UNIVERSITY OF BERGAMO

School of Doctoral Studies

Doctoral Degree in Engineering and Applied Sciences

XXXVI Cycle

SSD: ING-INF/05

Enforcing security boundaries and protecting application data

Advisor

Prof. Stefano Paraboschi

Doctoral Thesis

Gianluca OLDANI

Student ID 1040358

Academic year 2022/2023

Acknowledgments

The research described in the contents of this thesis has been supervised by Prof. Stefano Paraboschi (Università degli Studi di Bergamo). The works described have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825333 (Mosaicrown) and Horizon Europe – 2.4 programme under agreement No. 101070141 (GLACIATION).
I would like to thank Prof. Stefano Paraboschi for the precious guidance offered to me during this journey. My gratitude also goes to my colleagues of the Security Lab of Università degli Studi di Bergamo: Marco Abbadini, Michele Beretta, Dario Facchinetti and Matthew Rossi; their collaboration has been crucial to complete a large portion of the works described in this thesis. A final thank is due to all the people that supported me during my research, since it is thank to them that the most difficult parts of my journey have been overcome.

To my family, Chiara and the friends that supported me

Table of contents

Introduction

In the recent years, the main programming languages chosen by developers to develope their applications are high level scripting languages, such as: JavaScript, Python, PHP and Ruby. This is mainly due to the fact that application demands quick development times, in order to comply with fast evolving requirements and new features demand.
The aforementioned languages share their ease of use and the possibility of integrating third-party code into applications through package managers(e.g., *npm*, *pip*). In this thesis, great focus will be on JavaScript, due to the fact that it is one of the most diffused language for the development of applications. One of the main reason of its popularity in this scenario can be found in the fact that, while it is born as a scripting language to be run in the client-side of a web application, the creation of JavaScript runtimes enabled the execution of JavaScript server-side. Thank to this property, the entirety of the application stack can be developed through the same language, allowing front-end developers to reuse their skills also for back-end components.
While these properties are welcomed from a software development perspective, the characteristics of JavaScript, when used in the back-end, have led to severe security vulnerabilities. Among the main reasons of this problem, researchers have identified: *i)* common behaviors that can be used when JavaScript is executed with restricted privileges in the browser, are not secure when applied through JavaScript runtimes, *ii)* developers employ extensive use of third-party dependencies, even when only a small portion of these packages are necessary, leading to larger attack surfaces.
After acknowledging this issue, the research effort aimed at solving it has been mainly focused on resolving vulnerabilities introduced by JavaScript code, in order to solve the problems which can be mitigated within the language itself. While these efforts have been a huge improvement in right direction concerning the security of applications written in JavaScript, the source of vulnerabilities is not only limited to code written in JavaScript.
A common kind of third-party dependencies are libraries and tools written in low-level languages(e.g., C/C++ and Rust). This kind of component is usually employed when capabilities beyond the reach of JavaScript are needed(e.g., performance delivered, hardware capabilities).
The studies presented in the first chapters of this thesis are part of the line of work performed by the research community in this direction: mitigating security risk of JavaScript application when native components are employed. The first work presented is *Cage4Deno*, a set of modifications applied to the *Deno* JavaScript runtime in order to restrict privileges of native code run as a subprocess, following the least-privilege principle. After the preliminary work performed in *Cage4Deno*, it has been investigated the possibility of extending its security features to native libraries. `WASM` is a popular proposal which is able to run untrusted native code in a secure

sandbox. In this scenario, the thesis exposes the work done in order to achieve better security during the interaction between `WASM` and the underlying OS through `WASI`. While the results achieved in this scenario are in line with the performance already obtained through modern `WASI`-compliant runtimes, running `WASM` introduces a noticeable overhead compared to native code. This lead to the development of *NatiSand*, a set of modification for JavaScript runtimes which can apply access restriction to utilities that are employed both under form of subprocesses and shared libraries.
While the collection of these first research efforts is able to limit the impact of a successful breach of an application, the protection enforced on native components follows the least-privilege principle. This enhances the security of such dependencies, since they are no longer able to reach resources which are not needed to perform their duty. The highlighted issue in this scenario is that it has to be taken into consideration that access is still granted to resources actually needed by the native component: these are still reachable by attackers in case of a successful exploit. For this reason, in order to have a comprehensive protection that also includes resources inside the enforced security boundaries, the thesis addresses the problem of allowing access to resources through methods that are able to provide security guarantees but with a limited impact on the quality of the service provided by the protected application (e.g., latency and result quality). In order to take full advantage of the kind of deployments that are commonly used in a modern application scenario (e.g. the cloud), the protection techniques explored are applicable through distributed technologies that can be leveraged through all the major cloud providers.
The protection of data collection described in the last chapters of this thesis has been explored in order to achieve two different objectives: privacy protection of data points and information confidentiality. The contribution to the first topic details how it is possible to leverage Apache Spark to distribute a well-know single thread algorithm to enforce k-Anonymity: Mondrian. Concerning the technique employed to enforce data confidentiality, the thesis explores how it is possible to create indexes that can be stored on the client of a remote application in order to allow the execution of query on encrypted data, without sharing any decryption key with the host that may have been compromised.

Chapter 1. Presentation

1.1 Document Structure

This thesis is structured in seven chapters.

Chapter 1 describes structure of the thesis, gives an overview of the contents of each chapter and lists the publications that are the foundations of its contributions.

Chapter 2 describes the first research efforts done to create a sandbox for native components employed through the Deno JavaScript runtime. The product of this research direction is Cage4Deno: a set of modifications to Deno to enable the creation of fine-grained sandboxes for the execution of subprocesses. During the design of this first proposal, the following properties have been set as primary goals: compatibility, transparency, flexibility, usability, security, and low performance overhead. The realization of these requirements has been possible thanks to the usage of Landlock and eBPF, two robust and efficient Linux Security Modules. In addition to this, particular attention has been paid to the design of a flexible and compact policy model consisting of `RWX` permissions. The process of formulating such policies is supported by a toolchain, which can automatically create ruleset based on a test-suite provided by developers.

- Section 2.1 introduces the security gap concerning native utilities employed by JavaScript applications and highlight the contributions of Cage4Deno
- Section 2.2 describes in detail the current state of the Deno JavaScript runtime and gives an overview of the properties of the leveraged Linux Security Modules, Landlock and eBPF
- Section 2.3 gives a high level description of the architecture of Cage4Deno, describes the threat model addressed and the properties that are guaranteed
- Section 2.4 describes the policy model enforced by Cage4Deno and gives a detailed description of the enforcement procedure of both permissions and prohibitions
- Section 2.5 proposes a method to automatically generate policy rules for Cage4Deno given a test-suite
- Section 2.6 illustrates the experiments that have been done to assess that Cage4Deno is both able to successfully mitigated recent vulnerabilities and to incur in low overhead when its protections are in place

- Section 2.7 discusses related works
- Section 2.8 concludes the chapter

Chapter 3 describes how part of the techniques explored in Chapter 2 can also be applied to WebAssembly in order to guarantee additional security properties when executing native code. The use case for this research is similar to the one of Cage4Deno, since WebAssembly was originally designed to be run inside web browsers and the introduction of modern runtimes like Wasmtime and WasmEdge allows the execution of WebAssembly directly on various systems. In order to access system resources with a universal hostcall interface, a standardization effort named WebAssembly System Interface (WASI) is currently undergoing. With specific regard to the file system, runtimes must prevent hostcalls to access arbitrary locations, thus they introduce security checks to only permit access to a predefined list of directories. Previous works have assessed that this approach suffers from poor granularity, is recognized as error-prone and has led to several security issues. In this work it is illustrated how it is possible to replace the security checks in hostcall wrappers with eBPF programs, enabling the introduction of fine grained per-module policies. Preliminary experiments confirm that the approach introduces limited overhead to existing runtimes.

- Section 3.1 introduces the security gap concerning the execution of WebAssembly outside of the browser and illustrates the core idea of using BPF programs to strengthen WASI security properties.
- Section 3.2 describes in details the current state of WASI runtimes
- Section 3.3 explains the motivations of this research and details the threat model
- Section 3.4 illustrates the architecture developed to perform the proposed security checks
- Section 3.5 shows the results of the performance evaluation of our approach when applied to popular WebAssembly runtimes
- Section 3.6 discusses related works
- Section 3.7 concludes the chapter

Chapter 4 further extends the research done in Chapter 2 and 3, since it continues the research performed on JavaScript runtimes and also complements the file system protection with security guarantees concerning network and IPC resources. In this case, the research is again conducted on JavaScript runtimes due to the possibility of executing shared libraries. When these are leveraged, similarly to subprocess, no isolation guarantees are provided. This limitation affects many popular runtimes including Node.js, Deno, and Bun.

In this chapter it is described NatiSand, a component for JavaScript runtimes that leverages *Landlock*, *eBPF*, and *Seccomp* to control the filesystem, Inter-Process Communication (IPC), and network resources available to binary programs and shared libraries. Similarly to Cage4Deno, NatiSand does not require changes to the application code and offers to the user an easy interface. To demonstrate the effectiveness and efficiency of the proposed approach, the prototype of NatiSand has been integrated into Deno. The security assessment has been conducted through the reproduction of several vulnerabilities affecting shared libraries, showing how they are mitigated by NatiSand. In addition to that, an extensive experimental evaluation to assess the performance of NatiSand has been conducted, proving that the proposal is competitive with state-of-the-art code sandboxing solutions. The implementation is available open source.

- Section 4.1 introduces the security gap concerning the execution of native libraries through JavaScript runtimes and lists the contributions of the research exposed in the chapter.
- Section 4.2 describes Seccomp BPF
- Section 4.3 explains the motivations of this research and details the threat model
- Section 4.4 illustrates the objectives of the research exposed in the chapter and details the architecture of NatiSand
- Section 4.5 describes the policy model used to express the security guarantees enforced by NatiSand and proposes an approach to automatically generate policies
- Section 4.6 discusses the implementation of the prototype of NatiSand inside the Deno JavaScript runtime
- Section 4.7 illustrates the experiments that have been done to assess that NatiSand is both able to mitigate recent vulnerabilities while also exhibiting better performances compared to state-of-the-art solutions
- Section 4.8 discusses related works
- Section 4.9 concludes the chapter and discusses future research directions

Chapter 5 explores a solution to address the issue of handling in a secure way data that is included in the security boundary enforced through the research exposed in the previous chapters. Particularly, k-Anonymity and ℓ-diversity are the techniques investigated. These are two well-known privacy metrics that guarantee protection of individual data points in a dataset by obfuscating information that can disclose identities and sensitive information. Existing solutions for enforcing them implicitly assume to operate in a centralized scenario, since they require complete visibility over the dataset to be anonymized, and can therefore have limited applicability in anonymizing large datasets. In order for this techniques to be applied in an environment more similar to the one of previous chapters, the research presented proposes a solution that extends Mondrian (an efficient and effective approach designed for achieving k-anonymity) for enforcing both k-anonymity and ℓ-diversity over large datasets in a distributed manner, leveraging the parallel computation of multiple workers.

- Section 5.1 introduces the limitations of executing Mondrian in a centralized fashion and exposes the contributions of the research
- Section 5.2 gives an overview of k-anonymity, ℓ-diversity and the discusses the Mondrian algorithm
- Section 5.3 describes form a high level perspective the architecture implementation to distribute the Mondrian algorithm
- Section 5.4 illustrates the techniques employed to distributed data among the cloud workers and discusses alternative approaches
- Section 5.5 presents the anonymization techniques supported by the developed prototype
- Section 5.6 explains the metrics used to assess the quality of the developed approach and how these are calculated
- Section 5.7 describes the technical details of the prototype implemented through Apache Spark
- Section 5.8 exposes and discusses experimental results
- Section 5.9 discusses related works
- Section 5.10 concludes the chapter

Chapter 6 presents an approach for indexing encrypted data stored on the host of a vulnerable web application to enable provider-side evaluation of queries. The approach designed supports the evaluation of point and range conditions on multiple attributes. The core component of the proposal is a spatial-based algorithm that partitions tuples to produce a clustering that ensures efficient query execution. Query translation and processing require the client to store a compact map. The experiments, evaluating query performance and client-storage requirements, confirm the efficiency enjoyed by our solution.

- Section 6.1 introduces the addressed problem and the core ideas of the proposed architecture to enable queries over encrypted data store on a remote host

- Section 6.2 illustrates basic concept and presents the definition of the terms used in the chapter
- Section 6.3 describes the partitioning procedure applied over protected data collections
- Section 6.4 illustrates how indexes are created from the partitioned dataset
- Section 6.5 presents format of the maps that are stored on the client of the web application
- Section 6.6 describes how query are translated in order to be executed on the encrypted data collections
- Section 6.7 exposes the technical details of the implemented prototype and discusses experimental results
- Section 6.8 discusses related works
- Section 6.9 concludes the chapter

Chapter 7 draws the conclusions of the thesis and discusses future work.

1.2 Publications

In this section are listed the publications produced during the PhD course that compose the basis for this thesis.

- Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi and Stefano Paraboschi. **Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses** . "*ASIA CCS '23: ACM Asia Conference on Computer and Communications Security*", pp. 149–162. 2023.
- Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi and Stefano Paraboschi. **Leveraging EBPF to Enhance Sandboxing of WebAssembly Runtimes**. "*ASIA CCS '23: ACM Asia Conference on Computer and Communications Security*", pp. 1028–1030. 2023.
- Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi and Stefano Paraboschi. "**Lightweight Cloud Application Sandboxing**" . *IEEE International Conference on Cloud Computing Technology and Science (IEEE CLOUDCOM 2023).*

- Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi and Stefano Paraboschi. **NatiSand: Native Code Sandboxing for JavaScript Runtimes** . To appear in "*26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023)*".

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi and Pierangela Samarati. "**Scalable distributed data anonymization**". *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 401-403. IEEE, 2021.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi and Pierangela Samarati. "**Artifact: Scalable distributed data anonymization**" . *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 450-451. IEEE, 2021.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Giovanni Livraga, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi and Pierangela Samarati. "**Scalable Distributed Data Anonymization for Large Datasets**" . *IEEE Transactions on Big Data (Volume: 9, Issue: 3, 01 June 2023)*, pp. 818 - 831. IEEE, 2022.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi and Pierangela Samarati. "**Multi-dimensional indexes for point and range queries on outsourced encrypted data**" . *2021 IEEE global communications conference (GLOBECOM)*. IEEE, 2021.

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi and Pierangela Samarati. "**Multi-Dimensional Flat Indexing for Encrypted Data Databases**". Under submission.

- Marco Abbadini, Michele Beretta, Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi and Pierangela Samarati. "**Supporting Data Owner Control in IPFS Networks**" . *2024 IEEE International Conference on Communications (ICC).*

Chapter 2. Protect Deno subprocesses through LSMs

2.1 Introduction

JavaScript is currently among the most popular programming languages [160]. 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 [67, 161, 173, 174, 184], 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* [101], 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 [57]. 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 [59].

Research has shown that third-party code accounts for a great portion of a JavaScript application codebase [173]. The 2022 State of Open Source Security [157] 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* [61], which enables the execution of Node packages in Deno. Third-party modules often depend on subprocesses [66, 82, 161]. There is a broad spectrum of programs that fall into this category, ranging from Linux utilities to handle files [130], to programs that process media (e.g., image/video conversions [129], metadata removal [128]), 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. Focusing on the records of the last 5 years, it has been possible to identify 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 malicious actions such as: privilege escalation on the host, open reverse shells, perform local file inclusion, and corrupt the filesystem.

Contributions of this research The following sections describe the development of a proposal to address the issues just mentioned, *Cage4Deno*: an extension of the Deno security functions aimed at mitigating vulnerabilities that may be introduced when subprocesses are used. The prototype implementing Cage4Deno is open-source and available at `https://github.com/unibg-seclab/cage4deno`. The design and implementation targets Linux systems, since it 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, great attention has been paid offer a tool that is usable 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, it is not expected from them to be fully aware of its functioning under the hood. To this end, the proposal of Cage4Deno also includes the design of 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. To support developers in the procedure of generating policy rules for Cage4Deno, the prototype includes the implementation of an auxiliary open-source tool named `dmng` [6]. This addition part of the toolchain can be used to generate policies that conform to the least-privileged principle for each subprocess employed. `dmng` 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. As a final remark of the design of Cage4Deno: the set of extensions preserves backward compatibility, meaning that sandboxing of existing code, including direct or transient dependencies, can be achieved without code changes.

The security benefits of Cage4Deno are demonstrated by showing how it is capable of mitigating a several real-world vulnerabilities, which have been exploited against popular services (e.g., GitLab [45] and TikTok [88]). Finally, an experimental evaluation has been performed in order to compare Cage4Deno to both scenarios with no sandboxing, and sandboxing through state-of-the-art proposals, showing substantial performance improvements with respect to the available alternatives.

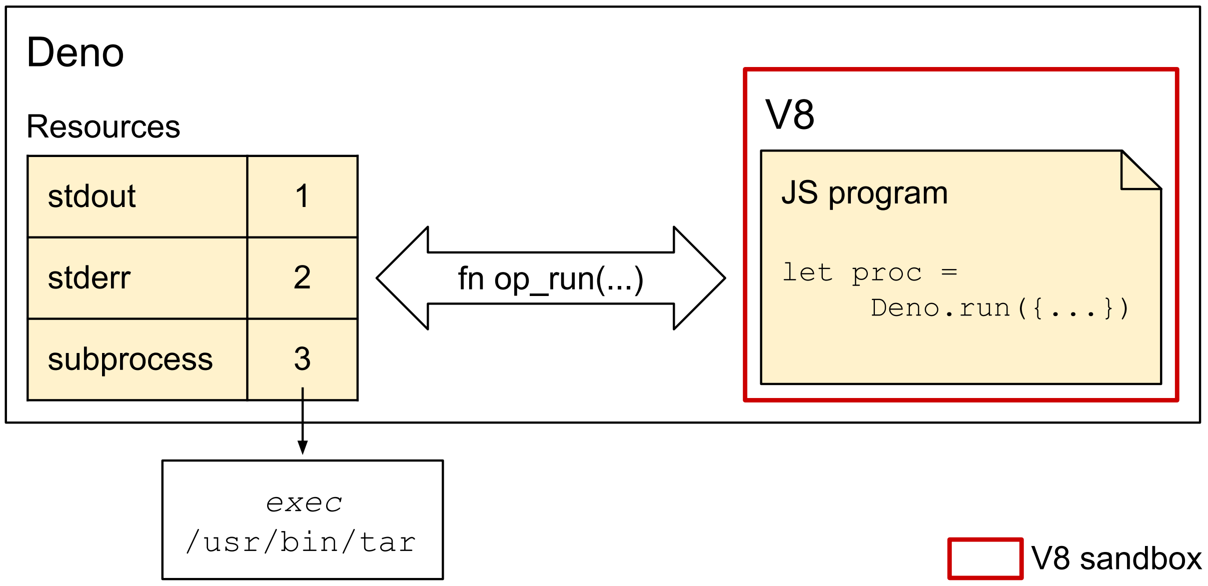


Figure 2.1: High level view of the Deno architecture

2.2 Background

This section overviews the frameworks used to implement Cage4Deno.

2.2.1 Deno

Deno [101] is a runtime for JavaScript and TypeScript embedding *V8* [169], 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 accessible through *ops* are stored by *deno_core*

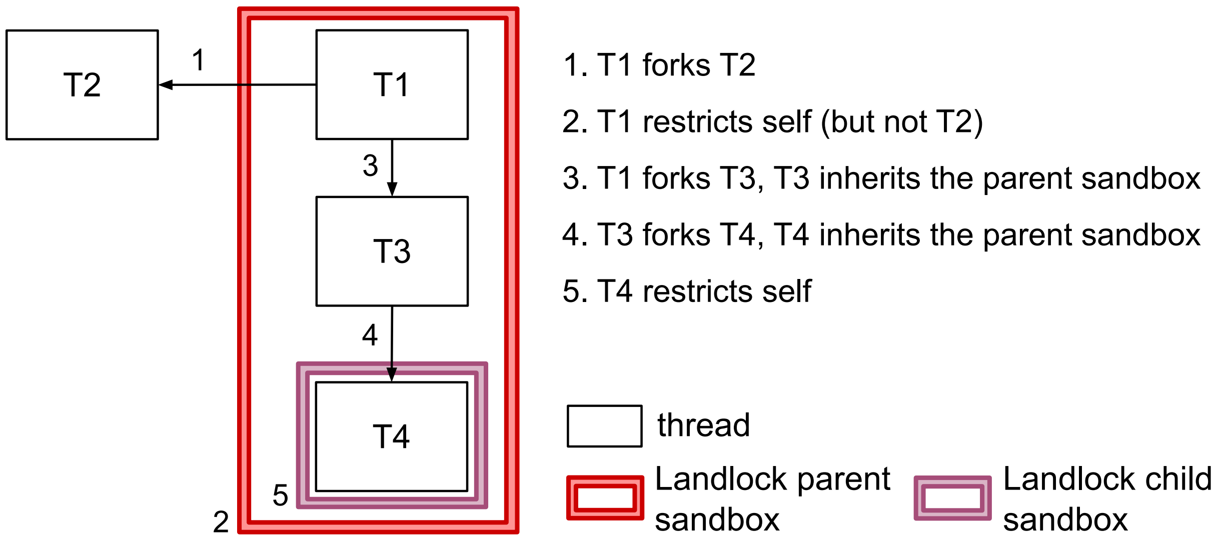

T2
1
T1
3
T3
4
T4
5
2
1. T1 forks T2
2. T1 restricts self (but not T2)
3. T1 forks T3, T3 inherits the parent sandbox
4. T3 forks T4, T4 inherits the parent sandbox
5. T4 restricts self
thread
Landlock parent sandbox
Landlock child sandbox

Figure 2.2: Creation and inheritance of a Landlock sandbox

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 2.1, 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 [57].

2.2.2 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 children created after sandbox activation, and the actions available can only be further restricted. The process is detailed in Figure 2.2.

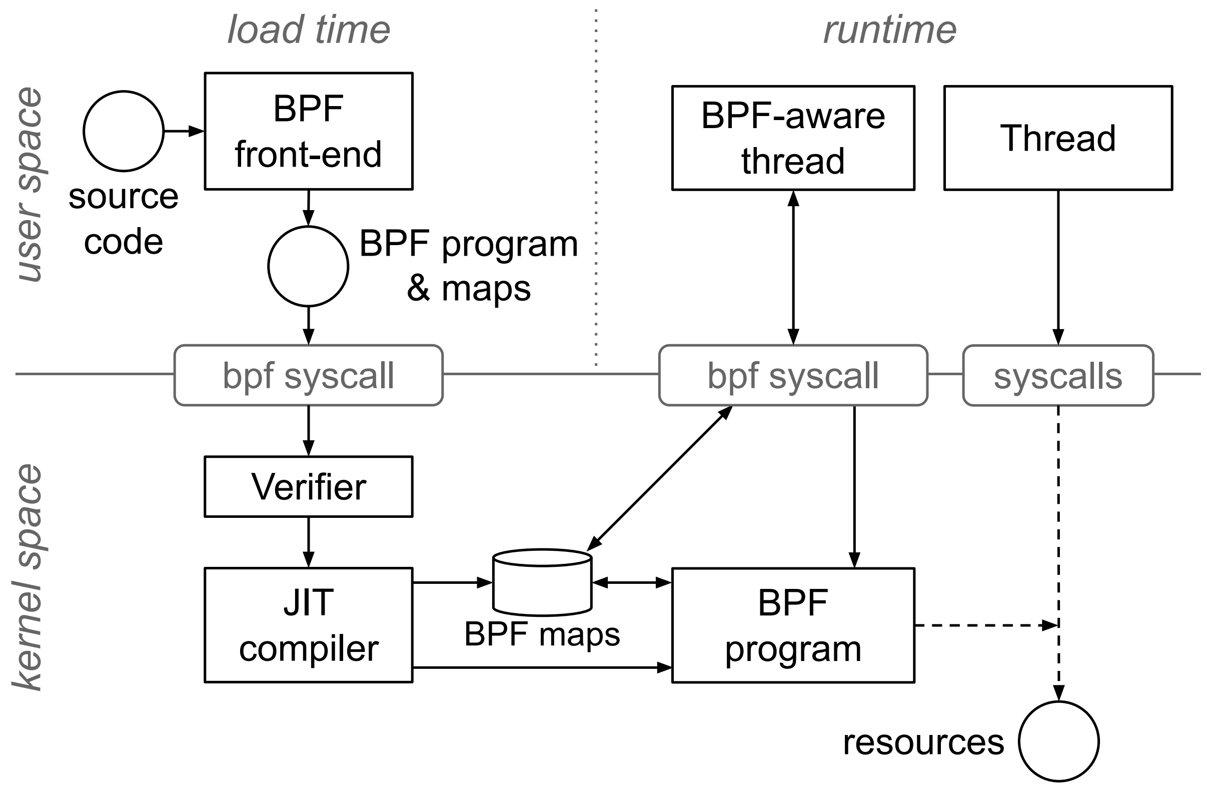


Figure 2.3: Overview of the BPF architecture

Although Landlock permits to sandbox a thread ensuring low performance overhead, there are a few limitations. The most relevant to the implementation of Cage4Deno 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.

2.2.3 eBPF

The *Extended Berkeley Packet Filter* (eBPF, henceforth referred to as BPF) [83, 96] 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. Predefined 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* [109, 125] as it elegantly addresses portability following a Compile Once – Run Everywhere approach [125]. 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) an d `CAP_NET_ADMIN` (to load networking-related programs) [4]. 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 2.3.

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

2.3 Cage4Deno

2.3.1 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 2.1, 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 2.2.1). 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. Cage4Deno performs this process thanks to the *sandboxer*, a new module added to Deno that leverages the Landlock LSM and the BPF framework (operating in stacking mode [187]) 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.[1] Compared to other well-known general-purpose sandboxing solutions like Minijail [80], and Google Sandbox2 [81], 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 2.4.

2.3.2 Threat model

Similarly to other research proposals [55, 162, 174], 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, developers of the utilities are not malicious actors, but, 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 decompression (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 [126]. 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 [54, 118, 191] 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 sub-

[1]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.

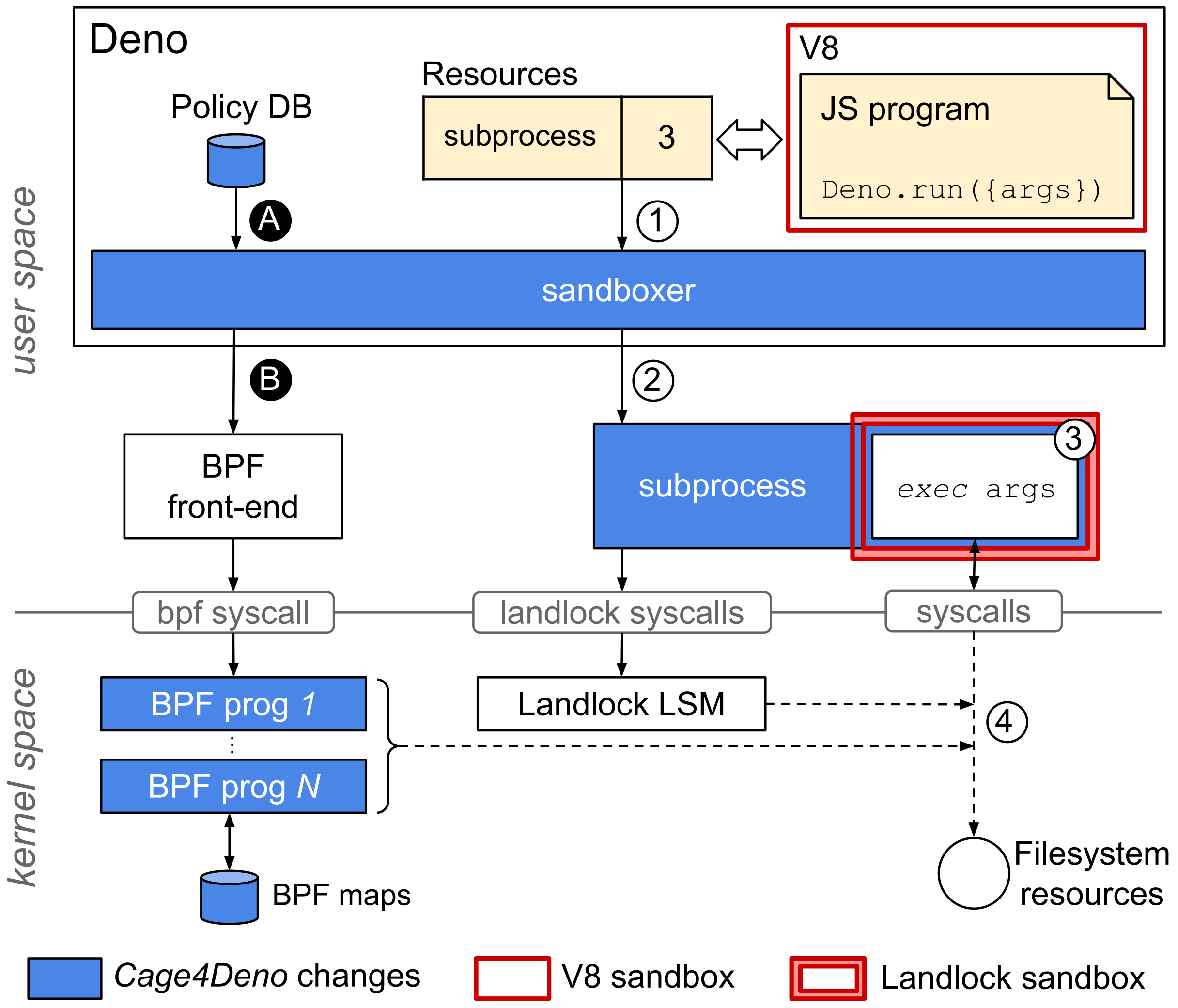


Figure 2.4: Overview of Cage4Deno implementation. Initial setup: *(A)* read the policy and *(B)* load BPF programs and write maps according to it. On subprocess request: *(1)* intercept subprocess creation, *(2)* create a Landlock-sandboxed subprocess with BPF deny, and *(3)* execute the utility. All the requests issued by the subprocess are restricted according to the policy *(4)*

process 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.

2.3.3 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 2.3.2 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.

2.4 Design and Implementation

This Section illustrates the design and implementation of Cage4Deno. The first component described is the interface used by the developer to input the policy. After that, it is illustrated what changes have been introduced to Deno in order to support `read`, `write`, `exec` and `deny` rules. 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 further detailed in Section 2.5, while additional BPF implementation details that focus on performance improvement are discussed in Appendix A.

2.4.1 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` instruction, 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 2.1. 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 2.3.1), 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 2.4, 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 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, the setup procedure is responsible for dropping these additional capabilities. This is done in order 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 the developed 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 2.1), and *(ii)* using the policy directly specified by the developer in the JS code using the newly introduced `policyId` option as

Listing 2.1: An example of child process in Deno

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

Listing 2.2: Restricting a child process using `policyId`

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

shown in Listing 2.2. In both cases, if no policy is found, Cage4Deno behavior falls 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**).

2.4.2 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 2.2.1), 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, to leverage Cage4Deno, developers are expected to detail the `read`, `write` and `exec` permissions inside the JSON policy file. Listing 2.3 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 functions `Deno.run()` and `Deno.Command.spawn()`, 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 [63] 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, Cage4Deno changes the procedure employed at process creation: a a closure (Rust equivalent of a C++ lambda expression) is scheduled to be run just before the `exec` function is invoked. The closure leverages the *sandboxer* module to create a new Landlock ruleset consisting of the `RWX` permissions found in the policy, and then invokes the `landlock_restrict_self()` syscall

Listing 2.3: `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"]
    }
  ]
}
```

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 2.2.2). 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 [28] and Control Groups [78], `RWX` rules also permit to grant access with file-level granularity on the whole filesystem, and not only on single volumes (Objective **O3**).

2.4.3 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 [148]. 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**), BPF has been identified 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 2.4.1).

This Section details the work that has been done to support deny rules. First, it is presented the problem of efficiently supporting access control decisions given a sequence of deny rules, then it is explained how to translate the approach into BPF programs, making clear how the *sandboxer* enforces access control at runtime.

```
       /
      / \
  home  media
    |
  user
   / \
data  lib
```

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

Deny listing approach

Similarly to what happens for `RWX` rules, developers leveraging Cage4Deno are expected to list the `deny` rules into the JSON policy file (see Listing 2.3). 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 2.1a. At runtime, access requests are simply evaluated traversing the trie. If a leaf node is visited, it is possible to 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,[2] 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 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 2.2.3). The most important, in the treated case, is related to the use of memory. The current implementation of BPF does not provide support for multi-level map-in-map [114]

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

structures required to implement the trie. To solve this issue, the trie is translated in BPF through a single-level hashmap Map_{policy} storing all the prefixes represented in the trie, as shown in Table 2.1b. 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 this aspect is further detailed, explaining how to adapt the maps 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 the proposal.

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* [109].
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 2.4.3).
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 2.2a 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

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 2.2: Hooks and tracepoints monitored by Cage4Deno

query, are executed when any of the hooks listed in Table 2.2b 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.

2.5 Policy generation

A key aspect of Cage4Deno proposal is usability by developers. The straightforward `RWX+D` permission model is functional to achieve ease of use (Objective **O2**), however, we a proposal of this kind also has to offer to 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 that has been done to support the process, explaining how our solution can be leveraged to generate the `RWX` rules shown in Listing 2.3.
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 [81] 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 Slim-Toolkit [141], where a containerized service is run against a test suite to automatically create Seccomp [32] and AppArmor [33] security profiles matching the least-privilege principle. In Cage4Deno, a similar approach has been adopted, which is able to retrieves the files a program should be able to read, write, or execute to work as intended. Moreover, developer have also the option of using 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 2.3.
The solution adopted was to develop `dmng`, a CLI tool written in Go ($\sim$3.5k lines of code), which ships within Cage4Deno, allowing the developer to automatically generate `RWX rules`,

Listing 2.4: Semi-automatic generation of `tar` policy (Listing 2.3)

```
# Set the active policy_name for a program
dmng -c tar --setcontext tarPolicy
# Add the dependencies of tar to the template
dmng -c tar -t -s
# Add the input to the template
dmng -c tar -a input.tgz -p r--
# Add the output to the template
dmng -c tar -a output -p -w-
# Add the denials
dmng -c tar -a output/misc -d
# Serialize the entries into a JSON file
dmng --serialize tar_policy_file
```

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 [11], the developer can use `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 `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 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 2.4). 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.1 reports the complete `tar` policy produced following the instructions in Listing 2.4.

It is worth noting that, while Cage4Deno employes dynamic analysis for the automatic generation of policies, this approach is widely known for producing results whose completeness depends on the test suite coverage [177]. To address this limitation, the current approach could be complemented by either source or binary code analysis. Relevant related works (e.g., [55]) use static analysis to pinpoint the system calls, however they do not keep track of parameter values. On the other hand, *AutoArmor* [107] 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, which is the Cage4Deno use case.

2.6 Experiments

In this Section is presented the experimental evaluation of Cage4Deno. The evaluation considers two aspects of the proposal: effectiveness in mitigating exploits (Section 2.6.1), and performance overhead (Section 2.6.2). The evaluation has been performed in 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. To be able to load every required BPF program, the pre-installed Linux Kernel v5.15 has been replaced with the same release of the kernel, but with BPF LSM available (Landlock LSM is enabled by default).

2.6.1 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 6 years, a sample of 15 vulnerabilities affecting common web application utilities has been selected. Among the targets there are several popular utilities such as ExifTool, FFmpeg, Git, GNU Tar, GraphicsMagick, Ghostscript, ImageMagick, OpenSSL, Pip, UnRAR, and Unzip. The CVEs considered are classified into three

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 2.3: Sample of CVEs mitigated by Cage4Deno. Despite being discovered in 2016, CVEs 1897, 1898, and 6321 have seen recent exploitation in 2018 [158] and 2021 [88]

categories: Remote Code Execution (RCE), Local File Read (LFR), and Arbitrary File Overwrite (AFO). Their details are reported in Table 2.3. These vulnerabilities have been selected as they represent examples of 0-click exploits, they can lead to severe security breaches, and they target utilities are extensively used by web applications. In general, similar considerations extend to a broader set of CVEs.

In this analysis, each one of the vulnerabilities has been reproduced by adapting public Proofs of Concepts. For each of them, is has been possible to create a single-click test case to showcase the sandboxing capabilities added to Deno. In particular, the experimental evaluation shows 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 that Cage4Deno introduces (Objective **O2**). The policy files used to restrict each utility have been automatically generated with the `dmng` tool described in Section 2.5. Table 2.4 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

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 2.4: Preliminary test showing the execution time of utilities reported in Table 2.3 over 500 runs (as $\mu \pm \sigma$)

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 CVE-2016-6321 is considered, in which a decompression of an archive permits the attacker to overwrite a target file, it is possible to notice how deny rules can be used to prevent filesystem corruption. As a final note, it is worth highlighting 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, some popular packages can be found: `astrodon` and `fast_forward` from `deno.land/x`, `fluent-ffmpeg` and `gm` from `npm` (executed in Node compatibility mode). Experiments 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.

2.6.2 Performance evaluation

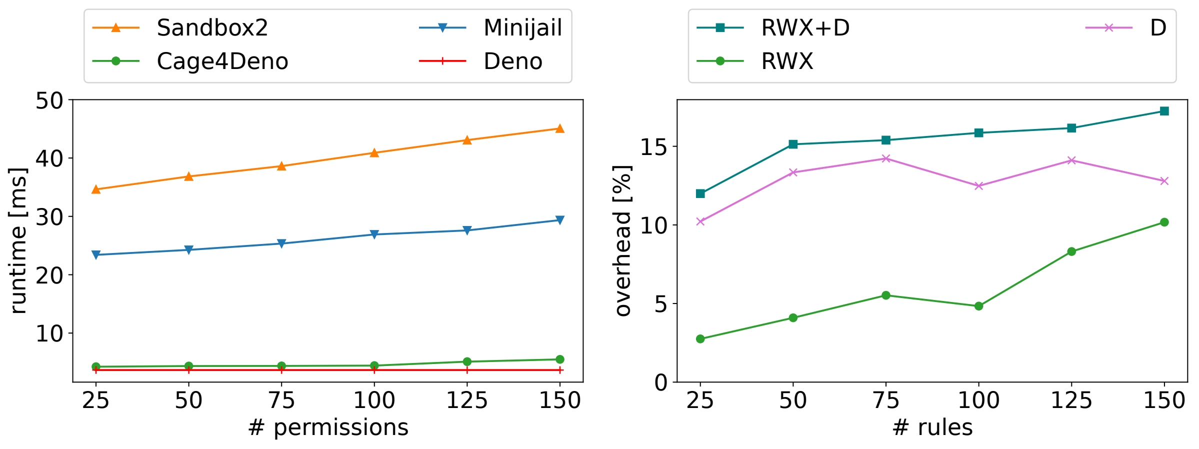
A requirement of Cage4Deno is that it must not introduce a large runtime overhead compared to the scenario in which the developer relies solely on the basic functions offered by Deno (Objective **O6**). This is fundamental, as any additional delay could negatively affect the end-user experience or increase the cost to host the application. To investigate this aspect, the performed tests compare 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 the proposal compared to general purpose sandboxing solutions proposed by the industry. In the final part of the section, it is also evaluated how the proposed solution scales with the number of rules in the policy.

Runtime overhead

A result that is expected analysis of Cage4Deno is that the runtime overhead has to vary according to the size of the policy and the number of filesystem requests performed by the sandboxed utility. The preliminary test conducted a involves the utilities affected by the CVEs detailed in Section 2.6.1. For each of them, it has been measured the execution time of vanilla Deno and Cage4Deno. The results of the test are reported in Table 2.4. The data clearly show that the largest overhead is associated with the larger policies and the shorter execution time. To further highlight this aspect, it is necessary to perform tests in the worst case scenario for Cage4Deno: a simple and quick running binary is used to perform a single access to the filesystem. As shown in row 1 of Table 2.4, 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 Cage4Deno scales with a large number of rules in the policy, a second test has been developed. Due to the aforementioned characteristics, `cat` is leveraged to conduct this second experiment. In this scenario, additional rules have been added to its minimum set of permissions. These rules are composed of a varying number of `RWX` rules, ranging from 25 to 150. To measure the execution time, the benchmarking module provided by the Deno standard library [58] has been employed, and to ensure the statistical value of the results, tests have been repeated 500 times. The first result illustrated, is the comparison between the execution time of vanilla Deno and Cage4Deno. While the proposed sandboxing approach 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 the overhead introduced, Cage4Deno has also been compared with two general-purpose sandboxing solutions: *(i) Minijail* [80], and *(ii) Sandbox2*, a key component of



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

Figure 2.5: Deterioration of overhead varying the policy size

the Sandboxed API [81] framework. As shown in Figure 2.5a, 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, the execution of each tool has been profilled through `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.[3] This validates Cage4Deno design choices, proving that our solution can provide significantly better performance with respect to industrial general-purpose sandboxing solutions. Moreover, Figure 2.5a 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

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

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, an additional test has been designed. In this scenario it is 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, *JMeter* [16] has been used, a tool written in Java designed to load test and measure the performance of web applications. Specifically, the employed configuration generates 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` [14], a Linux utility to configure and shape the network traffic. The test is executed with four different 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 2.5b confirm the trend shown in Figure 2.5a, 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.

2.7 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, the focus has been gived to security gaps of solutions that target runtimes and are therefore applicable to the server-side scenario. The proposal exposed in this chapter complements them, providing effective isolation of subprocesses using recent technologies. In the following, the related works are discussed and separated 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 [11,60,115,133,134,136,164,173,174,184].

Secure EcmaScript (SES) [115] 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* [136]. 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* [173]. 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 [174] 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* [11] 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 [184] 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 [79, 89, 98, 126, 137, 144, 145, 151, 170, 175]. *Galeed* [144], *PKRU-Safe* [98], and *NoJITsu* [137] 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++). NoJITsu 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* [126] 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 [27,40,55,80,81,97,127].
TRON [27] 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* [97]. 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 [94], 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 [55, 141, 177]. This effectively limits the capabilities of an attacker expoiting the program, and reduces the attack surface of the kernel [108]. 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 2.6 we have already compared Cage4Deno to other general-purpose sandboxing solutions such as *Minijail* [80], and *Sandbox2* [81]. 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 Cage4Deno`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* [127] and *Bubblewrap* [40]; 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 [7, 20, 21, 26, 74, 75, 92].

BPFBox [75] and *BPFContain* [74] 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 [26] 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. [92] 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* [20] and *Falco* [21]. 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.

2.8 Conclusions

Web development is a fast-paced environment characterized by tight time constraints. In this vast ecosystem, it has been specifically considered the role of Deno, a modern and secure runtime for JavaScript and TypeScript. Cage4Deno proposes an approach 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. Great attention has been paid to reduce the additional work done by developers willing to strengthen the security of their 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. The research that has been illustrated in this chapter is further extended by the work described in Chapter 4, where the design of Cage4Deno is extended in order to be made general enough to be applied to other JavaScript runtimes, since they share the same design as Deno regarding the unconstrained execution of subprocesses (e.g., Node.js and Bun). In addition to a more general design, Chapter 4 also describes how the protection guarantees provided for the file system in Cage4Deno can be extended to other resources.

Chapter 3. Hardening WASI using eBPF programs

3.1 Introduction

The research efforts started with Cage4Deno have shown promising results. Continuing the research on the same topic, this chapter explores a proposal to mitigate some of the security gaps that the research community is recently exploring in WebAssembly (Wasm) [86].

This tool 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 [13], WasmEdge [38], Wasmer [180], and WAMR [12].

To answer the developers' need to access resources of the host system from within the runtime, a standardization effort called WebAssembly System Interface (WASI) [183] 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 [181], 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 [124]. 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 [29, 93, 106] proved the approach to be error-prone, leaving the system unprotected when a vulnerability was introduced in a hostcall wrapper (Figure 3.1). 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. It is also worth noting that there is no practical advantage in having several implementations of the same access control checks for different runtimes. The core idea illustrated in this chapter is to replace the user-space runtime-specific security checks with a single in-kernel implementation that leverages eBPF [166]. The initial considerations that supported this research are the following properties of such approach: (i) it permits to decouple the implementation of hostcall wrappers and the access control details, minimizing the risk of bugs [96, 145], (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 [125] and also operating systems, thanks to Microsoft's undergoing effort to port eBPF to Windows [123].

3.2 Background

This section gives a background on WASI runtimes. eBPF has already been treated, in Section 2.2.3, which can be consulted for additional details.

3.2.1 WASI

Check for repetitions between here and intro The Web Assembly System Interface(WASI) [183] is a set of standardized APIs with the objective of providing a modular, portable and safe interaction with the underlying system to WASM modules. The specified OS-agnostic interfaces are called hostcalls, and provide a wide set of functionalities related to OS resources. One of the most notable interface is the host filesystem access interface utilized by a WASM module to interact with filesystem resources. Another proposal that is gain growing interest is the definition of an interface to interact with network sockets [23]. WASM runtimes which aims at supporting the execution of WASI compliant modules, must implement the code executed when hostcalls are invoked by WASM. This level of abstraction allows to maintain the decoupling between WebAssembly code and the OS in which it is executed: the module is only responsible of invoking a specific hostcall, while the WASI runtime is the component actually responsible of the interaction with the underlying system. Among these positive feature, WASI also introduces

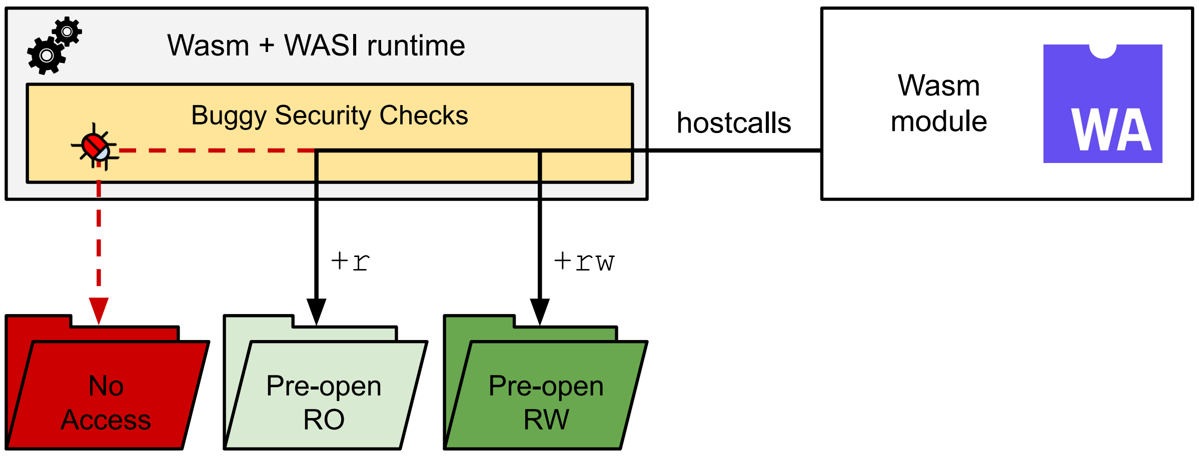


Figure 3.1: Current implementation of WASI by runtimes. A bug present in a hostcall wrapper permits the module to read the unauthorized directory on the left (red dotted arrow)

a critical aspect that must be taken into account: WebAssembly code, originally meant to be run in a well-defined sandbox, is now capable of interact with system resources. A common use case for WebAssembly is the execution of untrusted code, which can invoke hostcalls. To this end, WASI also defines a set of security guarantees that have to be respected by the implementations of the runtimes. In particular, the sandbox provided by the runtimes must be able to satisfy the following properties when interacting with the underlying OS: 1) file system isolation and 2) network isolation. In order to achieve `file system isolation`, runtimes must expose an interface to list the directories available to the WASM module along their *read* and *write* privileges. *Network isolation* is achieved through a similar method: runtimes allows users to specify an allow-list of network addresses that can be reached, along the protocols that can be used for the connection (e.g., UDP, TCP). Once the resources included in the sandbox have been defined, runtimes are responsible of implementing and enforcing security checks in order to avoid any access that violates the previously defined boundaries for both filesystem and network resources. Taking as an example the case of the filesystem: currently every WASI-compliant runtime implements the proposed file system interface with a libpreopen-like layer [124]. 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.

3.3 Motivation and threat model

3.3.1 Resource isolation

As illustrated in Section 3.2.1, currently WASI-compliant runtimes must implement the interaction with the host system and the necessary security checks. In detail, to apply protections to the filesystem, the majority of runtimes apply an approach similar to *libpreopen* [124]. In this case, users must provide the runtime with a list of directories, with the corresponding opening modes. Thanks to this list, the runtime is able to keep a list of all the paths that a WASM module can access. Each time an hostcall is performed, it is checked if the requested path is in the set of accessible resources. It is possible to note that this approach offers coarse granularity in terms of protection: it is necessary to give access to an entire directory even when a single file is necessary. In addition to this first issue, as noted in previous works [29, 93, 106], the sandbox of WASM runtimes may suffer of bugs that invalidates its security guarantees. For instance, in [93] is reported the effect of inconsistent security check enforcements when the hostcall `path_remove_directory` is invoked [39] through the Wasmer [180] runtime. If malicious code is able to utilize this interface, any file reachable by the user executing the WASM module may be removed, violating the security requirements that a WASI runtime should guarantee.

This is not the only bug that can be linked to this design decision. Errors may be caused by the fact that procedures normally handled at kernel level, now must be implemented in user space in order to perform the security checks correctly. One of these procedures is the *path resolution*, which implementation also caused known issues [120, 176] in Wasmer: in this case the introduced bug allows malicious modules to access the parent of a directory even if it is not part of the set of pre-opened paths. Finally, another drawback of this approach is that different implementations lead to different behaviors when the same WASM module is executed through different runtimes. For instance, since the procedure to implement the pre-open of multiple directories is not specified by WASI, the security checks performed by different runtimes are not the same, leading to different security boundaries [119, 155, 176]. The previously mentioned issues only affects a specific type of system resource: the filesystem. This chapter will focus on the effort to enhance the protection of this resource through eBPF. This is not the only system resource that can be exposed through WASI: runtimes such as WasmEdge and Wasmtime allows WASM modules to access network resources. While the WASI standard regulating this kind of resources is still being completed, in Chapter 4 is discussed how to extend the results of the proposal presented in this chapter also to network resources.

3.3.2 Threat Model

This work is based on assumptions that reflect the threat model employed by Wasm runtimes. It is assumed 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., [29]).

3.4 Architecture

The analysis presented in this chapter starts from the scenario illustrated in Figure 3.1. 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). The presented proposal follows a similar approach, since it requires 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 related to a directory, `read` translates to listing its content, `write` allows to create and delete files within it. The proposed approach has been tested thanks to the creation of extension for the following Wasm runtimes: Wasmtime and WasmEdge. The runtimes have been modified in order to be able to load the policy at startup, and, instead of pre-opening the directories available to the Wasm module, they enforce the policy through eBPF programs. Programs store information about access privileges in eBPF maps. Once these maps and eBPF programs are loaded, the runtime instantiates the Wasm module selected by the user (arrow Ⓐ in Figure 3.2). 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. A crucial property of this design is that the policy is activated before the runtime executes the module (i.e., before untrusted code is interpreted). This procedure guarantees that, from this point on, all the hostcalls performed by the Wasm module are restricted by the 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

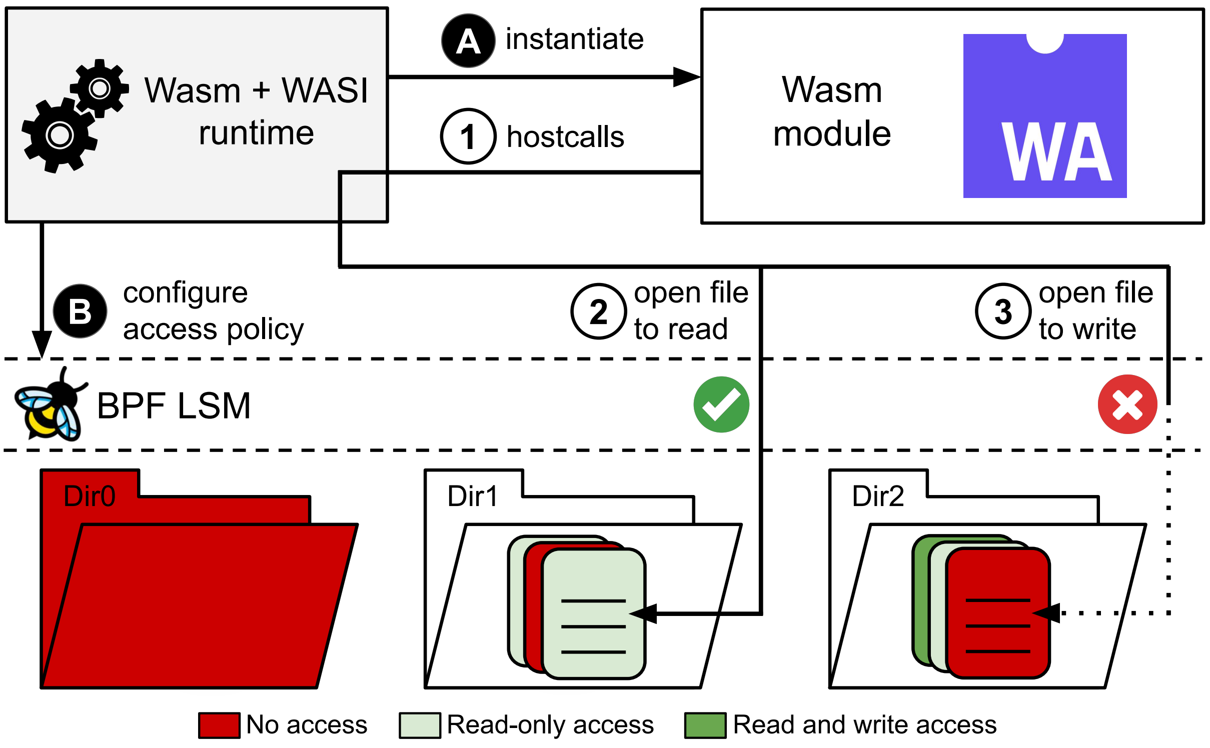


Figure 3.2: Proposal workflow. The runtime instantiates the Wasm module (Ⓐ), and configures the associated policy calling the traced user probe (Ⓑ). After the Wasm module is run, all the hostcalls issued by the program (①) are restricted by eBPF (②, ③)

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 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 [120], wrong directory removal [39]). Then, by leveraging kernel hook points [166], the presented 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.

3.5 Experiments

To investigate the overhead introduced by the proposed solution, the extensions for both WasmEdge and Wasmtime have been benchmarked. 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. No kernel upgrade or modification was required, as eBPF is already available in the default OS image. In order to assess the performance, the tests have been performed through one of the most popular binary that can be compiled to Wasm with support to WASI: `uutils coreutils`, the porting of the coreutils in Rust [43]. First, the binary has been compiled using with the `wasm32-wasi` target; runtime-specific optimizations have been applied (with `wasmedgec` [3] for WasmEdge and with `wasmtime compile` [2] for Wasmtime) to obtain optimal performances. Then, the benchmarks reported in the coreutils repository have been reproduced, 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 [72]). Finally, the proposed protection has been applied to the tested runtime. The Hyperfine benchmarking tool [47] has been used to perform the measurements, with 1000 performed runs after 100 warmups executions. As shown from the results in Table 3.1, the illustrated 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`). It also possible to observe that there are notable differences between the WasmEdge and Wasmtime test execution time for some utilities (e.g., `ls` and `cut`); from the performed analysis it is possible to conclude that these differences are mostly caused by the specific post-compilation optimizations.

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

3.6 Related Work

There are several successful solutions that leverage Wasm to sandbox untrusted code [77, 126, 138]. RLBox [126] is a framework that facilitates the isolation of third-party libraries in pre-existing software. eWASM [138] optimizes the execution of Wasm in embedded systems with constrained resources. Sledge [77] enables efficient Wasm-based serverless execution on the edge. The use of the approach presented in this chapter for restricting access to the file system within these frameworks can strengthen their security guarantees.
The memory safety guarantees of Wasm depend on the runtime implementation [106]. Hence, Bosamiya et al. [29] explore the problem of producing provably safe sandboxes. WaVe [93] 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 the presented eBPF-based solution can be directly integrated into existing runtimes employed in the industry.

3.7 Conclusions

The results achieved by the research presented in this chapter 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 the applied approach can extended be extended also to network sockets, as exposed in Chapter 4. Since this component of WASI is currently under definition, having solutions to be able to protect network resources is a promising research direction.

Chapter 4. Extend the protection of Deno native code

4.1 Introduction

As established in Chapter 2, 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 [142], and the solution identified by the community was to configure the resources available to an application with simple permission flags [62]. This change also influenced the design of Node.js, which introduced a similar flag-based control model[1] two years later [135]. 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 [62].
This chapter focuses on an argument treated also by previous research [73, 162]: JavaScript modules frequently depend on components written in native languages such as C or C++. One advantage of such practice is that 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. There are several notable examples, such as: the node-sqlite3 [167] and deno-sqlite3 [102] database drivers; modules to perform image/video conversions, such as sharp [132], fluent-ffmpeg [128] and gm [129]; OCR engines like Tesseract [165]; and the cryptography modules relying on bcrypt [131]. The 2022 State of Open Source Security [157] 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, the research presented in this chapter has 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 Similarly to what has been exposed in Chapter 2 and 3, this research focus on the security gap in the way modern JS runtimes execute native code, as neither Node.js, nor Deno, nor Bun sandbox it. In this work it is proposed NatiSand, a framework to provide

[1]Node.js support for creating security policies is still experimental as of 2023.
[2]The list is reported in Table 4.3.

strong isolation guarantees against the execution of native code. Differently from Cage4Deno, NatiSand allows the developer to control access not only to the filesystem, but also to Inter-Process Communication, and network, effectively reducing the risks coming from the execution of untrusted native code. These additional system resources protected are not the only source of novelty introduced by NatiSand. While Cage4Deno focused only on providing security guarantees for subprocesses, NatiSand is also albe to sandbox the execution of shared libraries. The solution described in this chapter is characterized by a compact, generic architecture that fits nicely with modern runtimes. Internally, it leverages *Seccomp* [32] and Linux Security Modules (LSMs), such as *Landlock* [122] and *eBPF* [41] to restrict access to protected resources. Again, in the design phase of NatiSand, specific attention has been paid to usability by developers. To this end, the automatic policy generation approach described for Cage4Deno has been extended in order to include the additional protected resources.
A prototype of NatiSand has been implemented and integrated into Deno and is available at `https://github.com/unibg-seclab/natisand`. Concerning the performance of NatiSand, the runtime overhead have been measured for both subprocesses and shared libraries; in both scenario, popular utilities have been used to perform the benchmark. Due to the addition of shared libraries as a protection target, the baseline for the benchmarks are not only the ones used in Chapter 2, but Wasm modules, which are the most popular alternative to the usage of native libraries. The benchmarks performed to assess the performance of show that, compared to the alternatives, NatiSand exhibits substantial performance improvements.

4.2 Background

A detailed background on JavaScript runtimes, Landlock and eBPF has already been discussed in Section 2.2. This gives an overview on the additional component used to create the NatiSand sandbox: Seccomp BPF.

Seccomp

Seccomp BPF [32], henceforth Seccomp, is a mechanism provided by the Linux kernel to restrict the set of 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) [91]. 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 based on their arguments. 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 [70].

4.3 Security motivation

JavaScript runtimes let developers specify the set of access privileges given to a web application [62, 135]. 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 [62].

There is a broad variety of applications that rely on the use of native libraries. 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 JavaScript 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 [100]. Previous work [162] 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.

The following paragraph describe in detail the three scenario addressed in this chapter.

Filesystem compromise Guaranteeing the integrity and confidentiality of the application host filesystem is crucial to mitigate risks of data corruption and exfiltration [27]. 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 file system 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 [30, 153]. Given the potential of this attack vector, it is important to limit the scope and set of IPC channels available to a 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 [31]. In addition, attackers may open network channels to remotely recover data obtained on the vulnerable host [152]. Restrictions on how a 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.

4.3.1 Threat model

The addressed threat model 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 JavaScript code is out of the scope of our proposal, since JavaScript engines and the permissions system enforced by JavaScript runtimes can be considered able to securely render JS code. NatiSand aims to constrain the execution of potentially malicious, or vulnerable, binary utilities and functions used by JavaScript applications. This native code accesses system resources unconstrained by the security mechanisms offered by the JavaScript runtime, and its actions may cause severe security breaches. Moreover, the input processed by the web application is often untrusted and can be not sanitized, 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 JavaScript application to the vulnerable components. The goal of the proposal is to prevent the execution of malicious payloads and 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.

4.4 Design and implementation

In this section it is presented NatiSand, the proposal developed to enable the isolation of native code for JS runtimes.

4.4.1 Objectives

From the experience gained from Cage4Deno, driving the design through a set of core objectives has proven to be a successful approach. NatiSand follows some of the objectives already exposed in 2.3.3, while expanding on others. In the following paragraphs, only the newly addressed goals are exposed.

Security applied to multiple resources 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 native code in 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.

Compatibility with multiple runtimes 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.

4.4.2 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 BPF. Finally, BPF 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 [156], 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 4.1. 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 are initialized and loaded into the kernel. A pool of isolated contexts is also prepared by the sandbox. Only then the web application is started. At runtime, NatiSand intercepts all the calls to native code performed by the application and executes them safely in the proper isolated context.

4.4.3 Integration with JS runtimes

Similarly to the naming convention applied in Cage4Deno, the component responsible of enforcing NatiSand security constraints is called *sandboxer*, a component that extends the enhanced JavaScript runtime. Another task perfomed by the *sandboxer* is to parse security policies in order to enforce the isolation of native code accordingly. In the following we explain the operations performed by the runtime to use it, referring to Figure 4.1.

Bootstrap This phase is similar to the one described in Section 2.4.1: the JSON file containing security boundaries is parsed by the JavaScript runtime and BPF components(i.e., programs and maps) are loaded into the Kernel. After these steps, the Linux capabilities needed to perform these operations are dropped. In addition to these steps shared with Cage4Deno, the bootstrap procedure of NatiSand performes additional operations, mainly addressed to the creation of security contexts. Each context is an OS thread with permissions restricted through the following mechanisms: (i) Landlock rulesets are enforced through the dedicated syscalls, (ii) Seccomp filters are installed in order to limit the set of syscalls available to the thread, and (iii) uprobes are invoked in order to enforce the security checks performed by BPF programs on the security context. This architecture has been chosen to avoid paying the performance cost to instantiate each thread at the exect moment of the invocation of executables and shared libraries: the sandboxer allocates and configures beforehand all the security contexts necessary and parks them in a context pool (ⓐ, ⓑ, ⓒ). This design choice minimizes latency, one of the highlighted goal expressed in Section 2.3.3.

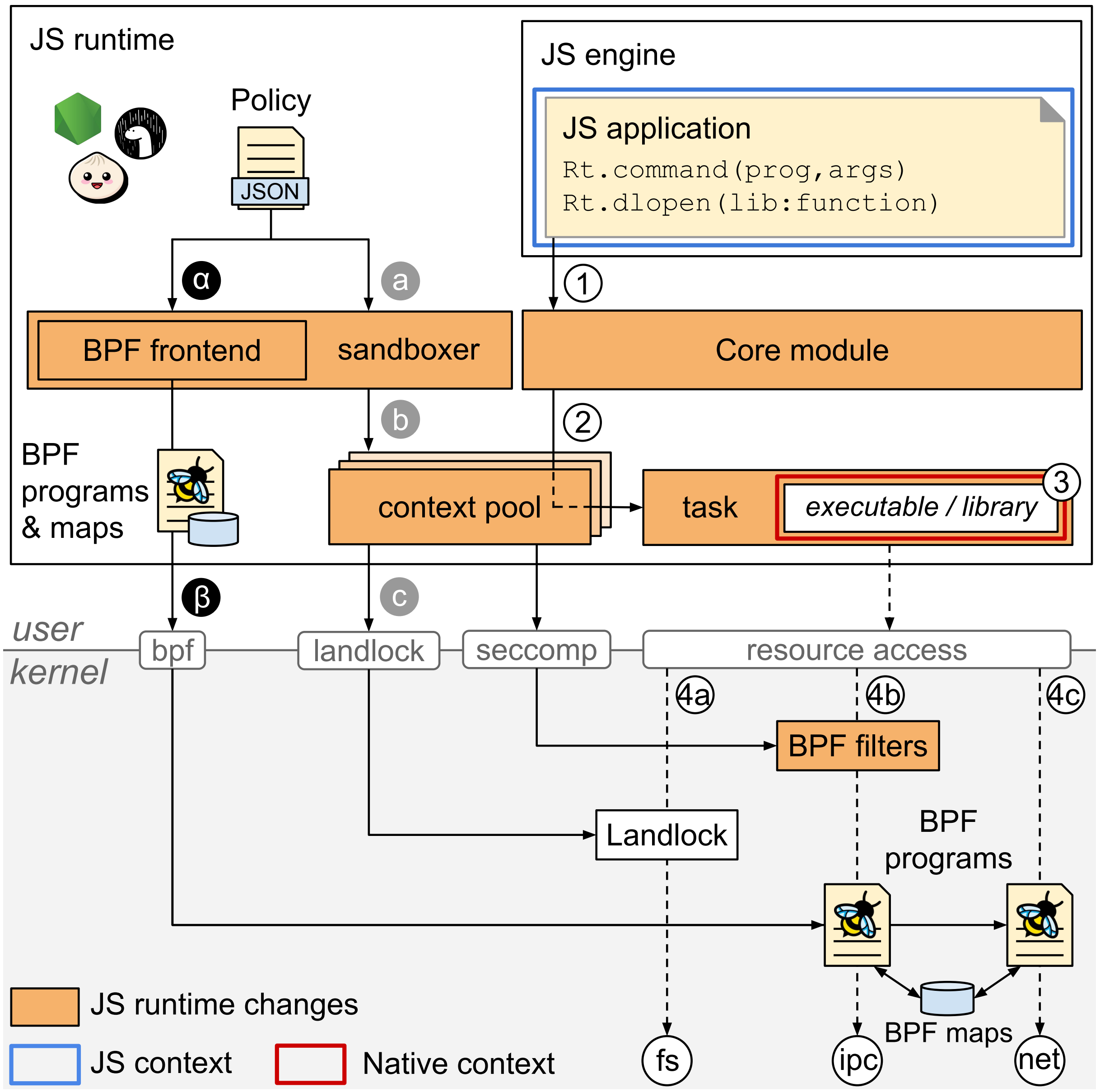


Figure 4.1: Integration of NatiSand in the JavaScript runtimes. Bootstrap: import security contexts (α, β), creation of the context pool (a, b, c). Application runtime: isolated execution of binary programs and shared library functions (1, 2, 3, $4a$, $4b$, $4c$)

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

Table 4.1: Hooks and tracepoints monitored by NatiSand

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 isolated context ②. Based on the type of request, a dedicated task inheriting the selected security context is launched, finally it is used to execute native code ③. The consequence is that any request to access filesystem, IPC, and network resources will be subject to the restrictions imposed by the LSMs ((4a), (4b), (4c)). 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.

4.4.4 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 and filesystem protection The same concerns explained in Section 2.4.3 also applies for NatiSand: Landlock and Seccomp guarantee policy inheritance after a `clone` syscall is performed, but BPF does not have such guarantee out of the box, since BPF maps responsible of tracking the restricted contexts must be updated explicitly. To this end, NatiSand relies on the same mechanisms based on the security hooks mentioned in Table 2.2(a).

Concerning the filesystem, NatiSand restricts access to the filesystem using Landlock. The properties of this tools have already been described in detail in Section 2.4.2: the sandbox enforces a `read`, `write`, `exec` (RWX) permission model, specified with three allow-list vectors (e.g., lines 5, 6, and 7 in Listing 4.1). After the security context has been activated to limit access privileges of a security context, the available permissions can only be further restricted.

IPC To explain the isolation features NatiSand provides, it is first necessary to describe 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 [112]. 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 [113]. 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 [70]). 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 BPF programs attached to the hooks reported in Table 4.1 (which are not affected by TOCTOU issues). In particular, in the case of UNIX sockets the programs are attached to the `lsm/socket_bind` and `lsm/socket_connect` hooks, while with fifo files, the kernel function `fifo_open` is used. A summary of the IPC mechanisms controlled by NatiSand, along with the LSM leveraged to perform the security checks, is reported in Table 4.2.

IPC	**Subclass**	**Linux system call**	**Seccomp**	**BPF**
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 4.2: 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

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 BPF 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 4.1. The approach described in this section can be efficiently applied to the protection mechanism described in Chapter 3 in order to protect also network resources.
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. 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 BPF program that checks the opening of a connection performs a sequence of lookups in a BPF map storing the reachable (i.e., policy allowed) hosts. Each network resource is uniquely identified by its IP address and port. Internally, the value zero for the port is used to allow opening a connection on every port.
Up to now we have discussed the restrictions applied to a context 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 `bind`. We decided to allow a security context to bind a socket only when the policy grants access to localhost. The rationale is that the developer should specify in the

policy which ports are available to a security context. No restrictions are instead 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 [111]. The same applies to `accept`, which is used to extract the first connection request from the queue of pending connections [110].

4.5 Policy

In this section it is presented the structure of the policy file. After that, it is proposed an approach to automatically generate the permission rules.

4.5.1 Policy structure

JavaScript applications executed through 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 4.1 (the complete example is reported in Appendix B.2), while a summary of the most relevant policy features is described next.

name and 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 binary program, library, and function names, 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 name, or when different functions from the same library are invoked. Listing 4.1 shows a policy that is enforced every time the application runs the curl program.

fs The structute of the policy in this part is similar to what has been described in Section 2.4.2, but contained in the *fs* element. It stores three optional arrays: *read*, *write* and *exec*. Filesystem paths are used as array values. As an example, the context detailed in Listing 4.1 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

Listing 4.1: Example of JSON policy file with single context

```
[{
   "name": "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]
   }]
}]
```

`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, the developer is provided with a simple interface consisting in flags that can be turned on and off based on security requirements. Six optional flags are available in the policy: *fifo*, *message*, *semaphore*, *shmem*, *signal*, and *socket*. For example, in Listing 4.1, 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 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 the developer to restrict the permissions associated with her application. Similarly to the filesystem case, the developer can use a coarser granularity by setting the *ipc* element to `true`, enabling all communication mechanisms.

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 4.1, the process executing curl is only allowed to connect to `https://www.example.com` on port 443.

4.5.2 Policy generation

Similarly to what described in Section 2.5, the approach proposed to automatically generate policies is similar to the one applied in SlimToolkit [141], where a service is run against a test suite to automatically generate the least privilege security profile. For filesystem rules, the same approach already applied for Cage4Deno can be applied. Concerning the protection of IPC mechanisms, the `dmng` utility has been extended to support the generation of the required set of policy flags by analyzing the results of multiple test cases where the Seccomp filter and BPF programs of the sandboxer are set to *auditing mode* (i.e., instead of blocking an access request, the event is reported in system logs). Finally, network rules generation can be automated by observing the execution of the binary with BPF programs. In fact, with this approach is possible to retrieve 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, a `uprobe` has been attached to the `getaddrinfo` function of the libc library, for IPs, network socket connections are observable using `kprobes` on `socket_bind` and `socket_connect` LSM hooks.
It is important to recall the limits of the proposed approach to automatically generate policies: (i) test suite with limited coverage might provide overly strict policies not allowing the execution of legitimate code, (ii) policy generation for malicious code produces overly permissive policy and obviously cannot be trusted.

4.6 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. 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, in order to create a function proof of concept, NatiSand has been integrated into only one of them to demonstrate the achievement of the set objectives (Section 4.4.1). In this section it is explained the rationale of the decision that has been taken. After that, the main architectural changes applied to the chosen runtime are detailed.

4.6.1 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 and (iv) greater familiarity with the code base, the Deno runtime has been selected. The main motivation for not selecting Bun is the fact that it is still in the early stages of its lifecycle (version v1.0 has just been released at the time of writing of this thesis), so it is the least used by developers and the one that may have major changes in the future. 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 JavaScript applications to access system resources, and although it recently introduced a module-based permission model, the feature is experimental [135]. Deno was instead designed with the protection of the host as one of its main goals [142], thus no access to privileged system resources is given to JavaScript applications unless the developer explicitly grants it. Deno provides the *Node Compatibility Mode* [61], 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 all three dimensions. 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, making Deno the ideal candidate to implement the prototype of NatiSand.

4.6.2 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. This stage has been modified in order to also read and parse the policy file specified with the new `native-sandbox` flag. After that, the permissions associated with each security context are added to the *global state* stored by the runtime. To complete the bootstrap phase, the steps to load the necessary BPF programs and to initialize the pool of isolated contexts have also been integrated, as mentioned in Section 4.4.3.

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. The requests that are required to execute native code (e.g., `command`, `dlopen`, `run`) have been identified and modified, in this way *deno_core* is able to apply the security restrictions enforced by NatiSand. The *op_code* along with the arguments are used to select the proper security context.

Listing 4.2: 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
```

4.6.3 Support to fast JS calls

In October 2020 V8 announced the support to fast JS calls [168]. The function allows V8 to directly invoke optimized native functions without leveraging the bindings that connect V8 and the embedder (e.g., a JavaScript runtime). This permits to obtain substantial performance gains, since native function calls can be resolved in nanoseconds.

Deno has introduced unstable support to fast JavaScript calls in July 2022 [99]. 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 JavaScript runtime, invalidating the security reference monitor of Deno. In the developed prototype, this Deno security issue has been addressed by 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 fast calls but allow to select the JavaScript 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, a new API named `Deno.nativeCall()` has been introduced in the runtime. The API receives, as first argument, the name of the function to be sandboxed, along with the list of its arguments. Listing 4.2 shows how to sandbox the functions from the native database driver *sqlite3*.

4.7 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, it is first shown how the solution is able to protect web applications relying on binary programs and shared libraries affected by high severity vulnerabilities (Section 4.7.1), then we investigate the performance of our approach (Section 4.7.2). Both tests use a server with Ubuntu 22.04 LTS, an AMD Ryzen 3900X CPU, 64 GB RAM, and 2 TB SSD.

4.7.1 Exploit mitigation

To conduct the analysis, it has built a representative sample of vulnerabilities targeting executables and libraries widely used in web applications consinsting in the 32 CVEs reported in Table 4.3. 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). It is important to note 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`.
Similarly to the analysis perfomed in Section 2.6, the public Proofs of Concept of the CVEs in Table 4.3 have been retrieved and successfully used to exploit the vulnerable version of the utilities. Then, it has been analyzed whether the vulnerabilities were exploitable sending the malicious payload through the JavaScript module interface, and confirmed the feasibility of the attack. The *Node compatibility mode* was leveraged to execute in Deno the modules downloaded from npm. Finally, the experiment is repeated while the security functions provided by NatiSand are in place, in order to verify 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 introduced in the experiment was the specification of a security policy through the `native-sandbox` CLI argument. The policy was generated using the approach described in Section 4.5.2. 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

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 4.3: Sample of CVEs mitigated by NatiSand

confidential files, open reverse shells, interact with privileged services through IPC, and transfer data to unauthorized network hosts. It is important to note that, while sandboxing limits the privileges an attacker can gain from exploiting a vulnerable program, it cannot eliminate vulnerabilities, nor it can make infeasible to use them in an exploit chain.

4.7.2 Performance evaluation

To assess the performance of NatiSand, a broad set of programs has been considered, including several GNU Core Utilities, executables to process multimedia, database drivers, and Object

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**

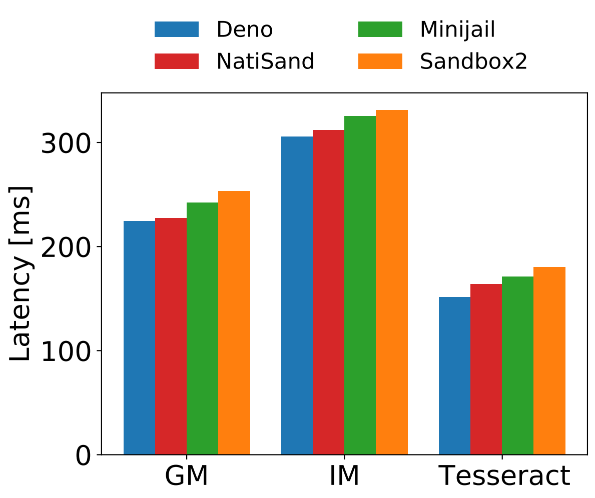
Table 4.4: Average execution time for common Linux utilities

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 constrained ambient rights. A handful of general purpose sandboxers can be adopted to achieve a comparable degree of protection. In the evaluation, *Minijail* [80] and *Sandbox2* [81] have been used as alternatives. The features of these two tools can be consulted in detail in Section 2.6.2

Benchmark I In the first benchmark, a JavaScript application has been implemented 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. The results are shown in Table 4.4 (tests are ordered by increasing execution time). As expected, the cost of activating the sandbox is amortized



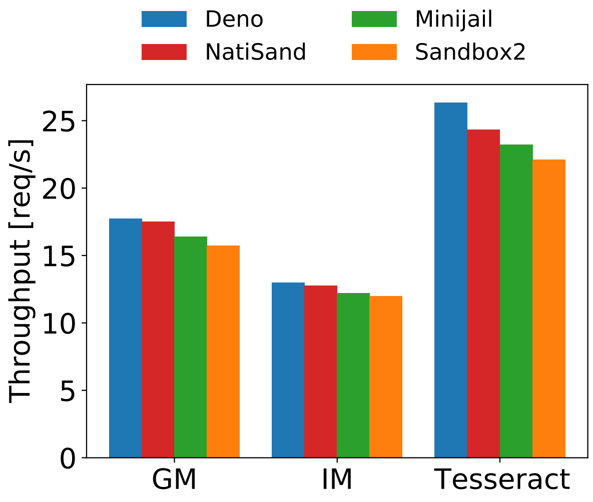


Figure 4.2: Average latency and throughput for microservices that execute subprocesses

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 is designed to show the overhead experienced by a remote client. To this end, three microservices have been used, each one representing a real use case scenario of high performance native programs. Two microservices rely on 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 4.2 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,

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

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, it has been measured the overhead associated with libraries. Contrary to the default JavaScript runtime behavior, NatiSand transparently executes native library functions in dedicated contexts with limited ambient rights. As explored in Chapter 3, the most popular alternative approach to isolate libraries is to compile them to WebAssembly (Wasm). As explored in related works, 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 [126].

Benchmark III Similarly to Benchmark I, a JavaScript application has been implemented 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 4.5. 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 interpreted language remains slower than its native counterpart. Remarkable are the cases of opus and sqlite3, which used `nativeCall` and demonstrate its efficiency.

Extend the protection of Deno native code

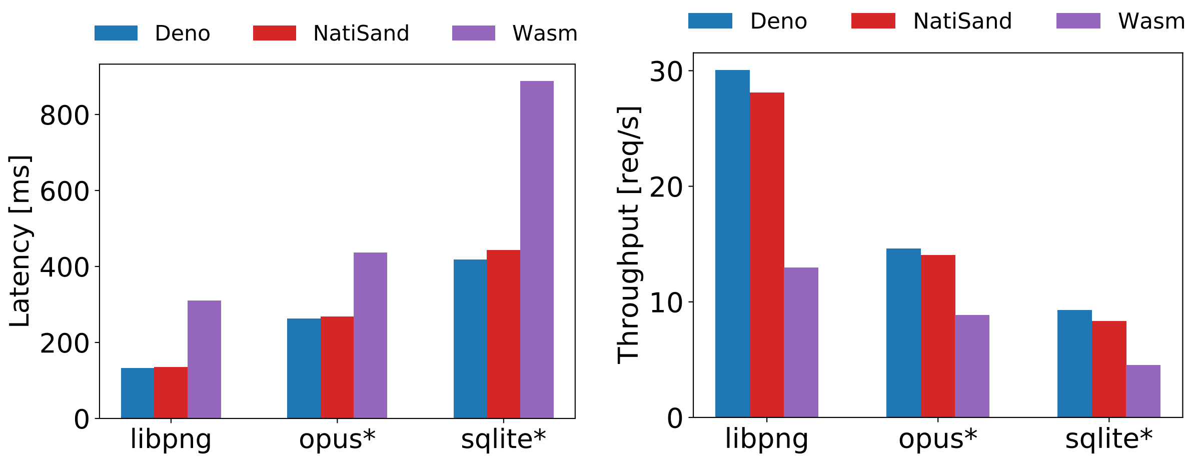


Figure 4.3: Average latency and throughput for microservices that execute native functions (* marks the use of `nativeCall`)

Benchmark IV To understand the slowdown perceived by a remote client, the functionalities of the libpng, opus, and sqlite3 libraries have been exposed through microservices. For each of them, a client has been configured 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 4.3. 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. The main reason for such performance overhead is due to the Wasm interpreter and the additional memory management required to exchange data between the JavaScript application and Wasm code.

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 the evaluation described in this chapter, it has been used a precompiled version of sqlite3, but it has been also necessary to manually compile opus and libpng using the Emscripten toolchain [71] and the WASI Sdk [182], respectively. Moreover, current implementations of the WebAssembly System Interface (WASI) can only restrict ambient rights programmatically, and, as noted in Chapter 3, filesystem privileges work at directory granularity. Lastly, Wasm requires the developer to explicitly allocate, write, and read bytes from the Wasm module linear memory.

4.8 Related Work

The majority of previous research efforts that are related to NatiSand, have been already exposed in Section 2.7. In addition to the ones already mentioned, the BinWrap framewrok [36], proposed by Christou et al., shares a lot of the consideration that lead to the creationg of NatiSand. In particular, the usage of threads to enforce security guarantees over native code in Node.js, testifies the generality of the NatiSand design. In BinWrap, the authors propose an approach that is able to separate the runtime execution of and untrusted component from the rest of the application. Similarly to NatiSand, the separation is achieved using different execution threads. While focusing on memory protection, BinWrap also leverages Seccomp in order to limit the set of available syscalls to threads that are responsible for the execution of code that may be affected by vulnerabilities.

4.9 Conclusions

The increase in scale and complexity of modern web applications has led to the introduction of new security mechanisms in JS runtimes. Unfortunately, the execution of native code 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 binary programs and shared libraries. The proposal is not dependent on a particular JavaScript runtime, and was designed to be integrated into different architectures. Considerable attention was dedicated to usability; little effort is required by the developer to sandbox her applications. Indeed, no specific security expertise is necessary to benefit from the protection, nor are changes to the application source code.

The research effort describe in this paper enhance both the protections of what have been described in Chapters 2 and 3, showing how novel LSMs can be used to protect a wide variety of programs and system resources.

While these results are already a considerable step in the right direction in terms of protection against attackers, it has to be taken into account that resources that are part of the boundaries of the applied sandbox can still be reached after a succefull exploit of a vulnerable utility. For instance, database drivers such as *sqlite3* can reach sensitive data even when protections that applies the least-privilege principle are in place. In order to protect against such security risk, it is also necessary to handle this kind of data with particular attention. The next chapters of this thesis details methods that can be applied to address the concerns just exposed.

Chapter 5. Data Anonymization for Large Datasets

5.1 Introduction

Previous chapters have explored approaches that can be used to enforce a well defined security boundary to protect a device that hosts vulnerable software. While valuable, these techniques do not always prevent the and exploit from being successful, thus they only limit the resources exposed to the attacker only to the one that reachable by the compromised application component. If the exploited component has access to data collections(e.g., a database driver), this breach may rise privacy concerns for the subjects that are represented by data points in the leaked data collection. In particular, in order to obtain privacy guarantees in datasets containing possible identifying and sensitive information, it is required not only to refrain from publishing explicit identities, but also to obfuscate data that can disclose such identities as well as their association with sensitive information. k-Anonymity [37, 149, 150, 163], extended with ℓ-diversity [116], is a technique that can offer such protection. k-Anonymity requires generalizing values of the *quasi-identifier* attributes (i.e., attributes that can expose to linkage with external sources and leak information on respondents' identities) to ensure each quasi-identifier combination of values to appear at least k times. ℓ-Diversity considers each sensitive attribute in grouping tuples for quasi-identifier generalization so to ensure each group of tuples (whose quasi-identifiers will then be generalized to the same values) be associated with at least ℓ different values of the sensitive attribute.

While simple to express, k-anonymity and ℓ-diversity are far from simple to enforce, given the need to balance privacy (in terms of the desired k and ℓ) and utility (in terms of information loss due to generalization). Also, the computation of an optimal solution requires evaluating (based on the dataset content) which quasi-identifying attributes generalize and how, and hence demands complete visibility of the whole dataset. Hence, existing solutions implicitly assume to operate in a centralized environment. Such an assumption clearly does not fit the web application scenario addressed in the previous chapters, where large scale systems operate on a huge amount of collected data (e.g., smart cars are reported to upload to the cloud 25GB per hour). The core idea exposed in this chapter is that the scalable distributed architectures commonly employed in a web application scenario, can help in performing computation on such large datasets, but their usage in computing an optimal k-anonymous solution requires careful design. In fact, a simple distribution of the anonymization load among workers would affect either the quality of the solution or the scalability of the computation, for instance it may require expensive synchronization and data exchange among workers [17]).

In the research illustrated in this chapter, it is addressed the problem of efficiently anonymizing large data collections. The proposed solution extends Mondrian [104], an efficient and effective approach originally proposed for achieving k-anonymity in a centralized scenario, to enforce both k-anonymity and ℓ-diversity in a distributed scenario. With the proposed method, anonymization is executed in parallel by multiple workers, each operating on a portion of the original dataset to be anonymized. The design of our partitioning approach aims at limiting the need for workers to exchange data, by splitting the dataset to be anonymized into as many partitions as the number of available workers. Then, each worker runs a version of Mondrian that has been updated in order to operate in a distributed environment; each node of the cluster is able to independently operate on their portion of the data. A distinctive feature of the proposal is that the partitioning approach does not require knowledge of the entire dataset to be anonymized. Rather, it can be executed on a sample of the dataset whose size can be dynamically adjusted. The approach is therefore applicable in scenarios where the dataset is very large, maybe even distributed, and does not entirely fit in main memory. In addition to that, the implemented approach is able to perform parallel execution on a dynamically chosen number of workers. The experimental evaluation confirms both the goodness of our partitioning strategy with respect to maintaining utility of the anonymized dataset and the scalability of our approach. The main contributions of this research are thus summarized as follows. First, it proposes and evaluates different partitioning strategies for distributing data to workers. Second, it extends the original Mondrian algorithm to operate in a distributed scenario without asking workers to interact, and to enforce both k-anonymity and ℓ-diversity. Third, it supports different strategies for managing generalization, including the use of generalization hierarchies that permit to produce semantically-aware anonymization. Fourth, the approach have been evaluated through different metrics for assessing the information loss caused by the distribution of the anonymization process.

5.2 Basic concepts

The proposed solution is based on three main components: k-anonymity, ℓ-diversity, and Mondrian.

k-Anonymity. k-Anonymity [149] is a privacy property aimed at protecting respondents identities in data publication. k-Anonymity starts from the observation that a dataset, even if de-identified (i.e., with explicit identifying information removed) can contain other attributes, called *quasi-identifiers* (abbreviated QI) such as gender, date of birth, and living area, that can be exploited for linking the dataset with other data sources and enable observers to reduce uncertainty on the identity (or identities) to whom the tuples in the de-identified dataset refer. k-Anonymity demands that no tuple in a released dataset can be related to less than a certain number k of respondents. k-Anonymity operates on the values of the QI attributes to ensure that no tuple can be uniquely associated with the identity of its respondent through its QI values, and vice versa. In practice, k-anonymity is enforced by ensuring (through generalization of data values) that each combination of values of the quasi-identifier in a dataset appears with at least k occurrences. In this way, any linking attack exploiting the quasi-identifier will always find at least k individuals to which each anonymized tuple can correspond and vice versa. k-Anonymity can be guaranteed in different ways. The original proposal of k-anonymity applies generalization to the QI attributes [149]. Generalization is a data protection technique that replaces attribute values with other, more general values. For instance, an individual's `Age` may be generalized in age ranges (e.g., replacing all age values from 25 to 30 with a single interval $[25, 30]$). While numeric attributes (i.e., attributes defined on a totally ordered domain) naturally generalize to ranges of values, the generalization of categorical attributes (i.e., attributes defined on a non-ordered domain) can leverage generalization hierarchies (e.g., Figure 5.1 illustrates a generalization hierarchy for attribute `Country`). Since generalization (while maintaining data truthfulness) removes details from data, it reduces the risk of finding unique correspondences for QI values with external data sources. For example, the dataset in Figure 5.2(c) is a 3-anonymous version of the dataset in Figure 5.2(a), considering attributes `Age` and `Country` as quasi-identifer. In the figure, quasi-identifying attributes `Age` and `Country` have been generalized so that their values appear with at least 3 occurrences (for readability, the i^{th} tuple in Figure 5.2(a) corresponds to the i^{th} tuple in Figure 5.2(c)). For example, the `Age` and `Country` of the respondents of the first three tuples have been generalized to range $[25, 30]$ and to value *Europe*, respectively. Note that neither the values for `Age` nor the values for `Country` of the last three tuples have been generalized to obtain 3-anonymity, since they already share the same values for the quasi-identifier (38 and *USA*, respectively). It is easy to see that no record in external data sources can be linked, through `Age` and `Country`, to less than three tuples in the 3-anonymous dataset.

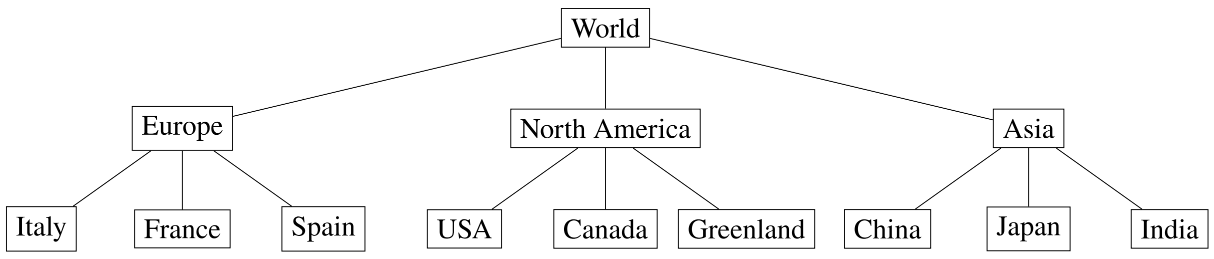
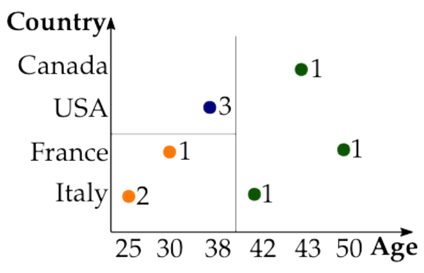


Figure 5.1: Generalization hierarchy for attribute `Country`

ℓ-Diversity. ℓ-Diversity [116] extends k-anonymity to prevent attribute disclosure, that is, to protect against possible inferences aimed at associating a value for the sensitive attribute to the respondent's identity. With reference to the datasets in Figure 5.2, suppose that the `TopSpeed` of the respondent aged 30 from *France* (third tuple) were 132. The 3-anonymous dataset in

Age	Country	TopSpeed
25	Italy	132
25	Italy	132
30	France	128
42	Italy	110
50	France	115
43	Canada	115
38	USA	126
38	USA	127
38	USA	140

(a)



(b)

Age	Country	TopSpeed
[25,30]	Europe	132
[25,30]	Europe	132
[25,30]	Europe	128
[42,50]	World	110
[42,50]	World	115
[42,50]	World	115
38	USA	126
38	USA	127
38	USA	140

(c)

Figure 5.2: An example of a dataset (a), its spatial representation and partitioning (b), and a 3-anonymous and 2-diverse version (c), considering quasi-identifier QI={`Age`,`Country`} and sensitive attribute `TopSpeed`

Figure 5.2(c) would have, for the first three tuples, the same sensitive value (132). While, thanks to the protection offered by 3-anonymity, no record in an external data source (e.g., a voter list) can be uniquely mapped to any of these tuples, this 3-anonymous dataset would still leak the fact that European respondents with `Age` between 25 and 30 have `TopSpeed` equal to 132. ℓ-Diversity extends k-anonymity by demanding that each equivalence class E (i.e., each set of tuples sharing the same generalized values for the quasi-identifier) have at least ℓ well-represented values for the sensitive attribute(s). Several definitions of *well-represented* have been proposed, and a natural interpretation requires at least ℓ different values for the sensitive attribute(s). For example, the 3-anonymous dataset in Figure 5.2(c) is also 2-diverse, since each equivalence class contains at least two different values for `TopSpeed`.

Mondrian. Mondrian [104] is a multi-dimensional algorithm that provides an efficient and effective approach for achieving k-anonymity. Mondrian leverages a spatial representation of the data, mapping each quasi-identifier attribute to a dimension and each combination of values of the quasi-identifier attributes to a point in such a space. Mondrian operates a recursive process to partition the space in regions containing a certain number of points (which corresponds to splitting the dataset represented by the points in the space in fragments that contain a certain number of records). In particular, at each iteration Mondrian cuts the set of tuples in each fragment F computed at the previous iteration (the whole dataset at the first step) based on the values (e.g., for numerical attributes, whether lower/higher than the median) for a quasi-identifying attribute chosen for each cut. The algorithm terminates when any further cut would generate only sub-fragments with less than k tuples, at which point values of the quasi-identifying attributes in each fragment are substituted with their generalization. Figure 5.2(b) shows the spatial representation and partitioning of the dataset in Figure 5.2(a), where the number associated with each data point is the number of tuples with such values for quasi-identifier `Age` and `Country` in the dataset. The 3-anonymous version of the dataset in Figure 5.2(c) has been obtained by first partitioning the dataset in Figure 5.2(a) based on attribute `Age`: fragment

$F_{\texttt{Age}\leq 38}$ includes all the tuples with Age less than or equal to the median value 38, and $F_{\texttt{Age}>38}$ the remaining tuples. $F_{\texttt{Age}\leq 38}$ is further partitioned based on attribute Country, obtaining $F_{\texttt{Age}\leq 38,\ \texttt{Country IN}\ \{\mathit{Canada,USA}\}}$ including all tuples with Country equal to *Canada* or *USA*, and $F_{\texttt{Age}\leq 38,\ \texttt{Country IN}\ \{\mathit{France,Italy}\}}$ including the remaining tuples. No further partitioning is possible (all fragments include exactly three tuples), and the quasi-identifying attributes in each fragment can be generalized. The dataset in Figure 5.2(c) has been obtained generalizing the dataset in Figure 5.2(a) according to the partitioning in Figure 5.2(b) and leveraging the generalization hierarchy in Figure 5.1 for attribute Country.

5.3 Distributed anonymization

The considered scenario consists of a large and possibly distributed dataset $\mathcal{D}$ that needs to be anonymized. $\mathcal{D}$ may not entirely fit into the main memory of a single machine. The objective of the research presented in this chapter is then to distribute the anonymization of $\mathcal{D}$ to a set $W = \{w_1, \ldots, w_n\}$ of workers so that they can operate in parallel and independently from one another, to have benefits in terms of performance while not compromising on the quality of the solution (with respect to a traditional centralized anonymization of $\mathcal{D}$). To this purpose,Mondrian has been extended to operate in such a way that workers in W are assigned (non-overlapping) partitions of $\mathcal{D}$ (i.e., sets of tuples of $\mathcal{D}$, defined as *fragments* in the rest of the chapter) and can operate limiting the need for data exchanges with other workers. Each worker $w \in W$ can independently anonymize its fragment satisfying k-anonymity and ℓ-diversity, with the guarantee that the combination of the anonymized fragments is a k-anonymous and ℓ-diverse version of $\mathcal{D}$.

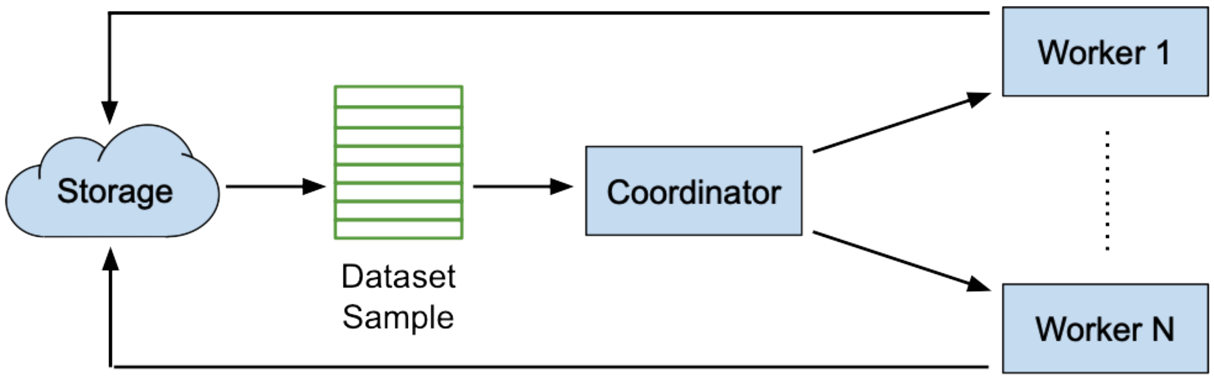


Figure 5.3: Overall view of the distributed anonymization process

The overall process is overseen by a Coordinator, and includes a *pre-processing phase* (which partitions the dataset $\mathcal{D}$ in fragments and assigns fragments to workers) and a *wrap-up phase* (which collects the anonymized fragments from the workers, recombines them, and evaluates the quality of the computed solution). The reference scenario of the approach (as graphically represented in Figure 5.3) is then characterized by a *(distributed) storage platform, storing and managing the dataset $\mathcal{D}$ to be anonymized (as well as its anonymized version* $\hat{\mathcal{D}}$, after workers have anonymized their fragments), the anonymizing *workers*, and the Coordinator. In the remainder of this chapter, given a dataset $\mathcal{D}$ with quasi-identifier QI=$\{a_1, \ldots, a_q\}$ and privacy parameters k and ℓ, the following notation is adopted:

- $\hat{\mathcal{D}}$ refers to the k-anonymous and ℓ-diverse version of $\mathcal{D}$
- $\hat{t}\in\hat{\mathcal{D}}$ represents the generalized version of t in the anonymized dataset, $\forall t\in\mathcal{D}$
- $\hat{F}$ denotes the anonymized version of fragment F (i.e., $\forall t \in F$, $\exists \hat{t} \in \hat{F}$ s.t. $\hat{t}$ is the generalized version of t)

The pre-processing phase is crucial to achieve distributed anonymization. The first problem to be addressed is the definition, by the Coordinator, of a fragmentation strategy, regulating which tuples belong to which fragment. An effective strategy, as demonstrated by the experimental results (Section 5.8), is to fragment $\mathcal{D}$ based on the values of (some of the) quasi-identifying attributes $a_1, \ldots, a_h$, in such a way that a tuple t of $\mathcal{D}$ is assigned to a fragment F based on the values of $t[a_1], \ldots, t[a_h]$. To illustrate, consider the dataset in Figure 5.2(a) and suppose to define two fragments F_1 and F_2 based on the values of Age. The Coordinator may define a strategy such that F_1 contains all tuples of $\mathcal{D}$ with values lower than or equal to 38 (i.e., the first three and last three tuples of the table), and F_2 the remaining tuples of $\mathcal{D}$. In principle, this would require the Coordinator to have complete visibility over $\mathcal{D}$ for defining fragments. However, $\mathcal{D}$ might be too large to fit into the main memory of the Coordinator. The proposed approach also presents a strategy in which the Coordinator can define the conditions that regulate the fragmentation of $\mathcal{D}$ based on a *sample* of $\mathcal{D}$, whose size can be dynamically adjusted according to the storage capabilities of the Coordinator. The Coordinator then communicates the conditions to the workers, which will then download the tuples in $\mathcal{D}$ that satisfy such conditions directly from the storage platform (i.e., without the need for the Coordinator to send any dataset to the workers). With reference to the example above, where fragments are defined based on the values of Age, the Coordinator communicates to workers the value ranges (e.g., lower than or equal to 38, and greater than 38) for the tuples in their fragments. The anonymization phase following the pre-processing operates in parallel at the workers. For the design of this phase, special attention has been given on the support, for categorical attributes, of generalization hierarchies to the aim of producing semantically-aware generalized (anonymous) data. The pre-processing phase is discussed in Section 5.4, while the anonymization and wrap-up phases are described in Sections 5.5 and 5.6, respectively.

5.4 Data pre-processing

The pre-processing phase of the proposed approach operates on a sample D of $\mathcal{D}$, whose size is tuned depending on the storage capabilities of the Coordinator. For the sake of readability, the discussion presented in this chapter refers to a generic dataset D with the note that D is a sample of the original dataset $\mathcal{D}$ to be anonymized (clearly the quasi-identifier attributes considered for D are those defined for $\mathcal{D}$).

5.4.1 Partitioning strategies

Selecting a strategy for partitioning the dataset D is crucial, since a random partitioning may cause considerable information loss. Indeed, if fragments include tuples with heterogeneous values for the quasi-identifier, each worker (which independently operates on its fragment) would need a considerable amount of generalization to satisfy k-anonymity. On the contrary, information loss is mitigated if partitioning does not spread across fragments tuples that assume similar values for the quasi-identifier. To illustrate, consider a dataset with four tuples $t_1, \ldots, t_4$ having values 25, 25, 60, and 60 for Age, which needs to be partitioned in two fragments. If partitioning generates two fragments $F_1 = \{t_1, t_2\}$ and $F_2 = \{t_3, t_4\}$, no generalization is needed to enforce 2-anonymity. On the contrary, fragments $F_1 = \{t_1, t_3\}$ and $F_2 = \{t_2, t_4\}$ requires generalizing Age to the range [25,60] in each fragment, causing higher information loss.
To limit the information loss implied by the partitioning of a dataset D with quasi-identifier QI among a set $W = \{w_1, \ldots, w_n\}$ of workers, the proposed approach adopts one of the two following strategies:

- The first one takes the name of *quantile-based approach*, it selects an attribute a from the QI, and partitions D in n fragments $F_1, \ldots, F_n$ according to the n-quantiles of a in D.

- The second one is named *multi-dimensional approach*, it recursively partitions D in a similar way as the Mondrian approach (see Section 5.2). Given a fragment F (the entire dataset D at the first iteration), the multi-dimensional strategy selects an attribute $a \in$QI and partitions F in two fragments according to the median value of a in F. Each of the resulting fragments is then further partitioned, until n fragments have been obtained.

Both the *quantile-based* and the *multi-dimensional* partitioning approaches rely on an ordering among the values that the attribute a selected for partitioning assumes in D to compute quantiles (for the quantile-based approach) and median values (for the multi-dimensional approach). In fact, given a sample D of the dataset and the attribute a selected for partitioning, the tuples in D are first ordered according to their value of a, establishing a ranking among the attribute values. Quantiles and the median values are then computed on such a ranking. When a is numerical,

```
Q_PARTITION($D, W$)
1: let $a$ be the attribute used to partition $D$
2: $R := \{rank(t[a]) \mid t \in D\}$ /* rank of $a$'s values in the ordering */
3: let $q_i$ be the $i^{th}$ $|W|$-quantile for $R, \forall i = 1, \ldots, |W|$
4: $F_1 := \{t \in D \mid rank(t[a]) \leq q_1\}$
5: for each $i = 2, \ldots, |W|$ do
6:     $F_i := \{t \in D \mid q_{i-1} < rank(t[a]) \leq q_i\}$
```

Figure 5.4: Quantile-based partitioning

ordering among values is naturally defined. When a is categorical and has a generalization hierarchy $\mathcal{H}(a)$, attribute values are considered with the order in which they appear in the leaves of $\mathcal{H}(a)$, aiming at keeping in the same fragment values that generalize to a more specific value. Indeed, leaf values that are close in the hierarchy will have a common ancestor (to which they would be generalized) at a lower level in the hierarchy (see Section 5.5), thus limiting information loss. For instance, with reference to the hierarchy in Figure 5.1, it is possible to use the order ⟨*Italy*, *France*, *Spain*, *USA*, *Canada*, *Greenland*, *China*, *Japan*, *India*⟩. This ordering would combine in the same fragment values *Italy* and *France*, which generalize to a more specific value (i.e., *Europe*) than a fragment with values *Italy* and *Canada* (i.e., *World*).

Figure 5.4 illustrates the procedure implementing quantile-based partitioning executed by the Coordinator. Given a dataset D and a set W of workers, the procedure selects the attribute a for partitioning, orders the tuples in D according to $t[a]$, and determines the rank $rank(t[a])$ of each value $t[a]$ (lines 1–2). It then computes the $|W|$-quantiles of such ranking *R* (line 3). The first fragment F_1 is obtained by including all the tuples $t \in D$ with rank of $t[a]$ lower than or equal to the first computed quantile (line 4). The remaining fragments $F_2, \ldots, F_{|W|}$ are obtained by including in F_i all the tuples $t \in D$ with ranks of $t[a]$ in the interval $(q_{i-1}, q_i]$, with q_i the i^{th} $|W|$-quantile of the computed ranks (lines 5–6). To illustrate, consider partitioning in 4 fragments the dataset in Figure 5.2(a) with the quantile-based approach over attribute Age (for simplicity, the dataset in Figure 5.2(a) is considered as a sample). For the first tuple t_1, it is possible to notice that $rank(t_1[\texttt{Age}])=rank(25)=1$, since 25 is the first value (i.e., the smallest) in the ordering for Age. Similarly, for the third tuple t_3, $rank(t_3[\texttt{Age}])=rank(30)=2$, since 30 is the second value in the ordering for Age. The 4-quantiles $q_1, \ldots, q_4$ for such ranks are $q_1=2$, $q_2=3$, $q_3=4$, and $q_4=6$. The first fragment F_1 then includes all the tuples t such that $rank(t[\texttt{Age}]) \leq 2$, that is, all the tuples such that $t[\texttt{Age}] \leq 30$. Fragment F_2 includes all tuples t such that $2 < rank(t[\texttt{Age}]) \leq 3$. Fragments F_3 and F_4 are computed in a similar way.

Figure 5.5 illustrates the recursive procedure implementing multi-dimensional partitioning executed by the Coordinator. The procedure takes as input a dataset D, the set W of workers, and the recursive level of iteration i (1 at the first invocation). The procedure first selects the attribute a for partitioning, orders the tuples in D according to $t[a]$, and determines the rank

```
**M_PARTITION**$(D, W, i)$
1: **let** $a$ be the attribute used to partition $D$
2: $R := \{rank(t[a]) \mid t \in D\}$ /* rank of $a$'s values in the ordering */
3: **let** $m$ be the median of $R$
4: $F_1 := \{t \in D \mid rank(t[a]) \leq m\}$
5: $F_2 := \{t \in D \mid rank(t[a]) > m\}$
6: **if** $i < \lceil \log_2 |W| \rceil$ **then**
7:     **M_Partition**$(F_1, W, i+1)$
8:     **M_Partition**$(F_2, W, i+1)$
```

Figure 5.5: Multi-dimensional partitioning

$rank(t[a])$ of each value $t[a]$ (lines 1–2). It then computes the median value m for *R* (line 3), and defines two fragments F_1, including the tuples t of D having rank for $t[a]$ lower than or equal to m, and F_2, including the remaining tuples (lines 4–5). The procedure then recursively calls itself on the two computed fragments (lines 7–8) with $i+1$ to further fragment them, unless the necessary number of iterations have already been executed (i.e., $i = \lceil \log_2 |W| \rceil$, since at each iteration the number of fragments doubles and $\lceil \log_2 |W| \rceil$ fragments are sufficient to assign at least a fragment to each worker) (line 6). To illustrate, consider partitioning in 4 fragments the (sample) dataset in Figure 5.2(a) with the multi-dimensional approach, and suppose that at the first iteration (i=1) the attribute chosen for partitioning is `Age`. The median of the ranks for the values of `Age` is 3, and the sample is split in two fragments F_{12} and F_{34} such that F_{12} includes all tuples t for which $rank(t[\texttt{Age}]) \leq 3$, and F_{34} the remaining ones. At the second (and last) iteration (i=2), the procedure selects an attribute (which could possibly be different from `Age`) for fragment F_{12} and an attribute for F_{34}, and partitions each fragment in two more fragments based on the median of the ranks of such attribute values in F_{12} (F_{34}, respectively). Since 4 fragments have been obtained, the partitioning process terminates.

The quantile-based and the multi-dimensional partitioning exhibit different behavior in the definition of the fragments. The quantile-based partitioning ensures balancing among the fragments, since all n fragments will include (approximately) the same number of tuples, but its application can be limited by the domain of the attribute a chosen for partitioning (it cannot be used if the number n of workers is larger than the domain of a). On the contrary, while being always applicable, the multi-dimensional approach may result in some workers being assigned twice the workload of other workers. Since multi-dimensional partitioning doubles the number of fragments at each iteration, when the number n of workers is a power of 2, the recursive process can be executed $\log_2 n$ times, obtaining n fragments of (approximately) the same size. However, n may not be a power of 2. Aiming at using all the workers, the partitioning strategy stops when $n \leq 2^i$ (multi-dimensional partitioning generates 2^i fragments at the i-th iteration) and $2^i - n$ workers are assigned two fragments, resulting in some workers having twice the workload of the others. For instance, assume $W = \{w_1, \ldots, w_7\}$ and $|D| = 1000$. Multi-

dimensional partitioning needs 3 iterations for generating at least 7 fragments ($2^2 < 7 \leq 2^3$). Since $2^i - n = 8 - 7 = 1$, one worker (e.g., w_1) will be assigned two fragments, resulting in a workload of nearly 250 tuples for w_1 and of 125 tuples for each of the other workers.
The computational complexity of the two approaches is slightly different, with quantile-based partitioning resulting more efficient than multi-dimensional partitioning (as confirmed by the experimental results in Section 8). Procedure **Q_Partition** costs $O(|D|)$, while procedure **M_Partition** costs $O(|D| \log |W|)$. The first two steps (lines 1-2) are the same for the two procedures and cost $O(|D|)$, and the computation of quantiles has the same cost as the computation of the median and cost $O(|D|)$. The **for each** loop at line 5 of procedure **Q_Partition** has cost $O(|W|)$, and therefore the overall cost of quantile-based partitioning is $O(|D|)$, since the number of workers is smaller than the number of tuples in the dataset. The recursive calls of procedure **M_Partition** imply a cost of $O(|D| \log |W|)$ since the procedure recursively calls itself with i from 1 to $\lceil \log_2 |W| \rceil$ and, for each value of i, the overall size of the fragments input to the different recursive calls is $|D|$.

5.4.2 Fragments retrieval

While the Coordinator operates on a sample D of the dataset $\mathcal{D}$ to be anonymized for defining fragments, workers need to operate on the whole fragment assigned to them. To minimize communication overhead, the proposed approach defines fragments assigned to workers according to the *partitioning conditions* identified by the Coordinator to partition D. These conditions, comparing the attribute a selected for partitioning with the values in its domain corresponding to the quantiles or median value of the ranking of tuples in D according to a, are communicated to workers (see Section 5.7). Each worker can then retrieve the tuples in its fragment directly, without need for the Coordinator to retrieve and communicate such tuples. For instance, consider two fragments F_1 and F_2 computed over a sample D of $\mathcal{D}$ with the multi-dimensional approach, and assume to adopt attribute Age for partitioning and that the median of the ranking corresponds to value 38 in the attribute domain. The worker in charge of the anonymization of F_1 will retrieve all the tuples in $\mathcal{D}$ having Age≤38, while the worker in charge of the anonymization of F_2 will retrieve all the tuples in $\mathcal{D}$ having Age>38.
Given a set W=$\{w_1, \ldots, w_n\}$ of workers, c_i denotes the condition describing the fragment F_i assigned to $w_i, i = 1, \ldots, n$. When using the quantile-based approach, condition $c_i, i = 1, \ldots, n$, describes the values for a that are included in the i-th n-quantile of D (i.e., its endpoints). For example, with reference to the example in Section 5.4.1 for the quantile-based partitioning of the dataset in Figure 5.2(a) in 4 fragments over attribute Age, the conditions identifying fragments $F_1, \ldots, F_4$ would be defined as c_1="(Age≤30)"; c_2="(Age>30) AND (Age≤38)"; c_3="(Age>38) AND (Age≤42)"; and c_4="(Age>42) AND (Age≤50)". When using the multi-dimensional approach, condition $c_i, i = 1, \ldots, n$, is a conjunction of conditions of the form $a \leq v$

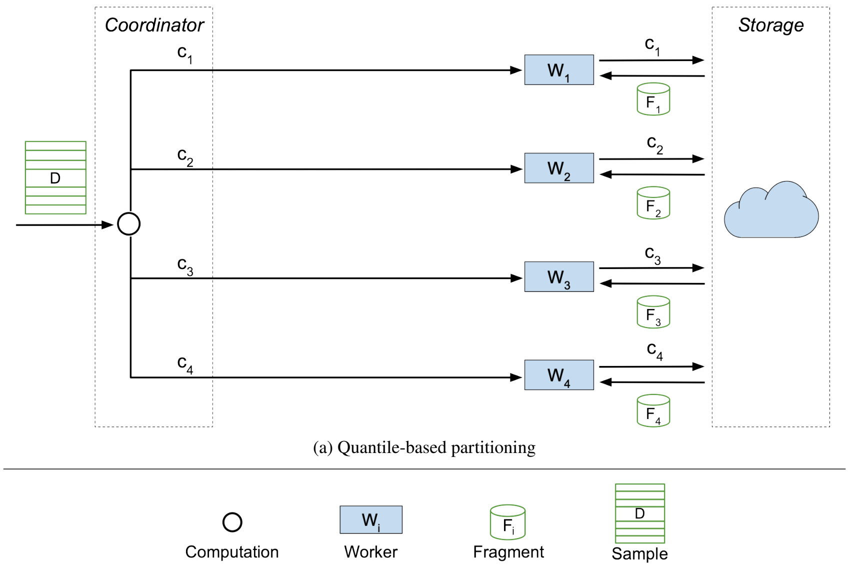
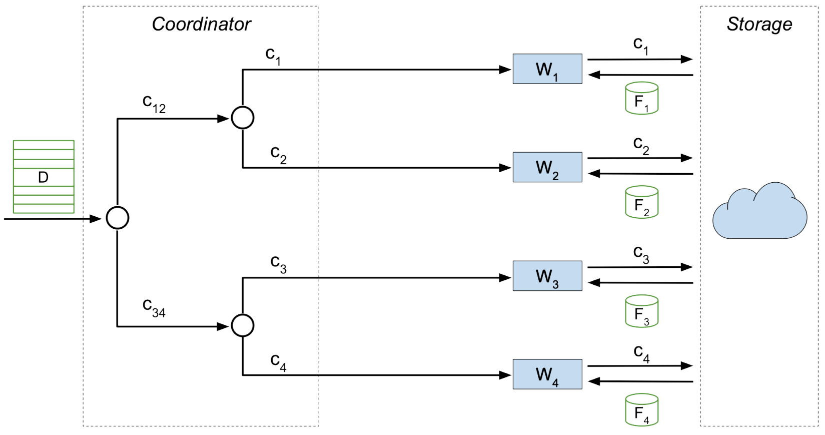


Figure 5.6: Partitioning strategies: Quantile-based partitioning

or $a > v$ describing the recursive partitioning performed by the Coordinator to obtain fragment F_i. Intuitively, each recursive call to **M_Partition** (Figure 5.5) partitions the input fragment D into two fragments F_1 and F_2, described by condition c AND $(a \leq v)$ or c AND $(a > v)$, respectively, with c the condition describing the input fragment D (empty at the first iteration), a the attribute selected for partitioning, and v the value in the domain of a corresponding to the median m of the ranking of the tuples in D according to a (i.e., $v{=}t[a]$ s.t. $rank(t[a]){=}m$). To illustrate, consider the example in Section 5.4.1 for the multi-dimensional partitioning of the dataset in Figure 5.2(a) in 4 fragments. The first partitioning, based on attribute `Age`, produces F_{12} and F_{34} described by conditions `Age`$\leq$38 and `Age`>38, respectively. At the second iteration, assume that F_{12} is split into fragments F_1 and F_2 according to the value of attribute `Country`, which is *Italy* or *France* in F_1 and *USA* or *Canada* in F_2. The conditions describing fragments F_1 and F_2 (and communicated to the corresponding workers) would be c_1 = "(`Age`$\leq$38) AND (`Country` IN $\{$*Italy*, *France*$\}$)"; c_2 = "(`Age`$\leq$38) AND (`Country` IN $\{$*USA*, *Canada*$\}$)". Figures 5.6 and 5.7 graphically illustrate the pre-processing operated by the Coordinator to partition a sample D of $\mathcal{D}$ assuming to produce 4 fragments (to be then assigned to 4 workers) adopting quantile-based and multi-dimensional partitioning. In the figures, it is denoted with $c_{i...j}$ the condition describing the fragment of D that will be further partitioned to generate $F_i, \ldots, F_j$.

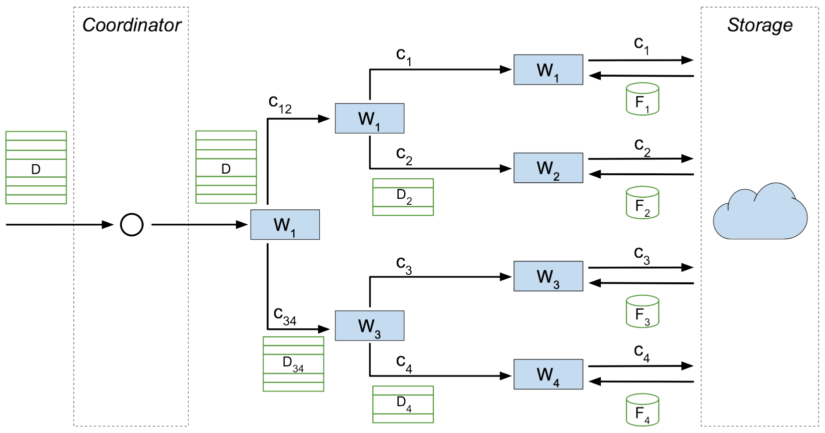
It is worth discussing an additional option: the possibility of leveraging the availability of multiple workers for the pre-processing. The multi-dimensional approach (Figures 5.5 and 5.6) could in fact be performed in parallel by workers (see Figure 5.8. Intuitively, each of the two fragments F_1 and F_2 produced by the partitioning of a fragment F (D at the first iteration) can be assigned to two different workers for further partitioning, so that partitioning can run



(b) Multi-dimensional partitioning

Figure 5.7: Partitioning strategies: Multi-dimensional partitioning

in parallel (i.e., one fragment can be partitioned by the same worker in charge of splitting F while the other can be delegated to a different worker). This strategy has been investigated, both theoretically and experimentally and, while clearly permitting to reduce the computation effort for the Coordinator, it would require data transfer among workers, resulting in lower performance than the traditional (non parallelized) multi-dimensional partitioning.



(c) Parallelized multi-dimensional partitioning

Figure 5.8: Partitioning strategies: Parallelized multi-dimensional partitioning

5.4.3 Attributes for partitioning

The first step for partitioning the dataset, regardless of the approach (i.e., quantile-based or multi-dimensional) adopted, is the selection of the quasi-identifying attribute a used to split (line 1 in procedures **Q_Partition** in Figure 5.4 and **M_Partition** in Figure 5.5).

For quantile-based partitioning, the attribute $a_i \in \mathsf{QI}$ with more distinct values in D is selected. This strategy distributes the values for a_i among different fragments, limiting the necessary amount of generalization over a_i. Indeed, generalization needs to operate only over the subset of values for a_i appearing in the fragment, which are expected to be close. On the contrary, partitioning according to a different attribute a_j having a limited number of values might cause excessive generalization for a_i, if a fragment has tuples with values at the extremes of the domain for the attribute.

For multi-dimensional partitioning, similarly to the original Mondrian approach, it is selected the attribute $a \in \mathsf{QI}$ that has, in the fragment F to be partitioned, the highest representativity of the values it assumed in D. If a is numerical, its representativity is defined as the ratio between the span (i.e., the width of the range) of the values in F and the span of the values in the entire dataset D. If a is categorical, its representativity can be defined as the ratio between the number of distinct values in F and the number of distinct values in D. Formally, the representativity $rep(a)$ of a quasi-identifying attribute $a \in \mathsf{QI}$ is defined as follows:

$$rep(a) = \begin{cases} \frac{\max_F\{t[a]\} - \min_F\{t[a]\}}{\max_D\{t[a]\} - \min_D\{t[a]\}} & \text{if } a \text{ is numerical} \\ \frac{\text{count}_F(\text{distinct } t[a])}{\text{count}_D(\text{distinct } t[a])} & \text{if } a \text{ is categorical} \end{cases} \tag{5.1}$$

where $\max_F\{t[a]\}$ and $\min_F\{t[a]\}$ ($\max_D\{t[a]\}$ and $\min_D\{t[a]\}$, resp.) are the maximum and minimum values for a assumed by the tuples in fragment F (in dataset D, resp.); and $\text{count}_F(\text{distinct } t[a])$ ($\text{count}_D(\text{distinct } t[a])$, resp.), is the number of distinct values assumed by attribute a in F (in D, resp.). For both numerical and categorical attributes, $rep(a) \in (0, 1]$.

For example, consider a fragment F with $\mathsf{QI} = \{a_1, a_2, a_3\}$, where a_1 is categorical and a_2 and a_3 are numerical. Suppose that: *i)* the distinct values for a_1 are 1000 in D and 100 in F; *ii)* the values for a_2 are in a range of width 1000 in D and of width 500 in F; and *iii)* the values of a_3 are in a range of width 1000 in D and of width 700 in F. Since $rep(a_1) = 100/1000 = 0.1 < rep(a_2) = 500/1000 = 0.5 < rep(a_3) = 700/1000 = 0.7$, a_3 is chosen for partitioning F.

Note that, at the first iteration of multi-dimensional partitioning, all quasi-identifying attributes have representativity equal to 1 because F=D (the entire dataset D is to be partitioned). In these cases, select the attribute with the maximum number of distinct values in the dataset is selected (similarly to what is done in quantile-based partitioning).

5.5 Data anonymization

During the anonymization phase, each worker anonymizes the fragment assigned to it, independently (i.e., without interacting with other workers) executing the distributed version of the Mondrian algorithm. In particular, this version of the algorithm enforces also ℓ-diversity, besides k-anonymity considered by the original approach [104]. To this end, the recursive partitioning at the core of the anonymization algorithm (Section 5.2) terminates when any further sub-partitioning would generate fragments that have less than k occurrences for the combination of values for the quasi-identifying a ttributes (which would violate k-anonymity), or less than ℓ values for the sensitive attributes (which would violate ℓ-diversity). The anonymization phase of the proposed distributed Mondrian has computational complexity $O(|F| \log |F|)$, with F the fragment input to function **Anonymize** in Figure 5.9. Indeed, the cost of lines 1–8 is $O(|F|)$, as discussed in Section 5.4. Due to the recursive calls on a partition of F including two fragments of size $\frac{|F|}{2}$, the overall complexity is $O(|F| \log |F|)$, which is in line with the complexity of the original Mondrian algorithm [104].

```
ANONYMIZE($F$)
 1: if no partitioning can be done without violating
    $k$-anonymity or $\ell$-diversity then
 2:   generalize $F$[QI]
 3: else
 4:   let $a$ be the attribute for partitioning
 5:   $R := \{rank(t[a]) \mid t \in F\}$ /* rank of $a$'s values in the ordering */
 6:   let $m$ be the median of $R$
 7:   $F_1 := \{t \in F \mid rank(t[a]) \leq m\}$
 8:   $F_2 := \{t \in F \mid rank(t[a]) > m\}$
 9:   Anonymize($F_1$)
10:   Anonymize($F_2$)
```

Figure 5.9: Anonymization algorithm for a fragment F

Figure 5.9 illustrates the anonymization algorithm executed by each worker for anonymizing the fragment assigned to it. The recursive partitioning (lines 3–10), which operates according to the same logic as multi-dimensional partitioning in the pre-processings phase (Section 5.4), terminates when any further sub-partitioning of a fragment would violate k-anonymity or ℓ-diversity (lines 1–2). The attribute a with maximum representativity (Equation 5.1) is chosen for partitioning (line 4). Clearly, since during the anonymization phase each worker w_i has visibility only on its fragment F_i, representativity is computed over F_i (in contrast to the entire dataset D in Equation 5.1). Like in multi-dimensional partitioning, at the first recursive call, representativity is equal to 1 for all quasi-identifying attributes. The implemented approach then selects the attribute that has the highest number of distinct values in the fragment. When a fragment F cannot be further partitioned as this would violate k-anonymity or ℓ-diversity, the anonymization algorithm produces the anonymized version of F, obtained generalizing the values of the quasi-identifying attributes to guarantee that all the tuples share the same (generalized) quasi-identifier values (line 2). The proposed distributed Mondrian approach supports the following generalization strategies:

- *Generalization hierarchies*: applicable to categorical attributes only, the values for attribute $a \in$QI are substituted with their lowest common ancestor in the generalization hierarchy $\mathcal{H}(a)$ defined for a (Section 5.3). To illustrate, consider the dataset in Figure 5.2(a) and suppose, for simplicity, it is a fragment retrieved by a worker for anonymization. Its anonymized version in Figure 5.2(c) is obtained generalizing attribute `Country` according to the generalization hierarchy in Figure 5.1, considering the partitions in Figure 5.2(b). For example, values *Italy*, *Italy*, and *France* of the first three tuples (partition at the bottom-left of Figure 5.2(b)) are generalized to their lowest common ancestor in the hierarchy (i.e., *Europe*).

- *Common prefix:* applicable to categorical and numerical attributes interpreted as strings, the values for attribute $a \in \mathsf{QI}$ are replaced with a string that includes their common prefix (substituting with a wildcard character the characters that differ). For example, values 10010, 10020, 10030 for attribute `ZIP` can be generalized to $100{*}{*}$, maintaining common prefix 100 and redacting the last two characters.

- *Set definition:* applicable to both categorical and numerical attributes, the values for attribute $a \in \mathsf{QI}$ are replaced with the set of values including all of them. For example, values *Italy*, *Italy*, and *France* for attribute `Country` can be generalized to the set $\{$*Italy*, *France*$\}$.

- *Interval definition:* applicable to numerical attributes defined on a totally ordered domain, the values for attribute $a \in \mathsf{QI}$ are replaced with a range of values including all of them. While the smallest range containing all the values to be generalized (i.e., the one delimited by the minimum and maximum value) is the most natural choice, also larger (possibly pre-defined) intervals may be adopted. Note that this generalization is different from set definition: while the set definition explicitly maintains all the (original) values generalized in a set, interval definition maintains only the extremes of the range. To illustrate, consider the dataset in Figure 5.2(a) and suppose, for simplicity, it is a fragment retrieved by a worker for anonymization. Its anonymized version in Figure 5.2(c) is obtained generalizing attribute `Age` by grouping its values in intervals, considering the partitions in Figure 5.2(b). For example, values 42, 50, and 43 (fourth, fifth, and sixth tuples, corresponding to the partition on the right-hand-side of Figure 5.2(b)) are generalized to [42,50].

5.6 Wrap up and information loss assessment

The wrap-up phase of the illustrated approach is aimed at collecting anonymized fragments, and at assessing information loss. To this purpose, each worker stores the anonymized fragment $\hat{F}$ assigned to it in the storage platform, and computes the information loss implied by the anonymization of F. The information loss characterizing the anonymized fragments are collected and combined by the Coordinator to assess the information loss of the entire dataset. The following parts of this section describe the information loss metrics adopted. For the sake of readability, the formulas refer to a dataset D and its anonymized version $\hat{D}$, with the note that each worker operates on the fragment F (and its anonymization $\hat{F}$) assigned to it.

- *Discernibility Penalty* (DP [24, 104]) assigns a *penalty* to each tuple in D based on the size of the equivalence class E (the larger an equivalence class, the larger the penalty) to which the tuple belongs (i.e., number of tuples generalized to the same values). Formally, the Discernibility Penalty of an anonymized dataset $\hat{D}$ is computed as follows:

$$\mathsf{DP}(\hat{D}) = \sum_{E \in \hat{D}} |E|^2 \tag{5.2}$$

- *Normalized Certainty Penalty* (NCP [186]) assigns penalties based on the amount of generalization (more generalization resulting in higher penalties) applied to the values of the quasi-identifying attributes. NCP is applicable to numerical attributes generalized in intervals, and to categorical attributes for which a generalization hierarchy exists. Given a tuple $t \in D$, the normalized certainty penalty $\mathsf{NCP}_a(\hat{t})$ of its generalization $\hat{t} \in \hat{D}$ for attribute $a \in \mathsf{QI}$ is computed as follows:

$$\mathsf{NCP}_a(\hat{t}) = \begin{cases} \frac{v_{max} - v_{min}}{Range(a)} & \text{if } a \text{ is numerical} \\ \frac{|Ind(\hat{v})|}{|Dom(a)|} & \text{if } a \text{ is categorical} \end{cases} \tag{5.3}$$

  where $\hat{t}[a]$=$[v_{max}, v_{min}]$ and $Range(a)$ is the range of the values assumed by a, if a is a numerical attribute generalized in intervals; $\hat{t}[a]$=$\hat{v}$ and $Ind(\hat{v})$ is the set of values in $Dom(a)$ that could be generalized to $\hat{v}$ (i.e., the number of leaves of the generalization hierarchy $\mathcal{H}(a)$ for a that are descendants of $\hat{v}$), if a is a categorical attribute with generalization hierarchy.
  Given an anonymized dataset $\hat{D}$ with quasi-identifier QI, the Normalized Certainty Penalty of $\hat{D}$ is computed summing the Normalized Certainty Penalties of the attributes in QI for all the tuples in $\hat{D}$ as follow:

$$\mathsf{NCP}(\hat{D}) = \sum_{\hat{t} \in \hat{D}} \sum_{a \in \mathsf{QI}} \mathsf{NCP}_a(\hat{t}) \tag{5.4}$$

Given the information loss measures $\mathsf{DP}(\hat{F}_1), \ldots, \mathsf{DP}(\hat{F}_{|W|})$ ($\mathsf{NCP}(\hat{F}_1), \ldots, \mathsf{NCP}(\hat{F}_{|W|})$, resp.) for the fragments, the Coordinator can compute the information loss for the whole dataset by simply summing them. For evaluating the approach presented in this chapter, both DP and NCP metrics have been taken into consideration, since they take a different approach in the assessment: DP is independent from the amount of generalization adopted and only considers the size of equivalence classes, while NCP precisely assesses the amount of generalization, regardless of the number of tuples in equivalence classes.

5.7 Implementation

In this section, it is illustrated the architectural design and the deployment of the implemented prototype for the proposed distributed anonymization approach. The implementation is available at `https://github.com/mosaicrown/mondrian`.

5.7.1 Architecture

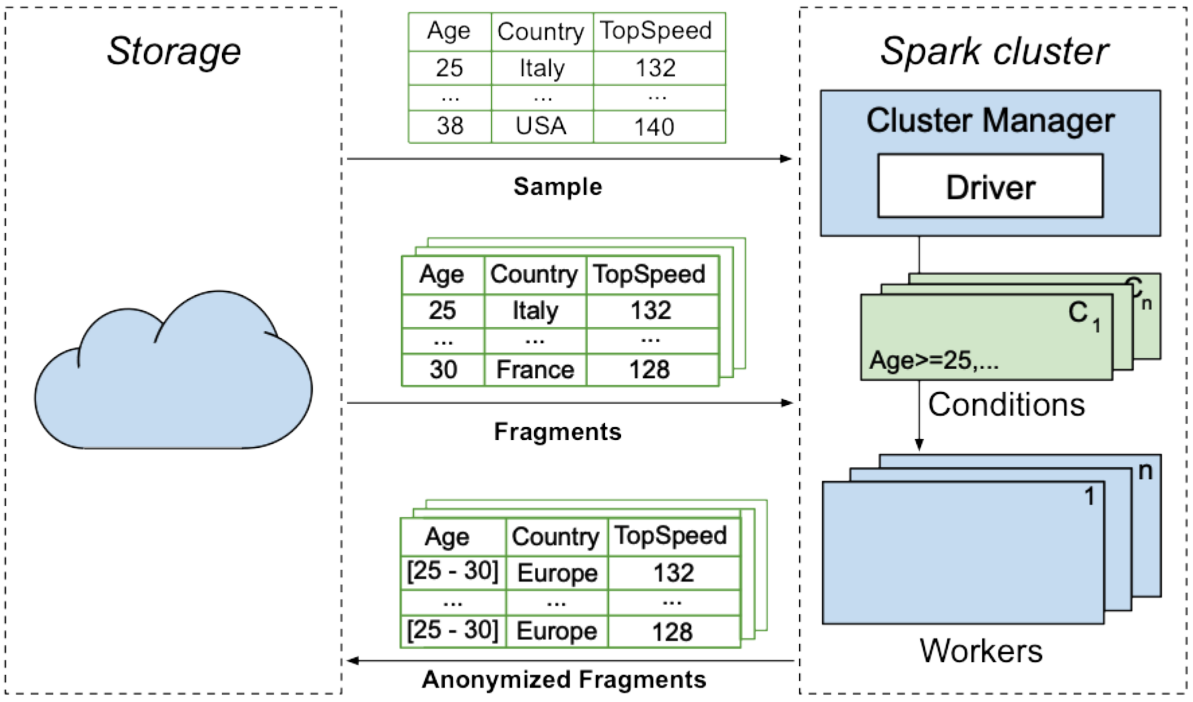


Figure 5.10: Spark-based distributed anonymization system

The implementation is based on an *Apache Spark* cluster, whose nodes perform the three phases (pre-processing, anonymization, and wrap-up) of the proposal. Figure 5.10 illustrates the components and working of the prototype. The dataset $\mathcal{D}$ to be anonymized can be stored on any storage platform (centralized or distributed) reachable by Apache Spark with a URL. The Spark cluster includes a *Spark Cluster Manager*, which coordinates the cluster, and a set W of *Spark Workers*, which perform the tasks assigned to them by the Cluster Manager. The implementation of the proposed distributed anonymization in Python. This allows to leverage the Pandas framework [121, 143], which can be conveniently used for managing very large datasets.

In the Spark cluster architecture in Figure 5.10, the *Spark Driver* plays the role of our Coordinator: it is responsible for calling the *Spark Context* in the user-written code, implementing the partitioning process. (Note that the Spark Driver and the Spark Cluster Manager are not necessarily hosted on the same node of the cluster.) The Spark Driver is also responsible for translating the code to be executed in the cluster into jobs, which are further divided into smaller execution units, called *tasks* (which actually implement the distributed anonymization), executed by the workers.
To anonymize a dataset $\mathcal{D}$, first the Spark Driver downloads from the storage platform a sample D of $\mathcal{D}$ that fits into its memory. It then locally executes the pre-processing phase. In particular, the Spark Driver locally partitions the downloaded sample D running procedure **Q_Partition** in Figures 5.4 (if quantile-based approach is adopted), or procedure **M_Partition** in Figure 5.5 (if multi-dimensional approach is adopted), keeping track of the conditions describing the computed fragments. The Spark Driver then defines a set of $|W|$ Spark Tasks. Each task corresponds to the anonymization of a fragment F_i and includes the conditions defining F_i. Once the Spark Driver has terminated the pre-processing phase, the Spark Cluster Manager selects the workers that will be involved in the distributed anonymization process. The Spark Driver then sends to each identified worker w_i its anonymization task, enabling w_i to download from the storage platform the tuples in the fragment F_i assigned to it. Each Spark Worker w then retrieves from the storage platform the fragment of $\mathcal{D}$ satisfying the conditions included in the task assigned to it. The Spark Worker anonymizes the fragment through the distributed Mondrian (Figure 5.9, Section 5.5), computes the amount of information loss caused by anonymization (Equations 5.2 and 5.4, Section 5.5), and stores the results in the storage platform using the services of the Spark Driver. The Spark Driver combines the information loss computed by the Spark Workers to obtain the information loss of the overall dataset.

5.7.2 Deployment

Aiming at creating a solution that can be easily deployed in a cloud infrastructure (e.g., AWS or Google Cloud), the prototype is delivered as a multi-container application, leveraging Docker containers, in order to achieve an architecture that is easy to deploy. The deployed Spark architecture is shown in Figure 5.10 and is composed of: *i)* a Docker container for the Spark Driver; *ii)* a Docker container for the Spark Cluster Manager; *iii)* a variable number of Docker containers for the Spark Workers; and *iv)* a Docker container to expose a *Spark History Server*, for additional information about the task scheduling and assignment performed by Spark.
In this prototype, Docker containers are spawned through Docker Compose. To manage the distribution of the containers to the nodes (machines) of the Spark cluster, different orchestrator tools (e.g., Kubernetes) can be adopted. Due to its simplicity, for the prototype it has been leveraged Docker Swarm. Figure 5.11 illustrates an example of the Docker Swarm distribution

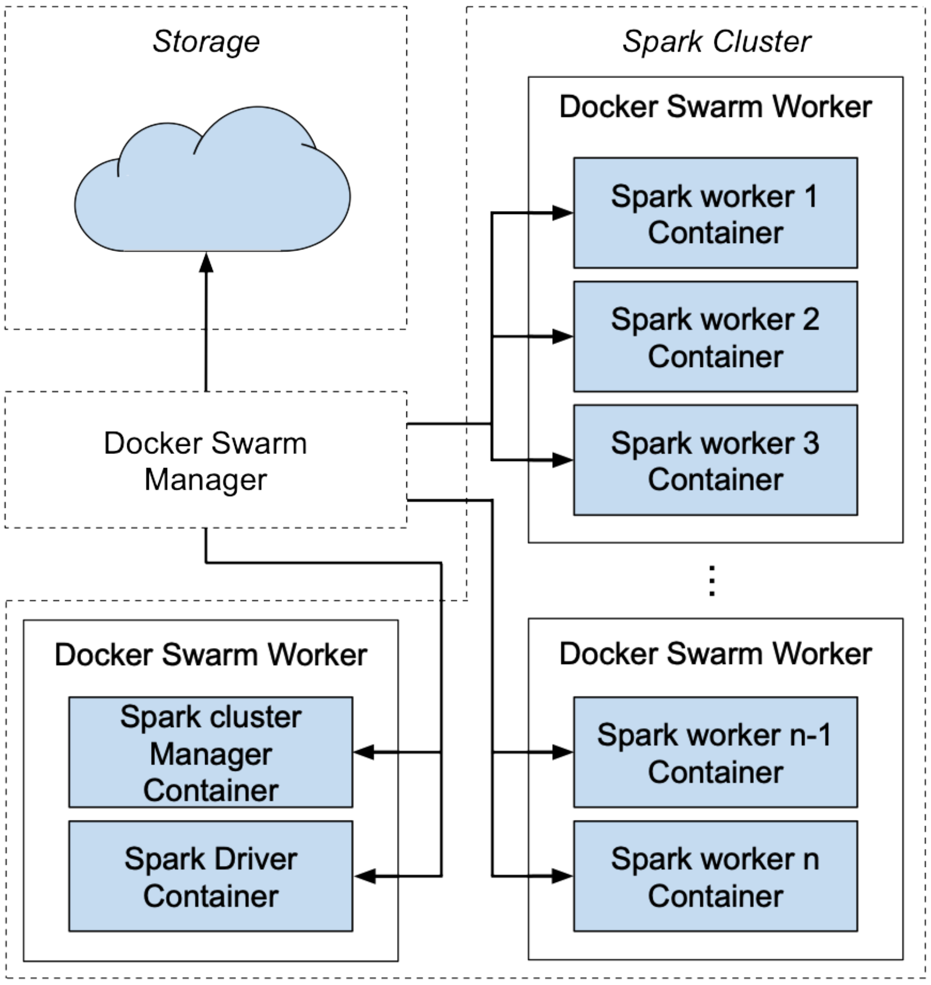


Figure 5.11: Container deployment in a cloud environment

of Docker containers to the nodes of a Spark cluster. In the figure, white solid boxes represent the nodes in the Spark cluster, while blue (gray, in b/w printouts) boxes represent Docker containers. Note that each node in the Spark cluster can spawn more than one Docker container. One of the nodes in the cluster acts as *Docker Swarm Manager*, while the other nodes act as *Docker Swarm Workers*. The Docker Swarm Manager coordinates and is in charge of distributing the workload on the Docker Swarm Workers, which in turn are used to spawn the Docker containers that act as Spark Manager, Spark Driver, and Spark Workers. One of the Docker Swarm Workers is then dedicated to spawn one container for the Spark Cluster Manager and one container for the Spark Driver. The other Docker Swarm Workers spawn the containers modeling Spark Workers.

5.8 Experimental results

Several experiments have been performed to evaluate the scalability and applicability of the proposed distributed Mondrian anonymization approach, compared to the traditional centralized

Mondrian algorithm, considered as a baseline.

5.8.1 Experimental settings

Server specifications and cloud deployment. Since the solution does not require any specific cloud environment, to obtain reproducible results, in the experiments a cloud environment have been simulated using Docker Compose. Experiments have been run on a machine equipped with an AMD Ryzen 3900X CPU (12 physical cores, 24 logical cores), 64 GB RAM and 2 TB SSD, running Ubuntu 20.04 LTS, Apache Spark 3.0.1, Hadoop 3.2.1, and Pandas 1.1.3. Each worker is equipped with 2GB of RAM and 1 CPU core. Centralized Mondrian relies on 1 CPU core, with no limitation on the use of the RAM. To prove the applicability and scalability of our solution in a real-world distributed environment, the prototype has also been deployed in Amazon Elastic Compute Cloud (*t2.medium* instances equipped with 2 cores, 4GB of RAM, and 8GB of gp3 SSD, running Ubuntu 20.04 LTS, and located in the *us-east-1* region). The results obtained in this real-world cloud environment confirm the ones obtained in our simulated environment illustrated in Section 5.8.2.

Storage platform. *Hadoop Distributed File System (HDFS)* has been used as the distributed storage platform for storing the dataset to be anonymized. The HDFS cluster has been deployed leveraging Docker containers, with one container for the Hadoop Namenode (responsible for the cluster management); and multiple containers for the Hadoop Datanodes (responsible for storing data and servicing read and write requests).

Dataset. The considered datasets are the Poker Hand dataset [34] and ACS PUMS USA 2019 dataset [146]. The choice has been dictated by the need to consider very large datasets, to test the scalability of our approach. In addition to these results, in Appendix C are illustrated the steps necessary to reproduce the results presented in this chapter on a smaller sample of the well-known ACS PUMS USA 2018 dataset [146].
The Poker Hand dataset is composed of 1,000,000 tuples. Each tuple represents the cards in a hand of Poker. Each card is described through 2 attributes: the seed (an integer value in the range $\{1, \ldots, 4\}$) and the rank (an integer value in the range $\{1, \ldots, 13\}$). These attributes have been considered as the quasi-identifier in the experiments. An additional attribute identifying the entire hand (an integer value in the range $\{0, \ldots, 9\}$) has been treated as the sensitive attribute during the evaluation.
For the ACS PUMS USA 2019 dataset, a sample of 1,500,000 tuples has been used. Each tuple of the dataset represents an individual respondent with attributes `ST`, `OCCP`, `AGEP`, and `WAGP`, representing respectively the respondent's US State of residence, occupational status (expressed with a numeric code), age, and annual income.`ST`, `OCCP`, `AGEP` has been considered as the quasi-identifier, and `WAGP` as the sensitive attribute. While `OCCP`, `AGEP`, and `WAGP`

are numeric attributes, ST is categorical. For attribute ST, a generalization hierarchy $\mathcal{H}(\texttt{ST})$ defined according to the criteria adopted by the US Census Bureau [1] has been employed. In this hierarchy States are at the leaf level and are grouped in Divisions, which are in turn grouped in Regions. For example, States NJ, NY, and PA (leaf level) can be generalized to MiddleAtlantic (which is their parent in the hierarchy), which in turn can be generalized to Northeast, which in turn can be generalized to US (which is the root of the hierarchy).

5.8.2 Results

To assess the scalability of the approach described in the chapter, it has been analyzed the computation time of our distributed Mondrian varying the number of workers. Also, to assess the quality of the solution computed by the proposed distributed Mondrian, it has been analyzed the information loss varying the size of the sample used for partitioning the dataset among workers. Both computation times and information loss values reported in this section have been obtained as the average over 5 runs. The values are compared with centralized Mondrian, considered as the *baseline* for the evaluation performed.

Computation time. Figure 5.12 compares the computation times of the proposed distributed Mondrian anonymization algorithm over Poker Hand dataset, while Firgue 5.13 shows the results obtained over the ACM PUMS USA 2019 sample. Both evaluations are performed for the two partitioning methods, then they are compared with centralized Mondrian anonymization, and results have collected varying the number of workers and values of parameters k and ℓ. In particular, the values for k are $\{5, 10, 20\}$, while the ones for ℓ are $\{2, 3, 4\}$. The numbers of of workers considered are between 2 and 12 workers (12 is the largest number of partitions that quantile-based partitioning can produce due to the domain of the attributes of the Poker Hand dataset). As expected, for the Poker Hand dataset, the execution time decreases when the number of workers grows, with savings with respect to centralized Mondrian between 28% and 98%, confirming the scalability of the proposed distributed approach. Similar results can be observed on the ACS PUMS USA 2019 dataset, with savings between 45% and 98%. As visible from the figures, the chosen pre-processing strategy does not significantly affect computation times. Quantile-based and multi-dimensional partitioning exhibit the same execution time when using two workers. Indeed, both approaches would perform the same partitioning. It is also possible to note that, when using quantile-based partitioning, each additional worker provides a saving in computation time. On the contrary, when using multi-dimensional partitioning, computation time saving can be enjoyed when the number of workers reaches a power of 2 (i.e., it is necessary to have 2^i workers for saving on computation time). For instance, this is particulary visible in Figure 5.12, where there is a marginal saving when passing from 6 to 7 workers, but there is a considerable saving from 7 to 8 workers. This is due to the fact that, when the number of workers is not a power of 2, some workers are assigned twice the workload as the others (see

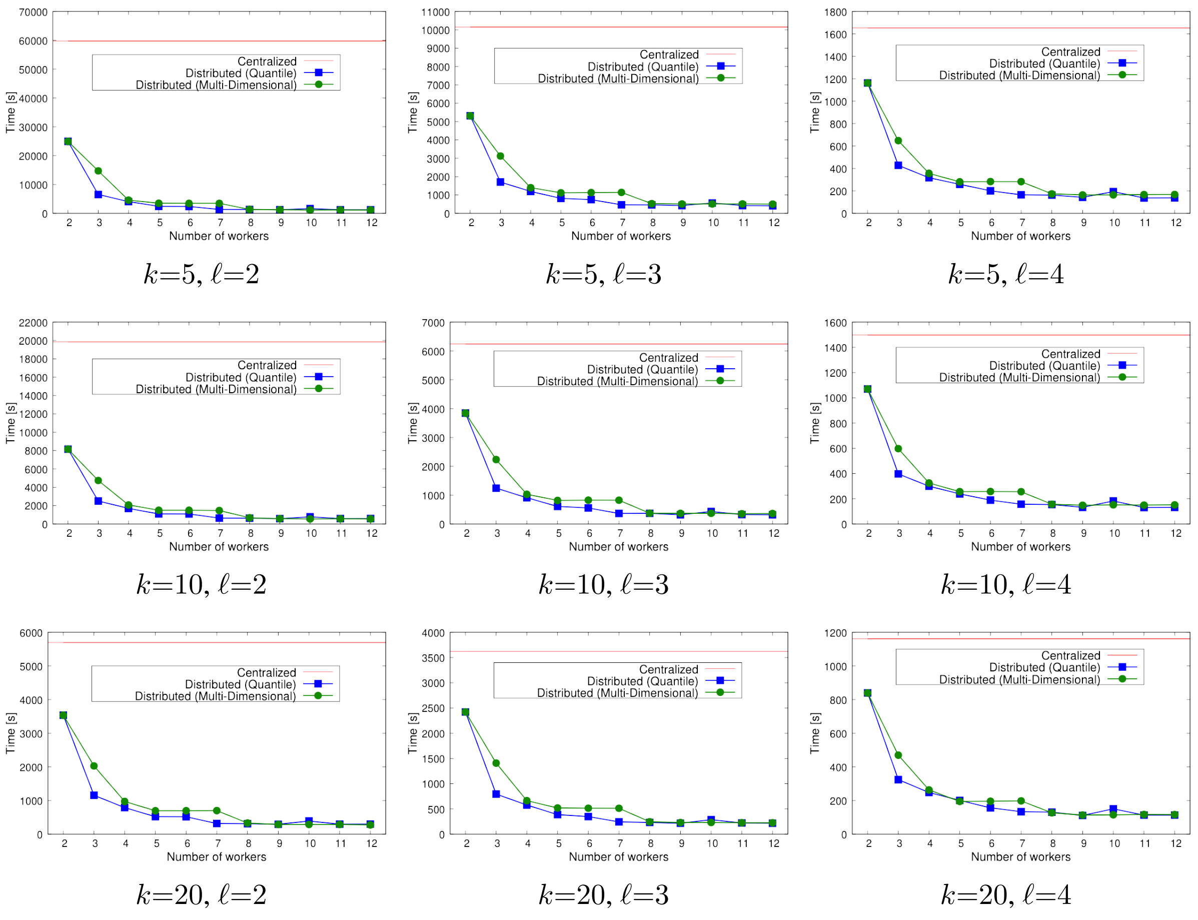


Figure 5.12: Execution times of centralized Mondrian and distributed Mondrian varying the number of workers, k, and ℓ (Poker Hand)

Section 5.4.1) and therefore represent a bottleneck.

Information loss. The usage of the proposed distributed Mondrian might cause additional information loss compared to the centralized Mondrian, since each worker independently operates on its fragment without coordinating with other workers. The evaluation performed allows to observe that the information loss caused by distribution can be impacted by: 1) the number of workers (and hence of fragments), and 2) the size of the sample used to partition the dataset. Figure 5.14 compares the average information loss (and its variance) obtained in 5 runs of the distributed (with 5 and 10 workers) Mondrian with the average information loss of the centralized Mondrian for computing a k-anonymous (with $k = 5$, $k = 10$, and $k = 20$) 2-diverse version of the Poker Hand dataset, assuming different sampling sizes (0.1%, 0.01%, 0.001%). The value used for ℓ is 2. The same results but produced over the ACM PUMS USA 2019 are visible in 5.15. The results in Figure 5.14 and 5.15 show that, for all values of k, sampling has a very limited impact on information loss. The results also confirm that, for all values of k, also the number of workers has negligible impact on information loss. More precisely, multi-dimensional

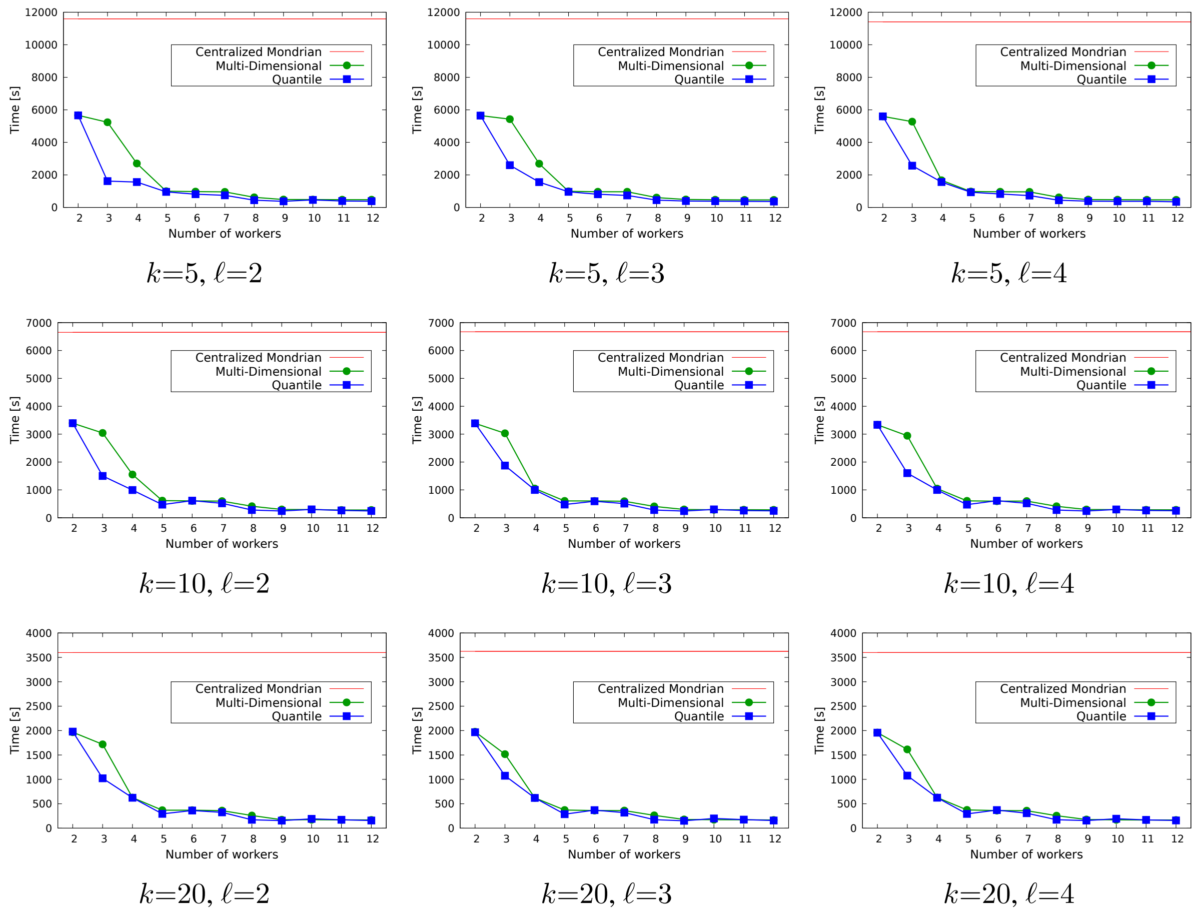


Figure 5.13: Execution times of centralized Mondrian and distributed Mondrian varying the number of workers, k, and ℓ (ACS PUMS USA 2019)

partitioning performs similarly for all tested numbers of workers, while quantile-based partitioning produces higher information loss when the number of workers grows. The reported values for DP reveal that multi-dimensional partitioning produces equivalence classes similar to the ones produced by centralized Mondrian, while the quantile-based approach produces slightly smaller equivalence classes, especially when the sample used for partitioning is small. The results also show that, in some of the tested scenarios, the values for DP are higher in the centralized scenario. Even if the difference is negligible, it reveals that, when using sampling for partitioning the dataset, (a subset of) the equivalence classes are smaller compared to the equivalence classes obtained without sampling. Smaller equivalence classes, however, do not imply less generalization, as testified by the values of NCP. Indeed, quantile-based partitioning, multi-dimensional partitioning, and the centralized algorithm present similar (small) values for NCP.

		Information Loss (DP)		**Information Loss** (NCP)	
Sampling	**Partitioning**	**5 workers**	**10 workers**	**5 workers**	**10 workers**
0.1%	Quantile	7.14e06 ± 5.13e02	7.10e06 ± 8.31e03	1.83e06 ± 3.18e02	1.97e06 ± 3.99e02
	Multi-dimensional	7.22e06 ± 1.63e04	7.22e06 ± 9.25e03	1.80e06 ± 2.46e03	1.80e06 ± 2.15e03
0.01%	Quantile	7.14e06 ± 2.78e04	7.10e06 ± 9.35e03	1.83e06 ± 3.41e03	1.96e06 ± 1.01e04
	Multi-dimensional	7.17e06 ± 1.93e04	7.15e06 ± 1.36e04	1.79e06 ± 4.02e03	1.80e06 ± 7.44e03
0.001%	Quantile	7.14e06 ± 2.78e04	7.13e06 ± 1.27e04	1.83e06 ± 3.41e03	1.91e06 ± 7.92e02
	Multi-dimensional	7.20e06 ± 5.02e04	7.20e06 ± 5.02e04	1.80e06 ± 3.82e03	1.80e06 ± 3.82e03
Centralized		7.23e06		1.50e06	

(a) k=5, ℓ=2

		Information Loss (DP)		**Information Loss** (NCP)	
Sampling	**Partitioning**	**5 workers**	**10 workers**	**5 workers**	**10 workers**
0.1%	Quantile	1.42e07 ± 7.44e03	1.41e07 ± 9.94e03	2.20e06 ± 1.38e03	2.34e06 ± 1.14e03
	Multi-dimensional	1.42e07 ± 2.39e04	1.43e07 ± 1.47e04	2.17e06 ± 6.50e02	2.17e06 ± 8.62e02
0.01%	Quantile	1.42e07 ± 2.16e04	1.41e07 ± 7.92e03	2.20e06 ± 1.67e03	2.34e06 ± 1.10e04
	Multi-dimensional	1.42e07 ± 2.35e04	1.42e07 ± 1.71e04	2.17e06 ± 4.90e03	2.17e06 ± 4.29e03
0.001%	Quantile	1.42e07 ± 2.16e04	1.41e07 ± 2.95e04	2.20e06 ± 1.67e03	2.24e06 ± 8.00e04
	Multi-dimensional	1.43e07 ± 2.36e04	1.47e07 ± 2.36e04	2.17e06 ± 4.67e03	2.17e06 ± 4.51e03
Centralized		1.43e07		1.82e06	

(b) k=10, ℓ=2

		Information Loss (DP)		**Information Loss** (NCP)	
Sampling	**Partitioning**	**5 workers**	**10 workers**	**5 workers**	**10 workers**
0.1%	Quantile	2.88e07 ± 2.82e04	2.86e07 ± 2.06e04	2.50e06 ± 3.30e01	2.66e06 ± 1.55e03
	Multi-dimensional	2.88e07 ± 4.44e04	2.88e07 ± 5.77e04	2.47e06 ± 1.56e03	2.46e06 ± 4.32e03
0.01%	Quantile	2.88e07 ± 6.73e04	2.85e07 ± 4.21e04	2.50e06 ± 1.32e03	2.65e06 ± 1.76e04
	Multi-dimensional	2.88e07 ± 5.03e04	2.88e07 ± 1.86e04	2.47e06 ± 5.75e03	2.46e06 ± 4.00e03
0.001%	Quantile	2.88e07 ± 6.73e04	2.86e07 ± 2.86e04	2.50e06 ± 1.32e03	2.63e06 ± 3.43e03
	Multi-dimensional	2.88e07 ± 7.23e04	2.88e07 ± 6.83e04	2.47e06 ± 8.32e03	2.46e06 ± 6.17e03
Centralized		2.88e07		2.07e06	

(c) k=20, ℓ=2

Figure 5.14: DP and NCP information loss varying the number of workers and k (Poker Hand)

The experiments confirm that our distributed Mondrian provides high scalability, while causing limited impact on information loss. When the number of workers is a power of 2, multi-dimensional and quantile-based partitioning exhibit similar computation time. However, when the number of workers is not a power of 2, quantile-based partitioning provides better performance. Both approaches have limited impact on information loss.

5.9 Related work

The problem of protecting privacy in data publishing has been widely studied (e.g., [64, 65, 68, 149, 171]). The solutions proposed in the literature include both syntactic (e.g., k-anonymity [149] and ℓ-diversity [117]) and semantic techniques (e.g., differential privacy [68] and its variations [69]). Traditional algorithms aimed at enforcing k-anonymity and/or ℓ-diversity (e.g., [90, 103, 104]) operate in centralized scenarios. The problem of distributing and

		Information Loss (DP)		**Information Loss** (NCP)	
Sampling	**Partitioning**	**5 workers**	**10 workers**	**5 workers**	**10 workers**
0.1%	Quantile	4.12e07 ± 1.06e06	4.21e07 ± 2.03e06	1.69e05 ± 5.22e03	1.72e05 ± 1.91e03
	Multi-dimensional	3.91e07 ± 1.51e06	3.95e07 ± 1.22e06	1.27e05 ± 9.43e03	1.31e05 ± 1.13e03
0.01%	Quantile	4.14e07 ± 1.21e06	4.24e07 ± 1.98e06	1.68e05 ± 3.02e03	1.76e05 ± 4.93e03
	Multi-dimensional	4.06e07 ± 1.21e06	4.16e07 ± 1.54e06	1.41e05 ± 2.81e04	1.46e05 ± 2.74e04
0.001%	Quantile	4.15e07 ± 1.15e06	4.22e07 ± 1.72e06	1.75e05 ± 2.44e03	1.76e05 ± 4.11e03
	Multi-dimensional	4.12e07 ± 1.53e06	4.13e07 ± 2.01e06	1.79e05 ± 2.91e03	1.81e05 ± 3.41e03
Centralized		3.87e07		1.21e05	

(a) k=5, ℓ=2

		Information Loss (DP)		**Information Loss** (NCP)	
Sampling	**Partitioning**	**5 workers**	**10 workers**	**5 workers**	**10 workers**
0.1%	Quantile	5.46e07 ± 1.33e04	5.70e07 ± 5.46e04	1.95e05 ± 7.41e03	2.03e05 ± 1.05e03
	Multi-dimensional	5.46e07 ± 3.44e05	5.46e07 ± 5.31e05	1.58e05 ± 1.01e04	1.67e05 ± 2.11e04
0.01%	Quantile	5.47e07 ± 1.72e06	5.55e07 ± 9.22e05	2.02e05 ± 3.62e03	2.04e05 ± 5.06e03
	Multi-dimensional	5.47e07 ± 2.92e05	5.47e07 ± 6.09e05	1.64e05 ± 1.81e04	1.83e05 ± 3.53e04
0.001%	Quantile	5.56e07 ± 1.25e06	5.57e07 ± 1.81e06	2.03e05 ± 2.17e03	2.07e05 ± 3.57e03
	Multi-dimensional	5.47e07 ± 3.23e05	5.47e07 ± 4.43e05	1.95e05 ± 2.05e04	1.98e05 ± 1.51e04
Centralized		5.46e07		1.46e05	

(b) k=10, ℓ=2

		Information Loss (DP)		**Information Loss** (NCP)	
Sampling	**Partitioning**	**5 workers**	**10 workers**	**5 workers**	**10 workers**
0.1%	Quantile	6.31e07 ± 9.82e05	6.42e07 ± 8.06e05	2.28e05 ± 1.03e04	2.32e05 ± 9.05e03
	Multi-dimensional	5.91e07 ± 7.44e05	6.21e07 ± 5.61e05	1.91e05 ± 2.08e04	1.96e05 ± 1.33e04
0.01%	Quantile	6.42e07 ± 8.73e05	6.55e07 ± 4.22e05	2.31e05 ± 5.8e03	2.36e05 ± 5.49e03
	Multi-dimensional	6.40e07 ± 3.22e05	6.38e07 ± 4.09e05	1.96e05 ± 4.17e04	2.11e05 ± 2.03e04
0.001%	Quantile	6.54e07 ± 5.15e05	6.61e07 ± 6.83e05	2.37e05 ± 3.31e03	2.37e05 ± 3.57e03
	Multi-dimensional	6.51e07 ± 3.23e05	6.57e07 ± 2.43e05	2.07e05 ± 1.09e01	2.12e05 ± 2.11e04
Centralized		5.73e07		1.81e05	

(c) k=20, ℓ=2

Figure 5.15: DP and NCP information loss varying the number of workers and k (ACM PUMS USA 2019)

parallelizing anonymization has been recently studied, to the aim of protecting also large datasets (e.g., [188, 190]). The approach in [190] partitions the dataset and anonymizes the resulting fragments, leveraging MapReduce [53] paradigm to parallelize the centralized anonymization solution in [76]. The proposal in [190] takes a different approach in partitioning with respect to the one presented in this chapter, since it aims at maintaining in each fragment the same value distribution as the whole dataset while the approach just illustrated aims at maintaining in each fragment homogeneous values for the quasi-identifier, so to reduce the amount of generalization needed to enforce k-anonymity (and hence the information loss due to generalization). The distributed anonymization approach in [188] partitions data so that fragments contain records that are semantically similar (leveraging Locally Sensitive Hashing, semantic distance measure, and k-member clustering), but it does not leverage Mondrian for computing fragments.

Different distributed anonymization approaches rely, similarly to the proposal illustrated in this chapter, on distributed architectures for parallelization (e.g., [15, 17, 18, 25, 35, 159, 189]). The approach in [17] parallelizes Mondrian through Apache Spark, but relies on data exchange among workers to coordinate anonymization of different portions of the original dataset distributed to workers. In the researched approach one of the objective is limiting data exchange among workers. The solution in [18] differs from the presented research since it uses hierarchical clustering and k-means to provide ℓ-diversity instead of performing partitioning according to Mondrian strategy. The approach in [15] considers Apache Spark for parallelizing different anonymization approaches, but does not discuss the Spark-based adaptation of Mondrian. The solution in [159] randomly assigns tuples to workers, while the solution just described specifically studies a strategy for distributing tuples to workers to minimize information loss. The first approach aimed at parallelizing Mondrian algorithm has been proposed in [35] and is based on MapReduce paradigm. This solution heavily relies on data exchanges among workers. Again, this is avoided in the proposed approach in order to minimize the need for workers to communicate for anonymization purposes so to reduce the delays and costs inevitably entailed by exchanging data over a network. A more recent approach relying on MapReduce for parallelizing Mondrian algorithm has been proposed in [189]. Besides the use of MapReduce in contrast to Apache Spark, this solution differs from the one presented in this chapter in the strategy adopted for splitting the dataset among workers. The approach in [189] operates on the whole dataset (and not on a sample of the same) and adopts a distributed algorithm for partitioning, using a tree structure shared among the workers in the cluster. Also, the proposal in [189] does not use quantiles for partitioning. The proposal in [25], which enforces Mondrian using Spark, is complementary to what has been exposed in this thesis, as it focuses on improving performances by optimizing data structures used by Spark.

5.10 Conclusions

This chapter explores a technique that can be used to enforce privacy guarantees over data collections that are part of the security boundaries defined through the technique exposed in Chaper 2, 3 and 4. The research produced a prototype of scalable approach for the distributed execution of the Mondrian algorithm. This allows developers to achieve data anonymization of very large datasets by fully utilizing the computational resources at their disposal.

Chapter 6. Enabling queries on encrypted data

6.1 Introduction

The technique described in Chapter 5 is effective in enforcing privacy guarantee of data points of a data collection. Such method can be applied when the dataset is composed of information that can be divided into sensitive and not. In general, data sanitization techniques are applicable when privacy is the main concern that has to be guaranteed. In a case in which every piece of information has to be protected, such approach is not able to enforce the correct security guarantees. The classical solution to this threat is represented by the use of encryption, so that the control of the physical representation of the data does not give access to the information content, as long as the attacker does not have access to the encryption key. A valuable addition to the sandbox that can be constructed through the technique described in Chapter 2, 3 and 4 is to adapt encryption scheme in such a way that compromised components are not able to reach access credentials(e.g. encryption keys).

Concerning data in the form of relational tables, the research and development community have dedicated significant effort to this problem, considering different lines of investigations. Possible approaches include: *i)* the use of searchable encryption (e.g., [139]), supporting the evaluation of conditions on encrypted data; *ii)* the use of trusted hardware components at the server (e.g., Intel SGX), offering a trusted execution environment residing at, but not accessible by, the server (e.g., [154]); *iii)* the association with the encrypted data of metadata working as indexes offering support for the evaluation of conditions (e.g., [46, 87]). All these approaches represent valid alternatives depending on the application scenario. The first two, while enjoying strong protection guarantees, suffer from a significant performance overhead, making them still not applicable in many practical scenarios. On the other hand, indexes, while applicable in practice, may suffer from a possible exposure to inferences, as they might leak information on the values behind them once the data collection has been leaked. The vulnerability of indexes typically resides in the frequencies of their occurrences, which can bear relationship with the plaintext values. Frequencies of both individual attributes values as well as combinations of them can be exposed to inference. A solution to this problem is guaranteeing indexes with collisions (i.e., mapping different plaintext values into the same index value), so to provide confusion and indistinguishability. Unfortunately, this is easier said than done, as constructing such an index requires addressing two (interconnected) aspects which are far from being trivial. First, an inevitable curse of dimensionality, while it can easily provide collisions and indistinguishability over one attribute, it is not so when multiple attributes need to be considered. The problem is complicated by the second aspect, which is the need to guarantee effectiveness of indexes (in terms of the limited overhead caused by spurious tuples returned to the clients due to collisions) and their efficiency (in terms of low performance overhead) for query execution. A third non trivial aspect is the need to limit the storage required at the client for (re)constructing indexes to translate queries on original plaintext data into queries on indexes at the server.

In this chapter, all these problems are addressed by proposing a *multi-dimensional index* (i.e., an index on multiple attributes) that is robust against inference exposure and, at the same time, performs well for query execution and requires limited storage at the client side. The designed multi-dimensional index ensures not only that index values on individual attributes are guaranteed to appear at least a given number of times (i.e., no peculiar frequencies can be exploited for inference attacks) but that the same holds for their combination. In other words, each combination of index values enjoys the property of having at least a given number of occurrences. Besides providing protection against static inference attacks (which can no longer exploit frequencies of index values), the approach illustrated in this chapter guarantees protection after the application host has been compromised, since the encryption keys are never shared with the server. The price to pay for such a protection is the overhead in query execution: being tuples with the same indexes indistinguishable one from the others since any query touching one of them would return all the others as well. It is therefore important to carefully group tuples for indexing so to limit the overhead in query execution and hence guarantee performance. While this can be trivial when only one attribute is to be indexed, it is far from being so (it is an NP-hard problem) when multiple attributes need to be indexed.

The illustrated approach for index construction employs a spatial-based representation of tuples to be outsourced and an algorithm performing recursive cuts on such space, resulting in a partitioning of tuples for indexing. As confirmed by the experimental evaluation, the proposal provides for effective and efficient query evaluation, enjoying limited overhead and limited storage requirements at the client side.

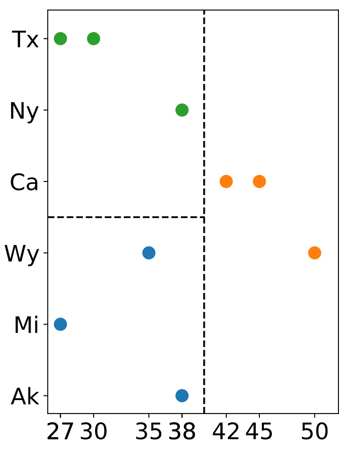
6.2 Basic concepts

This chapter focus on the protection of relational database systems. As anticipated, the protected relation is available in the security boundary of a vulnerable application that may be exploited by an attacker. The relation r is defined over schema $R(a_1, \ldots, a_n)$, where each attribute a_j is defined over a domain $\mathtt{d}(a_j)$, for $j = 1, \ldots, n$. In the following, it is used the notation $\mathtt{val}(a_j)$ to denote the set of values of attribute a_j stored in r (i.e., $\mathtt{val}(a_j)$ = SELECT DISTINCT a_j FROM R). As an example, Figure 6.1(a) illustrates a relation r with three attributes: `Name`, `State`, and `Age`. Here, $\mathtt{d}(\mathtt{Age})=\{0, \ldots, 120\}$ and $\mathtt{val}(\mathtt{Age})=\{27, 30, 35, 38, 42, 45, 50\}$. To protect the confidentiality of data even in the case in which the server application is compromised

r

	Name	State	Age
t_1	Ada	Ak	38
t_2	Bob	Mi	27
t_3	Coy	Wy	35
t_4	Dan	Ca	42
t_5	Eve	Ca	45
t_6	Fay	Wy	50
t_7	Gil	Ny	38
t_8	Hal	Tx	30
t_9	Ian	Tx	27

(a)



(b)

	State	Age
t_1	AkMiWy	[27,38]
t_2	AkMiWy	[27,38]
t_3	AkMiWy	[27,38]
t_4	CaWy	[42,50]
t_5	CaWy	[42,50]
t_6	CaWy	[42,50]
t_7	NyTx	[27,38]
t_8	NyTx	[27,38]
t_9	NyTx	[27,38]

(c)

r^e

	et	$i_{\texttt{State}}$	$i_{\texttt{Age}}$
t_1^e	but6yv	ω	α
t_2^e	lmoe!.	ω	α
t_3^e	p4?llq	ω	α
t_4^e	gbS1.X	γ	β
t_5^e	c493pw	γ	β
t_6^e	WD.23b	γ	β
t_7^e	Q.co43	τ	α'
t_8^e	de31As	τ	α'
t_9^e	xMe1!K	τ	α'

(d)

Map$_{\texttt{Age}}$

Bucket	Count
[27,38]	2
[42,50]	1

(e)

Map$_{\texttt{State}}$

Value	Buckets$^\sigma$
Ak	AkMiWy
Ca	CaWy
Mi	AkMiWy
Ny	NyTx
Tx	NyTx
Wy	AkMiWy, CaWy

(f)

Figure 6.1: Plaintext relation (a), its spatial representation (b), partitioning (c), encrypted and indexed relation (d), and maps for attribute `Age` (e) and `State` (f)

by an attacker, the owner of the data collection encrypts the relation at the tuple level before outsourcing it, using a symmetric encryption scheme with a key that is not shared with the application host. Queries on the encrypted relation are supported via a set of *indexes* associated with a set $\mathcal{I}=\{a_1,\ldots,a_l\}\subseteq R$ of attributes in the original relation on which conditions need to be evaluated in the execution of queries (`State` and `Age` for our running example). An encrypted and indexed relation is formally defined as follows.

Definition 6.2.1 (Encrypted and indexed relation) *Let r be a relation over schema $R(a_1,\ldots,a_n)$, and $\mathcal{I}=\{a_1,\ldots,a_l\}\subseteq R$ be a subset of the attributes in R. The* encrypted and indexed *version of r is a relation r^e over schema $R^e(et,i_1,\ldots,i_l)$ where $\forall t\in r$, $\exists t^e\in r^e$ such that $t^e[et]=E_k(t)$, with E_k a symmetric encryption function with key k, and $t^e[i_j]$ the index value derived from $t[a_j]$, $j=1,\ldots,l$.*

According to this definition, the encrypted and indexed version r^e of relation r has an attribute et, which is the encrypted representation of the tuples in the plaintext relation, and an attribute

i_j, which is the index for attribute a_j in $\mathcal{I}$, $j = 1, \ldots, l$. Figure 6.1(d) illustrates an example of encrypted and indexed version of the relation in Figure 6.1(a), where `State` and `Age` are indexed. For simplicity, in the example Greek letters are used to represent index values.
The goal of the research presented in this chapter, is to compute a multi-dimensional index that is effective and efficient for the execution of queries with support for equality ($=$) and range ($>, \geq, <, \leq$) conditions.

6.3 Multi-dimensional tuple partitioning

The designed approach for partitioning tuples for indexing employs an algorithm similar to the one used by the Mondrian anonymization algorithm [51, 105] explored in Chapter 5. While similar, the algorithm developed in this chapter bears differences to accommodate the fact that in the considered scenario it is needed to cluster tuples to produce obfuscated indexes performing well for query evaluation (in contrast to cluster tuples for semantically meaningful generalization). The partitioning process developed works then in a multi-dimensional space, with one dimension for each indexed attribute, and where tuples correspond to points in the multi-dimensional space where their coordinates correspond to the values of the indexed attributes in the tuples. Figure 6.1(b) shows the two-dimensional representation for the indexing of attributes `State` and `Age` of the relation in Figure 6.1(a). Since more tuples can have the same values for the indexed attributes, a point in the multi-dimensional space can correspond to more than one tuple, which is represented in the figure with the number of occurrences associated with it (this is omitted in the proposed example since it is always equal to 1).
To construct the multi-dimensional space on which the algorithm operates, by partitioning tuples in *boxes* of at least b tuples, attributes to be indexed are classified into two categories:

- *continuous* attributes (e.g., `Age` in Figure 6.1(a)), characterized by a total order relationship on their domain, and on which range conditions need to be supported;
- *nominal* attributes (e.g., `State` in Figure 6.1(a)), which do not have a semantic order in their domain and hence on which only equality conditions make sense.

When partitioning tuples in boxes, care must be taken to put as much tuples as possible that share the same values for an attribute in the same box. Also, for continuous attributes, close values should fall as much as possible in the same space. It is important to note that this objective might intrinsically not be possible for all attributes. Consistently with these observations, values of continuous attributes are considered in their natural order along the axis of their dimension, while values of nominal attributes are considered in increasing order of their relative frequencies in the tuples.

The partitioning process of the multi-dimensional space works recursively, cutting, at each step, a space (e.g., in the first step, the whole space is cut) with respect to a selected attribute and a value in its domain as threshold. The cut divides the space in two sub-spaces, each containing the points (i.e., the tuples) falling on its side of the cut.

The process is recursively repeated on each of the two resulting sub-spaces, and terminates when any further cut would generate a partition with less than b tuples. At each step, the attribute chosen for the cut is the one that, in the considered space, has the maximum *span*. For continuous attributes, the span is the distance between the minimum and maximum value that the tuples in the (sub-)space assume. For nominal attributes, it is the number of distinct values that the tuples in the (sub-)space assume. For instance, with reference to the relation in Figure 6.1(a), the span for attribute `Age` is $50 - 27 = 23$, while the span for attribute `State` is 6. The value chosen as threshold for the cut is the median for continuous attributes, and the value that splits the (sub-)space in two sub-spaces with nearly 50% of the tuples each for nominal attributes. Note that cuts change the relative frequency of values in the generated sub-spaces, and hence also the order in which values for nominal attributes are considered on their axis.

6.4 Index construction

At the end of the partitioning process, the tuples in r are grouped in non-overlapping boxes (i.e., disjoint groups of tuples whose union corresponds to r) such that each box contains at least b tuples. The dotted lines in Figure 6.1(b) denote the cuts performed and hence the resulting boxes for our example. Two cuts have been performed, resulting in three boxes, each containing three tuples.

Intuitively, boxes determine the tuples that will be mapped to the same combination of index values. In other words, for each indexed attribute, all the values that fall in the same box will be mapped to the same index values. Since this applies to all indexed attributes, this implies that all tuples in the same box will be mapped to the same combination of index values. In the following, the notation $\mathcal{B}$ is used to denote the set of all boxes, and $B_j \in \mathcal{B}$ to denote the j-th box. Also, for each box $B \in \mathcal{B}$ and attribute $a \in \mathcal{I}$, it is denoted with $B[a]$ the set of values of a, called *bucket*, covered by B. A bucket is expressed as an interval for continuous attributes and as a set of values for nominal attributes. Formally, for each box B and attribute a:

- $B[a] = [v, v']$ such that $v = \min\{t[a] \mid t \in B\}$ and $v' = \max\{t[a] \mid t \in B\}$, if a is a continuous attribute;

- $B[a] = \{t[a] \mid t \in B\}$, if a is a nominal attribute.

Figure 6.1(c) reports the buckets of the three boxes corresponding to the partitioning in Figure 6.1(b). For readability, each set is represent as a string composed of all its elements; for instance, AkMiWy stands for set $\{$Ak,Mi,Wy$\}$.

Note that, different boxes might be associated with the same bucket for one or more of their attributes. Formally, it is possible to have $B_x[a]$ = $B_y[a]$, with $x \neq y$. For instance, in the proposed example, box B_1 containing the first three tuples and box B_3 containing the last three tuples have $B_1[\texttt{Age}]$ = $B_3[\texttt{Age}]$ = [27,38].

Since, for each attribute, index values in different boxes must be different, the generation of indexes cannot depend only on the bucket. To map the same bucket of different boxes to different index values, the procedure designed combines buckets for their indexing with a salt. This is formalized by the following definition.

Definition 6.4.1 (Index function) *Let r be a relation, $a \in \mathcal{I}$ be an indexed attribute, $\mathcal{B}$ be the set of boxes of relation r, and h_k be a cryptographic hash function with key k. An* index function *for attribute a is a function ι_a:$\mathcal{B} \rightarrow I_a$ such that:*

- *$\forall B \in \mathcal{B}$, $\iota_a(B)$=$h_k(B[a]||\sigma)$, with σ a randomly generated salt;*
- *$\forall a, a' \in \mathcal{I}$, $\forall B, B' \in \mathcal{B}$, with $\iota_a(B)$=$h_k(B[a]||\sigma)$, $\iota_{a'}(B')$=$h_k(B'[a']||\sigma')$, if $B[a] = B'[a']$ and ($a \neq a'$ or $B \neq B'$), then $\sigma \neq \sigma'$.*

Indexes are then computed as the result of a cryptographic hash function on the concatenation of the bucket to be indexed and a salt. The second bullet in the definition dictates the use of a different salt for buckets that are equal but refer to different attributes (i.e., dimensions) or for a same bucket that appears in different boxes for the same attribute. Satisfaction of such a condition is guaranteed by generating salts using a pseudo-random generation function with a different seed for each attribute and using a different salt in the sequence for different occurrences of a same bucket. Hence, different attributes will be associated with a different sequence of randomly generated salts. For each attribute $a \in \mathcal{I}$, we denote with $\sigma_a(j)$ the j-th salt generated by function σ with the seed of attribute a. Each bucket $B_x[a]$ is then associated with the j-th salt $\sigma_a(j)$, with $j - 1$ the number of boxes $B_y \in \mathcal{B}$ such that $B_x[a]$=$B_y[a]$ and $y < x$. For instance, with reference to the example in Figure 6.1, bucket $B_3[\texttt{Age}]$ is combined with salt $\sigma_{\texttt{Age}}(2)$ since $B_1[\texttt{Age}]$=$B_3[\texttt{Age}]$ and $1 < 3$. Figure 6.1(d) shows the encrypted and indexed version of the plaintext relation in Figure 6.1(a). Here, combinations of index values $\langle\omega, \alpha\rangle$, $\langle\gamma, \beta\rangle$, and $\langle\tau, \alpha'\rangle$ are those computed for the three boxes in Figure 6.1(c). Indexes α and α' represent different salted versions of the same bucket (i.e., [27,38]).

6.5 Client-side maps

The process illustrated in the previous sections of this chapter enables the creation of indexes to be associated with the tuples in the plaintext relation that has to be protected on a vulnerable host. The encrypted and indexed relation (Def. 6.2.1) will then have, for each tuple in the original plaintext relation, its encrypted version and the values of the indexes computed as illustrated (Def. 6.4.1). For simplicity, in the proposed examples tuples are maintained in the same order in both the original and outsourced relations; it is worth noting that tuples should be shuffled before upload them to the storage provider.
The next problem that has to be addressed is the definition of the information to be stored at the client side to enable the translation of queries on the plaintext relation into queries on the encrypted and indexed relation that can then be executed by the application host without the need of sharing any decryption key. According to Def. 6.4.1, such information comprises:

- the cryptographic hash function h along with the corresponding key k;
- the function σ_a used for salt generation;
- a map, denoted Map$_a$, enabling the translation of plaintext attribute values into index values.

While the first two bullets only require a client to memorize one function, the third one, requires the creation of a map for the attributes, thus it needs more consideration. Being maps stored client side, it is important to maintain them compact, to limit the storage needed at the client. In Section 6.7, it will be possible to notice that the implemented maps enjoy such compactness. The attributes' maps are maintained in a compact form, compliant to what is defined in the following.
Continuous attributes. For each continuous attribute a, Map$_a$ is a set of pairs (`Bucket`,`Count`), reporting the buckets in which the attribute has been divided and, for each bucket, the number of boxes in which it appears. Formally, Map$_a$ = $\{\langle B[a], c\rangle \mid B \in \mathcal{B}, c = |\{B' \in \mathcal{B} \mid B'[a] = B[a]\}|\}$.
The counter associated with each bucket gives the number of distinct index values corresponding to the bucket and hence the number of salts to be used for reconstructing such indexes for query translation. Figure 6.1(e) illustrates the map for attribute `Age` that contains the information about the two buckets resulting from partitioning (i.e., [27,38] and [42,50]). Bucket [27,38] has 2 occurrences and the sequence of salts used in the generation of the corresponding index values is $\sigma_{\texttt{Age}}(1)$ and $\sigma_{\texttt{Age}}(2)$. Hence, the index values corresponding to bucket [27,38] are h_k([27,38]$||\sigma_{\texttt{Age}}(1)$)=$\alpha$ and h_k([27,38]$||\sigma_{\texttt{Age}}(2)$)=$\alpha'$ (see Figure 6.1(d)).

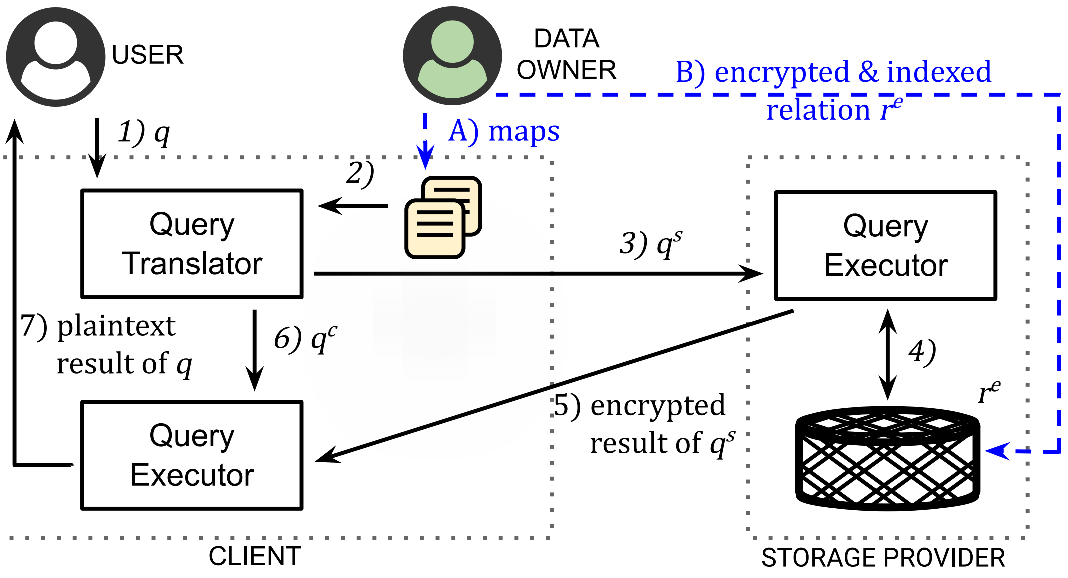


Figure 6.2: Query execution process

Nominal attributes. For each nominal attribute a, Map$_a$ is a set of pairs (`Value`, `Buckets`$^\sigma$) for each value v in a, the buckets $B[a]$ that include v, concatenated with the corresponding salt value. Formally, Map$_a$ = $\{\langle v, \{(B[a] \mid\mid \sigma)\}\rangle \mid v \in \mathtt{val}(a), B \in \mathcal{B} : v \in B[a], \iota_a(B) = h_k(B[a] \mid\mid \sigma)\}$.
Figure 6.1(f) illustrates the map for attribute `State`. For simplicity, in the figure, noting that in the proposed example all buckets have only one occurrence, and therefore only one salt is to be used, the unsalted bucket values is reported.

6.6 Query translation and execution

Once the (encrypted and indexed) relation is stored in the web application host, only the information needed for query translation (i.e., the maps) will be stored client side. Each query q on R, formulated at the client side, will then need to be translated into a query q^s operating on indexes at server side. The retrieved result (encrypted tuples whose indexes satisfy q^s) will be decrypted and query q^c will be executed to eliminate possible spurious tuples, where q^c is the same as q but executed on the decryption of the result of q^s instead of R. Figure 6.2 illustrates the overall architecture and query execution process.
Translating q into q^s requires mapping each of the conditions appearing in its WHERE clause into a condition on indexes. In the following it is illustrated such a mapping depending on whether the condition is on a continuous or nominal attribute.

Continuous attribute. Continuous attributes support both *point* (i.e., equality) as well as *range* conditions. Conditions can then be of the form “a *op* v” or “a BETWEEN v_x AND v_y”, with $a \in \mathcal{I}$,

v,v_x,$v_y$$\in$d($a$), and *op* $\in \{>, \geq, <, \leq\}$.
To illustrate the mapping, it is taken into consideration a general form capturing all the cases above and illustrate the mapping of condition "$a$$\in$Range", where Range is an (open or closed) interval specified by two values v_l and v_r, with $v_l \leq v_r$. Such a general form captures all the conditions above, by simply considering $v_l = v_r = v$ for point conditions; v_l the lowest value in val(a) for $<$ or $\leq$ conditions; v_r the highest value in val(a) for $>$ or $\geq$ conditions; and the interval open on the left (right, resp.) side if values equal to v_l (v_r, resp.) should be excluded. For instance, Age$<$30, is equivalent to Age$\in$[27,30).
A condition of the form "$a$$\in$Range" is translated into a condition on a's index i_a, requesting it to be in the set of index values to which the a's buckets intersecting Range have been mapped, that is, in condition:

- i_a IN ($h_k(b||\sigma_a(j))$ s.t. $\langle b, c\rangle \in$ Map$_a$, $b \cap$ Range $\neq \emptyset$, $j = 1, \ldots, c$)

Here $\langle b, c\rangle \in$ Map$_a$ denotes the different buckets in Map$_a$, $b \cap$ Range $\neq \emptyset$ restricts the consideration to the ones intersecting Range, and c expresses the number of salts to be used for each bucket b (i.e., the number of distinct index values to which the bucket has been mapped).
For instance, consider attribute Age and its map in Figure 6.1(e). Condition "Age=30" is translated to "$i_{\texttt{Age}}$ IN (α, α')", with α=h_k([27,38]$||\sigma_{\texttt{Age}}$(1)) and α'=h_k([27,38]$||\sigma_{\texttt{Age}}$(2)). Condition "Age$>$39" is translated to "$i_{\texttt{Age}}$ IN (β)", with β=h_k([42,50]$||\sigma_{\texttt{Age}}$(1)).

Nominal attribute. Nominal attributes support the evaluation of point conditions only. A condition"a=v" is translated into a condition on a's index i_a, requesting it to be in the set of index values to which v has been mapped, that is, in condition:

- i_a IN($\{h_k(set_j)$ s.t. $m$$\in$Map$_a$, with m[Value]=v and $set_j \in m$[Buckets$^\sigma$], $j = 1, \ldots, |m$[Buckets$^\sigma$]$|\}$)

For instance, consider attribute State and its map in Figure 6.1(f). Condition "State='Wy'" is translated to "$i_{\texttt{State}}$ IN (ω, γ)", with ω=h_k(AkMiWy) and γ=h_k(CaWy). Condition "State='Ca'" is translated to "$i_{\texttt{State}}$ IN (γ)".

6.7 Implementation and experiments

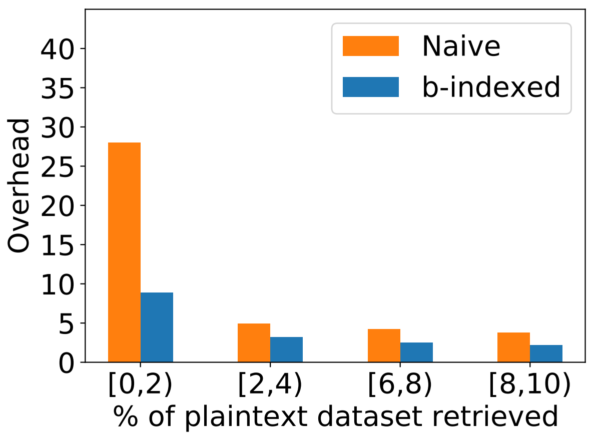
In order to perform the evaluation described in this Section, it has been built a prototype in Python that implements both the generation of the encrypted and indexed relation and the query evaluation process. The code prototype is available at https://github.com/unibg-seclab/flat-index. The tests have been perfomed on a sample of the PUMS USA ACS 2019 dataset containing 3.2M tuples [146]. The dataset schema includes two nominal attributes, namely (`State` and `Occupation`) and two continuous ones: (`Age` and `Income`). The experiments analyze the overhead in query execution and the size of the client-side maps.

Indexing and encryption. In order to realize a distributed version of the partitioning process illustrated in Section 6.3, it has been employed an architecture similar to the one illustrated in Chapter 5. Thank to this decision, the tool implemented uses a scalable Apache Spark platform and parallelizes the indexing process relying on an arbitrary number of workers. Partitions of the dataset are assigned to the workers, which independently apply the indexing process on the tuples assigned to them. Each worker computes index values (Def. 6.4.1) using the Blake2b [19] hash function and a different 16-byte salt for each attribute. The encrypted tuple (attribute *et*) is computed using XSalsa20 with Poly1305 MAC (to guarantee both confidentiality and integrity) with a 32-byte high-entropy key. The tool also generates a fresh nonce for each encryption invocation to provide indistinguishability of tuples with the same values for all encrypted attributes. The encryption functions are implemented by *PyNacl*. The encrypted and indexed relations generated by each worker are uploaded, in randomized order, to a containerized PostgreSQL DBMS.

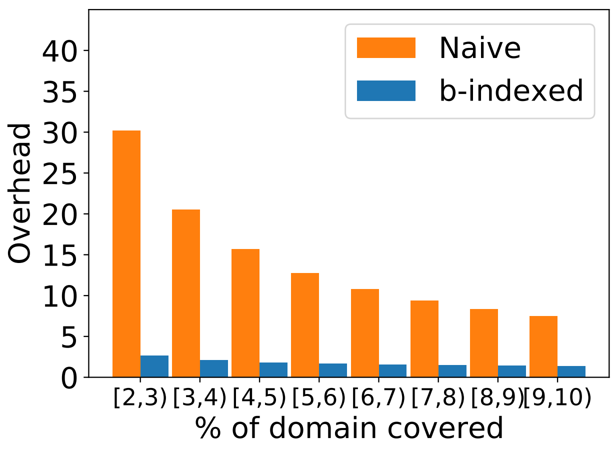
Query translation. Each client side query q is parsed using *sqlparse*, a SQL parser for Python, and translated as described in Section 6.6. Query q^s is submitted to the PostgreSQL DBMS hosted at the storage provider, and its result is decrypted and checked for integrity by the client. The client executes query q^c on an in-memory SQLite DB to filter spurious tuples and project the attributes of interest.

Query overhead. The solution described in this chapter has been compared with a naive approach that builds boxes of b tuples, each by ordering tuples according to the values of a sequence of attributes and then splitting the ordered dataset in boxes of b contiguous tuples. Two kinds of query have been run for the evaluation: *1)* point queries for each attribute a in the dataset schema (i.e., `State`, `Occupation`, `Age`, `Income`), and each value v in `val`(a); *2)* range queries for attribute `Age` and for each range [v_i,v_j] of values in `val(Age)`. Figure 6.3 compares the overhead in query execution for the naive approach and for the proposal prototype, called b-indexed, approach. Figures 6.3(a,c,e) refer to point queries and Figures 6.3(b,d,f) refer to range queries. The overhead is measured in terms of the average ratio between the number of tuples returned by the query on index i_a and the original query on a, considering b with values 10, 25, and 50. For point queries, such an overhead is measured depending on the selectivity (%age of tuples returned) of the original query on the plaintext data (plotted on the x axis). It is assumed that each query to be issued as many times as the frequency of the requested value. For range queries, Figures 6.3(b,d,f), the overhead is measured depending on the percentage of domain values covered by the range condition (plotted on the x axis).

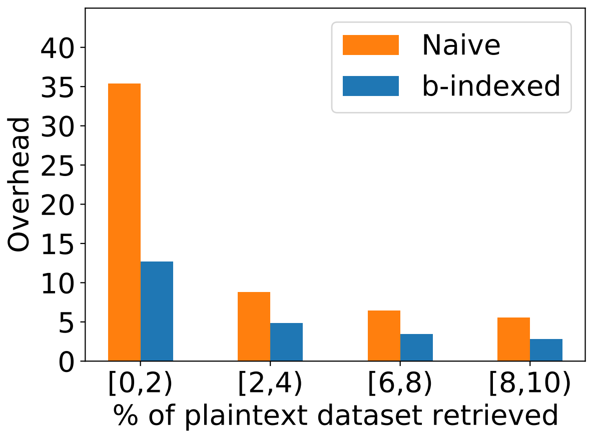
As visible from the figures, the proposed approach largely outperforms the naive one, whose overhead compared is between 1.5x and 3x larger for point queries and between 5x and 10x for



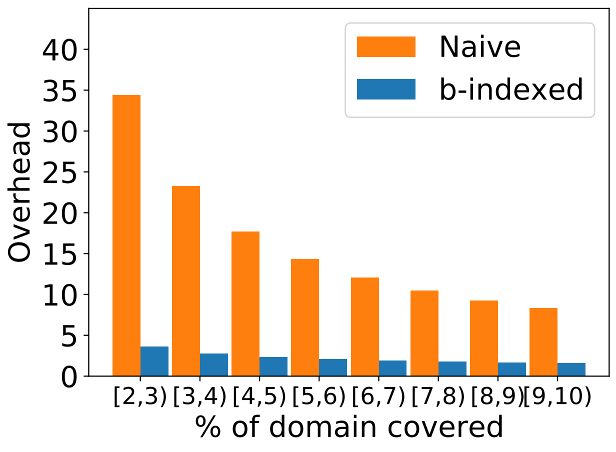
(a) b=10



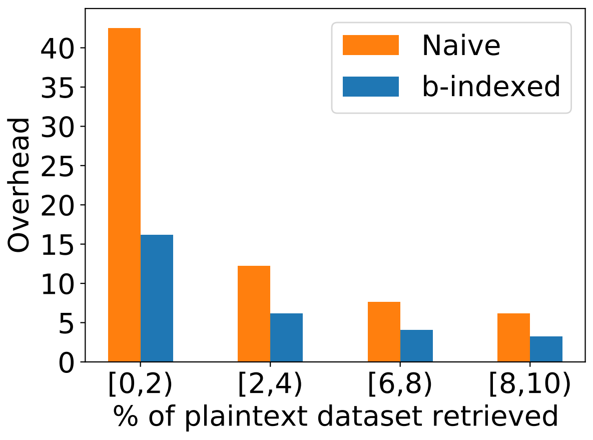
(b) b=10



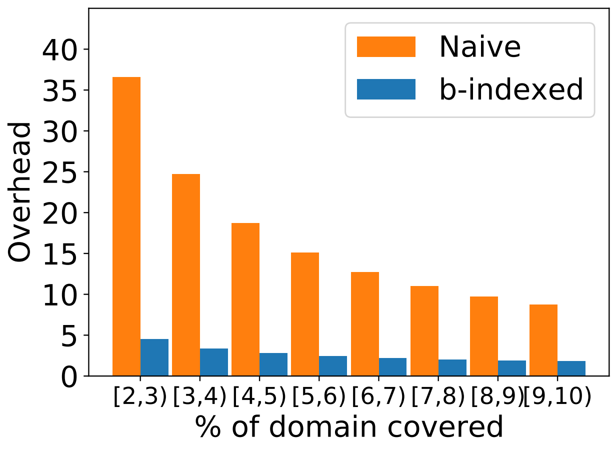
(c) b=25



(d) b=25



(e) b=50



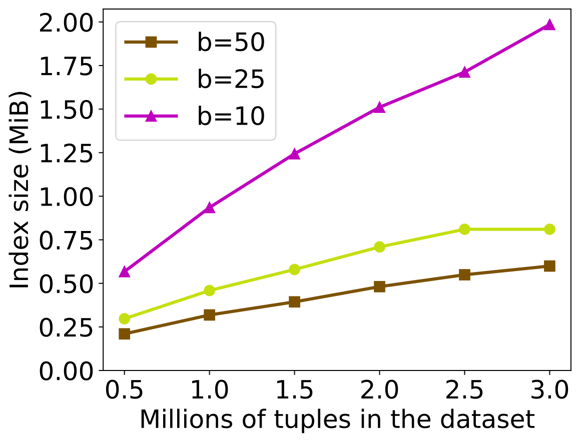
(f) b=50

Figure 6.3: Point (a,c,e) and range (b,d,f) queries overhead

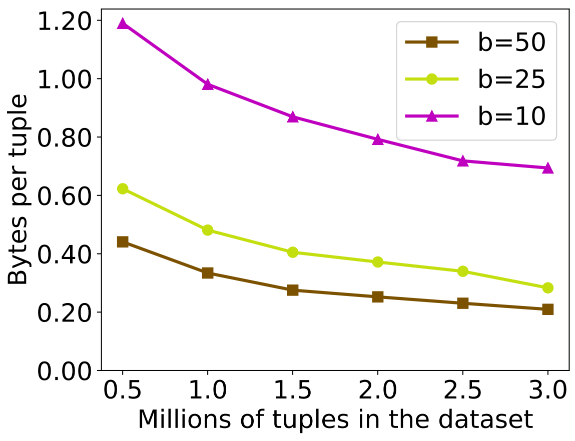
range queries.

It is interesting to note the limited overhead provided by the b-indexed approach, which is much smaller than the value of b.

Local data structure. Figure 6.4(a) illustrates the size of the maps stored at the client side,



(a)



(b)

Figure 6.4: Absolute (a) and relative (b) size of the maps

varying the number of tuples in the dataset between 0.5 and 3 millions and b equal to 10, 25, and 50. As visible from Figure 6.4(b), the implemented maps require in almost all the analyzed configurations less than 1 byte per tuple. Note that the storage space per tuple required by the map decreases as the number of tuples increases.

6.8 Related work

Several research efforts have addressed the problem of supporting queries on encrypted data stored on a remote host through the definition of indexing techniques or specific cryptographic schemes (e.g., [46, 87, 139, 140, 179]). The definition of efficient solutions also depends on the specific queries to be supported. In particular, the support of range queries requires to define techniques that consider the order relationship characterizing the domain of the attributes on which queries have to be executed, which can complicate the definition of such techniques (e.g., [46, 56, 172]). The approach illustrated in this chapter share the goal of these solutions proposed by the research community: supporting queries over encrypted data; in the considered scenario, the shortcoming of these works is that they operate on a single attribute. The described approach instead is based on a multi-dimensional interpretation of the dataset that allows the definition of indexes over multiple attributes. The problem of indexing multi-dimensional datasets has been already considered and resulted in the definition of multi-dimensional indexes for supporting queries with conditions on multiple attributes (e.g., [178]). These solutions, however, differ from the one just described since they define one index only for the whole set of attributes/dimensions considered. Differently, the techniques illustrated defines a multi-dimensional index with a component for each attribute, considering the intrinsic multi-dimensional nature of relations.

A line of work close to what just presented is represented by approaches aimed at supporting query evaluation over data organized in multiple relations and/or fragments that cannot be joined by non-authorized subjects (e.g., [44, 52, 185]). These solutions reduce the precision of join operations introducing a degree k of uncertainty (i.e., it is never possible to reconstruct a tuple in the join result with uncertainty lower than $1/k$). Protection is obtained by grouping tuples in the relations/fragments and performing joins at the group (in contrast to tuple) level.

6.9 Conclusions

This Chapter have addressed the problem of allowing clients to query encrypted data that are stored server side in a vulnerable host. The approach illustrated create indexes that are able to work on a relation composed of multiple attributes. This guarantees protection against attacker that are able to recover the data collection reachable through an exploited application, while providing effective and efficient query execution, with support for both point and range conditions. The experimental evaluation of the approach has been performed on a large publicly available dataset and confirms the validity of the proposal and its applicability in practical scenarios.

Chapter 7. Conclusions

This thesis presents novel techniques to leverage Linux Security Modules to enforce security boundaries in applications that leverages components written in unsafe languages. Since the enforced protection follows the least-privilege principle, this thesis also proposes techniques to further protects data collections that are still reachable in case of a succefull exploit.

Concerning the protection of JavaScript applications leveraging unsafe utilities under the form of subprocesses, it has been described Cage4Deno, a set of extensions for the Deno JavaScript runtime. This set of extensions is able to create a sandbox for subprocesses spawned thorugh Deno by leveraging Landlock and eBPF LSM. For Cage4Deno, the security boundaries enforced concern file system resources. Since the proposal has as main objective usability, this thesis has described a simple policy language for Cage4Deno that realizes a simple `RWX+D` permission system. In addition to that, it has been also proposed an approach to automate the creation of policies for Cage4Deno. Finally, the experimental evaluation demonstrates that Cage4Deno is able to protect the system against attacks leveraging recent CVEs while also suffering a performance overhead that is much smaller compared to alternative solutions.

After this initial research effort, techniques similar to the ones used to realize Cage4Deno have been employed to protect application run through WebAssembly runtimes. In particular, the thesis proposes a method to strengthen the security guarantees of runtimes that implements the WebAssembly System Interface. In this case, only eBPF programs have been employed to sandbox the execution of code that interact with the host file system. Again, the performed performance evaluation has shown that the proposed approach introduces limited overhead compared to a scenario in which no protection is applied.

Then, it has been explored how to extend the protection applied to the file system also to other system resources (i.e., IPC and network). NatiSand realizes this objective by leveraging Landlock, eBPF LSM and Seccomp BPF to create a sandbox that protects all the aforementioned resources and can be applied to both subprocesses and native libraries. Similarly to Cage4Deno, the proposal is capable of protecting vulnerable hosts compromised through recent CVEs while being affected by an acceptable overhead.

In the final part of the thesis, it has been taken into account the protection of data collections that are inside the security boundaries defined through the techniques treated in the previous chapters. In particular, two methods have been proposed to address security concerns over these resources: the first one protects the privacy of data points, the second ensures confidentiality. Both methods achieve the objective of providing security guarantees while suffering low performance overhead.

7.1 Future Work

This section concludes the thesis with a discussion on the future works that can be done by grouping the research done in three areas: *creation of fine-grained sandboxes*, *enforcing privacy guarantees* and *enabling operations on encrypted data*.

Creation of fine-grained sandboxes –Chapter 2 and 4 illustrated the usage of modern LSMs to create fine-grained sandboxes for native components leveraged by JavaScript runtimes. While the architecture described in this thesis is versatile, as shown by similar works that extend Node.js [36], an evaluation on the use of these techniques to protect Node native addons(e.g., Node-API) would be of great interest. In addition to that, Chapter 3 has shown the flexibility of the approach outside of JavaScript runtimes, thus would be interesting to explore how to enable similar protection in other environments such as Android [10]. Finally, another interesting future work would be the integration with memory protection techniques, in order to cover a wider range of use cases.

Enforcing privacy guarantees –Chapter 5 explored how to enforce privacy guarantees on data collections that are reachable by vulnerable applications. During the chapter it has been presented a scalable approach to apply sanitization to large collections of sensors data. As a future work, it would be interesting to extend the set of criteria used to perform the cut and determine the partitions, for instance by considering the usage of K-means clustering [18]. From a technical perspective, the support of Kubernetes as an orchestrator to ease the deployment of the solution on popular cloud providers would open the possibility of investigating the impact of a well designed mechanism to position workers on nodes.

Enabling operation on encrypted data –Chapter 6 detailed an approach to support the execution of point and range queries over encrypted data. As a future work, it would be interesting to explore how to enable additional operations on data that is not strictly a relational table [6, 22]. Among the future directions considered concerning this topic, implementing a block rotation strategy to support the insertion, deletion and update of records would be an interesting addition.

Chapter A. Cage4Deno BPF implementation details

In the following it is detailed, as a complement, some of the issues that has been found while working with the current implementation of BPF, explaining how they have been addressed.

A.1 Hashing & collision handling

As anticipated in the design and implementation of `deny` rules (Section 2.4.3), at the time of writing, the implementation of the BPF framework does not provide support for map-in-map structures [114]. Moreover, support for using string keys in a map is still limited [85]. At the time of writing, a patch for providing this option, along with efficient ternary search support to lookup the map [84], are under active development on `bpf-next`, the branch dedicated to the features and improvements that should eventually land in BPF. Nonetheless, it has been necessary to cope with the temporary limitation of using an integer key to lookup the Map_{policy}, hence, during the research it has been developed a method 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, it has been 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 $i-1$. The function that has been employed is *djb2* [95], a non-cryptographic low-complexity incremental hash function proposed by D. J. Bernstein.

The use of a hashing function also required 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, in here it is illustrated the following example. Assume to have three deny rules in the policy: `/home/user/data`, `/home/user/lib` and `/media`. Also, assume that the djb2 hash of `/home/user/lib` and `/media` collide. Instead of using the deny rules to build a prefix tree, it is possible to 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 lookups. In case no lookup is successful, the verifier concludes that no deny rule was hit, hence the access request is granted. Conversely, the

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:      $task \leftarrow bpf\_get\_current\_task\_btf()$
3:      if $\neg task \in Map_{task}$ then
4:          return 0                                          ▷ Grant request
5:      end if
6:      $Map_{policy} \leftarrow Map_{task}[task]$
7:      for each $prefix$ in $S_r$ do
8:          if $djb2(prefix) \in Map_{policy}$ then
9:              $collision\_chain \leftarrow Map_{policy}[djb2(prefix)]$
10:             for $D_r$ in $collision\_chain$ do
11:                 if $isPrefix(D_r, prefix)$ then
12:                     return -EPERM                         ▷ Deny request
13:                 end if
14:             end for
15:         end if
16:     end for
17:     return 0                                              ▷ Grant request
18: end procedure
```

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 [95], the results presented in the Experimental Evaluation performed showed a limited impact associated with the presence of collisions (see Section 2.6). 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 2.4.3 and A.1 it has been illustrated the 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 it has been opted for the `TASK_STORAGE` map type, which permits to be modified calling the following BPF helpers: `bpf_task_storage_get()` and `bpf_task_storage_delete()`. 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 in this case the map type employed is the `HASH` 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, each access path outside is stored outside of the BPF stack, through 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).

Chapter B. Policy files samples

This appendix collects the full extract of policy files used in Chapter 2 and 4

B.1 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 2.4.

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"
      ]
    },
}
```

B.2 curl policy

Listing B.2 reports the policy associated with the execution of the `curl` utility to reach the URL: `https://www.example.com`. The policy has been automatically generated using the approach described in Section 4.5.2.

Listing B.2: Example of policy associated with `curl`

```
[{
  "name": "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]
  }]
}]
```

Chapter C. Distributed Mondrian experiments

C.1 Introduction

This appendix describes the contents of the repositoty publicly available at `https://github.com/mosaicrown/mondrian`, that implements the proposal described in Chapter 5, and how to reproduce the experimental results on a small dataset.

C.2 Hardware and software requirements

Hardware requirements. The deployment of the prototype requires a machine having:

- a CPU with at least one logical core for each worker;
- at least 2 GB of RAM for each worker.

Software requirements. The deployment of the prototype requires a machine with Linux operating system (the experimental results have been obtained using Ubuntu 20.04 LTS) and the following packages installed:

- *make*, version 4.3;
- *git*, version 2.27.0;
- *zip*, version 3.0, and *gzip*, version 1.10-2;
- *python3*, version 3.8.6;
- *python3-venv*, version 3.8.6;
- *gnuplot*, version 5.2 patchlevel 8.

When these packages are available, the environment set up should be finalized through the following steps:

1. install and set up *docker* and *docker-compose* (for more details on this step, see Section "Prerequisites" in the provided repository);
2. run `sudo usermod -aG docker <USER>`;
3. reboot the system;

4. check that the following commands run without root privileges:
   `docker run hello-world`
   `docker-compose -version`

C.3 Deployment of the prototype

The steps for deploying the prototype are the following:

1. clone the repository through command

   ```
   git clone --depth 1 --branch \
   percom2021_artifact \
   https://github.com/mosaicrown/mondrian.git
   ```

2. run `make` to verify that all the software requirements illustrated in Section C.2, which are needed for the distributed (Spark-based) version of the algorithm, are satisfied by the environment;

3. run `make start` to pull and build a copy of the Docker images necessary to the prototype.

The prototype uses the following Docker containers:

- *Hadoop Namenode* at `http://localhost:9870`
  (the web page available at this url permits to check the status of the *Hadoop Datanode* and to browse the distributed file system);

- *Hadoop Datanode* at `http://localhost:9864`;

- *Spark History Server* at `http://localhost:18080`;

- *Spark Cluster Manager* (and thus *Spark worker*s) at
  `http://localhost:8080`.

C.4 Use of the prototype

The prototype implements the centralized and the distributed version of the Mondrian algorithm. The prototype is complemented with a web UI that can be deployed running command `make ui`. The web UI is available at `http://localhost:5000` and can be used to run customized experiments. A complete user guide to the web UI is available in the provided repository. In the following, it is described how to use the prototype to reproduce the experimental results presented on a small dataset.

IPUMS USA dataset. The experiments run by following the contents of this appendix use a sample from the IPUMS USA dataset [147]. The dataset is available at `https://ipums.org/`, together with a detailed guide for its download. With the steps detailed in the following, it is possible to obtain a data sample that can be anonymized through a one-click command implemented in the repository. As a first step, go to IPUMS website `https://usa.ipums.org/usa/` and click on "Get Data". Then, select the attributes of interest (harmonized variables `State FIP Code`, `Age`, `Education Number`, `Occupation`, and `Income` in the case of the automated procedure) and add them to the cart. For convenience, it is possible to use the direct links at `https://github.com/mosaicrown/mondrian#usa-2018-dataset` (each variable name is a link that redirects to the page at ipums.org that permits to add the variable to the cart). Select the sample of interest (among USA samples, *2018 ACS* in the case of the automated procedure) and create the data extract. To customize the sample size, set parameter *Persons* (to run the automated procedure *Persons*, set this parameter to 510 in order to obtain a dataset with at least 500,000 tuples). Among the formats available for downloading the dataset, select the CSV format and save the downloaded gzip archive in the root folder of the project, with name *usa_<extract_number>.csv.gz*.
Note that the sample of the dataset is randomly extracted at each download from the IPUMS USA web site. Hence, results may differ from the different experimental evaluations.

Prototype execution. The repository implements a procedure to run the experiments in an automated way and can be started running the command `make artifact_experiments` from the root folder of the project. The procedure operates as follows:

1. it cleans the test environment stopping every Docker container that is still running and removing from HDFS the results produced by the previous runs;
2. it extracts the sample of IPUMS USA dataset to be anonymized from the archive and copies it to the *Spark Driver* volume;
3. it runs the centralized and distributed version of the Mondrian algorithm (see below), and measures the execution time and information loss, storing the results with the following directory structure:
```
mondrian/
 |-- percom_artifact_experiments/
 |-- |-- results/
 |-- |-- |-- runtime_results_<TIMESTAMP>/
 |-- |-- |-- loss_results_<TIMESTAMP>/
```

4. it shuts down all the containers except the *Spark History Server*, which remains available to keep track of the previous runs of the prototype.

Centralized version. The centralized version of Mondrian corresponds to the baseline of the experimental results illustrated in Chapter 5. The execution of the algorithm can be monitored through the messages showed on the terminal, which report:

1. the schema and the first few tuples of the input dataset;
2. each decision taken by Mondrian to cut the dataset;
3. the schema and the first few tuples of the anonymized dataset;
4. a summary of the information loss measures and the execution time of the algorithm.

The anonymized dataset is in folder *local/anonymized*.

Distributed version. Given a number n of workers available in the distributed system, the prototype performs the following steps to execute the distributed version of the Mondrian algorithm:

1. start all the Docker services, initialize HDFS, and submit to the *Spark Driver* Spark Application implementing the distributed version of Mondrian;
2. recover the dataset from HDFS and show its structure;
3. retrieve the n-quantiles of the best-scoring attribute of the dataset, showing the score used to decide the optimal cut and the size of the partitions;
4. show the first few tuples of the dataset, complemented with a new attribute containing the id of the quantile to which each tuple belongs and hence the worker to which the tuple is assigned;
5. anonymize the dataset;
6. show the first few tuples of the anonymized dataset, with a summary of the execution time.

The anonymized dataset is in folder *distributed/anonymized*.

C.5 Experimental results produced

Execution time. This experiment measured the execution time when computing a 3-anonymous and 2-diverse version of a sample of the IPUMS USA dataset. The results of the experiments are stored in folder `runtime_results_<TIMESTAMP>`. First, the implemented prototype runs the centralized version of the Mondrian algorithm. The results are saved in file *centralized_results.csv*. Then, the prototype runs the distributed (Spark-based) version of the Mondrian algorithm, varying the number of workers from 2 to 20. The results are saved in file *spark_based_results.csv*. Besides generating the .csv files with the execution time of the centralized and distributed versions of the algorithm, the prototype plots these results generating file *comparison.pdf*.

Information loss. This experiment measured the information loss when computing a 5-anonymous and 2-diverse version of a sample of the IPUMS USA dataset. The results of the experiments are stored in folder `loss_results_<TIMESTAMP>`. The prototype first runs the centralized version of the Mondrian algorithm, storing the results in file *centralized_results.csv*. Then, it runs the distributed version (with 5, 10, and 20 workers), using a sample including 0.01% of the dataset to determine the most suitable attribute and compute the n-quantiles (with $n = 5$, $n = 10$, and $n = 20$, respectively) for partitioning the dataset among the workers. The results obtained from five runs of the distributed version of the algorithm are stored in file *spark_based_results.csv*. The prototype also generates file *loss_table.csv*, which reports the average and the variance (in the form $\mu \pm \sigma$) of the results in file *spark_based_results.csv*.

References

[1] Census regions and divisions of the united states, 2020.

[2] Cli options - wasmtime, 2023.

[3] wasmedgec aot compiler - wasmedge, 2023.

[4] A. Starovoitov. CAP_BPF, 2020.

[5] 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.

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

[7] 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 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 1028–1030, New York, NY, USA, 2023. Association for Computing Machinery.

[8] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Cage4deno: A fine-grained sandbox for deno subprocesses. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 149–162, New York, NY, USA, 2023. Association for Computing Machinery.

[9] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Natisand: Native code sandboxing for javascript runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023.

[10] Y. Agman and D. Hendler. Bpfroid: robust real time android malware detection framework. *arXiv preprint arXiv:2105.14344*, 2021.

[11] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action platforms. In *USENIX Security*, 2021.

[12] B. Alliance. WAMR Runtime, 2023.

[13] B. Alliance. Wasmtime Runtime, 2023.

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

[15] S. Antonatos, S. Braghin, N. Holohan, Y. Gkoufas, and P. Mac Aonghusa. Prima: an end-to-end framework for privacy at scale. In *Proc. of ICDE 2018*, Paris, France, April 2018.

[16] Apache. JMeter, 2022.

[17] F. Ashkouti, K. Khamforoosh, and A. Sheikhahmadi. DI-Mondrian: Distributed improved Mondrian for satisfaction of the ℓ-diversity privacy model using Apache Spark. *Information Sciences*, 546:1–24, 2021.

[18] F. Ashkouti, K. Khamforoosh, A. Sheikhahmadi, and H. Khamfroush. DHkmeans-ℓ-diversity: distributed hierarchical k-means for satisfaction of the ℓ-diversity privacy model using Apache Spark. *The Journal of Supercomputing*, 78(6):2616–2650, 2021.

[19] J. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.

[20] T. C. Authors. Cilium, 2023.

[21] T. F. Authors. Falco, 2022.

[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 16th International Conference on Availability, Reliability and Security*, ARES '21, New York, NY, USA, 2021. Association for Computing Machinery.

[23] D. Bakker. wasi-sockets, 2023.

[24] R. J. Bayardo and R. Agrawal. Data privacy through optimal k-anonymization. In *Proc. of ICDE 2005*, Tokoyo, Japan, April 2005.

[25] S. U. Bazai, J. Jang-Jaccard, and H. Alavizadeh. A novel hybrid approach for multi-dimensional data anonymization for Apache Spark. *ACM TOPS*, 25(1):5:1–5:25, 2021.

[26] M. Bélair, S. Laniepce, and J. Menaud. Snappy: Programmable kernel-level policies for containers. In *SAC*, 2021.

[27] A. Berman, V. Bourassa, and E. Selberg. Tron: Process-specific file protection for the unix operating system. In *USENIX ATC*, 1995.

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

[29] J. Bosamiya, W. S. Lim, and B. Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *USENIX Security*, 2022.

[30] 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 *Proceeding of the USENIX Security Symposium (USENIX Security)*, 2018.

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

[32] C. Canella, M. Werner, D. Gruss, and M. Schwarz. Automating Seccomp Filter Generation for Linux Applications. In *CCSW*, 2021.

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

[34] R. Cattral and F. Oppacher. Poker Hand Data Set, UCI machine learning repository, 2007. https://archive.ics.uci.edu/ml/datasets/Poker+Hand.

[35] A. Chakravorty, T. W. Wlodarczyk, and C. Rong. A scalable k-anonymization solution for preserving privacy in an aging-in-place welfare intercloud. In *Proc. of IC2E 2014*, Boston, MA, USA, March 2014.

[36] 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 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 429–442, New York, NY, USA, 2023. Association for Computing Machinery.

[37] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati. k-Anonymity. In T. Yu and S. Jajodia, editors, *Secure Data Management in Decentralized Systems*. Springer-Verlag, 2007.

[38] CNCF. WasmEdge Runtime, 2023.

[39] B. Coenen. feat(wasi): add rename for a directory + fix remove_dir, 2021.

[40] containers. Bubblewrap, 2022.

[41] J. Corbet. BPF: the universal in-kernel virtual machine, 2014.

[42] J. Corbet. KRSI, 2019.

[43] T. coreutils Authors. uutils coreutils, 2023.

[44] G. Cormode, D. Srivastava, T. Yu, and Q. Zhang. Anonymizing bipartite graph data using safe groupings. *PVLDB*, 1(1), Aug. 2008.

[45] CVE Mitre. Gitlab Exiftool vulnerability, 2021.

[46] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of ACM CCS*, Washington, DC, USA, Oct. 2003.

[47] P. David. hyperfine, 3 2023.

[48] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Livraga, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Scalable distributed data anonymization for large datasets. volume 9, pages 818–831, 2023.

[49] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Artifact: Scalable distributed data anonymization. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 450–451, 2021.

[50] 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 *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021.

[51] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Scalable distributed data anonymization. In *Proc. of IEEE PerCom*, Mar. 2021.

[52] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Fragments and loose associations: Respecting privacy in data publishing. *PVLDB*, 3(1), Sept. 2010.

[53] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[54] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *MSR*, 2018.

[55] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In *Research in Attacks, Intrusions and Defenses*, 2020.

[56] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical private range search revisited. In *Proc. of ACM SIGMOD*, San Francisco, CA, USA, June–July 2016.

[57] Deno Land. Deno Permission Model, 2022.

[58] Deno Land. Deno standard library for testing, 2022.

[59] Deno Land. Deno Subprocess, 2022.

[60] Deno Land. Deno Workers, 2022.

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

[62] Deno Land. Deno Permission Model, 2023.

[63] Docs.rs. Tokio, 2022.

[64] J. Domingo-Ferrer and V. Torra. Ordinal, continuous and heterogeneous k-anonymity through microaggregation. *Data Mining and Knowledge Discovery*, 11(2):195–212, 2005.

[65] J. Domingo-Ferrer and V. Torra. A critique of k-anonymity and some of its enhancements. In *Proc. of ARES 2008*, Barcelona, Spain, March 2008.

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

[67] 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 *NDSS*, 2021.

[68] C. Dwork. Differential privacy. In *Proc. of ICALP 2006*, Venice, Italy, July 2006.

[69] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[70] J. Edge. Seccomp and deep argument inspection, 2020.

[71] Emscripten Contributors. Emscripten toolchain, 2023.

[72] A. Ene, M. Kolny, and A. Brown. wasi-threads, 2023.

[73] 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.

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

[75] W. Findlay, A. Somayaji, and D. Barrera. bpfbox: Simple precise process confinement with ebpf. In *Cloud Computing Security Workshop*, 2020.

[76] B. Fung, K. Wang, and P. Yu. Top-down specialization for information and privacy preservation. In *Proc. of ICDE 2005*, March, Tokyo, Japan 2005.

[77] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *International Middleware Conference*, 2020.

[78] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang. Houdini's escape: Breaking the resource rein of linux control groups. In *CCS*, 2019.

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

[80] Google. Minijail, 2022.

[81] Google. Sandbox2, 2022.

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

[83] B. Gregg. BPF internals. 2021. USENIX LISA.

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

[85] H. Tao. BPF: Support for string key in hash-table, 2022.

[86] 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 *Programming Language Design and Implementation*, 2017.

[87] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of ACM SIGMOD*, Madison, WI, USA, June 2002.

[88] hackerone. External SSRF and Local File Read due to vulnerable FFmpeg, 2021.

[89] 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.

[90] B. Hore, R. C. Jammalamadaka, and S. Mehrotra. Flexible anonymization for privacy preserving data publishing: A systematic search based approach. In *Proc. of SIAM SDM 2007*, Minneapolis, MN, USA, April 2007.

[91] J. Edge. A seccomp overview, 2015.

[92] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. F. ldman Fitzthum, D. Skarlatos, D. Gruss, and T. Xu. Programmable system call security with ebpf. *arXiv*, 2023.

[93] 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 *IEEE Security and Privacy*, 2022.

[94] Z. Junyuan and R. Guo. Phantom attack: Evading system call monitoring. 2021. DEFCON.

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

[96] M. Kehoe. eBPF: The next power tool of SREs. USENIX Association, 2022.

[97] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX ATC*, 2013.

[98] 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 *EuroSys*, 2022.

[99] D. Land. Deno 1.24 release notes – improved ffi call performance, 2022.

[100] D. Land. Deno 1.25 Release Notes – FFI API improvements, 2022.

[101] D. Land. Deno: JavaScript runtime, 2022.

[102] D. Land. sqlite3 bindings for Deno, 2023.

[103] K. LeFevre, D. DeWitt, and R. Ramakrishnan. Incognito: Efficient full-domain k-anonymity. In *Proc. of SIGMOD 2005*, Baltimore, MA, USA, June 2005.

[104] K. LeFevre, D. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Proc. of ICDE 2006*, Atlanta, GA, USA, April 2006.

[105] K. LeFevre, D. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Proc. of ICDE*, Atlanta, GA, USA, Apr. 2006.

[106] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of webassembly. In *USENIX Security*, 2020.

[107] 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.

[108] 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.

[109] libbpf. libbpf, 2022.

[110] Linux manual. accept, 2023.

[111] Linux manual. listen, 2023.

[112] Linux manual. pipe, 2023.

[113] Linux manual. socketpair, 2023.

[114] M. K. Lau. BPF map-in-map support, 2017.

[115] M. S. Miller. Draft proposal for ses (secure ecmascript), 2022.

[116] A. Machanavajjhala, J. Gehrke, and D. Kifer. ℓ-diversity: Privacy beyond k-anonymity. In *Proc. of ICDE 2006*, Atlanta, GA, USA, April 2006.

[117] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 1(1):3:1–3:52, 2007.

[118] 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. *SIGPLAN*, 2015.

[119] M. McCaskey. Order of preopened directories matters, 2019.

[120] M. McCaskey. Prevent parent directory from being opened without being preopened wasi, 2019.

[121] W. McKinney. Data structures for statistical computing in Python. In *Proc. of SciPy 2010*, Austin, TX, USA, July 2010.

[122] Mickaël Salaün. Landlock: unprivileged access control, 2022.

[123] Microsoft. ebpf for windows, 2023.

[124] MUSEC. libpreopen, 2023.

[125] A. Nakryiko. BPF CO-RE, 2021.

[126] 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 *USENIX Security*, 2020.

[127] netblue30. Firejail, 2022.

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

[129] Npm. gm. `https://www.npmjs.com/package/gm`, 2022.

[130] Npm. sane. `https://www.npmjs.com/package/sane`, 2022.

[131] npm. bcrypt, 2023.

[132] npm. sharp, 2023.

[133] G. Ntousakis, S. Ioannidis, and N. Vasilakis. Detecting third-party library problems with combined program analysis. In *CCS*, 2021.

[134] OpenJS Foundation. Worker threads, 2022.

[135] OpenJS Foundation. Node Permissions, 2023.

[136] P. Simek. Proposal for VM2: Advanced vm/sandbox for Node.js, 2022.

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

[138] 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*, 2020.

[139] G. Poh, J. Chin, W. Yau, K. Choo, and M. Mohamad. Searchable symmetric encryption: Designs and challenges. *ACM CSUR*, 50(3), May 2017.

[140] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of SOSP*, Cascais, Portugal, Oct. 2011.

[141] K. Quest. Slimtoolkit, 2022.

[142] R. Dahl. 10 Things I Regret About Node.js. 2018. JSConf EU.

[143] J. Reback et al. pandas-dev/pandas: Pandas, February 2020. https://doi.org/10.5281/zenodo.3509134.

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

[145] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi. SEApp: Bringing Mandatory Access Control to Android Apps. In *USENIX Security*, 2021.

[146] S. Ruggles, S. Flood, R. Goeken, J. Grover, E. Meyer, J. Pacas, and M. Sobek. IPUMS USA: Version 10.0 [dataset], 2020. https://doi.org/10.18128/D010.V10.0.

[147] S. Ruggles, S. Flood, R. Goeken, J. Grover, E. Meyer, J. Pacas, and M. Sobek. IPUMS USA: Version 10.0 [dataset], 2020. https://doi.org/10.18128/D010.V10.0.

[148] M. Salaün. Landlock, 2022.

[149] P. Samarati. Protecting respondents' identities in microdata release. *IEEE TKDE*, 13(6):1010–1027, November/December 2001.

[150] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. 1998.

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

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

[153] 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*, 2016.

[154] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In *Proc. of ACM CCS*, Dallas, TX, USA, Oct./Nov. 2017.

[155] V. Shymanskyy. Overlapping preopens, 2020.

[156] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 2001.

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

[158] Snyk. Zip slip vulnerability, 2022.

[159] U. Sopaoglu and O. Abul. A top-down k-anonymization implementation for Apache Spark. In *Proc. of IEEE Big Data 2017*, Boston, MA, USA, December 2017.

[160] Stack Overflow Insights. Annual survey of the Stack Overflow community, 2022.

[161] C. Staicu, M. Pradel, and B. Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *NDSS*, 2018.

[162] C. Staicu, S. Rahaman, Á. Kiss, and M. Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. *USENIX Security*, 2023.

[163] L. Sweeney. k-anonymity: A model for protecting privacy. *International journal of uncertainty, fuzziness and knowledge-based systems*, 10(05):557–570, 2002.

[164] J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript(js.js): Sandboxing Third-Party Scripts. In *WebApps*, 2012.

[165] tesseract-ocr. Tesseract, 2023.

[166] The kernel development community. LSM eBPF Programs, 2023.

[167] TryGhost. Asynchronous, non-blocking SQLite3 bindings for Node.js, 2023.

[168] V8 project. Unsafe fast JS calls, 2020.

[169] V8 project. What is v8?, 2022.

[170] 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.

[171] J. Vaidya and C. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *Proc. of ACM KDD 2002*, Edmonton, Alberta, Canada, July 2002.

[172] H. Van Tran, T. Allard, L. d'Orazio, and A. El Abbadi. FRESQUE: A scalable ingestion framework for secure range query processing on clouds. In *Proc. of EDBT*, Nicosia, Cyprus, Mar. 2021.

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

[174] 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 *CCS*, 2021.

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

[176] W. W. Reading directory permissions, 2019.

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

[178] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li. Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index. In *Proc. of ACM ASIACCS*, Kyoto, Japan, June 2014.

[179] H. Wang and L. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *Proc. of VLDB*, Seoul, Korea, Sept. 2006.

[180] I. Wasmer. Wasmer Runtime, 2023.

[181] WebAssembly. WASI Libc, 2023.

[182] WebAssembly. Wasi SDK, 2023.

[183] WebAssembly. The webassembly system interface, 2023.

[184] 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.

[185] X. Xiao and Y. Tao. Anatomy: Simple and effective privacy preservation. In *Proc. of VLDB*, Seoul, Korea, Sept. 2006.

[186] J. Xu, W. Wang, J. Pei, X. Wang, B. Shi, and A.-C. Fu. Utility-based anonymization for privacy preservation with less information loss. *ACM SIGKDD Explorations Newsletter*, 8(2):21–30, 2006.

[187] W. Zhang, P. Liu, and T. Jaeger. Analyzing the overhead of file protection by linux security modules. In *ASIACCS*, 2021.

[188] X. Zhang, C. Leckie, W. Dou, J. Chen, R. Kotagiri, and Z. Salcic. Scalable local-recoding anonymization using locality sensitive hashing for big data privacy preservation. In *Proc. of CIKM 2016*, Indianapolis, IN, USA, October 2016.

[189] X. Zhang, L. Qi, W. Dou, Q. He, C. Leckie, R. Kotagiri, and Z. Salcic. MRMondrian: Scalable multidimensional anonymisation for big data privacy preservation. *IEEE TBD*, 8(1):125–139, 2022.

[190] X. Zhang, L. T. Yang, C. Liu, and J. Chen. A scalable two-phase top-down specialization approach for data anonymization using MapReduce on cloud. *IEEE TPDS*, 25(2):363–373, 2013.

[191] 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.

List of figures

List of tables