ABSTRACT

Deno is a runtime for JavaScript and TypeScript that is receiving great interest by developers, and is increasingly used for the construction of back-ends of web applications. A primary goal of Deno is to provide a secure and isolated environment for the execution of JavaScript programs. It also supports the execution of subprocesses, unfortunately without providing security guarantees.

In this work we propose Cage4Deno, a set of modifications to Deno enabling the creation of fine-grained sandboxes for the execution of subprocesses. The design of Cage4Deno satisfies the compatibility, transparency, flexibility, usability, security, and performance needs of a modern sandbox. The realization of these requirements partially stems from the use of Landlock and eBPF, two robust and efficient security technologies. Significant attention has been paid to the design of a flexible and compact policy model consisting of RWX permissions, which can be automatically created, and deny rules to declare exceptions. The sandbox effectiveness is demonstrated by successfully blocking a number of exploits for recent CVEs, while runtime experiments prove its efficiency. The proposal is associated with an open-source implementation.

CCS CONCEPTS

• Security and privacy → Software and application security; Access control; File system security; Usability in security and privacy.

KEYWORDS

Sandboxing, Access control, JavaScript runtime, Deno, Subprocess

1 INTRODUCTION

JavaScript is currently one of the most popular programming languages [68]. 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 [24, 69, 74, 75, 78], 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 [44], 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 [17]. 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 [19].

Research has shown that third-party code accounts for a great portion of a JavaScript application codebase [74]. The 2022 State of Open Source Security [66] 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 [21], which enables the execution of Node packages in Deno. Third-party modules often depend on subprocesses [23, 32, 69]. There is a broad spectrum of programs that fall into this category, ranging from Linux utilities to handle files [56], to programs that process media (e.g., image/video conversions [55], metadata removal [54]), 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\textsuperscript{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 dimg 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 \cite{14} and TikTok \cite{36}). 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.

2 BACKGROUND

This section overviews the frameworks used in our proposal.

\textsuperscript{1}https://github.com/unibg-seclab/cage4deno

2.1 Deno

Deno \cite{44} is a runtime for JavaScript and TypeScript embedding V8 \cite{72}, an open-source high-performance JavaScript and Web-Assembly 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\textunderscore v8, which defines the set of bindings to the V8’s API; (ii) deno\textunderscore core, a package built on top of rusty\textunderscore v8 implementing the abstractions required to run JavaScript (i.e., the \textit{JsRuntime}); and (iii) deno, a package providing the Deno executable and the user-facing API.

Among the abstractions implemented in deno\textunderscore 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 \textit{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 accessible through \textit{ops} are stored by deno\textunderscore core in a resource table, and are uniquely identified by integers, similarly to the Unix concept of file descriptor.

The \textit{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(\textit{\ldots}) 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 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 \cite{17}.

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 mit-
igate 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 lock_create_ruleset() and lock_add_rule() system calls, and subsequently activated with lock_restrict_self(). Rules are inherited by the children created after sandbox activation, and the actions available can only be further restricted. The process is detailed in Figure 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.

2.3 eBPF

The Extended Berkeley Packet Filter (eBPF, henceforth referred to as BPF) [33, 41] 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 [47, 51] as it elegantly addresses portability following a Compile Once – Run Everywhere approach [51]. 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_NET_ADMIN (to load tracing-related programs) and CAP_PERFMON (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.

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) [13], also known as BPF LSM. This feature permits to attach BPF programs to LSM hooks, and thus enforce access control.

3 CAGE4DENO

This Section gives an overview of Cage4Deno, clarifying the threat model. We also introduce our design objectives (Section 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 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 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 [79]) 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.\footnote{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.} Compared to other well-known general-purpose...
sanding solutions like Minijail [30], and Google Sandbox2 [31]. 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 4.

3.2 Threat model
Similarly to other research proposals [16, 70, 75], 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 [52]. The constant and extensive exposure to untrusted inputs in web applications may enable the attacker to trigger these vulnerabilities, 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 [15, 50, 80] 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 compromised 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 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 3.2 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.

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 5, while additional BPF implementation details that focus on performance improvement are discussed in Appendix A.

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 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 3.1), 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 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(1); 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 (2). Due to this additional step, this is the only part of the execution in which Cage4Deno requires additional privileges compared to Deno. These privileges 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 (2) 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 (2). 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 (2). 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 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 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).

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 (Section 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 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 [22] 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 2.2). Should any of them try to access a filesystem location not listed in the policy, the request is denied.

---

Listing 1: An example of child process in Deno

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

Listing 2: Restricting a child process using policyId

```
let a = Deno.run({ cmd: ["tar", "zxf", "input.tgz", "-C", "output"], policyId:"tarPolicy" });
await a.status();
```
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 [8] and Control Groups [28], RWX rules also permit to grant access with file-level granularity on the whole filesystem, and not only on single volumes (Objective O3).

### 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 [64]. 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 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.

#### 4.3.1 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). 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/ibf 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 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 $D_i$ in the policy, the maximum depth of the trie is $N/2$ (in the case of a path structured as [ / ] $^*$). When a hashmap is used to implement the trie, each lookup (used to jump from the parent to the child) takes O(1) time, thus traversing the trie takes O(N) time (worst case upper bound).

#### 4.3.2 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 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 [48] 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 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 · 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.

#### 4.3.3 BPF policy attachment

To enforce access control at runtime, the sandboxer must be able to retrieve the $Map_{policy}$ according to

---

Listing 3: RWX+D permissions associated with tar

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

---

Table 1: Deny prefix tree (a) and BPF Map (b)

<table>
<thead>
<tr>
<th>r</th>
<th>Trie prefix $p_i$</th>
<th>Hash $h(p_i)$</th>
<th>isLeaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/</td>
<td>$h(p_1)$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>/home</td>
<td>$h(p_2)$</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>/home/user</td>
<td>$h(p_3)$</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>/home/user/data</td>
<td>$h(p_4)$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>/home/user/lib</td>
<td>$h(p_5)$</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>/media</td>
<td>$h(p_6)$</td>
<td>1</td>
</tr>
</tbody>
</table>

---

The maximum path length is bounded to 4096 chars in `linux/limits.h`.
the argument provided by the developer to the Deno.run(), and then to attach it to the sandbox 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\cite{47}. 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 4.3.2).

The first group of BPF programs, namely the ones used to trace the process lifecycle, are responsible for maintaining in memory the map \textit{Map}$_{\text{task}}$ of processes running on the system that are subject to the restrictions imposed by Cage4Deno. Specifically, \textit{Map}$_{\text{task}}$ associates each thread identifier to the proper \textit{Map}$_{\text{policy}}$. The entries in \textit{Map}$_{\text{task}}$ are updated when one of the hooks listed in Table 2a is reached. For instance, when a traced process issues the clone syscall, the tracepoint \texttt{tp_btf/sched_process_for}k is reached, the related BPF tracing program executed, and the new child process, inheriting the policy of the parent, is added to \textit{Map}$_{\text{task}}$.

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 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 \textit{Map}$_{\text{task}}$, and accordingly to perform the sequence of lookups in \textit{Map}$_{\text{policy}}$ to allow or deny the request.

Table 2: Hooks and tracepoints monitored by Cage4Deno

<table>
<thead>
<tr>
<th>Access control hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>lsm/path_mknod</td>
</tr>
<tr>
<td>lsm/path_mkdir</td>
</tr>
<tr>
<td>lsm/path_link</td>
</tr>
<tr>
<td>lsm/path_synlink</td>
</tr>
<tr>
<td>lsm/file_open</td>
</tr>
<tr>
<td>lsm/path_rename</td>
</tr>
<tr>
<td>lsm/path_rmdir</td>
</tr>
<tr>
<td>lsm/path_unlink</td>
</tr>
</tbody>
</table>

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 \cite{31} leverages \texttt{ldd} (acronym for \textit{List Dynamic Dependencies}) to retrieve the dependencies used by programs available on the host as \textit{Executable and Linkable Format} (ELF) files; or in SlimToolkit \cite{61}, where a containerized service is run against a test suite to automatically create Seccomp \cite{9} and AppArmor \cite{10} 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.

The solution we adopted was to develop \texttt{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, \texttt{dmng} relies on both \texttt{ldd} and \texttt{strace}. The former was preferred to \texttt{objdump} and \texttt{readelf} since it can be used to recover the so called \textit{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 \cite{3}, the developer can use \texttt{dmng} to run a test suite against a program (or a script). This approach allows the developer to reuse existing tests to generate RWX rules that satisfy multiple execution paths of the binary. By using \texttt{strace}, \texttt{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 of threads, along with their arguments, are captured and saved to a log file. The log is then used by \texttt{dmng} to generate the RWX rules accordingly. The list of syscalls monitored comprises the ones wrapped by the standard system functions: \texttt{execve()}, \texttt{open()}, \texttt{create()}, \texttt{link()}, \texttt{mkdir()}. The \texttt{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.

Listing 4: Semi-automatic generation of tar policy (Listing 3)

```bash
# Set the active policy_name for a program
1 dmng -c tar --setcontext tarPolicy

# Add the input to the template
2 dmng -c tar -t -s

# Add the output to the template
3 dmng -c tar -a output -p "-w"

# Add the dependencies
4 dmng -c tar -a output/misc -d

# Serialize the entries into a JSON file
5 dmng --serialize --tar_policy_file
```
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-execute ($W^X$) 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 until 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 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 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 [77]. To address this limitation, the current approach could be complemented by either source or binary code analysis. Relevant works (e.g., [16]) use static analysis to pinpoint the system calls, however they do not keep track of parameter values. On the other hand, AutoArmor [45] 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.

### 6 EXPERIMENTS

In this Section we present the experimental evaluation of Cage4Deno. The evaluation considers two aspects of our solution: exploit mitigation (Section 6.1), and performance overhead (Section 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).

#### 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.

<table>
<thead>
<tr>
<th>CVE ID</th>
<th>Utility</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-1897</td>
<td>FFmpeg v3.2.5</td>
<td>Video processing</td>
</tr>
<tr>
<td>CVE-2016-1898</td>
<td>FFmpeg v3.2.5</td>
<td>Video processing</td>
</tr>
<tr>
<td>CVE-2019-12921</td>
<td>GraphicsMagick v1.3.31</td>
<td>Image processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVE-2016-6321</td>
<td>GNU Tar v1.29</td>
<td>Archive decompression</td>
</tr>
<tr>
<td>CVE-2019-20916</td>
<td>Pip v19.0.3</td>
<td>Dependency fetch</td>
</tr>
<tr>
<td>CVE-2022-30333</td>
<td>UnRAR v6.11</td>
<td>Archive decompression</td>
</tr>
</tbody>
</table>

#### Table 3: Sample of CVEs mitigated by Cage4Deno. Despite being discovered in 2016, CVEs 1897, 1898, and 6321 have seen recent exploitation in 2018 [67] and 2021 [36]

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 functions 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 5. Table 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 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 astroden and fast_forward from the former, fluent-ffmpeg and gm from the latter (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.

## 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 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 sandboxing solutions: (i) Minijail [30], and (ii) Sandbox2, a key component of the Sandbox API [31] framework. As shown in Figure 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, which further degrades performance. The other hand, Cage4Deno only pays the overhead of evaluating the Landlock security checks. This validates our design choices, proving that our solution can provide significantly better performance with respect to industrial general-purpose sandboxing solutions. Moreover, the flame graphs used for the analysis are available in the repository of the project.

### Table 4: Preliminary test showing the execution time of utilities reported in Table 3 over 500 runs (as $\mu \pm \sigma$)

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

Figure 5: Deterioration of overhead varying the policy size

---

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

6.2.2 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` [5], 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` [4], a Linux utility to configure and shape the network traffic. The test is executed four times with the following policy configurations: (i) 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 5b confirm the trend shown in Figure 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 node-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.

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 [3, 20, 49, 57–59, 71, 74, 75, 78].

Secure EcmaScript (SES) [49] 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 [59]. 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 [74]. 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. Compartment policies 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 [75] 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 [3] 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 [78] 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 [29, 37, 43, 52, 60, 62, 63, 65, 73, 76]. Galeed [62], PKRU-Safe [43], and NojitS [60] 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++). NojitS 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 [52] 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 eases 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 [7, 12, 16, 30, 31, 42, 53]. TRON [7] 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 sandboxers is MBOX [42]. 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 intercepts system calls using Seccomp filters, and relies on ptrace to enforce permissions. However, the use of ptrace is prone to TOCTOU attacks [39], 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 [16, 61, 77]. This effectively limits the capabilities of an attacker exploiting the program, and reduces the attack surface of the kernel [46]. 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 filesystem resources. These solutions can be used in conjunction with Cage4Deno to reduce the attacker capabilities and the attack surface of the kernel.

In Section 6 we have already compared Cage4Deno to other general-purpose sandboxing solutions such as Minijail [30], and Sandbox2 [31]. 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 R0X 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 established 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 [53] and Bubblewrap [12]; 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 [2, 6, 11, 25–27, 38]. BPFBox [27] and BPFContain [26] 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 R0X permissions so to guarantee low runtime overhead, and (iii) we provide a tool for the automatic generation of policies. Snappy [6] 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. [38] 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 [11] and Falco [25]. 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.

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.

ACKNOWLEDGMENTS

We thank our shepherd Dokyung Song and the reviewers for their valuable feedback. The work was supported by the European Commission within the GLACIATION project (No 101070141).
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 4.3), the current implementation of the BPF framework does not provide support for map-in-map structures [48]. Moreover, support for using string keys in a map is still limited [35]. At the time of writing, a patch for providing this option, along with efficient ternary search support to lookup the map [34], 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\(\text{policy}\), 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 \(djb2\) [40], 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\(\text{policy}\), 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: \texttt{/home/user/data, /home/user/1ib} and \texttt{/media}. Also, let us assume that the \(djb2\) hash of \texttt{/home/user/1ib} and \texttt{/media} collide. Instead of using the deny rules to build a prefix tree, we can form the string prefixes stored in the trie to fixed size integers. A cryptographic low-complexity incremental hash function proposed by D. J. Bernstein.

\(\texttt{djb2}(\text{prefix})\) means that every time a key is found in the policy map Map\(\text{policy}\), the verifier incurs an \(O(N \cdot L)\) extra time, where \(L\) is the length of the longest collision chain. Similarly to other research proposals [40], the results presented in our Experimental Evaluation showed a limited impact associated with the presence of collisions (see Section 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 4.3 and 4.1 we illustrated our construction of the verifier, detailing the role of Map\(\text{task}\) and Map\(\text{policy}\). 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\(\text{task}\) 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\(\text{policy}\), 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

<table>
<thead>
<tr>
<th>BPF Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deny rule hash</td>
</tr>
<tr>
<td>\texttt{djb2(D_r1)}</td>
</tr>
<tr>
<td>\texttt{djb2(D_r2, D_r3)}</td>
</tr>
</tbody>
</table>

Table 5: Map\(\text{policy}\) with separate hash chaining (assuming the hashes of \texttt{/home/user/1ib} and \texttt{/media} colliding)

Algorithm 1 Modified deny rule verifier

1. procedure DenyVerifier\((S_r)\)
2. \(\text{task} \leftarrow \text{bpf\_get\_current\_task\_btf()}\)
3. if \(\neg\text{task} \in \text{Map}\_\text{task}\) then
   4. return 0 \(\triangleright\) Grant request
5. \(\text{Map}\_\text{policy} \leftarrow \text{Map}\_\text{task}\ [\text{task}]\)
6. for each prefix in \(S_r\) do
7. \(\text{if} \ \text{djb2(prefix)} \in \text{Map}\_\text{policy}\) then
8. collision\_chain \(\leftarrow\) Map\(\text{policy}\ [\text{djb2(prefix)}]\)
9. \(\text{for} \ D_r \text{ in collision\_chain do}\)
10. \(\text{if isPrefix}(D_r, \text{prefix}) \) then
11. return -EPERM \(\triangleright\) Deny request
12. return 0 \(\triangleright\) Grant request

Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses ASIA CCS ’23, July 10–14, 2023, Melbourne, VIC, Australia
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 5 reports the policy associated with GNU Tar. The policy has been generated using dmng according to the procedure described in Listing 4.

```json
Listing 5: 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"
      ]
    }
  }
}
```