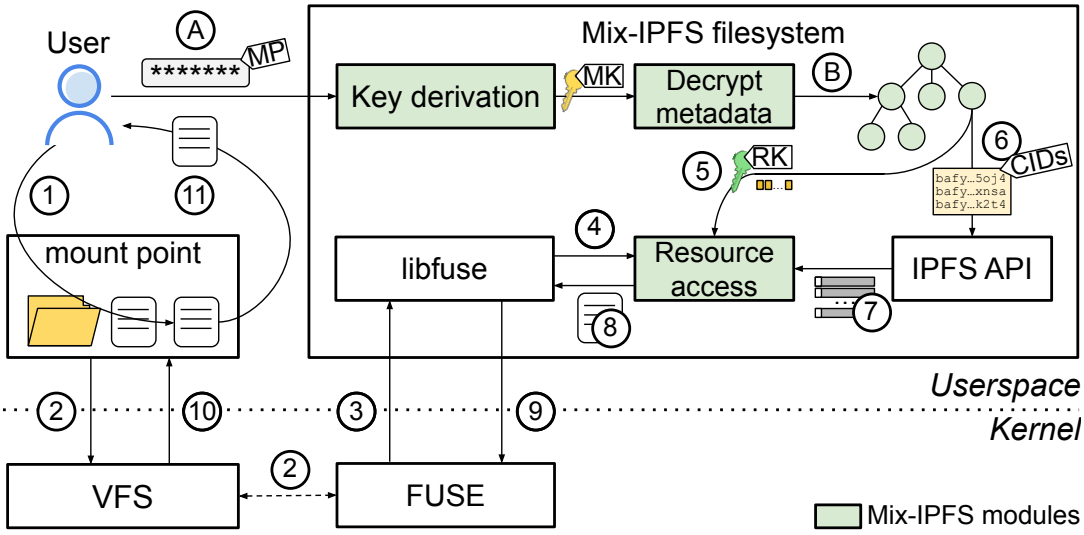


Figure 7.4: Access a resource in Mix-IPFS

Mix-IPFS easily manages also updates to resources by combining the upload and access processes illustrated above. In fact, any update to an IPFS chunk implies an update to its CID, and hence to the CID of the resource. Delete operations, as already noted, are managed by permanently removing the golden fragment(s) of the resource.

7.5 Experimental evaluation

To assess the performance overhead introduced by Mix-IPFS, we conducted a wide experimental analysis comparing access times to files stored in IPFS in plaintext, when using a traditional block cipher, and when using Mix-IPFS (Section 7.5.1). We

also analyzed the local storage overhead due to the golden fragment and additional file system metadata (Section 7.5.2).

For the experimental evaluation, we used a machine with Ubuntu 22.04 (kernel 5.19), IPFS 0.19.1, `libfuse` 3.10.5, and equipped with an AMD Ryzen 9 7900X, 64 GB of RAM, and 2 TB of SSD. The machine is located in southern Europe (note that IPFS performance varies depending on the region, as shown in [182]). The resources used for our experiments have been replicated on different public IPFS nodes.[1] In our experimental evaluation, we used 10 different IPFS nodes to test the benefit of the parallelization of our Mixing, which leverages the distributed nature of IPFS. We considered IPFS chunks of 256 KiB and assumed to locally store in the golden fragment 1 KiB for each macroblock.

7.5.1 Latency

To analyze the latency of Mix-IPFS, which represents a crucial aspect when accessing resources, we consider two scenarios that differ in how resources are protected: *1)* encryption through AES in CBC and CTR mode; and *2)* encryption through Mixing. Note that, in our experiments, AES operates at the macroblock level, to leverage the same parallelism implemented in our solution and enable a fair comparison among the considered scenarios. We analyzed separately the overhead introduced by the protection techniques of these two scenarios when uploading and when accessing a resource. The overhead is computed with respect to the case where resources are stored in IPFS in plaintext, which is our baseline. The results illustrated in the following have been obtained as the average of 20 runs. The colored areas in the figures represents the standard deviation obtained in the experimental evaluation.

Upload. The upload of a resource can be considered as operating in two independent steps: *i)* protect the resource, and *ii)* publish the protected resource on IPFS. Since IPFS natively implements the second step, and the size of the encrypted resource does not change significantly, the overhead introduced when uploading a resource only depends on the performance of the protection technique adopted (i.e., AES in CBC and CTR mode or Mixing). Considering a resource of size between 1 KiB and 10^6 KiB, Figure 7.5a illustrates the average overhead, computed as the ratio between the time necessary to protect the resource and our baseline (i.e., the time spent to publish the resource on IPFS), when using AES in CBC and CTR mode and when using Mixing. As expected, the overhead increases with the size of the resource, since the time spent for encrypting a resource depends on its size, while IPFS publication is less susceptible to it. We note, however, that the overhead

[1]List of public IPFS nodes: https://ipfs.github.io/public-gateway-checker/

of Mixing is similar to the overhead of traditional AES encryption (2% overhead for 1GB resources), while providing higher protection guarantees.

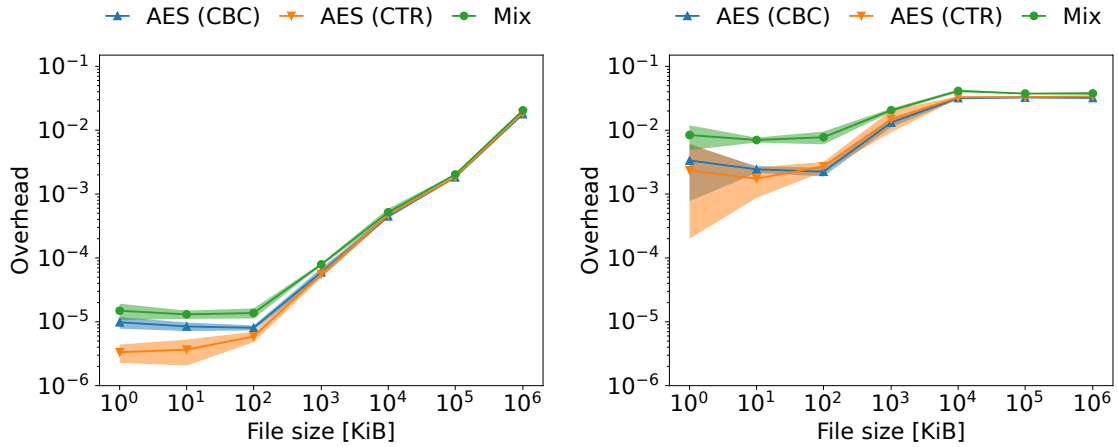
Access. Similarly to the upload operation, the access to a resource operates in two steps: *i)* retrieve and download from IPFS the chunks composing the resource, which is natively implemented in IPFS, and *ii)* remove the protection layer. Considering a resource of size between 1 KiB and 10^6 KiB, Figure 7.5b illustrates the average overhead, computed as the ratio between the time necessary to remove the protection layer (i.e., AES in CBC and CTR mode or Mixing), and the time spent to retrieve the resource (baseline). The overhead increases with the size of the resource, but the increase is less steep compared to what observed for upload operations in Figure 7.5a. This smoother trend is due to the fact that IPFS takes longer to fetch large resources (step *i)* above), as shown in Figure 7.6b that reports the average absolute time necessary to download a resource varying its size. Figure 7.5b confirms that our approach is competitive with AES also for access operations. For small resources, which cannot take advantage of parallelization since they fit in one macroblock, we observe a higher cost when using Mixing compared to AES. The performance overhead is however limited (0.95% compared to AES in CBC mode, 0.85% compared to AES in CTR mode). This is confirmed by Figure 7.6a, showing the average absolute time for removing the protection layer (step *ii)* above).

7.5.2 Storage

We also analyze the additional space needed for the adoption of Mix-IPFS (i.e., the space required for storing the golden fragment and the file system metadata).

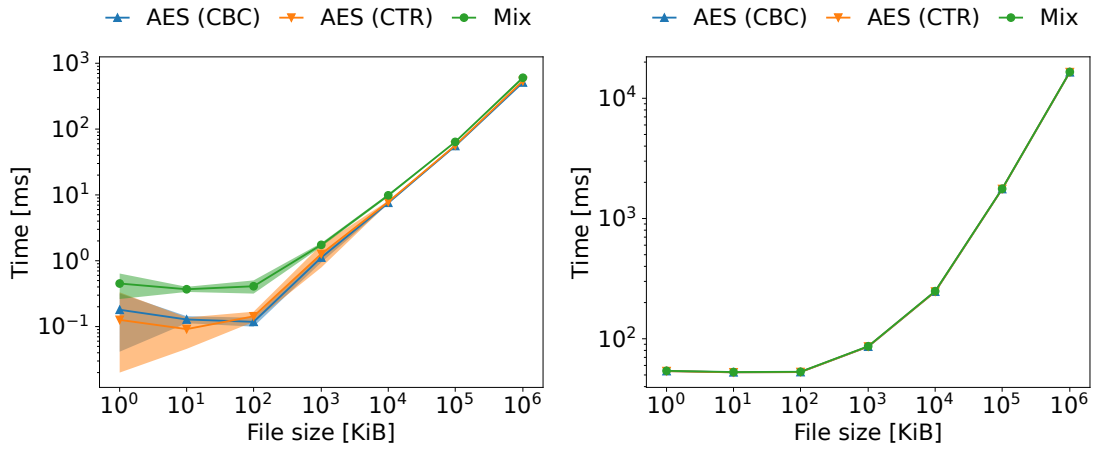
Golden fragment. Assuming to locally store 1 KiB for each macroblock and to use IPFS chunks (and macroblocks) of 256 KiB, Figure 7.7a illustrates the overhead, computed as the ratio between the size of the golden fragment and the size of the (plaintext) resource, varying the size of the resource between 1 KiB and 10^6 KiB. As expected, the overhead due to the local storage of the golden fragment is higher for small resources, especially for resources having size smaller than 256 KiB. In fact, Mix-IPFS applies padding to reach a size that is a multiple of the macroblock size before encrypting the resource. For instance, considering a resource of size 20 KiB, the overhead of the golden fragment is 1/20. The figure shows that the storage overhead quickly approaches constant value 1/256 (i.e., 0.39% of the resource size).

Metadata. Figure 7.7b illustrates the overhead, computed as the ratio between the on-disk size of file system metadata and the size of the plaintext resources handled by Mix-IPFS, varying the overall size of plaintext resources between 10^3 KiB and 10^4 KiB. The overhead remains extremely low when managing small sets of resources



(a) Upload overhead **(b) Access overhead**

Figure 7.5: Average overhead of Mixing and of AES in uploading and accessing a resource, varying the size of the resource



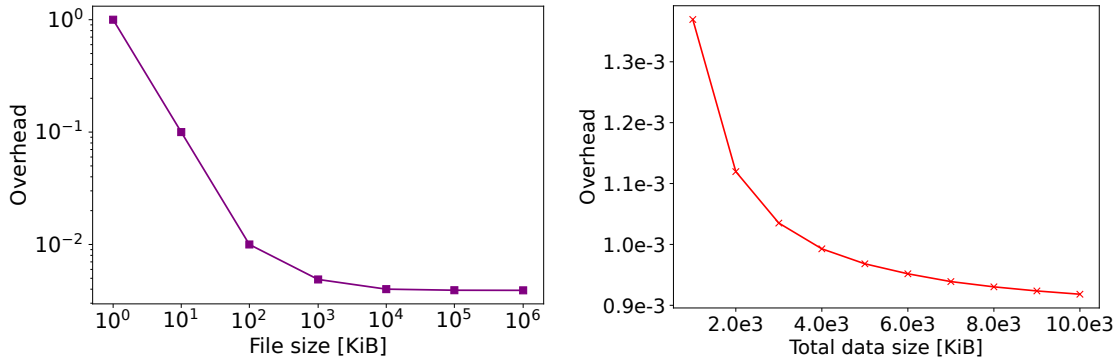
(a) Decryption time **(b) Download time**

Figure 7.6: Average absolute times for accessing a resource, varying the size of the resource

(less than 0.14% in our experiments), and decreases as the overall size of resources grows.

7.6 Related work

The problem of protecting data confidentiality in modern P2P networks and decentralized storage networks has been recently addressed by several approaches (e.g., Freenet [178], Filecoin [79], Arweave [17], and Sia [162]). While presenting similarities with Mix-IPFS, these approaches do not allow the resource owners to be in control of resource deletion. The work in [22] addresses the problem of scheduling



(a) Golden fragment overhead **(b) Metadata overhead**

Figure 7.7: Space overhead of the golden fragment and of the file system metadata

the release of confidential data in a decentralized network, even when some of the nodes may not follow the P2P protocol. Contrary to our scenario, this solution relies on economic incentives and penalties to guarantee persistent storage to the client.

The use of AONT for providing data confidentiality in decentralized systems has been recently addressed, with approaches that enforce client-side encryption in AONT mode. The proposals in [20, 19] introduce a model based on AONT and data replication for providing both security and availability guarantees to resources stored in a decentralized cloud storage. SAFE network [97, 151] is a decentralized cloud storage network that adopts convergent encryption for data protection. These solutions, while effective in providing data confidentiality, differ from our approach since they rely on the nodes in the network for enforcing resource deletion, which is therefore not fully under the owner control. Similarly, the adoption of linear network coding aims at protecting resources stored in a distributed cloud storage by splitting each resource into multiples fragments allocated at different nodes [163]. The adoption of a scheme similar to secret sharing for splitting resources provides data confidentiality, but resource deletion is not under the owner control.

The problem of providing a file system that transparently manages security mechanisms has been widely studied (e.g., [90, 107]). Among existing solutions, EncFS [90] offers an encrypted userspace file system relying on FUSE. OramFS [107] is instead an encrypted (and optionally authenticated) Oblivious RAM file system. Differently from our proposal, these approaches do not adopt AONT encryption mode to guarantee strong interdependency, and are not designed to operate on P2P networks.

Another line of works related to our proposal is aimed to provide confidentiality of outsourced data (e.g., [54, 53]). The approach in [43] provides protection of confidential information by maintaining a portion of the data at the owner side.

This solution completely departs from encryption, assuming that what is sensitive is the association among data.

7.7 Conclusions

We proposed Mix-IPFS, an approach that aims at protecting confidentiality of data stored in IPFS, and at enabling owners to remain in control of their resources and to force delete operations over them. Data confidentiality is guaranteed by applying AONT encryption at the owner-side, and the control over delete operations is preserved by locally storing a small fragment of the encrypted resource. Being integrated in the file system, Mix-IPFS guarantees transparency to users. Our experimental evaluation confirms the limited overhead of Mix-IPFS, both in terms of access time and of local storage space.

8. Conclusions and future work

As cloud applications grow in size and the number of their dependencies increases, it is important to provide execution environments that are secured against vulnerabilities inside third-party packages, avoiding the risks of system compromises and data leakages. Focusing our attention on the most used cloud runtimes, like NodeJs or Deno, several works have pointed out the necessity to protect systems from memory unsafe libraries that can compromise an otherwise safe application [41, 191, 132, 171]. Similarly, system access control through access policies and applications' isolation following the *least privilege* principle have received growing interest by researchers [188, 8, 35] and process sandboxing represents a promising research field to provide more secure and reliable applications. WebAssembly is gathering more and more interest in web and cloud applications thanks to its high performance and security-oriented design, however Wasm runtimes relying on the WASI interface are still premature and on development and do not provide robust isolation and fine-grained control over the resources of the system accessed by a Wasm module. For these reasons research on WebAssembly access control has a significant role in development of modern cloud applications. Moreover, maintaining the control over the resources uploaded on modern network systems, whether these systems are cloud-based or based on other paradigms (e.g., decentralized) to support cloud systems, is fundamental in a scenario where, in future, more and more data will be stored online and where maintaining control over them, and protecting them against untrusted or *honest but curious* providers, its becoming more and more difficult. For these reasons research in data protection and system security (with particular attention to cloud systems) represents an important research field that deserves to be explored further with more in-depth future works. In this dissertation, advances in cloud computing sandboxing and data protection have been showcased. One first contribution is presented in Chapter 3 where we proposed Cage4Deno as a first attempt to provide a secure sandboxing technique for the Deno runtime. In Chapter 5 NatiSand has been therefore proposed as a more powerful extension that provides better guarantees for all the main JavaScript and TypeScript-based runtimes. Similarly, our contribution on WebAssembly sandboxing, in Chapter 4, represents one of the first attempts to provide Wasm runtimes with a more secure interaction with the filesystem and our policy-based proposals, extended further in Chapter 6, allow for a flexible and easy integration with existing systems. Finally, in Chapter 7 Mix-IPFS is proposed as a solution to keep control over data upload on decentralized filesystems and to allow

a safe and efficient data deletion. Considering our proposals and the interest of the research community, more advances can be done by proposing more efficient, robust and effective solutions, and with a deeper understanding of modern computing systems.

Acknowledgments

The research work done during my PhD and reported in this document was supported in part by the European Commission under projects GLACIATION (101070141) and MARSAL (101017171), by the italian Ministry for Universities and Research (MUR) under PRIN project POLAR, by projects GRINS (PE00000018) and SERICS (PE00000014) under the NRRP MUR program funded by the EU – Next Generation EU. The research was supervised by Prof. Stefano Paraboschi (Università degli Studi di Bergamo)

I am extremely grateful to my advisor Prof. Stefano Paraboschi that guided and supported me through my studies. I also wish to thank my colleagues of the Security Laboratory of the University of Bergamo with whom I shared study, research, work and pleasant moments: Dario Facchinetti, Matthew Rossi, Gianluca Oldani and Michele Beretta.

Finally, I want to express my most sincere gratitude to my family that supported me during this journey, and in particular to my mom who has always supported me and continues to do so wherever she is now. *I know you would have been proud of my achievement.*

A. BPF details

In the following we detail, as a complement, some of the issues we found working with the current implementation of BPF, explaining how we solved them.

A.1 Hashing & collision handling

As anticipated in the design and implementation of `deny` rules (Section 3.4.3), the current implementation of the BPF framework does not provide support for map-in-map structures [120]. Moreover, support for using string keys in a map is still limited [93]. At the time of writing, a patch for providing this option, along with efficient ternary search support to lookup the map [92], are under active development on `bpf-next`, the branch dedicated to the features and improvements that should eventually land in BPF. Nonetheless, we had to cope with the temporary limitation of using an integer key to lookup the Map_{policy}, hence, we had to find a way to transform the string prefixes stored in the trie to fixed size integers. A typical approach to solve this problem is to use a hash function. Given the need to preserve the ability to search for prefixes efficiently, we opted for using an incremental hash function. Given S_r, the string representing the access path, an incremental hash function permits to compute $hash(S_r[i])$, the hash of the prefix terminating at character i, in constant time given the hash of the prefix terminating at $i-1$. In our case we relied on *djb2* [103], a non-cryptographic low-complexity incremental hash function proposed by D. J. Bernstein.

The use of a hashing function also required us to handle the collisions explicitly. Indeed, collisions may occur during map creation, and at runtime upon receiving an access request. Basically, this means that every time a key is found in the policy map Map_{policy}, a collision check has to be performed to determine whether a deny rule was hit. The best trade-off between query response time and size of the policy map has been achieved adapting the separate hash chaining approach, in which a fixed size array is used to store a sequence of colliding paths. To explain how it works we introduce the following example. Let us assume to have three deny rules in the policy: `/home/user/data`, `/home/user/lib` and `/media`. Also, let us assume that the djb2 hash of `/home/user/lib` and `/media` collide. Instead of using the deny rules to build a prefix tree, we can build the policy map Map_{policy} as shown in Table A.1. Upon receiving an access request S_r, the verifier computes the incremental hash for each of its prefixes, and then performs a sequence of map

BPF Map	
Deny rule hash	**Array of colliding paths**
$djb2(D_{r1})$	`/home/user/data` $\rightarrow \emptyset$
$djb2(D_{r2}, D_{r3})$	`/home/user/lib` $\rightarrow$ `/media` $\rightarrow \emptyset$

Table A.1: Map_{policy} **with separate hash chaining (assuming the hashes of** `/home/user/lib` **and** `/media` **colliding)**

Algorithm 1 Modified deny rule verifier

```
1:  procedure DENY_VERIFIER(S_r)
2:      task ← bpf_get_current_task_btf()
3:      if ¬task ∈ Map_task then
4:          return 0                                   ▷ Grant request
5:      Map_policy ← Map_task [task]
6:      for each prefix in S_r do
7:          if djb2(prefix) ∈ Map_policy then
8:              collision_chain ← Map_policy [djb2(prefix)]
9:              for D_r in collision_chain do
10:                 if isPrefix(D_r, prefix) then
11:                     return -EPERM                  ▷ Deny request
12:     return 0                                       ▷ Grant request
```

lookups. In case no lookup is successful, the verifier concludes that no deny rule was hit, hence the access request is granted. Conversely, the verifier must check whether a deny rule was hit, or a collision was found. To do that, the verifier compares the access request S_r, and each of the colliding paths associated with the lookup key. Algorithm 1 details the procedure.

The time complexity of the proposed approach varies according to the presence of collisions. When none occurs, the worst case time complexity is given by the total hashing time $O(N)$, plus the total lookup time $O(N)$ (i.e., in the worst case S_r is structured as $[/c]^+$, hence $N/2$ lookups each taking $O(1)$ time are performed). Instead, each time a collision occurs (or a deny rule is hit), the verifier incurs in an $O(N \cdot L)$ extra time, where L is the length of the longest collision chain. Similarly to other research proposals [103], the results presented in our Experimental Evaluation showed a limited impact associated with the presence of collisions (see Section 3.6). It is worth mentioning that the number of collisions can be reduced using a cryptographic hash function, at the expense of a lower efficiency. With regard to the size of the policy map, given M the number of deny rules in the policy, a space of $O(M \cdot N)$ is required.

A.2 Map types

BPF provides several map types to the developer. Each of them is characterized by distinct performance and functions. In Sections 3.4.3 and A.1 we illustrated our construction of the verifier, detailing the role of Map_{task} and Map_{policy}. The former permits to keep track of the processes subject to the control of Cage4Deno and to associate them with a policy map, while the latter stores the restrictions listed in a policy. Since the content and number of entries of Map_{task} varies at runtime, in the implementation we opted for the `TASK_STORAGE` map type, which permits to be modified calling the `bpf_task_storage_get()` and `bpf_task_storage_delete()` BPF helpers. With regard to Map_{policy}, no modification is permitted after the creation of the map. The only parameter to be minimized is the lookup time, hence we opted for the `HASH` map type.

A.3 Stack limitation

The maximum path length on a Linux system is bounded to 4096 characters (in `linux/limits.h`). Unfortunately, the stack size of a BPF program is limited to 512 bytes. To circumvent this limitation, we stored each access path outside of the BPF stack, in a dedicated `PERCPU_ARRAY` map. As the name suggests, each CPU core executing a BPF program has an instance of the `PERCPU` map, which can hold a different state. However, the content of an instance cannot be modified for the whole duration of the check performed by the verifier (except for the verifier itself), as BPF programs are non-preemptable. Therefore, the use of the `PERCPU` type bypasses the BPF stack limitation without requiring any additional concurrency primitive (e.g., spinlocks).

B. GNU Tar policy file

Listing B.1 reports the policy associated with GNU Tar. The policy has been generated using `dmng` according to the procedure described in Listing 3.4.

Listing B.1: Example of policy associated with `tar`

```
{
  "policies": [
    {
      "policy_name": "tarPolicy",
      "read": [
        "/usr/local/bin/tar",
        "/usr/lib/locale/locale-archive",
        "/usr/share/locale/locale.alias",
        "/usr/bin/gzip",
        "/lib/x86_64-linux-gnu/libc.so.6",
        "/lib64/ld-linux-x86-64.so.2",
        "/etc/ld.so.cache",
        "/home/user/input.tgz",
      ],
      "write": [
        "/home/user/output"
      ],
      "exec": [
        "/usr/local/bin/tar",
        "/usr/bin/gzip",
        "/lib/x86_64-linux-gnu/libc.so.6",
        "/lib64/ld-linux-x86-64.so.2"
      ],
      "deny": [
        "/home/user/output/output/misc"
      ]
    },
}
```

C. Policy generated for the `curl` command

Listing C.1 reports the policy associated with the execution of the `curl https://www.example.com` command. The policy has been automatically generated using the utility described in Section 5.5.2.

Listing C.1: Example of policy associated with curl

```
[{
  "name": "/usr/bin/curl",
  "fs": {
    "read": [
      "/etc/gai.conf",
      "/etc/host.conf",
      "/etc/hosts",
      "/etc/ld.so.cache",
      "/etc/localtime",
      "/etc/nsswitch.conf",
      "/etc/passwd",
      "/etc/resolv.conf",
      "/etc/ssl/certs/ca-certificates.crt",
      "/lib/x86_64-linux-gnu",
      "/lib64/ld-linux-x86-64.so.2",
      "/usr/bin/curl",
      "/usr/lib/locale/locale-archive",
      "/usr/lib/ssl/openssl.cnf"
    ],
    "exec": [
      "/lib/x86_64-linux-gnu",
      "/lib64/ld-linux-x86-64.so.2",
      "/usr/bin/curl"
    ]
  },
  "net": [{
    "name": "https://www.example.com",
    "ports": [443]
  }]
}]
```

References

[1] A. Starovoitov. CAP_BPF. `https://lwn.net/Articles/820560/`, 2020.

[2] M. Abbadini, M. Beretta, S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Supporting data owner control in IPFS networks. In *Proceeding of the IEEE International Conference on Communications (IEEE ICC 2024)*, 2024.

[3] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Lightweight cloud application sandboxing. In *Proceeding of the 14th IEEE International Conference on Cloud Computing Technology and Science (IEEE CLOUDCOM 2023)*, 2023.

[4] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Poster: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.

[5] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Cage4Deno: A fine-grained sandbox for Deno subprocesses. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.

[6] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. NatiSand: Native code sandboxing for JavaScript runtimes. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023.

[7] Adobe Inc. Sandbox protections. `https://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/sandboxprotections.html`, 2023.

[8] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven trigger-action platforms. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[9] M. Albanese, A. De Benedictis, D. D. de Macedo, and F. Messina. Security and trust in cloud application life-cycle management. *Future Generation Computer Systems (FGCS)*, 111, October 2020.

[10] A. AlHamdan and C. Staicu. SandDriller: A fully-automated approach for testing language-based JavaScript sandboxes. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2023.

[11] B. Alliance. Cli options for wasmtime. `https://docs.wasmtime.dev/cli-options.html`, 2023.

[12] B. Alliance. Wasmtime Runtime. `https://wasmtime.dev/`, 2023.

[13] W. Almesberger. Linux network traffic control – implementation overview, 1999.

[14] N. Andrii. BPF portability and CO-RE. `https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html`, 2020.

[15] Apache. JMeter. `https://jmeter.apache.org/`, 2022.

[16] Apple. JavaScriptCore. `https://developer.apple.com/documentation/javascriptcore`, 2023.

[17] Arweave. Meet Arweave: Permanent information storage. `https://www.arweave.org/`, 2023.

[18] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Mix&Slice: Efficient access revocation in the cloud. In *Proc. of ACM CCS*, October 2016.

[19] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Dynamic allocation for resource protection in decentralized cloud storage. In *Proc. of GLOBECOM*, December 2019.

[20] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Securing resources in decentralized cloud storage. *IEEE TIFS*, 15(1), December 2020.

[21] E. Bacis, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, and P. Samarati. Mix&Slice for efficient access revocation on outsourced data. *IEEE 2TDSC*, 2023. *Early Access*.

[22] E. Bacis, D. Facchinetti, M. Guarnieri, M. Rosa, M. Rossi, and S. Paraboschi. I told you tomorrow: Practical time-locked secrets using smart contracts. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2021.

[23] D. Bakker. wasi-sockets. `https://github.com/WebAssembly/wasi-sockets`, 2023.

[24] M. Bauer and C. Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2021.

[25] M. Bélair, S. Laniepce, and J. Menaud. SNAPPY: Programmable kernel-level policies for containers. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2021.

[26] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Proc of EUROCRYPT*, May 1994.

[27] J. Benet. IPFS-content addressed, versioned, P2P file system. Technical report, Protocol Labs, 2014.

[28] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 1995.

[29] E. W. Biederman. Multiple Instances of the Global Linux Namespaces. In *Ottawa Linux Symposium (OLS)*, 2006.

[30] J. Bosamiya, W. S. Lim, and B. Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2022.

[31] V. Boyko. On the security properties of OAEP as an All-or-Nothing transform. In *Proc. of CRYPTO*, 1999.

[32] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in JavaScript bindings. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

[33] T. Bui, S. P. Rao, M. Antikainen, V. M. Bojan, and T. Aura. Man-in-the-Machine: Exploiting ill-secured communication inside the computer. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018.

[34] A. Bulekov, R. Jahanshahi, and M. Egele. Saphire: Sandboxing PHP applications with tailored system call allowlists. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[35] A. Bulekov, R. Jahanshahi, and M. Egele. Saphire: Sandboxing PHP applications with tailored system call allowlists. In *30th USENIX Security Symposium*, 2021.

[36] Bun. Bun is a fast all-in-one JavaScript runtime. `https://bun.sh/`, 2023.

[37] C. Canella, M. Werner, D. Gruss, and M. Schwarz. Automating seccomp filter generation for Linux applications. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2021.

[38] Canonical. AppArmor. `https://apparmor.net`, 2022.

[39] V. Casola, A. De Benedictis, M. Rak, and U. Villano. Monitoring data security in the cloud: A security sla-based approach. In *Security and Resilience in Intelligent Data-Centric Systems and Communication Networks*, Intelligent Data-Centric Systems. 2018.

[40] V. Casola, A. De Benedictis, M. Rak, and U. Villano. Security-by-design in multi-cloud applications: An optimization approach. *Information Sciences*, 454-455, July 2018.

[41] G. Christou, G. Ntousakis, E. Lahtinen, S. Ioannidis, V. P. Kemerlis, and N. Vasilakis. BinWrap: Hybrid protection against native Node.js add-ons. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.

[42] Chromium. Sandbox. `https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md`, 2023.

[43] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *Proc. of ESORICS*, September 2009.

[44] B. Coenen. feat(wasi): add rename for a directory + fix remove_dir. `https://github.com/wasmerio/wasmer/commit/e0e12f9d9ff41a512e44bd497324e`, 2021.

[45] D. Coldewey. Cloudflare DNS goes down, taking a large piece of the internet with it. `https://techcrunch.com/2020/07/17/cloudflare-dns-goes-down-taking-a-large-piece-of-the-internet-with-it`, 2020.

[46] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[47] containers. Bubblewrap. https://github.com/containers/bubblewrap, 2022.

[48] J. Corbet. File-based capabilities. https://lwn.net/Articles/211883/, 2006.

[49] J. Corbet. BPF: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/, 2014.

[50] J. Corbet. KRSI. https://lwn.net/Articles/808048/, 2019.

[51] CVE Mitre. Gitlab Exiftool vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22205, 2021.

[52] P. David. hyperfine. https://github.com/sharkdp/hyperfine, 3 2023.

[53] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Multi-dimensional indexes for point and range queries on outsourced encrypted data. In *Proc. of GLOBECOM*, December 2021.

[54] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Encryption policies for regulating access to outsourced data. *ACM TODS*, 35(2), April 2010.

[55] Debian. Service sandboxing. https://wiki.debian.org/ServiceSandboxing, 2023.

[56] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the Mining Software Repositories Conference (MSR)*, 2018.

[57] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[58] Deno Land. Deno 1.24 release notes – improved ffi call performance. https://deno.com/blog/v1.24#improved-ffi-call-performance, 2022.

[59] Deno Land. Deno 1.25 release notes – FFI API improvements. https://deno.com/blog/v1.25#ffi-api-improvements, 2022.

[60] Deno Land. Deno: JavaScript runtime. https://deno.land/, 2022.

[61] Deno Land. Deno Permission Model. https://deno.land/manual/getting_started/permissions#permissions, 2022.

[62] Deno Land. Deno standard library for testing. https://deno.land/std/testing, 2022.

[63] Deno Land. Deno Subprocess. https://deno.land/manual@v1.26.0/examples/subprocess, 2022.

[64] Deno Land. Deno Workers. https://deno.land/manual@v1.26.0/runtime/workers, 2022.

[65] Deno Land. Node compatibility mode. https://deno.land/manual/node/compatibility_mode, 2022.

[66] Deno Land. Deno API. https://doc.deno.land/deno/stable/, 2023.

[67] Deno Land. Deno Permission Model. https://deno.land/manual/getting_started/permissions, 2023.

[68] Deno Land. Rusty V8 bindings. https://github.com/denoland/rusty_v8, 2023.

[69] Deno Land. sqlite3 bindings for Deno. https://deno.land/x/sqlite3, 2023.

[70] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[71] Docs.rs. Tokio. https://docs.rs/tokio/0.2.0/tokio/index.html, 2022.

[72] dsherret. dax. https://github.com/dsherret/dax, 2022.

[73] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[74] J. Edge. Seccomp and deep argument inspection. https://lwn.net/Articles/822256/, 2020.

[75] Emscripten Contributors. Emscripten toolchain. https://emscripten.org/, 2023.

[76] A. Ene, M. Kolny, and A. Brown. wasi-threads. `https://github.com/WebAssembly/wasi-threads`, 2023.

[77] Falco. Falco, 2022.

[78] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. Containing malicious package updates in npm with a lightweight permission system. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021.

[79] Filecoin. Filecoin. `https://filecoin.io/`, 2023.

[80] W. Findlay, D. Barrera, and A. Somayaji. BPFContain: Fixing the soft underbelly of container security. *arXiv*, 2021.

[81] W. Findlay, A. Somayaji, and D. Barrera. bpfbox: Simple precise process confinement with eBPF. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2020.

[82] FUSE Project. FUSE: The Linux kernel documentation. `https://www.kernel.org/doc/html/next/filesystems/fuse.html`, 2023.

[83] FUSE Project. libfuse: Reference implementation for communicating with the fuse kernel module. `https://github.com/libfuse/libfuse`, 2023.

[84] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. Sledge: A serverless-first, light-weight Wasm runtime for the edge. In *Proceedings of the International Middleware Conference (MIDDLEWARE)*, 2020.

[85] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang. Houdini's escape: Breaking the resource rein of Linux control groups. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[86] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion. Enclosure: Language-based restriction of untrusted libraries. In *ASPLOS*, 2021.

[87] Google. Minijail. `https://google.github.io/minijail/`, 2022.

[88] Google. Sandbox2, 2022.

[89] Google. zx. `https://github.com/google/zx`, 2022.

[90] V. Gough. EncFS: an Encrypted Filesystem for FUSE. `https://vgough.github.io/encfs/`, 2023.

[91] B. Gregg. BPF internals. 2021. USENIX Large Installation System Administration Conference (USENIX LISA).

[92] H. Tao. BPF: Introduce ternary search tree for string key. https://lore.kernel.org/bpf/20220331122822.14283-1-houtao1@huawei.com, 2022.

[93] H. Tao. BPF: Support for string key in hash-table. https://lore.kernel.org/bpf/20211219052245.791605-1-houtao1@huawei.com, 2022.

[94] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[95] HackerOne. External SSRF and Local File Read via video upload due to vulnerable FFmpeg HLS processing. https://hackerone.com/reports/1062888, 2021.

[96] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *USENIX ATC*, 2019.

[97] D. Irvine. Maidsafe distributed file system. Technical report, MaidSafe, 2010.

[98] J. Edge. A seccomp overview. https://lwn.net/Articles/656307/, 2015.

[99] 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. *arXiv*, 2023.

[100] 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.

[101] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. WaVe: A verifiably secure WebAssembly sandboxing runtime. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2022.

[102] Z. Junyuan and R. Guo. Phantom attack: Evading system call monitoring. 2021. DEF CON.

[103] J. Karásek, R. Burget, and O. Morský. Towards an automatic design of non-cryptographic hash function. In *TSP*, 2011.

[104] M. Kehoe. eBPF: The next power tool of SREs. 2022. USENIX SREcon.

[105] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[106] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2022.

[107] Kudelski Security. OramFS: ORAM filesystem written in Rust. `https://github.com/kudelskisecurity/oramfs`, 2023.

[108] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of WebAssembly. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[109] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen. Automatic policy generation for Inter-Service access control of microservices. In *USENIX Security*, 2021.

[110] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. In *USENIX ATC*, 2017.

[111] libbpf. libbpf, 2022.

[112] Linux manual. bpf(2). `https://man7.org/linux/man-pages/man2/bpf.2.html`, 2022.

[113] Linux manual. ldd(1). `https://man7.org/linux/man-pages/man1/ldd.1.html`, 2022.

[114] Linux manual. strace(1). `https://man7.org/linux/man-pages/man1/strace.1.html`, 2022.

[115] Linux manual. accept. `https://man7.org/linux/man-pages/man2/accept.2.html`, 2023.

[116] Linux manual. hier. `https://man7.org/linux/man-pages/man7/hier.7.html`, 2023.

[117] Linux manual. listen. `https://man7.org/linux/man-pages/man2/listen.2.html`, 2023.

[118] Linux manual. pipe. `https://man7.org/linux/man-pages/man2/pipe.2.html`, 2023.

[119] Linux manual. socketpair. https://man7.org/linux/man-pages/man2/socketpair.2.html, 2023.

[120] M. K. Lau. BPF map-in-map support. https://www.mail-archive.com/netdev@vger.kernel.org/msg159387.html, 2017.

[121] M. S. Miller. Draft proposal for ses (secure ecmascript). https://github.com/tc39/proposal-ses, 2022.

[122] M. Salaün. Landlock: unprivileged access control. https://docs.kernel.org/userspace-api/landlock.html, 2022.

[123] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The Java unsafe API in the wild. *Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN)*, 2015.

[124] E. Mathews. Amazon cloud outage hits major websites, streaming apps. https://www.reuters.com/article/amazon-com-outages-idCAKBN2IM1U0, 2021.

[125] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Conference (USENIX)*, 1993.

[126] M. McCaskey. Prevent parent directory from being opened without being preopened wasi. https://github.com/wasmerio/wasmer/pull/463, 2019.

[127] Microsoft. ebpf for windows. https://microsoft.github.io/ebpf-for-windows/, 2023.

[128] J. Mogul, R. Rashid, and M. Accetta. The packer filter: An efficient mechanism for user-level network code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[129] MozillaWiki. Sandbox Architecture. https://wiki.mozilla.org/Security/Sandbox/Process_model, 2023.

[130] MUSEC. libpreopen. https://github.com/musec/libpreopen, 2023.

[131] A. Nakryiko. BPF CO-RE. https://nakryiko.com/posts/bpf-core-reference-guide/, 2021.

[132] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[133] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S.Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium*, 2020.

[134] netblue30. Firejail. `https://firejail.wordpress.com/`, 2022.

[135] NIST. Application Container Security Guide. `https://csrc.nist.gov/publications/detail/sp/800-190/final`, 2023.

[136] npm. Npm packages. `https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages.html`, 2020.

[137] npm. fluent-ffmpeg. `https://www.npmjs.com/package/fluent-ffmpeg`, 2022.

[138] npm. gm. `https://www.npmjs.com/package/gm`, 2022.

[139] npm. sane. `https://www.npmjs.com/package/sane`, 2022.

[140] npm. bcrypt. `https://www.npmjs.com/package/bcrypt`, 2023.

[141] npm. sharp. `https://www.npmjs.com/package/sharp`, 2023.

[142] G. Ntousakis, S. Ioannidis, and N. Vasilakis. Detecting third-party library problems with combined program analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[143] OpenJS Foundation. Worker threads. `https://nodejs.org/api/worker_threads.html`, 2022.

[144] OpenJS Foundation. Node js API. `https://nodejs.org/docs/latest/api/`, 2023.

[145] OpenJS Foundation. Node Permissions. `https://nodejs.org/api/permissions.html`, 2023.

[146] OpenJS Foundation. Node.js V8 APIs. `https://nodejs.org/api/v8.html`, 2023.

[147] OpenJS Foundation and Joyent. Node.js. `https://nodejs.org`, 2022.

[148] oven sh. Webcore bindings. `https://github.com/oven-sh/bun/tree/main/src/bun.js/bindings/webcore`, 2023.

[149] P. Simek. Proposal for VM2: Advanced vm/sandbox for Node.js. `https://github.com/patriksimek/vm2`, 2022.

[150] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz. Nojitsu: Locking down javascript engines. In *NDSS*, 2020.

[151] G. Paul, F. Hutchison, and J. Irvine. Security of the MaidSafe vault network. In *Wireless World Research Forum Meeting 32*, May 2014.

[152] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova. eWASM: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11), October 2020.

[153] R. Dahl. 10 things i regret about Node.js. 2018. JSConf EU.

[154] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow. Keeping safe rust safe with galeed. In *ACSAC*, 2021.

[155] R. L. Rivest. All-or-nothing encryption and the package transform. In *Proc. of FSE*, January 1997.

[156] M. Rosemain and R. Satter. Millions of websites offline after fire at French cloud services firm. `https://www.reuters.com/article/us-france-ovh-fire-idUSKBN2B20NU`, 2021.

[157] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi. SEApp: Bringing mandatory access control to Android apps. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[158] M. Salaün. Landlock. `https://landlock.io/`, 2022.

[159] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *USENIX Security*, 2022.

[160] F. Schwarz and C. Rossow. SENG, the SGX-Enforcing network gateway: Authorizing communication from shielded clients. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[161] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The misuse of Android Unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[162] Sia. Sia, decentralized data storage. `https://sia.tech/`, 2023.

[163] M. Sipos, F. H. Fitzek, D. E. Lucani, and M. V. Pedersen. Distributed cloud storage using network coding. In *IEEE CCNC*, January 2014.

[164] Slim.AI. SlimToolKit. `https://github.com/slimtoolkit/slim`, 2023.

[165] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 2001.

[166] Snyk. State of Open Source Security 2022. `https://snyk.io/reports/open-source-security/`, 2022.

[167] Snyk. Zip slip vulnerability. `https://snyk.io/research/zip-slip-vulnerability`, 2022.

[168] Stack Overflow Insights. Annual survey of the Stack Overflow community. `https://survey.stackoverflow.co/2022/`, 2022.

[169] C. Staicu, M. Pradel, and B. Livshits. Synode: Understanding and automatically preventing injection attacks on Node.js. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[170] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2023.

[171] C. A. Staicu, S. Rahaman, A. Kiss, and M. Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. In *32nd USENIX Security Symposium*, 2023.

[172] J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript(js.js): Sandboxing third-party scripts. In *Proceedings of the USENIX Conference on Web Application Development (USENIX WebApps)*, 2012.

[173] tesseract-ocr. Tesseract. `https://github.com/tesseract-ocr/tesseract`, 2023.

[174] The Cilium Authors. Cilium. `https://cilium.io`, 2023.

[175] The Cilium Authors. Cilium GitHub repository. `https://cilium.io`, 2023.

[176] The coreutils Authors. uutils coreutils. `https://github.com/uutils/coreutils`, 2023.

[177] The Falco Authors. What is the performance overhead or resource utilization of Falco? https://falco.org/about/faq/#what-is-the-performance-overhead-or-resource-utilization-of-falco, 2023.

[178] The Freenet Project. Freenet. https://freenetproject.org/, 2023.

[179] The kernel development community. LSM eBPF Programs. https://docs.kernel.org/bpf/prog_lsm.html, 2023.

[180] The Kubernetes Authors. Restrict a Container's Syscalls with seccomp. https://kubernetes.io/docs/tutorials/security/seccomp/, 2023.

[181] The Tetragon authors. Tetragon. https://tetragon.cilium.io/, 2023.

[182] D. Trautwein, A. Raman, G. Tyson, I. Castro, W. Scott, M. Schubotz, B. Gipp, and Y. Psaras. Design and evaluation of IPFS: A storage layer for the decentralized web. In *Proc. of SIGCOMM*, August 2022.

[183] TryGhost. Asynchronous, non-blocking SQLite3 bindings for Node.js. https://www.npmjs.com/package/sqlite3, 2023.

[184] V8 project. Unsafe fast JS calls. https://v8.dev/blog/v8-release-87#unsafe-fast-js-calls, 2020.

[185] V8 project. What is v8? https://v8.dev/, 2022.

[186] V8 project. Webassembly compilation pipeline. https://v8.dev/docs/wasm-compilation-pipeline, 2023.

[187] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security*, 2019.

[188] N. Vasilakis, B.Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Proceedings of the 2018 NDSS*, 2018.

[189] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, flexible application compartmentalization. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[190] N. Vasilakis, C. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel. Preventing dynamic library compromise on Node.js via RWX-based privilege reduction. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[191] N. Vasilakis, C. A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel. Preventing dynamic library compromise on node.js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC CCS*, 2021.

[192] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *EuroSys*, 2022.

[193] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li. Mining sandboxes for linux containers. In *ICST*, 2017.

[194] WasmEdge. wasmedgec AOT compiler. `https://wasmedge.org/book/en/cli/wasmedgec.html`, 2023.

[195] I. Wasmer. Wasmer Runtime. `https://wasmer.io/`, 2023.

[196] WebAssembly. WASI Libc. `https://github.com/WebAssembly/wasi-libc`, 2023.

[197] WebAssembly. WASI SDK. `https://github.com/WebAssembly/wasi-sdk`, 2023.

[198] WebAssembly. The webassembly system interface. `https://wasi.dev`, 2023.

[199] WinFsp. Windows file system proxy. `https://winfsp.dev/`, 2023.

[200] C. Wright, C. Cowan, J. Morris, James, S. Smalley, and G. Kroah-Hartman. Linux Security Module framework. In *Ottawa Linux Symposium*, 2002.

[201] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2012.

[202] E. Wyss, A. Wittman, D. Davidson, and L. De Carli. Wolf at the Door: Preventing Install-Time Attacks in npm with Latch. In *ASIACCS*, 2022.

[203] W. Zhang, P. Liu, and T. Jaeger. Analyzing the overhead of file protection by Linux Security Modules. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[204] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *USENIX Security*, 2019.

[205] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.