



UNIVERSITY OF BERGAMO

School of Doctoral Studies

Doctoral Degree in Engineering and Applied Sciences

XXXVI Cycle

SSD: ING-INF/05

**Fine-grained Access Control Technologies to Protect  
Resources in Mobile and Cloud Applications**

Advisor

*Prof. Stefano Paraboschi*

Doctoral Thesis by

*Matthew Rossi*

Student ID 1025234

Academic Year 2022/2023



*To my family and the friends that supported me*



# Abstract

Over the years operating systems security has greatly evolved and has been able to address many of the threats originating by an extensive and varied set of adversaries.

The mitigation of security threats is particularly important for mobile operating systems, due to their wide deployment and the confidential information they hold. Focusing on Android, we notice that components belonging to the same application share access to the app internal storage and system services. While this may not be an issue when the developer trusts all the code belonging to their application, it clearly becomes one when third-party code is included to achieve monetization and fast-paced development. Thus, we propose SEApp, a mechanism allowing developers to isolate the internal components of Android apps and regulate their permissions on a per-component basis. This is a crucial step to provide strong user privacy guarantees and meeting data privacy regulations despite the use of third-party code.

With the research conducted on Android it soon became clear that, while securing Android apps on the user device was very important, it was as much important to secure the cloud applications those applications interact with. Indeed, modern cloud applications can quickly grow to an intricate tangle of services, and unfortunately current technologies prove to be overly coarse to effectively restrict access control of system resources. To address this problem, we propose an approach that restricts access to file system resources with a resource-based granularity. Then, we further explore the topic in the context of WebAssembly runtimes (e.g., Wasmtime and WasmEdge). Specifically, we highlight the security implications of enabling access to system resources through the WebAssembly System Interface (WASI), and identify opportunities for improvement. Finally, we consider the use of JavaScript (JS) and TypeScript (TS) for the implementation of cloud applications thanks to JS runtimes (i.e., Node.js, Deno and Bun). These software securely renders JS code in an isolated sandbox, however access to system resources and the execution of native code raise security concerns, since they break the JS application isolation. To address these security issues, we propose NatiSand, a component for JavaScript runtimes to control the filesystem, Inter-Process Communication (IPC), and network resources available to binary programs and shared libraries.

The technologies described in this thesis advance the state of the art in fine-grained resource protection in mobile and cloud applications. The implementations have been released under open-source licenses and can be easily integrated with existing real systems.



# Table of Contents

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>11</b>
1.1 Document structure . . . . .	13
1.2 Publications . . . . .	16
<b>2 Methodology</b>	<b>19</b>
<b>3 Fine-grained Access Control in Android Applications</b>	<b>21</b>
3.1 Introduction . . . . .	21
3.2 Android security for apps . . . . .	22
3.3 Motivation . . . . .	24
3.3.1 Use cases . . . . .	24
3.3.2 Modular app compartmentalization . . . . .	27
3.3.3 Compatibility with Android design . . . . .	28
3.3.4 Compatibility with other proposals . . . . .	28
3.4 Policy language . . . . .	28
3.4.1 Choice of policy language . . . . .	29
3.4.2 Definition of types and type-attributes . . . . .	29
3.4.3 Policy constraints . . . . .	30
3.5 Policy configuration . . . . .	32
3.5.1 SEAndroid policy structure . . . . .	33
3.5.2 SEApp policy structure . . . . .	34
3.6 Implementation . . . . .	37
3.6.1 Policy compilation . . . . .	37
3.6.2 Runtime support . . . . .	40
3.7 Experimental results . . . . .	43
3.7.1 App installation . . . . .	43
3.7.2 Runtime performance . . . . .	46
3.8 Related work . . . . .	47
3.9 Conclusions . . . . .	49
<b>4 Lightweight Cloud Application Sandboxing</b>	<b>51</b>
4.1 Introduction . . . . .	51
4.2 Motivation . . . . .	52
4.2.1 Threat model . . . . .	52

4.2.2	Dependency identification . . . . .	53
4.2.3	Mitigation of bugs . . . . .	53
4.2.4	Performance and usability . . . . .	54
4.3	Approach overview . . . . .	55
4.4	Cloud application instrumentation . . . . .	57
4.4.1	Ptrace-based instrumentation . . . . .	57
4.4.2	eBPF-based instrumentation . . . . .	57
4.5	Policy . . . . .	59
4.6	Application sandboxing . . . . .	62
4.7	Experiments . . . . .	63
4.7.1	Mitigation of vulnerabilities . . . . .	63
4.7.2	Overhead . . . . .	64
4.8	Related work . . . . .	65
4.9	Conclusions . . . . .	67
<b>5</b>	<b>Enhancing the Sandbox of WebAssembly Runtimes</b>	<b>69</b>
5.1	Introduction . . . . .	69
5.2	Threat model . . . . .	70
5.3	Architecture . . . . .	71
5.4	Experiments . . . . .	72
5.5	Related work . . . . .	73
5.6	Conclusions . . . . .	74
<b>6</b>	<b>Native Code Sandboxing for JavaScript Runtimes</b>	<b>75</b>
6.1	Introduction . . . . .	75
6.2	Background . . . . .	77
6.2.1	JS runtimes . . . . .	77
6.2.2	Components for resource protection . . . . .	78
6.3	Security motivation . . . . .	81
6.3.1	Threat model . . . . .	83
6.4	Design and implementation . . . . .	83
6.4.1	Objectives . . . . .	83
6.4.2	High level architecture . . . . .	84
6.4.3	Integration with JS runtimes . . . . .	85
6.4.4	Isolation features . . . . .	88
6.5	Policy . . . . .	91
6.5.1	Policy structure . . . . .	91
6.5.2	Policy generation . . . . .	94
6.6	Case study: Deno runtime . . . . .	95



6.6.1	Runtime selection . . . . .	95
6.6.2	Deno integration . . . . .	96
6.6.3	Support to fast JS calls . . . . .	96
6.7	Experiments . . . . .	97
6.7.1	Exploit mitigation . . . . .	98
6.7.2	Performance evaluation . . . . .	99
6.8	Related work . . . . .	104
6.9	Conclusions . . . . .	106
<b>7</b>	<b>Conclusions and future work</b>	<b>107</b>
7.1	Future work . . . . .	108
	<b>Acknowledgments</b>	<b>111</b>
<b>A</b>	<b>Practical showcase of SEApp capabilities</b>	<b>113</b>
A.1	UC#1: fine-granularity in access to files . . . . .	113
A.2	UC#2: fine-granularity in access to services . . . . .	115
A.3	UC#3: isolation of vulnerability-prone components . . . . .	117
A.4	Policy module . . . . .	119
<b>B</b>	<b>Policy generated for the curl command</b>	<b>123</b>
	<b>References</b>	<b>125</b>



# List of Figures

2.1	High-level overview of the research methodology . . . . .	20
3.1	Evolution of the MAC policy in Android . . . . .	23
3.2	Security Enhanced App . . . . .	25
3.3	SEApp policy structure . . . . .	34
3.4	Installation process . . . . .	39
3.5	Application launch . . . . .	40
3.6	File relabeling . . . . .	41
3.7	Installation time overhead for apps with different complexity . . . . .	43
3.8	Cumulative install time overhead of the top 100 free apps with our policies	44
3.9	Install time overhead for the three policy sizes . . . . .	45
4.1	Overview of our approach . . . . .	55
4.2	Architecture of the ptrace-based instrumentation . . . . .	58
4.3	Architecture of the eBPF-based instrumentation . . . . .	59
4.4	Landlock sandbox setup and inheritance . . . . .	62
4.5	Latency of image processing operations . . . . .	65
4.6	Latency of video processing operations . . . . .	66
5.1	Current implementation of WASI by runtimes . . . . .	70
5.2	eBPF-based restriction of Wasm modules . . . . .	72
6.1	Execution of binary programs and shared libraries by JS runtimes . . . . .	79
6.2	Overview of the eBPF architecture . . . . .	80
6.3	Integration of NatiSand in JS runtimes . . . . .	86
6.4	Average latency and throughput of subprocess-based microservices . . . . .	101
6.5	Average latency and throughput of native-functions-based microservices .	103
A.1	Showcase app main view . . . . .	113
A.2	UC#1 view (initiation, exploitation, and mitigation) . . . . .	115
A.3	UC#1 SELinux denial message in the system log . . . . .	115
A.4	UC#2 views . . . . .	116
A.5	UC#2 malicious gadget retrieves location data . . . . .	117
A.6	UC#2 SELinux denial message in the system log . . . . .	117
A.7	UC#2 activity termination due to SELinux denial . . . . .	117
A.8	UC#3 views (initiation, exploitation, and mitigation) . . . . .	118
A.9	UC#3 SELinux denial message in the system log . . . . .	118



# List of Tables

3.1	Application policy module CIL syntax . . . . .	29
3.2	SEApp macros to grant permissions to local types . . . . .	32
3.3	Policy size . . . . .	45
3.4	Cold and warm start performance for activities and services . . . . .	46
3.5	File creation performance . . . . .	47
4.1	List of file system traced hook points . . . . .	60
4.2	Sample of CVEs reproduced in our evaluation . . . . .	64
5.1	Average execution time of coreutils with and without our approach . . . . .	73
6.1	Hooks and tracepoints monitored by NatiSand . . . . .	88
6.2	LSMs used by NatiSand to restrict Linux IPC . . . . .	91
6.3	Sample of CVEs mitigated by NatiSand . . . . .	99
6.4	Average execution time for common Linux utilities . . . . .	101
6.5	Average execution time for common native libraries . . . . .	103



# 1. Introduction

Over the years operating systems security has greatly evolved and has been able to address many of the threats originating by an extensive and varied set of adversaries.

The mitigation of security threats is particularly important for mobile operating systems, due to their wide deployment and the confidential information they hold. Focusing on Android, which is open source and more accessible to researchers, we notice that components belonging to the same application share access to the app internal storage and system services. So, third-party code included by developers in their applications for monetization and fast-paced-development purposes have the same access to internal data and system services as the rest of the application. This represents a threat to the security of applications and the privacy of users. Thus, we propose SEApp, a mechanism allowing developers to isolate the internal components of Android apps and regulate their permissions on a per-component basis. As a natural evolution of the security mechanisms already available in Android, SEApp design requires to (i) preserve the security of system components, (ii) limit how it may affect the development of applications, and (iii) minimize the performance impact. Our evaluations show our proposal meet these requirements, and SEApp is a crucial step to provide strong user privacy guarantees and comply to data privacy regulations despite the use of third-party code. This claim is supported by the current direction Google is following for the development of its Privacy Sandbox on Android [21].

With the research conducted on Android it soon became clear that, while securing Android apps on the user device was very important, it was as much important to secure the cloud applications those applications interact with. Indeed, modern cloud applications can quickly grow to an elaborate and intricate tangle of services, and keeping track of all the moving parts and their security boundaries can be very challenging. Nevertheless, in order to mitigate the impact of security incidents companies need to pay close attention to the security of their cloud applications. Several research works and industrial standards recommend the integration of least privilege policies to prevent disruptions such as file system tampering. Unfortunately current technologies prove to be overly coarse to effectively restrict access control of system resources. To address this problem, we propose an approach that restricts access to file system resources with a resource-based granularity. Specifically, we provide an instrumentation solution to retrieve all the file system resources required by an application component. Then, we demonstrate how this information can be used

to create fine-grained access policies, and introduce sandboxing using recent kernel security modules, thus strengthening the security boundary of the whole application.

Even though effective in restricting access to an entire microservice, the approach above can be further improved by considering the specifics of emerging technologies that enable finer-grained compartmentalization of cloud applications. In the context of WebAssembly runtimes (e.g., Wasmtime and WasmEdge), we highlight the security implications of enabling access to system resources through the WebAssembly System Interface (WASI), and identify opportunities for improvement. Here, not only our approach permits to introduce fine-grained policies to restrict file system access, it also replaces error-prone userspace implementations of the security checks (and the security issues stemming from them) with a unified eBPF implementation.

In the context of using JavaScript and TypeScript for the development of cloud applications, we investigate how JS runtimes secure the applications they host, and identify a few limitations. Modern runtimes (i.e., Node.js, Deno and Bun) render JavaScript code in a secure and isolated environment, however access to system resources and the execution of native code raise security concerns, since they break the JS application isolation. Developers commonly rely on native code to speed up the development and execution of their applications and, despite that, JS runtimes do not provide built-in solutions for the isolation of native code, so we propose the introduction of NatiSand; a component for JavaScript runtimes to control the filesystem, Inter-Process Communication (IPC), and network resources available to binary programs and shared libraries. Our solution is characterized by a compact, generic architecture that fits nicely with modern runtimes. Internally, it leverages *Seccomp* [63] and Linux Security Modules (LSMs), such as *Landlock* [138] and *eBPF* [73] to restrict access to protected resources.

During the research work, considerable attention was dedicated to the usability, effectiveness, and performance of our proposals. Indeed, in most of our proposals, we allow developers to take full advantage of the high level security mechanisms without requiring them to fully comprehend the advanced security features employed underneath. The developer is only required to provide a concise and readable policy file, which details the access privileges available to the components of their application and may be automatically generated. Moreover, we provide extensive evaluations showcasing the mitigation capabilities of our proposals with respect to common use cases and severe CVEs. Alongside the security guarantees introduced, we evaluate also their performance footprint, and can confirm the introduction of very limited overhead with respect to the original unsecure systems and significant improvements with respect to alternative approaches. To promote the reproducibility of our experimental evaluations, and facilitate the integration of our solutions with existing real systems our prototype implementations are all available open source.



## 1.1 Document structure

The thesis is organized in six chapters.

**Chapter 1** illustrates the document structure and the publications that set the basis for this thesis.

**Chapter 2** showcases the methodological approach employed to investigate the use of fine-grained access control technologies to protect resources in mobile and cloud applications.

**Chapter 3** describes SEApp [167], a proposal that provides developers with a mechanism to isolate the internal components of Android apps and regulate their permissions on a per-component basis. This is achieved by first executing components in dedicated processes, and then restricting access to the app and system resources with ad hoc SELinux policies. Specifically, it is possible to declare rules to regulate access to both the internal app storage and system services, which are otherwise shared between all the application components, including third-party libraries the app depends upon. This is a crucial step to provide strong user privacy guarantees despite relying on third-party code to achieve monetization and fast-paced development of Android applications. The prototype implements a patch to the Android Open Source Platform (AOSP) to demonstrate the feasibility, effectiveness and efficiency of the novel approach.

The chapter is organized as follows.

- Section 3.1 provides an overview of how the security architecture of mobile systems evolved with the maturity of the ecosystem.
- Section 3.2 introduces the techniques currently enforcing access control in Android.
- Section 3.3 presents the motivation for the introduction of intra-app isolation in Android. Specifically, a set of use cases is used to showcase the security measures introduced by SEApp.
- Section 3.4 details the SEApp policy module syntax, and its constraints to ensure proper integration of app and system policies.
- Section 3.5 illustrates the policy configuration files of SEAndroid and how to use them in SEApp.
- Section 3.6 discusses the changes the SEApp implementation introduced in Android platform.
- Section 3.7 presents the experimental evaluation, in which we measure both the installation time and runtime overhead introduced by SEApp.

- Section 3.8 discusses the major differences between SEApp and other literature proposals.
- Section 3.9 concludes the chapter.

**Chapter 4** highlights the limitations of the cloud technologies available at the time of writing with regard to fine-grained access control of system resources. Then, it addresses the problem proposing an approach that restrict application access to file system resources with a resource-based granularity. To this end, we develop Dmng, a flexible and intuitive tool that relies on instrumentation to collect and audit the activity traces generated by microservices. Finally, we demonstrate how this information can be used to create fine-grained access control policies and strengthen the security boundary of the cloud application.

The chapter is organized as follows.

- Section 4.1 presents the challenges of securing cloud applications and briefly discusses the state of the art to identify opportunities for improvement.
- Section 4.2 discusses the objectives of the solution together with its requirements and trust assumptions.
- Section 4.3 illustrates an overview of the approach by highlighting its integration with both staging and production environments.
- Section 4.4 presents two different techniques to instrument cloud applications and trace their activity.
- Section 4.5 discusses the generation of access control policies from activity traces.
- Section 4.6 details how we implement the lightweight enforcement of the policies.
- Section 4.7 presents an empirical evaluation of the mitigation capabilities of our approach and its performance.
- Section 4.8 discuss proposals from the literature and how they compare to ours.
- Section 4.9 concludes the chapter.

**Chapter 5** explores the use of WebAssembly outside of web browsers thanks to WebAssembly runtimes (e.g., Wasmtime and WasmEdge). Specifically, it highlights the security implications of enabling access to system resources through the WebAssembly System Interface (WASI) and it identifies opportunities for improvement. With specific regard to file system resources, runtimes must prevent hostcalls (i.e., functions provided by the runtime to the guest WebAssembly application) from accessing arbitrary locations. Current implementations introduce security checks to

only permit access to a predefined list of directories. This approach not only suffers from poor granularity, it is also error-prone and has led to security issues. In this chapter we propose to replace the security checks in hostcall wrappers with eBPF programs, enabling the introduction of fine-grained per-module policies.

The chapter is organized as follows.

- Section 5.1 introduces WebAssembly and the challenges WebAssembly runtimes are facing with restricting access to system resources.
- Section 5.2 models the threat an attacker can pose to the implementations of WASI in WebAssembly runtimes.
- Section 5.3 details the design of current WASI implementations and proposes an alternative approach.
- Section 5.4 evaluates the overhead introduced by our novel approach in two different WebAssembly runtimes.
- Section 5.5 discusses the related work.
- Section 5.6 concludes the chapter.

**Chapter 6** considers the use of JavaScript (JS) and TypeScript (TS) for implementing the backend of cloud applications. In this setting, the execution of JS code on the server-side is enabled by JS runtimes (i.e., Node.js, Deno and Bun). These sophisticated software securely renders JS code in an isolated sandbox, however access to system resources and the execution of native code raise security concerns, since they effectively break the isolation between the JS application and the host OS. Identifying these clear security issues, this chapter proposes 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. We demonstrate the effectiveness and efficiency of our approach by implementing and integrating it into Deno, a modern, security-oriented JavaScript runtime.

The chapter is organized as follows.

- Section 6.1 presents the development of web applications with the use of JavaScript runtimes and their limitations with regard to controlling native code access to system resources.
- Section 6.2 overviews the structure of modern JS runtimes and provides background on the components used by NatiSand to build the sandbox.
- Section 6.3 motivates the need of ad hoc access restrictions for native code highlighting the security implications of an otherwise overpermissive behavior.
- Section 6.4 presents NatiSand objectives and architecture.

- Section 6.5 details the policy structure and explains how to generate its permission rules.
- Section 6.6 showcases the achievement of the aforementioned objectives by integrating NatiSand into the Deno runtime.
- Section 6.7 presents the ability of NatiSand to mitigate real-world vulnerabilities while introducing only a negligible overhead compared to a permissive scenario.
- Section 6.8 discusses the major differences between NatiSand and other literature proposals.
- Section 6.9 concludes the chapter.

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

## 1.2 Publications

This section presents the list of publications authored during the Ph.D. course that set the basis for this thesis.

### Articles in journals

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Giovanni Livraga, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi, Pierangela Samarati. “**Scalable Distributed Data Anonymization for Large Datasets**”. *IEEE Transactions on Big Data (TBD)*, Volume 9, Issue 3. IEEE, 2022.
- 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**”. Under submission.

### Papers in proceedings of international conferences

- Sabrina De Capitani di Vimercati, Dario Facchinetti, Sara Foresti, Gianluca Oldani, Stefano Paraboschi, Matthew Rossi, and Pierangela Samarati. “**Scalable distributed data anonymization**”. *Proceedings of the 2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 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**”. *Proceedings of the 2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 2021.
- Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, and Stefano Paraboschi. “**SEApp: Bringing Mandatory Access Control to Android Apps.**”. *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*. USENIX, 2021.
  - Enrico Bacis, Dario Facchinetti, Marco Guarnieri, Marco Rosa, Matthew Rossi, and Stefano Paraboschi. “**I Told You Tomorrow: Practical Time-Locked Secrets using Smart Contracts**”. *Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES)*. ACM, 2021.
  - 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**”. *Proceedings of the 2021 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2021.
  - Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. “**Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses**”. *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2023.
  - Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. “**Leveraging eBPF to enhance sandboxing of WebAssembly runtimes**”. *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2023.
  - Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. “**NatiSand: Native Code Sandboxing for JavaScript Runtimes**”. *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 2023.
  - Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. “**Lightweight Cloud Application Sandboxing**”. *Proceedings of the 14th IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM)*. IEEE, 2023.

- 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**”. *Proceedings of the 2024 IEEE International Conference on Communications (ICC)*. IEEE, 2024.

## 2. Methodology

This chapter outlines the foundational methodology employed throughout the thesis. The research design adheres to a core set of principles, fostering a continuous cycle of exploration, refinement, and validation. The approach commences with the meticulous identification of security challenges associated with resource protection within the diverse domains of mobile and cloud applications. This initial phase involves in-depth analysis of existing access control methods, exposing limitations concerning their ability to provide fine-grained control. Then, we identify threat models associated with the coarse granularity. While there are inherent differences depending on the specific context investigated, there is a common bottomline associated with the exploitation of vulnerabilities, and how pervasive the use of third-party software is in current mobile and cloud applications. Indeed, running application components with broader permissions translate into more attractive targets for attackers, as they can reach a larger set of sensitive assets and escalate the attack. Moreover, the common practice of using third-party software to speed up development further exacerbates the problem, with developers lagging behind with dependency updates and often disregarding the trust implications of using external dependencies.

Informed by this analysis, the next phase involves the design of novel access control mechanisms that specifically target fine-grained enforcement. The design process encompasses the meticulous evaluation of enforcement models, creation of security policies, and careful consideration of integration strategies for seamless operation within the established system architectures. Despite the differences between mobile and cloud architectures, when we consider Android (i.e., the mobile operating system holding 69.94% of the market share [178]), Figure 2.1 shows they share a common foundation: the Linux kernel. By leveraging the built-in security features of the Linux kernel (e.g., seccomp, eBPF, Landlock, SELinux), this thesis proposes an efficient approach to establishing fine-grained access control across mobile and cloud environments. This approach not only addresses specific challenges in each domain but also demonstrates the versatility and potential of the Linux kernel as a foundation for diverse application domains.

Following the design of the solution, the next step involves its implementation utilizing programming languages, frameworks, and tools specifically chosen for their suitability within the targeted environments. Then, extensive evaluation follows, utilizing a combination of security testing and performance measurement techniques. Security testing meticulously evaluates the capability of the solution to enforce secu-

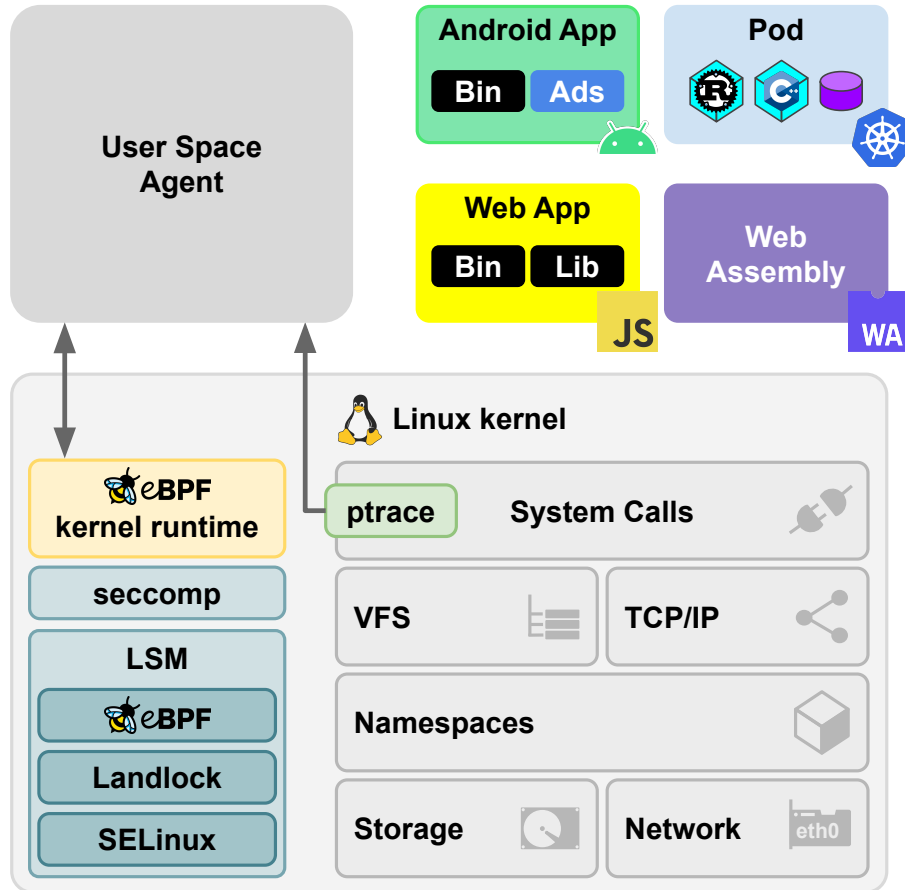


Figure 2.1: High-level overview of the methodology for integrating the use of fine-grained access control technologies in mobile and cloud applications

rity policies and prevent unauthorized resource access. This consists of reproducing common vulnerabilities classes and public proofs of concept to appraise the ability to mitigate attacks. Performance evaluation examines the overheads introduced by the solution, assessing metrics such as execution time and throughput. The findings serve as the cornerstone for the iterative refinement and improvement of the proposed solutions. This cyclical process addresses any identified shortcomings, optimizes efficiency, and ultimately ensures the robustness of the fine-grained access control mechanisms. Prototype implementations are open-source to promote reproducibility of our experimental results and ease of integration with existing systems.

While the specific technical details and implementation nuances vary between mobile and cloud environments due to their inherent differences, the overarching methodological framework described in this chapter provides a concise summary of the research approach adopted, but also highlights the potential of leveraging shared underlying features to address security challenges across diverse technological domains.



# 3. Fine-grained Access Control in Android Applications

## 3.1 Introduction

Security in operating systems has greatly evolved and has been able to address many of the threats originating by an extensive and varied collection of adversaries.

The mitigation of security threats is particularly important for *mobile operating systems*, due to their wide deployment and the confidential information they hold.

Both Android and iOS have seen significant investments toward the realization of advanced security techniques, which have led to a great increase in the level of protection offered to users [140]. The strength of security and the value of protected resources is testified, for instance, by the payouts associated with working exploits in markets like Zerodium [207], where the payouts for mobile operating systems are the highest<sup>1</sup>.

A peculiar threat that characterizes mobile operating systems is the need to balance on one side the high sensitivity of the information, and on the other hand the need for users to install into the system a large number of applications (called simply *apps* in this domain) often produced by unknown developers, which may hide malicious functions. A first level of protection is offered, both in iOS and Android, by a preliminary screening of apps before they are made available on the platform market [14] or installed to a device, but this approach cannot provide a strong guarantee. Security mechanisms internal to the operating system are needed in order to constrain the apps to only operate within the boundaries specified by the device owner at installation time.

The approach used in the design of mobile operating systems considers as the first requirement the protection of system resources. Focusing on Android, which is open source and more accessible to researchers, we notice a significant evolution in its internal security architecture. This architecture is quite rich and consists of many security measures [100, 140]. In this environment, we specifically look at the role of SELinux. SELinux implements the *Mandatory Access Control* (MAC) mechanism, which relies on a system-level policy to declare the operations that a process can execute over a resource based on the security labels associated with them.

---

<sup>1</sup>At the time of writing, US\$2.5M and US\$2M are paid for a zero click solution able to subvert the security of Android and iOS, respectively.

Compared to classical *Discretionary Access Control* (DAC), still used in Android in an extensive way, MAC is more rigid and provides stronger guarantees against unwanted behaviors. When SELinux was introduced into Android 4.3 in 2013 (see Figure 3.1), it used a limited set of system domains and it was mainly aimed at separating system resources from user apps. In the next releases, the configuration of SELinux has progressively become more complex, with a growing set of domains isolating different services and resources, so that a bug or vulnerability in some system component does not lead to a direct compromise of the whole system.

The introduction of SELinux into Android has been a clear success. Unfortunately, the stronger protection benefits do not extend to regular apps which are assigned with a single domain named `untrusted_app`. Since Android 9, isolation of apps has increased with the use of categories, which guarantees that distinct apps operate on separate security contexts. Our proposal, SEApp, builds upon the observation that giving app developers the ability to apply MAC to the internal structure of the app would provide more robust protection against other apps and internal vulnerabilities.

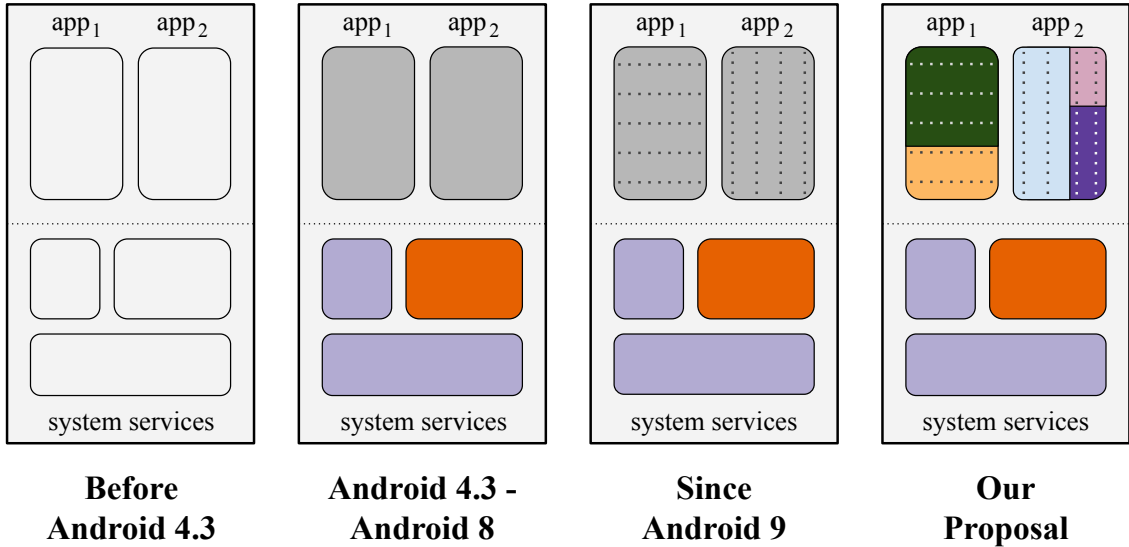
## 3.2 Android security for apps

One of the major requirements considered in the design of mobile operating systems is the need to constrain the ability of apps to manipulate the execution environment. Apps may hide functions that are meant to gain system privileges or capture valuable information from other apps. Compared to classical desktop operating systems, there is greater reliance on the use of apps to access resources or get services, with more attention paid to limit the ability of apps to operate in the system. Advancements in this context can have an impact on how security for applications is managed in other domains [7].

The basic principle adopted to manage the threat introduced by apps is the design of a *sandbox*, a restricted environment for app execution, where anomalous actions by the app are not able to access resources beyond what has been authorized at app installation time. The sandbox can be considered a realization of the “least privilege” security principle.

The construction of the app sandbox is based on three access control mechanisms: Android permissions [27, 100, 101], Discretionary Access Control (DAC) [66], and Mandatory Access Control (MAC) [168]; each of them roughly aligning with how users, developers, and the platform grant consent, respectively.

Permissions restrict access to sensitive data and services. In the `AndroidManifest.xml` [29], each app statically lists the Android permissions needed



**Figure 3.1: Evolution of the MAC policy in Android. Before 4.3, MAC was not used. Starting with 4.3, MAC protects system components. Since 9, categories offer rigid MAC protection for apps. Our proposal offers flexible MAC protection to apps.**

to fully operate. Not all of them may be granted; depending on the threat they pose from a security and privacy standpoint, they may be granted as part of the installation procedure, or prompted to the user when the app needs them.

DAC restricts access to resources based on user and group identity. By assigning each application a unique UNIX user ID (UID) and a dedicated directory, Android isolates apps from each other and from the system. However, UID sandboxing has a number of shortcomings. As an example, processes running as root are not subject to these restrictions. For this reason, when such a process is misbehaving, for instance due to a bug, it can access private app data files. DAC discretionality itself is a problem. Indeed, as apps and system processes could override safe defaults, they are more susceptible to dangerous behavior, such as leaking files or data across security boundaries via IPC or fork/exec. Despite its deficiencies, UID sandboxing is still the primary enforcement mechanism that separates apps from each other, establishing the foundation upon which further sandbox restrictions have been built.

MAC dictates which actions are allowed based on the security policy defined by the system. Specifically, only actions explicitly granted by the policy are permitted. To decide whether to permit or deny an action, a set of policy rules concerning the *security contexts* (i.e., collections of security labels that classify resources) of the involved parties is evaluated.

In Android, MAC is implemented using SEAndroid, a set of kernel modifications part of the Linux Security Module (LSM) framework [203]. Since its first introduc-

tion with the Security Enhanced Android (SEAndroid) project [173], SELinux has been extensively applied to protect system components. Initially, it was used to assert the security model requirements during compatibility testing, then its usage grew further at each release. In the current version Android 11, SELinux is also used to isolate the rendering of untrusted web content (by the `isolated_app` domain), to restrict `ioctl` system calls [122], thus limiting the reachability of potential kernel vulnerabilities, and to support multi-user separation and app sandboxing with SELinux categories. This last aspect permits to enforce app separation both at DAC and MAC. Android dynamically assigns categories to apps during app installation, so that: (i) an app running on behalf of a user cannot read or write files created by the same app on behalf of another user (since Android 6 [22]); and, (ii) an app cannot read or write files created by another app (since Android 9[24]). Before Android 9, this separation was only enforced at DAC level. This overlap of security measures is of extreme relevance to the enforcement of the Android Security Model and our proposal moves in the same direction. To bypass these protections, a process should be granted root permissions, `DAC_OVERRIDE` or `DAC_READ_SEARCH`, and run as SELinux `mlstrustedsubject`; only a few critical system services run in this configuration.

Android restricts the SELinux implementation to the policy enforcement, ignoring most policy management functions. The motivation is that the system policy only changes between releases, therefore support to runtime changes is not needed.

### 3.3 Motivation

As discussed above, SELinux and the MAC support have been a crucial factor in the realization of a secure design and the construction of a robust app sandbox. A limitation of the current design is that this is the only way that apps can benefit from MAC support. There is currently no option to let the app developer control the use of the MAC level, as only platform, vendor, ODM and OEM developers are allowed to introduce new policy segments [38]. Our solution overcomes this limitation, giving the application developer the power to specify new SELinux types and associated permissions.

#### 3.3.1 Use cases

We envision several scenarios that justify the use of SEApp. Many of them have been previously considered by researchers as motivations for the introduction in Android of dedicated components [49, 93, 120].

In this section, we give a tour of SEApp capabilities using a showcase app<sup>2</sup>. The architecture of the showcase app is shown in Figure 3.2. Our description is based on three use cases: fine-granularity in access to files, fine-granularity in access to services, and isolation of vulnerability-prone components. Each of the use cases emphasizes the intra-app security features introduced by SEApp. A dedicated description, along with policy files that show concretely how to enforce these use cases, appears in the Appendix A; we provide there a technical demonstration of how SEApp can provide protection against a number of common security problems in Android apps [113] that were implemented in the showcase app.

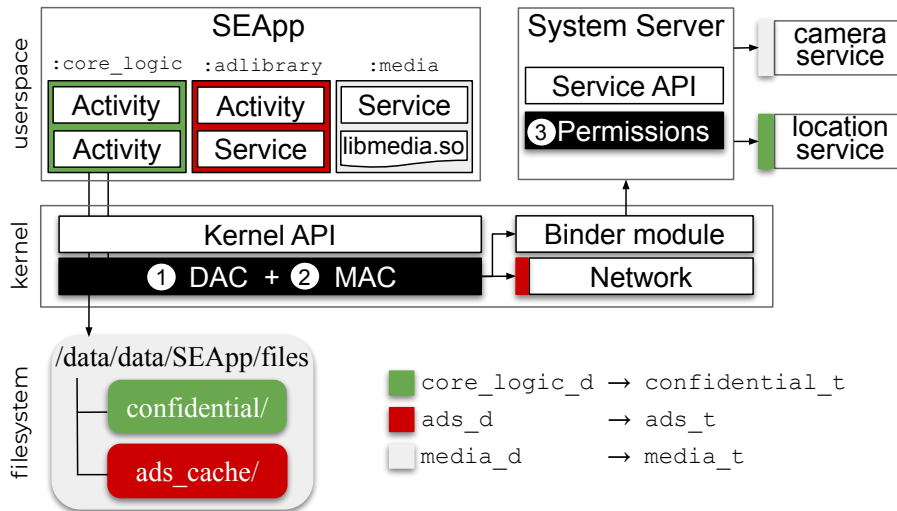


Figure 3.2: Security Enhanced App

### Fine-granularity in access to files

Android apps can collect data from multiple sources, and the system provides many options to store it. The default one is *Internal Storage*: a filesystem region, located at `/data/data/packageName`, reserved to each package. Its content is available to all app’s internal components and inaccessible to any other app. Since data can be extremely sensitive, the developer may be interested in restricting its visibility to only some internal components, labeling sensitive and non-sensitive data with distinct SELinux types (use case 1). Yet, in the current Android security model, apps do not have the option to assign distinct MAC labels to different resources, as all internal files are labeled `app_data_file`. SEApp allows the developer to introduce dedicated types, and to organize the app’s structure with a separation between components managing non-sensitive data and those requiring access to sensitive data. The sensitive components will be associated with a more stringent MAC domain.

<sup>2</sup>The showcase app is available in the SEApp repository along with the set of modifications to the AOSP.

Figure 3.2 shows an example in which the *confidential* files are made accessible to `:core_logic` processes and inaccessible to any other process.

In Appendix A.1 we give a demonstration of how *confidential* files are made inaccessible to non-confidential components in the presence of a path traversal vulnerability.

### **Fine-granularity in access to services**

Often developers introduce into their applications code coming from external sources, which they do not fully trust [105, 165, 84]. For instance, a common need of app developers is to get revenue from their apps and a simple approach is to include an Ad delivery library within the app. The library is a relatively complex piece of code, with local computation necessities and the need to manage a dialogue with remote servers. The app developer is clearly interested in supporting the execution of the library, but may want to have guarantees that the library cannot abuse the access privileges granted by the user to the whole application sandbox (use case 2). A common concern is preventing access to system services such as *location*. These requirements can be managed by SEApp with the definition of a separate MAC domain for the library. The process managing the delivery of Ads will be associated with this domain, which will provide only the necessary privileges to access the dedicated resources needed for the library execution. SELinux will then guarantee the confinement of the library, preventing access to the location service even if the `ACCESS_FINE_LOCATION` permission is granted to the app. Figure 3.2 shows an example in which the `:adlibrary` process is granted access to the network but is prevented from accessing `location service`.

In Appendix A.2 we give a demonstration of how the showcase app can support the execution of the Unity Ads [193] framework with a dedicated SELinux domain. We also describe in detail how SEApp prevents a malicious component, which was deliberately injected by us into the library process, to capture the device location.

### **Isolation of vulnerability-prone components**

App developers often have to consider that the input provided to the app can come from untrusted sources. A typical example is the rendering of complex Javascript code performed by WebView. The solution currently offered by Android is to execute these potentially dangerous actions within a sandbox using *isolatedprocess*, i.e., a special process that is isolated from the rest of the system and has no permissions of its own [18]. It runs under a dedicated UID and SELinux domain, and it can only interact with a restricted number of services [20].

A common need of app developers is to take advantage of complex media or processing libraries, components that are not considered malicious, but due to their size and complexity are more likely to have security bugs. The developer is then interested in isolating these potentially vulnerable components (use case 3). *Isolatedprocess* offers a high protection level in Android, however, its use imposes several restrictions on the developers. For instance, *isolatedprocess* cannot perform many of the core Android IPC functions, and the only way to interact with it is through the bound service API [19]. Also, *isolatedprocess* can only access already open app files received over Binder. Another shortcoming is that each invocation of an *isolatedprocess* requires the creation of a new process. If a series of requests are made by the app, the performance impact can be significant. SEApp offers an easier way to do this compared to *isolatedprocess*, as it permits to assign a domain to the process in which the component is executed, and then configure the required permissions at MAC level. In terms of performance, the management of multiple requests does not require the system to activate a new process with a new UID and a dedicated SELinux category. Figure 3.2 shows how to confine the *:media* component.

In Appendix A.3 we give a demonstration of how the showcase app can support the execution of media components relying on a native library in a dedicated process. We also describe how the developer can leverage SEApp to prevent the code of the library from the execution of unwanted or unintended operations, like opening a network connection.

### 3.3.2 Modular app compartmentalization

The motivations presented above become more frequent as apps increase their size and complexity, and several important apps see a continuous increase in these parameters. For instance, Facebook Messenger version 285 contains more than 500 components and WhatsApp Messenger version 2.20 more than 300. This increase in size and the need to manage it is testified by the development of App Bundles [16], Android’s new, official publishing format that offers a more efficient way to build and release modular applications.

In these large and modular apps, developers find it difficult to fully control which components of an app are using sensitive data<sup>3</sup>. The availability of a solution such as SEApp can greatly reduce such risk. A better compartmentalization can reduce the impact of internal vulnerabilities in modular apps, since each module can be associated with a dedicated policy fragment. From a security and software engineering standpoint, SEApp permits to separate the activities of security policy maintenance and development of new features.

<sup>3</sup>The topic was explicitly considered in [45], an interview with Android’s VP of Engineering.

### 3.3.3 Compatibility with Android design

Looking at the evolution of Android, it is clear that our proposal is consistent with the evolution of the operating system and the desire of its designers to let app developers have access to an extensive and flexible collection of security tools. The major obstacles, as perceived by OS developers, on offering to app developers the use of MAC services are: weakening of the protection of system components; performance impact; usability by app developers. The work we did solves these concerns: our approach guarantees that app policies do not have an impact on the system policy (Section 3.4.3); the app policy can be specified declaratively and attention has been paid to let developers adopt the approach in a convenient way (Section 3.5.2); and, experiments demonstrate the acceptable performance impact, with a quite limited overhead at app installation time, and a negligible runtime impact (Section 3.7).

### 3.3.4 Compatibility with other proposals

As presented in Section 3.3.1, SEApp by itself provides protection against a broad spectrum of attacks (see Appendix A), but its merit does not end there. As multiple literature proposals (e.g., [120, 51, 206]) build upon process isolation and use it to accomplish separation of privileges at the application layer, SEApp could be used as building block to enforce such restrictions at the MAC layer too, enabling defense in depth. Moreover, SEApp could also work in conjunction with other solutions that work at MAC level such as *FlaskDroid* [59], to benefit of its Userspace Object Managers (USOMs) coverage of the Android system services and provide finer granularity in access to services.

## 3.4 Policy language

To support the use cases presented in Section 3.3, we want the developer to have control of the SELinux security context of subjects and objects related to her security enhanced app. To each of them is assigned a *type* (also called *domain* when it labels processes). As types directly relate to groups of permissions, the evaluation of security contexts is the foundation of each security decision. Since apps may offer many complex functions, the policy language has to provide the flexibility of defining multiple domains with distinct privileges so that the app, according to the task it has to do, may switch to the least privileged domain needed to accomplish the job.

The app policy is specified in a module, provided by the app to describe its own types. The policy module is processed at app installation time by a component



Policy module syntax	
<i>blockStmt</i>	→ (block <i>blockId cilStmt</i> *)
<i>cilStmt</i>	→ <i>typeStmt</i>   <i>typeAttrStmt</i>   <i>typeAttrSetStmt</i>   <i>typeBoundsStmt</i>   <i>typeTransStmt</i>   <i>macroStmt</i>   <i>allowStmt</i>
<i>typeStmt</i>	→ (type <i>typeId</i> )
<i>typeAttrStmt</i>	→ (typeattribute <i>typeAttrId</i> )
<i>typeAttrSetStmt</i>	→ (typeattributeset <i>typeAttrId</i> (( <i>typeId</i>   <i>typeAttrId</i> )+))
<i>typeBoundsStmt</i>	→ (typebounds <i>parentTypeId childTypeId</i> )
<i>typeTransStmt</i>	→ (typetransition <i>sourceTypeId targetTypeId classId</i> [ <i>objectName</i> <i>defaultTypeId</i> ])
<i>macroStmt</i>	→ (call <i>macroId</i> ( <i>typeId</i> ))
<i>allowStmt</i>	→ (allow ⟨ <i>sourceTypeId</i>   <i>sourceTypeAttrId</i> ⟩ ⟨ <i>targetTypeId</i>   <i>targetTypeAttrId</i>   self⟩ <i>classPermissionId</i> +) )

Table 3.1: Application policy module CIL syntax

of the system, called *SEApp Policy Parser*, responsible to verify that the policy is correct and does not introduce vulnerabilities into the system. The addition of a policy module is managed by combining the new module with the platform policy and the previous installed ones, producing after policy compilation a single binary representation of the global policy.

In this section, we provide a description of the SEApp policy language and the restrictions each module is subject to. Policy configuration is detailed in Section 3.5, while policy compilation and runtime support are discussed in Section 3.6.

### 3.4.1 Choice of policy language

SEAndroid supports two languages for policies, Type Enforcement (TE) [189] and Common Intermediate Language (CIL) [139]. TE was the language available in the early implementations of SELinux, while CIL was later introduced to offer an easy to parse syntax that avoids the pervasive use of general purpose macro processors (e.g., M4 [107]). Another aspect that differentiates them is that, in Android, TE representations are internally converted into CIL before being compiled into the SELinux binary policy. To avoid the additional translation step being performed at each policy module installation, we decided to use CIL over TE.

### 3.4.2 Definition of types and type-attributes

CIL offers a multitude of commands to define a policy, but only a subset has been selected for the definition of an app policy module. This was done to control the impact of the policy module on the system and it may, as a side effect, facilitate the work of the app developer writing the policy.

The syntax is described in Table 3.1. To declare a *type*, the `type` statement can be used. This permits to declare the types involved in an access vector (AV) rule, which grants to a source type a list of permissible actions over a target type. AV rules are defined through the `allow` statement.

When writing a policy, there is frequently the need to assign the same set of authorizations to multiple types. To avoid the repetition of multiple `allow` declarations, it is convenient to refer to multiple types using a single entity, the *type-attribute*. Using the `typeattributeset` statement we associate with a `typeattribute` a set of types and type-attributes. Each type-attribute essentially represents the set of types that is produced by the (possibly multi-step) expansion of its definition. The semantics is that each of the types that directly or indirectly (using type-attributes) appears as the source of an allow rule will be authorized to operate with the specified permission on each of the types directly or indirectly appearing as the target. This improves the conciseness and readability of the policy.

After defining the domains with the least group of permissions necessary to fulfill the task, the developer can also configure the domain transitions using the `typetransition` statement. By doing so, it is possible to ensure that important native processes run in dedicated domains with limited privileges, leading to intra-app compartmentalization.

### 3.4.3 Policy constraints

The introduction of dedicated modules for apps raises the need to carefully consider the integration of apps and system policies. The first requirement is that an app policy must not change the system policy and can only have an impact on processes and resources associated with the app itself. To preserve the overall consistency of the SELinux policy, each policy module must respect some constraints. Since Android supports the side-loading of apps [15], we cannot rely on app markets to verify app policies. Therefore, the enforcement of constraints is done on the device, by both the SEApp Policy Parser and the SELinux environment. If any of these components raises an exception, during the verification or compilation of the policy, app installation is stopped.

To ensure that policy modules do not interfere with the system policy and among each other, a first necessity is that policy modules are wrapped in a unique namespace obtained from the package name. This is done through the `block` CIL statement, which prevents the definition of the same SELinux type twice, as the resulting global identifier is formed by the concatenation of the namespace and the local type identifier. Also, the use of a namespace specific for the policy module permits to discriminate between local types or type-attributes  $T_A$  (namespace equal

to the current app package name), types or type-attributes of other modules  $T_{A' \neq A}$  (namespace equal to some other app package), and system types or type-attributes  $T_S$  (system namespace). At installation time, the SEApp Policy Parser determines the origin of each type, with an explicit prohibition for policies to refer to types or type-attributes defined by other policy modules, while use of system types or type-attributes is subject to restrictions.

With regard to the `allow` statement, a dedicated analysis is performed by the SEApp Policy Parser. For each rule, the global origin of source and target types is determined. We refer to system origin  $S$ , when the type is directly or indirectly associated with a system type in the expansion of its definition, while to local origin  $A$  otherwise. Based on the origin of source and target of each rule, there are four cases. The case *AllowSS*, i.e., a permission with system origin both as source and target, is prohibited, as it represents a direct platform policy modification. The case *AllowAA* is always permitted, as it only defines access privileges internal to the app module. The cases *AllowAS* and *AllowSA* are more delicate.

An *AllowAS* originates when a local type needs to be granted a permission on a system type. A concrete example is shown in Section 3.3, where the `:media` process needs access to the `camera_service`. The case cannot be decided locally by the SEApp Policy Parser, therefore it is delegated to the SELinux decision engine during policy enforcement. This crucial postponed restriction depends on the constraint that all app types have to appear in a `typebounds` statement [48], which limits the bounded type to have at most the access privileges of the bounding type. As Android 11 assigns to generic third-party apps the `untrusted_app` domain, this is the candidate we use to bound the app types. If the *AllowAS* rule gives to the local type more privileges than those associated with `untrusted_app`, and at runtime these privileges are used, the SELinux decision engine identifies the policy violation and prohibits the action.

*AllowSA* rules are the key to regulate how system components access internal types. To be compliant with Android, the local types introduced by the app policy module must ensure interoperability with system services crucial to the app lifecycle. As an example *Zygote* [43], the native service which spawns and configures new app processes, can only execute processes labeled with the type-attribute `domain`, which is assigned by default to `untrusted_app`. However, giving app developers the freedom to directly define *AllowSA* rules would lead to two major issues: (i) the rules would depend on system policy internals, leading to a solution with limited abstraction and modularity; (ii) explicit *AllowSA* rules could lead to violations of the security assumptions of a system service, with the risk of introducing vulnerabilities (e.g., leading to a *confused deputy attack* [58]). For these reasons we prohibit their explicit use. To limit system types to only those already dealing with untrusted

Macro	Usage
<i>md_appdomain</i>	to label app domains
<i>md_netdomain</i>	to access network
<i>md_bluetoothdomain</i>	to access bluetooth
<i>md_untrusteddomain</i>	to get full untrusted app permissions
<i>mt_appdatafile</i>	to label app files

**Table 3.2: SEApp macros to grant permissions to local types**

content and simplifying the policy, we rely on CIL macros, a set of function-like statements that, when invoked by the SEApp policy module, produce a predefined list of policy statements. This approach permits to retain control on the rules produced, ensuring no violation of the default system policy. Also, it makes the work of the developer easier, by abstracting away system policy internal details. To preserve the interoperability with system services, third-party app functionality has been broken down into the CIL macros listed in Table 3.2. This list has been identified looking at the internal structure of the `untrusted_app` domain. With this design philosophy, the developer can grant a basic set of permissions to a type (by calling one or more macros), and then add to it fine-grained authorizations with *AllowAS* rules.

With regard to the `typeattributeset` statement, the SEApp Policy Parser uses a verification strategy similar to the one used for allow rules. First, the global origin of the type-attribute and of the set expression of types and type-attributes is determined. All statements that directly or indirectly relate to system types are blocked. This avoids implicit permission propagation from system and local types.

Similarly, for the `typetransition` statement, the SEApp Policy Parser verifies the origin of the types involved, with a prohibition for all the statements that relate to system types, as they may lead to an escalation of privileges.

## 3.5 Policy configuration

In this section, we explore the structure of application policy modules. Before describing the content of SEApp configuration files, we give a short description of how SEAndroid defines the security contexts of processes, files and system services. There are strong similarities between the structure of system and app policies. Indeed, we designed our solution as a natural extension of the approach used to protect the system. Also, our design maintains full backward compatibility. Developers who are not interested in taking advantage of MAC capabilities do not have to change their apps.

### 3.5.1 SEAndroid policy structure

Compared to a traditional Linux implementation, Android expands the set of configuration files where SELinux [31] security contexts are described, because a wider set of entities is supported. SEAndroid complements the common SELinux files (i.e., `file_contexts` and `genfs_contexts`) with 4 additional ones: `property_contexts`, `service_contexts`, `seapp_contexts` and `mac_permissions.xml`. Also, the implementation of the SELinux library (*libselinux*) [190] has been modified introducing new functions (to assign domains to app processes and types to their dedicated directory). We concisely describe the role of SEAndroid context files.

#### Processes

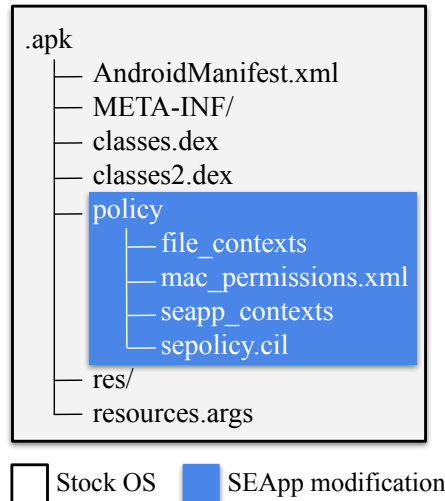
With reference to app processes, Android assigns the security context based on the class the app falls in. The specification of the classes and their security labels are defined in the `seapp_contexts` policy file. Most classes state two security contexts: one for the process (`domain` property) and the other one for the app dedicated directory (`type` property). A number of *input selectors* determine the association of an app with a class. Among these, `seinfo` filters on the tag associated with the X.509 certificate used by the developer to sign the app. The mapping between the certificate and the `seinfo` tag is achieved by the `mac_permissions.xml` configuration file. Since the enumeration of all third-party app certificates is not possible a priori, all third-party apps are labeled with the `untrusted_app` domain by default.

#### Files

SELinux splits the configuration of security contexts of files between `file_contexts` and `genfs_contexts`, with the former used with filesystems that support extended file attributes (e.g., `/data`), while the latter with the ones that do not (e.g., `/proc`). To apply `file_contexts` updates, two approaches are available: either rebuild the filesystem image, or run *restorecon* operation on the file or directory to be relabeled (this is the default method used by permissioned system processes). Conversely, to apply `genfs_contexts` changes, a reboot of the device or a sequence of filesystem *un-mount* and *mount* operations has to be performed.

#### Services

Unlike what happens for system processes, a system service requires the assignment of a security context to both its processes and its *Binder* [30], to be fully compliant with SEAndroid. The *Binder* is the lightweight inter-process communication primitive bridging access to a service. Its retrieval is enabled by the *servicemanager*, a



**Figure 3.3: SEApp policy structure**

process started during device boot-up to keep track of all the services available on the device. Based on the labels specified in the `service_contexts` file, it is then possible to control which processes can register (*add*) and lookup (*find*) a *Binder* reference for the service, and therefore connect to it. However, since *Binder* handles resemble tokens with almost unconstrained delegation, denying a process to get the *Binder* through the *servicemanager* does not prevent the process from obtaining it by other means (e.g., by abusing other processes that already hold it). Furthermore, preventing a process from obtaining a *Binder* reference prevents the process from using any functionality exposed by the service.

### 3.5.2 SEApp policy structure

Developers interested in taking advantage of our approach to improve the security of their apps are required to load the policy into their Android Package (APK). A predefined directory, `policy`, at the root of the archive, is where the SEApp-aware package installer will be looking for the policy module (see Figure 3.3). Inside this directory, the installer looks for four files (which we refer to as *local*), that outline a policy structure similar to the one of the system. Specifically, the developer is able to operate at two different levels: (i) the actual definition of the app policy logic using the policy language described in Section 3.4 (in the local file `sepolicy.cil`), and (ii) the configuration of the *security context* for each process (in the local files `seapp_contexts` and `mac_permissions.xml`) and for each file directory (in the local file `file_contexts`).

## Processes

SEApp permits to assign a SELinux domain to each process of the security enhanced app. To do this, the developer lists in the local `seapp_contexts` a set of entries that determine the security context to use for its processes. For each entry, we restrict the list of valid input selectors to `user`, `seinfo` and `name`: `user` is a selector based upon the type of UID; `seinfo` matches the app `seinfo` tag contained in the local `mac_permissions.xml` configuration file; `name` matches either a prefix or the whole process name. The conjunction of these selectors determines a class of processes, to which the context specified by `domain` is assigned. To avoid privilege escalation, the only permitted domains are the ones the app defines within its policy module and `untrusted_app`. As a process may fall into multiple classes, the most selective one, with respect to the input selector, is chosen. An example of valid local `seapp_contexts` entries is shown in Listing 3.1, which shows the assignment of the `unclassified` and `secret` domains to the `:unclassified` and `:secret` processes, respectively.

In Android, developers have to focus on components rather than processes. Normally, all components of an application run in a single process. However, it is possible to change this default behavior setting the `android:process` attribute of the respective component inside the `AndroidManifest.xml`, thus declaring what is usually called a *remote* component. Furthermore, with the specification of an `android:process` consistent with the local `seapp_contexts` configuration, we support the assignment of distinct domains to app components. To execute the component, the developer is only required to create the proper *Intent* object [34], as she would have already done on stock Android for remote components. The assignment to the process of the correct domain is handled by the system. This design choice allows us to support Android activities, services, broadcast receivers and content providers, while avoiding changes to the *PackageParser* [37], as there are no modifications to the manifest schema.

## Files

The developer states the SELinux security contexts of internal files in the local `file_contexts`. Each of its entries presents three syntactic elements, `pathname_regexp`, `file_type` and `security_context`: `pathname_regexp` defines the directory the entry is referred to (it can be a specific path or a regular expression); `file_type` describes the class of filesystem resource (i.e., directory, file, etc.); `security_context` is the security context used to label the resource. The admissible entries are those confined to the app dedicated directory and using types defined by the app policy module, with the exception of `app_data_file`. Due to the

regex support, a path may suit more entries, in which case the most specific one is used. Examples of valid local `file_contexts` entries are shown in Listing 3.2: the first line describes the default label for app files, second and third line respectively specify the label for files in directories `dir/unclassified` and `dir/secret`.

In SELinux, the security context of a file is inherited from the parent folder, even though `file_contexts` might state otherwise. Since, for our approach, it is essential that files are labeled as expected by the developer, we decided to enforce file relabeling at creation. Therefore, a new native service has been added to the system (see Section 3.6.2). We then offer to the developer an alternative implementation of class `java.io.File`, named `android.os.File`, which sets file and directory context upon its creation, transparently handling the call to our service.

### System services

To support any third-party app, the `untrusted_app` domain grants to a process the permissions to access all system services an app could require in the `AndroidManifest.xml`. As an example, in Android 11, the `untrusted_app_all.te` platform policy file [42] permits to a process labeled with `untrusted_app` to access `audioserver`, `camera`, `location`, `mediaserver`, `nfc` services and many more.

To prevent certain components of the app from holding the privilege to bind to unnecessary system services, the developer defines a domain with a subset of the `untrusted_app` privileges (in the local `sepolicy.cil` file), and then she ensures the components are executed in the process labeled with it. Listing 3.3 shows an example in which the `cameraserver` service is made accessible to the `secret` process.

**Listing 3.1: seapp\_contexts example**

```
1 user=_app seinfo=cert_id domain=package_name.unclassified
   name=package.name:unclassified
2 user=_app seinfo=cert_id domain=package_name.secret name=
   package.name:secret
```

**Listing 3.2: file\_contexts example**

```
1 .*          u:object_r:app_data_file:s0
2 dir/unclassified u:object_r:package_name.unclassified_file
   :s0
3 dir/secret     u:object_r:package_name.secret_file:s0
```

**Listing 3.3: Granting cameraserver access to secret domain**



```

1 (block package_name
2   (type secret)
3   (call md_appdomain (secret))
4   (typebounds untrusted_app secret)
5   (allow secret camerasever_service (service_manager (find)
6     ))
7   ...
7)

```

## 3.6 Implementation

In this section, we describe the main changes introduced in Android by SEApp. We first analyze the modifications required to manage policy modules, both during device boot and at app installation. We then describe how the runtime support was realized.

### 3.6.1 Policy compilation

#### Boot procedure

Since the introduction of Project Treble [23], policy files are split among multiple partitions, one for each device maintainer (i.e., platform, SoC vendor, ODM, and OEM). This feature facilitates updates to new versions of Android, separating the Android OS Framework from the device-specific low-level software written by the chip manufacturers. Yet, each time a partition policy (i.e., a segment) changes, an on-device compilation is required.

The *init* process divides its operations in three stages [32]: (i) *first stage* (early mount), (ii) SELinux setup, and (iii) *second stage* (*init.rc*). The first stage mounts the essential partitions (i.e., */dev*, */proc*, */sys* and */sys/fs/selinux*), alongside some other partitions specified as early mounted (since Android 10 using an *fstab* file in the first stage ramdisk, in Android 9 and lower adding *fstab* entries using device tree overlays). Once the required partitions are mounted, *init* enters the SELinux setup. As the name suggests, this is the stage where *init* loads the SELinux policy. As the */data* partition, where policy modules are stored, is not yet mounted, it is not yet possible to integrate them with the policy of the system. Then, as last operation of the SELinux setup stage, *init* re-executes itself to transition from the initial *kernel* domain to the *init* domain, entering the second stage. As the second stage starts, *init* parses the *init.rc* files and performs the builtin functions listed

there, among them mounting the `/data` partition. Now, the policy modules are available, and we can produce with *secilc* [40] (the SELinux CIL compiler) the binary policy consisting of the integration among the system policy, the SEApp macros and the app policy modules. To trigger the build and reload of the policy, we implemented a new builtin function, and modified the `init.rc` to call this function right after `/data` is mounted. The policy is considered immediately after the `/data` partition is available and this ensures that the policy modules are loaded far before an application starts, making the policy not bypassable.

Even though most Android devices supporting Android 10 were released with Treble support and, therefore, execute their SELinux setup stage on the `sepolicy.cil` fragments scattered among multiple partitions, *init* still supports the use of a legacy monolithic binary policy. For compatibility towards devices using a monolithic binary policy, additional changes are required, as SEApp needs the system policy written in CIL to be compiled alongside with app modules. To this end, we modified the Android build process to push the `sepolicy.cil` files onto the device even for non-Treble devices. New entries in the device tree were added to make the policy segments available during *init* SELinux setup stage [35].

As previously mentioned, we decided to store the policy modules in the `/data` partition; even if this choice required us to adapt the boot procedure of the device, it smoothly integrates SEApp with the current Android design. In fact, the `/data` partition is one of the few writable partitions, it is dedicated to hold the APK the user installs, as well as their dedicated data directories and, therefore, it represents the best option to contain also the app policy modules. Moreover, whenever a user performs a factory reset, Android automatically wipes the `/data` partition, removing the customization the user made to the device configuration, including the apps. By placing the app policy modules and the apps into the same partition, a factory reset removes the policy modules as well.

## App installation

As introduced in Section 3.5.2, the developer willing to define its own policy module is expected to load it in the app package. At app installation, the *PackageManagerService* [36] inspects the APK to identify whether or not the current installation involves a policy module, by looking for the `policy` directory at the root of the archive. When the app has a policy module attached to it (see Figure 3.4), the *PackageManagerService* extracts it (❶) and uses our *PolicyModuleValidator* to verify the respect of all the constraints on `sepolicy.cil` (through the *SEAppPolicyParser*, Section 3.4) and on the configuration files (Section 3.5). In case of a violation of the constraints, the app installation stops. Otherwise, the policy module is stored within

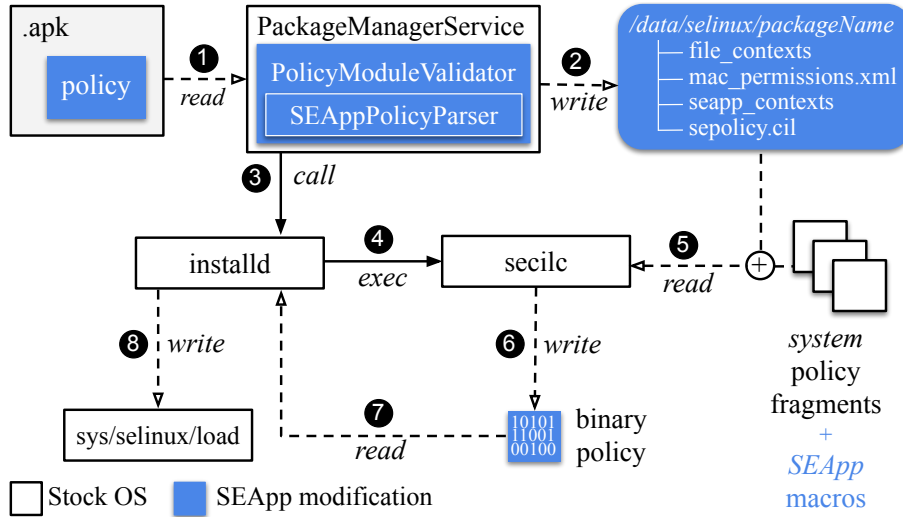


Figure 3.4: Installation process

`/data/selinux`, in a dedicated directory identified by the package name (2). Then, the `PackageManagerService` invokes `installld` [33] through the `Installer` to trigger the policy compilation with an `exec` call to the `secilc` program (3, 4). `Secilc` reads the system `sepolicy.cil` fragments, the SEApp macros and the `sepolicy.cil` fragments of the app policy modules in the `/data/selinux` directory (5), and builds the binary policy (6). When the `secilc` execution returns and no compilation errors have been raised, the binary policy is then read by `installld` (7) and loaded with `selinux_android_load_policy`, which writes the `sys/selinux/load` file (8).

To load the policy files after `init`, the implementation of SELinux in Android has been slightly modified. In particular, we modified the policy loading function within `libselinux` (function `selinux_android_load_policy`), and changed the system policy to allow `installld` to load the app policy module.

As for the policy configuration files, some changes were introduced to load the application `file_contexts`, `seapp_contexts` and `mac_permissions.xml`. `SELinux-MMAC` [41], i.e., the class responsible for loading the appropriate `mac_permissions.xml` file and assigning `seinfo` values to apks, was modified to load the new `mac_permissions.xml` specified within the app policy module. The loading of `file_contexts` and `seapp_contexts` was configured to treat system and app configuration files apart. So, SEApp-enhanced applications will load exclusively their configuration files, whereas the loading of system's and other apps' configuration files is not needed since their use is prohibited. System services and daemons, instead, load the base system configurations once, and then load the app policy module specific configuration files as they are needed. An example of this are `Zygote` and `restorecon` services, which need to retrieve at runtime `seapp_contexts` and

`file_contexts`, respectively (see Section 3.6.2).

Our implementation also supports the uninstallation of SEApp apps. The regular uninstallation process is extended with a step where the global policy is recompiled, in order to remove the impact of old modules on the overall binary policy. With reference to application updates, the native *installd* runs with the necessary permission to remove and apply new file types based on the content of the `file_contexts`.

### 3.6.2 Runtime support

In addition to the steps described above, other aspects have to be considered in order to extend SELinux support at the application layer.

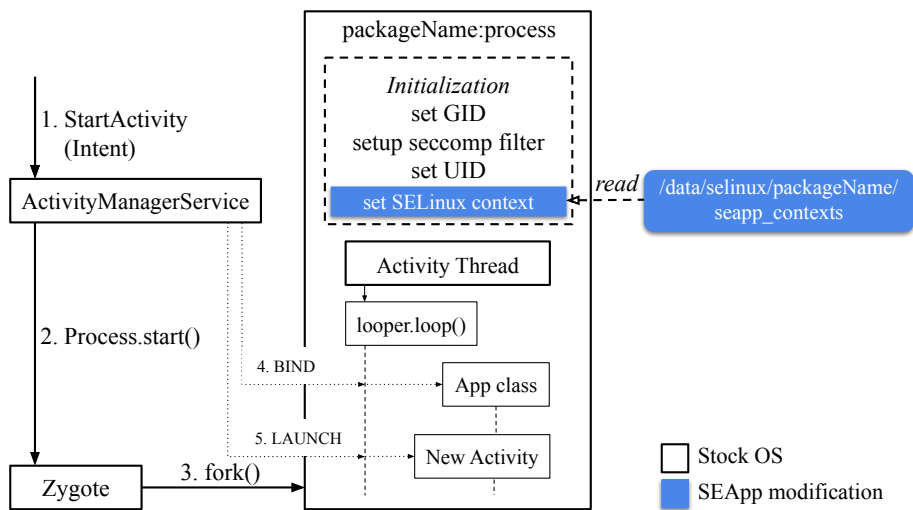


Figure 3.5: Application launch

#### Processes

Android application design is based on components. Each of them lives inside a process, and can be seen as an entry point through which the system or the user can enter the app.

To activate a component, an asynchronous message called intent, containing both the reference to the target component and parameters needed for its execution, has to be created. The intent is then routed by the system to the *ActivityManagerService* [25] via *Binder IPC*. Before delivering the intent request to the target component, the *ActivityManagerService* checks if the process in which the target component should be executed is already running; if not, the native service called *Zygote* [43] is executed. Its role is to spawn and correctly setup the new application process. To achieve this, it first replicates itself by performing a `fork()`, then, using the input provided by the *ActivityManagerService* (namely, package name, `seinfo`,

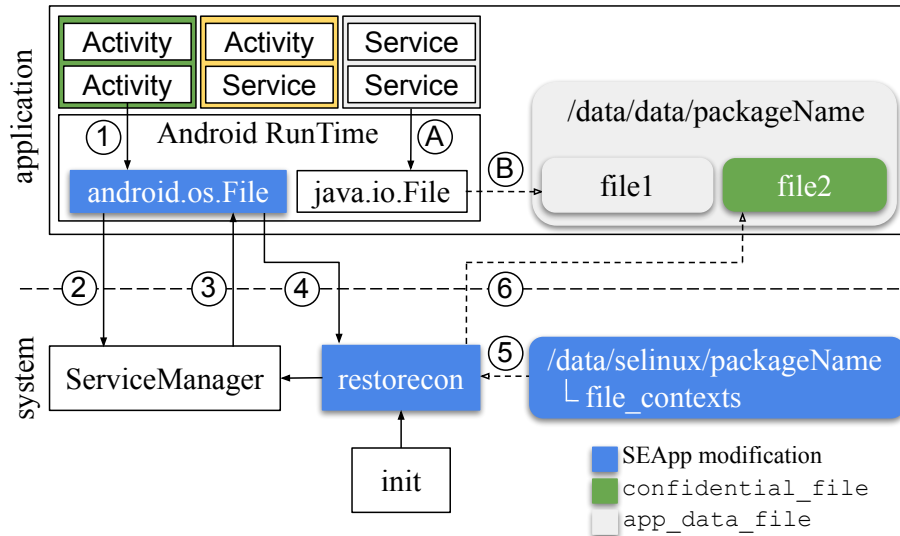


Figure 3.6: File relabeling

`android:process`, etc.), it starts configuring the process GID, the `seccomp` filter, the UID and finally the SELinux security context. We adapted the final configuration step, forcing `Zygote` to set the security context based on the `seapp_contexts` located at `/data/selinux/packageName` (i.e., the one provided by the developer for her app). Process name is used to assign the proper context to the process when it starts, before the logic of the process kicks in. In case the developer did not specify a domain, then `Zygote` uses the system `seapp_contexts` as fallback. After the correct labeling, the `ActivityManagerService` finishes the configuration by binding the application class, launching the component, and finally delivering the intent message. Figure 3.5 details the process.

This implementation design offers several benefits, including backward compatibility, support for all components, and ease of use. Indeed, a developer who wants to use our solution only has to configure some files; changes in the application code are reduced to a minimum, thus facilitating the introduction of SELinux in already existing apps.

In our study we have also explored other design alternatives, in which the developer could explicitly state a domain transition in the code, wherever she needs it. Although this category of solutions would give the developers more control over domain transitions, it also has some drawbacks. First, the developer would be expected to enforce the isolation among source and target domains managing the multi-threaded scenario, and second, this design implies granting too many permissions to the app (e.g., `dyntransition`, `setcurrent` and read/write access to `selinuxfs`). Moreover, such solution would introduce a new Android API, that would be quite delicate and, if not used correctly, it might be difficult to control.

## Files

Android applications aiming to create a file can use the `java.io.File` abstraction. Each file creation request that is generated is captured by the Android Runtime (ART) [28], and then converted into the appropriate syscall. The result is the creation of the target file, to which a security context inherited from the parent directory is assigned (see flow ①, ② of Figure 3.6). Since Android 9, the separation between files of different apps is enforced at MAC level (a unique context based on UID and SELinux category is assigned); however, all the files stored in the same app folder are labeled with the `app_data_file` type.

To make the most out of SELinux, SEApp complements Android with the implementation of a new service, which we called *restorecon* (to recall the SELinux *restorecon.c* tool). The *restorecon* service is spawned by *init* at boot, and works in its own SELinux domain. Its role is to create and label files as specified by the developer in the local `file_contexts`. To ease development, we implemented the new `android.os.File` abstraction, which exposes an interface equal to that of `java.io.File`, and transparently handles the call to our service. Figure 3.6 details the new control flow. A component running in a SEApp-enhanced process (highlighted in green in Figure 3.6) invokes `android.os.file`, and triggers a new file creation request (①). The new API first interacts with the *ServiceManager* (②) to get a handle of the *restorecon* service (③), then it interacts with the service using the AIDL [17] interface we defined for it, informing the *restorecon* of the target path (④). The *restorecon* service verifies whether the caller is the legitimate owner of the path, it reads the `file_contexts` file located at `/data/selinux/packageName` (⑤), and finally it creates the target file enforcing the correct labeling (⑥).

We also investigated three other implementation approaches: (i) change of the default security context inheritance behavior for the *ext4* filesystem, (ii) execution of the SELinux *restorecon* operation by the app, once the file is successfully created, and (iii) use of *restorecond* [39]. The first option would change the default behavior system-wide. As it might cause compatibility issues, we decided not to choose it. The second option is not ideal from a security standpoint, as it requires to grant the application too many permissions (e.g., `relabelfrom`, `relabelto`, as well as read/write access to `selinuxfs` to check the validity of the SELinux context). The third option refers to the use of *restorecond*, a system daemon that watches (inodes of) a configurable list of files and checks that they are labeled as stated in the system `file_contexts`. Although it may realize the control, *restorecond* was meant for a few system files, therefore its performance would hardly scale, especially considering that SEApp needs to manage all files created by SEApp-aware apps. Another major issue is that this approach is exposed to race conditions, because there is a delay

between file creation and its relabeling.

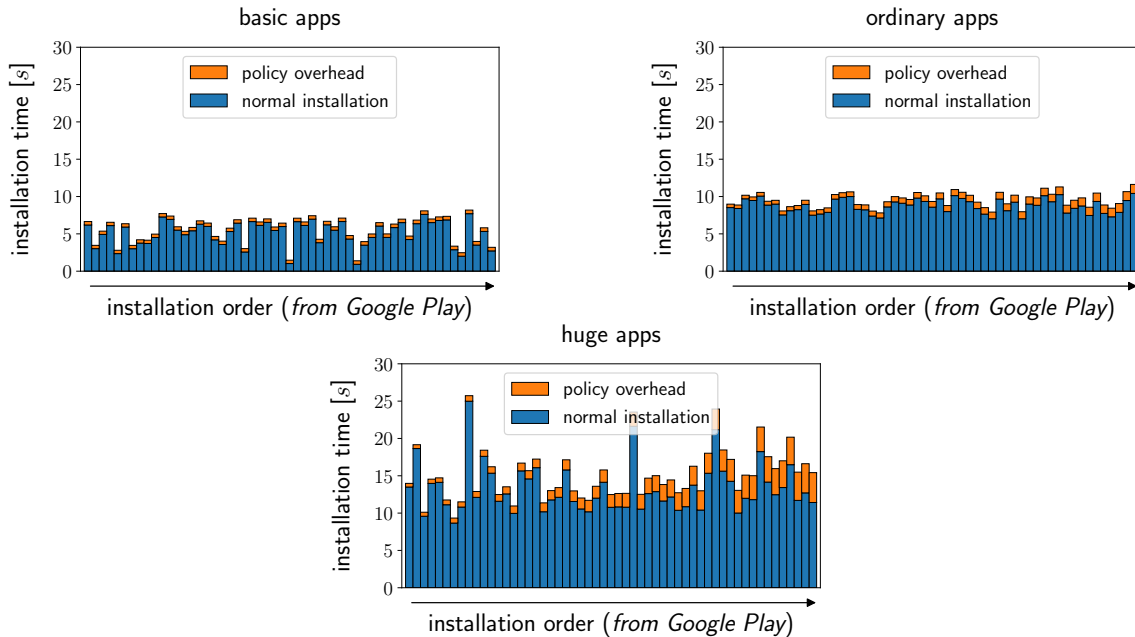


Figure 3.7: Installation time overhead for apps with different complexity

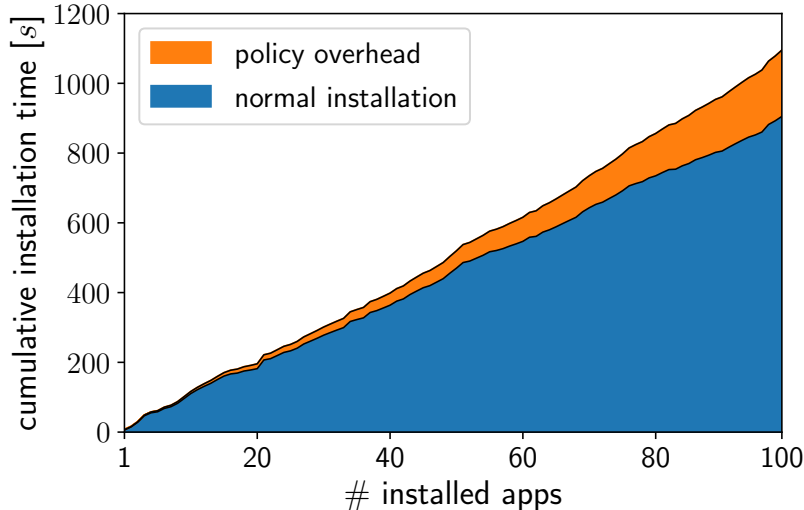
## 3.7 Experimental results

We now present a performance evaluation of SEApp. The experiments have been conducted on both Android 9 and 10, each with Linux kernel v4.9. However, all the measurements shown refer to Android 10 (release android-10.0.0\_r41). The device used to run the tests is a Google Pixel 3 (*bluepine*), in which the four *gold* cores frequency was set to 2.8 GHz, while the four *silver* ones were disabled. The change in CPU configuration has been performed to reduce the variability of measures. The confidence intervals provided have an associated confidence level of 99%.

### 3.7.1 App installation

The introduction of dedicated app policies implies further steps to be executed at app installation time, as each SEApp module has to be validated, compiled, and loaded. To evaluate the impact on performance, we wrote dedicated tests to stress the installation procedure with multiple application samples.

To build representative samples of a typical consumer scenario, we first downloaded the 150 most popular free apps from Google Play (retrieved in October 2020) [114]. The apps were subsequently divided into three buckets: *basic*, *ordinary* and *huge* apps, according to the weighted normalized average of the .apk size, the



**Figure 3.8:** Cumulative install time overhead when installing the top 100 free apps on Google Play Store with our policies

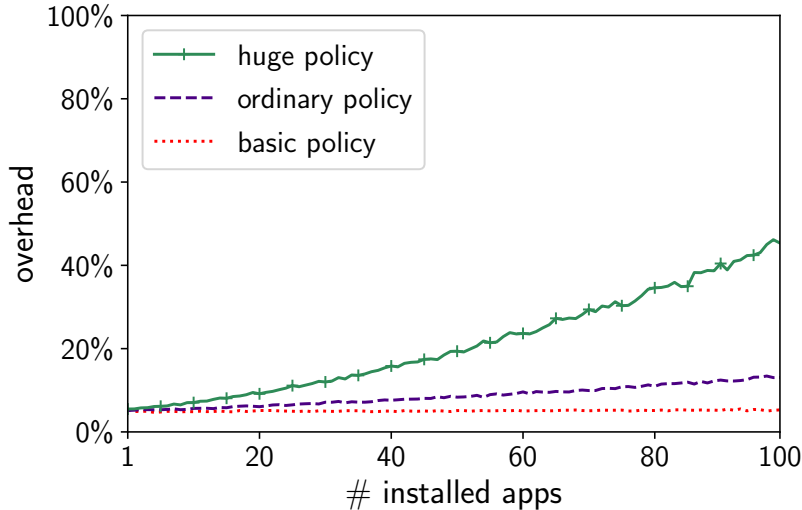
number of Android activities and the number of services. Based on the bucket, each app was equipped with one of the following policy configurations: (i) *basic*, 1 domain and 1 type per policy module, (ii) *ordinary*, 10 domains and 25 types, and (iii) *huge*, 20 domains and 100 types. The rationale is that larger apps can gain considerable benefit from the use of a large policy. The *basic* configuration mimics how third-party apps are currently handled, but with some key improvements, as it permits to define the subset of services the domain can use, and it permits to enforce app isolation, not only based on MAC category, but also through the specification of its own type. The *ordinary* and *huge* policy configurations are meant to take full advantage of intra-app isolation and flexibility via the definition of multiple domains. Each test was repeated five times, measuring the time each package took to install. The measurements were done with the `*nix date` utility.

**Test I.** To measure the overhead caused by the presence of the policy module, we performed on device installation of each of the previously described app buckets (*basic*, *ordinary* and *huge*) via Android Debug Bridge (`adb`) [26].

The results of Test I are illustrated in Figure 3.7. In detail, it shows in blue (i.e., the lower part of the bar) the time required by the system to install the current package without the dedicated policy module, while in orange (i.e., the top of the bar) the overhead caused by the presence of the policy module. The data report that a limited overhead is associated with apps with *huge* policies, at most  $3.59 \pm 0.04s$ , while *basic* and *ordinary* policy configurations exhibit a negligible slowdown, never exceeding  $1.22 \pm 0.02s$ .

**Test II.** To evaluate the overall impact of SEApp in a typical consumer scenario, we performed a test evaluating cumulative installations. At first, we repeated the





**Figure 3.9: Install time overhead for the three policy sizes**

installation of the top 100 apps on Google Play Store with the same policy configuration as in Test I (see Figure 3.8). In this case, we measured an overhead of  $20.98 \pm 1, 31\%$  on total installation time.

As explained in Section 3.6, each time a new application is installed, all policy fragments stored in the device have to be recompiled to produce the new binary policy. The installation time overhead then grows with the increase in the number of installed policy modules. To further analyze this aspect, we repeated the installation of the top 100 free apps adding to all the packages in three separate experiments the same *basic*, *ordinary*, and *huge* policy configurations. The experimental results illustrated in Figure 3.9, show that only the use of *huge* policy modules introduces a non-negligible overhead ( $45.35 \pm 2.44\%$  on total installation time). However, this policy configuration simulates an edge case, as we do not expect to find 100 of them in a real scenario. To give a comparison, the *huge* policy declares 100 types; `public/file.te`, i.e., the file used to define all the file types of the system, declares 314 types in Android 10.

In Table 3.3 we report the sizes of the overall policies for the three scenarios considered in this experiment. We report the number of MAC types, the number of produced AV rules, and the overall size in KBytes of the binary policy.

policy	#types	#avrules	KB
system	1536	29228	596
system + 100 basic	1836	47028	867
system + 100 ordinary	6036	213228	3512
system + 100 huge	15536	417228	7064

**Table 3.3: Policy size**

### 3.7.2 Runtime performance

We now evaluate the runtime overhead for an app taking full advantage of SEApp. We focus on the creation of processes and files, as they are the entities directly affected by the changes made in the implementation. The data shown refer to the creation time of each resource. The measurements have been acquired via *System.nanoTime* and have been repeated 100 times for each test. Also, all outliers diverging more than 3 standard deviations from the mean have been suppressed.

#### Processes

As discussed in Section 3.6, in SEApp the creation of a process is originated from the request of execution of an Android component. Thus, the slowdown occurs between the request for the component and the execution of the method *onCreate*, which is the time interval subject to measurement. Our evaluation is limited to activities and services, as these are the components most used by developers. Our analysis showed identical behavior for broadcast receivers and content providers, the other two components supporting the `android:process` attribute in the manifest.

Separate test cases have been identified based on the type of process that supports the component. We refer to *Local*, *Remote*, *Isolated* or *SEApp* components when we run components respectively in the current process, in another process, in another process with the `isolated_app` domain (using the *isolatedprocess* we described in Section 3.3.1), or in a package specific domain (declared in the app policy module). Furthermore, we cover *cold* and *warm* start scenarios. The *cold* start corresponds to the first time the application brings up the component, and the *warm* start to the subsequent times the app reuses a previously instantiated one.

<i>Component</i>	Cold start (ms)				Warm start (ms)			
	<i>Stock OS</i>		<i>SEApp</i>		<i>Stock OS</i>		<i>SEApp</i>	
	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$	$\mu$	$\sigma$
LocalActivity	39.102	1.094	38.689	0.980	21.052	6.046	18.685	5.001
RemoteActivity	123.468	3.176	124.649	3.526	15.722	2.682	15.933	3.256
SEApp Activity	-	-	127.356	3.542	-	-	15.188	2.394
LocalService	19.164	1.444	18.835	1.392	1.399	0.208	1.328	0.208
RemoteService	105.467	2.800	106.935	2.565	2.617	0.879	2.676	0.593
IsolatedService	103.923	2.425	104.260	3.727	-	-	-	-
SEApp Service	-	-	106.925	3.774	-	-	2.528	0.675

**Table 3.4: Cold and warm start performance for activities and services**

The results shown in Table 3.4 demonstrate that the performance of a stock version of the OS and SEApp are equivalent. Also, we observe that apps willing to benefit of the intra-app isolation feature get from the use of SEApp the same perfor-

mance they would get from the use of remote components. Our approach also proves to outperform the *IsolatedService*, as the *isolatedprocess* option forces the creation of a new process every time an *IsolatedService* that was previously *unBind*-ed is activated. This introduces a slowdown of  $102 \pm 1ms$  compared to the *SEAppService* warm start, which instead benefits from the system caching mechanism.

## Files

Alongside the usual creation method, SEApp introduces in Android the possibility of creating files with a security domain defined by the app dedicated `file_contexts`. Table 3.5 shows the time required to create a file, for each of the methods discussed. We observe no overhead on direct file creation, but the overall execution time becomes larger due to the invocation, as described in Section 3.6.2, of the *restorecon* service, which requires approximately  $374 \pm 30\mu s$ . This overhead only occurs at file creation and every subsequent operation on the file does not exhibit any performance degradation.

File creation		
<i>Test</i>	$\mu$ ( $\mu s$ )	$\sigma$ ( $\mu s$ )
Stock OS	57.077	5.174
SEApp	60.696	6.782
SEApp + <i>restorecon</i>	431.472	109.494

**Table 3.5: File creation performance**

## 3.8 Related work

In traditional desktop operating systems significant effort has been spent in retrofitting legacy code for authorization policy enforcement leveraging MAC. An approach is to place reference monitor calls to mediate sensitive access locations through the use of static and dynamic analysis [109, 147]. An evolution of this solution is the multi-layer reference monitor [119], in which the MAC policy is enforced at different levels (e.g., application, OS, Virtual Machine Manager). Another approach is to identify integrity-violating permissions through the use of information-flow analysis [170].

Android’s open source nature and popularity made it the target of careful security investigations (e.g., [7, 106, 98, 97]) and several proposals aiming at strengthen its security properties. In the following we discuss the ones that try to address app isolation and modularity, underlining the key differences with our methodology.

Our approach presents similarities with *Secure Application INTeraction (Saint)* proposed by Ongtang et al. in [158], in which the authors also try to address the issue of allowing developers to define policies that can be verified at both installation time and runtime, to better specify the permissions for each component of their app. However, since the paper has been published in 2010, *Saint* could not leverage SEAndroid [173], which was introduced later, thus the authors had to define their own Android security middleware, which would not fit into the current Android architecture [140].

*FlaskDroid* [59] defines a versatile middleware and kernel layer policy language. It is based on Userspace Object Managers (USOMs), which control access to services, intents and data stored in Content Providers. However, *FlaskDroid* does not focus on intra-app compartmentalization, a central aspect in our proposal.

*ASM* [118] and *ASF* [50] promote the need for a programmable interface that could serve as a flexible ecosystem for different security solutions. The generality of these solutions, however, requires to introduce several changes to the current Android security model.

*AppPolicyModules* [47] is another proposal that allows app developers to create dedicated policy modules. The authors focus more on how apps could use SEAndroid to better protect their resources from the system and from other apps, paying limited attention to internal compartmentalization.

*DroidCap* [76] is a recent contribution proposed by Dawoud and Bugiel, in which the authors propose to replace Android's UID-based ambient authority (DAC) with per-process Binder object capabilities. The proposal is interesting as it permits to achieve security compartmentalization between different app components. To introduce capability-based access control on files, *DroidCap* had to integrate *Capsicum for Linux* [110] in Android. Overall, *DroidCap* is a nicely engineered solution, which shares similar objectives with ours, and the two could work in parallel as they do not interfere with each other. However, as our proposal relies on SELinux and SEAndroid, which are already part of the Android security framework, our architecture appears to be more aligned with the natural evolution of the Android ecosystem.

*Boxify* [51] is a virtualization environment for Android apps, which could be used to achieve a higher level of privacy and better control over app permissions. The authors also describe how their solution could be used to compartmentalize Ads libraries to reduce the risk of sensible information leakage. Yet, since the virtualization environment acts as a mediator between the applications and the system, it extends the set of trusted components the app has to rely on.

*AFrame* [206] and *CompARTist* [120] propose to compartmentalize third-party libs from their host app using a separate process with a dedicated UID. In *AFrame* the Android Manifest is modified with the introduction of library ad-hoc permissions,

while *CompARTist* uses compile time app rewriting. Both proposals do not extend the protection at the MAC level.

To summarize, the main differences that characterize our proposal are: (i) we propose a natural extension of the role of SELinux to apps leveraging what is already used to protect the system itself, thus minimizing the impact on it, and (ii) we empower the developers while limiting the amount of changes an application must undergo in order to take advantage of our solution.

## 3.9 Conclusions

This chapter proposed an extension to the current MAC solution (SELinux) already available in Android. Developers can use SELinux to define domains that are internal to their apps, in such a way that it is possible to leverage the modules that are already providing protection to the system. By mapping SELinux domains to activities and services, developers can limit the impact that a vulnerability has on the app processes and files. We described in the chapter the changes that we introduced into Android, and our experimental evaluation shows that the overhead introduced by our proposal is compatible with the additional security guarantees.

## Availability

The implementation and the artifacts produced for the evaluation of our proposal are freely available at <https://github.com/matthewrossi/seapp>



# 4. Lightweight Cloud Application Sandboxing

## 4.1 Introduction

With the research conducted on Android it soon became clear that, while securing Android apps on the user device was very important, it was as much important to secure the cloud applications those applications interact with. Cloud applications are often built of many components, each implementing a specific set of business requirements. Components naturally evolve, their requirements may change, and as the overall scale of the application grows it can be challenging to ensure a high security standard. Many factors contribute to making this objective hard to achieve; the most common ones are: the presence of buggy components, the reliance on potentially vulnerable native libraries written with memory unsafe languages, and broad access to system resources.

Several research works have investigated the scenario (e.g., [177, 67, 6, 208]). For instance, Staicu et al. [177] explain the security problems that arise when web applications interact with native extensions. BinWrap [67] and NatiSand [6] analyze recent vulnerabilities and integrate new security measures in the Node.js and Deno runtimes to improve isolation and limit access to confidential resources. Lastly, Zimmermann et al. [208] present an extensive study of third-party dependencies, finding that large parts of the entire web ecosystem can be impacted by security issues even when individual packages are vulnerable or they include malicious code on purpose.

Recently, industry standards have also emerged to encourage organizations to proactively include new security measures. A notable example is the NIST SP 800-190 [151], which focuses on environments that adopt microservices, containers and Kubernetes. The production environment is given particular attention, with the goal of finding and stopping malicious threats in real time. The directive clearly indicates that there is a need for policies to defend against vulnerabilities that could lead to disruptions, such as modification of important files. The same regulation also provides instructions to prevent the tampering of the file system, prompting that applications and containers must be run with a set of permissions as minimal as possible, namely following the *least privilege* principle.

Powered by the recent eBPF kernel technology, frameworks like Tetragon [191]

and Falco [185] have been proposed to monitor cloud applications, identify unexpected security-relevant events, and act on them (e.g., denying access). While effective, they tend to introduce non-negligible overhead when fine-grained policy rules are used [186]. We argue that the combined use of classical operating system access control and sandboxing mechanisms may lead to a more resource efficient solution to isolate application components. For instance, the recent Landlock LSM [138] permits a process to restrict itself ensuring strong security guarantees with minimum performance footprint. Furthermore, the integration of Landlock, or an equivalent security mechanism, in cloud applications significantly mitigates the risk associated with the exploitation of a vulnerability, as the amount of resources available to a potential attacker is greatly reduced.

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

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

## 4.2 Motivation

This Section explains the threat model and the motivation.

### 4.2.1 Threat model

We assume that the code part of the cloud application (including native code executed by it) is trusted and not malicious, but potentially affected by vulnerabilities



due to bugs. In this scenario, an attacker may leverage web interfaces or programmatic APIs to send the application malicious payloads with the goal of exploiting such vulnerabilities. This attack vector may leave the application exposed to, e.g., arbitrary file read and write, file system compromise, and execution of arbitrary programs; all of which can lead to an inconsistent state of the application and its volumes. We aim at the creation of fine-grained, component-specific policies that are used by developers to gradually introduce sandboxing, mitigating the impact of vulnerabilities. This approach is complementary to the use of containers, as a compromised process running in a container can still damage all the resources available to it.

### 4.2.2 Dependency identification

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

### 4.2.3 Mitigation of bugs

Open source libraries and programs are used extensively while developing cloud applications. Examples include database drivers, media processing libraries, and encoding utilities. This software is often trusted, but it may be subject to vulnerabilities as explained in Section 4.2.1. When vulnerable third-party code is executed by the application, it can be targeted and compromised by an attacker who sends malicious payloads, as described in [11, 5, 102]. Popular cases are the Server-Side Request Forgery and Arbitrary File Read vulnerabilities found in FFmpeg and exploited against TikTok [117], and the Remote Command Execution vulnerability found in ExifTool and exploited against GitLab [74]. Other examples include: 1) CVE-2020-24020, CVE-2022-2566, CVE-2020-2499 associated with video processing software, 2) CVE-2022-1292, CVE-2022-2068, CVE-2022-2274 related to cryptographic software, and 3) CVE-2021-4118, CVE-2021-37678, CVE-2022-0845 targeting machine learning software.

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

#### 4.2.4 Performance and usability

Modern cloud applications are often deployed as Kubernetes Pod instances and executed in one or more containers. Kubernetes provides several tools to improve the security and isolation of Pods. Relevant to this scenario are the support for RBAC policies and the availability of restricted Seccomp profiles. RBAC policies by default permit to define access of human users, however, they can also be used to govern the behavior of software resources through *service accounts*. Unfortunately, these policies can only restrict access to Kubernetes APIs, therefore they are not suitable for fine-grained restriction of permissions at an application component level. The same limitation is shared with Seccomp profiles, which can limit the kernel interface available to the cloud application, but are only applied at container level [188] and cannot operate depending on the specific requested resource [123].

Recent solutions based on eBPF, like Tetragon and Falco, enable the introduction of fine-grained policy rules to overcome the previous limitations. To this end, they load into the kernel dedicated filters which are run system-wide every time a security event like the opening of a file or the execution of a program occurs. Based on the content of the policy, and therefore the filters, these frameworks can grant or deny a particular action, effectively restricting the privileges available to a cloud application. However, as mentioned in Falco's documentation [186], the main problem associated with this approach is that performance overhead can have a large variability. This is mainly due to the fact that filters are evaluated every time a certain hook point (e.g., a syscall) is triggered, and that many hooks may need to be controlled to enforce a given security policy. Furthermore, these frameworks do not assist the developer in the generation of application-specific policies.

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

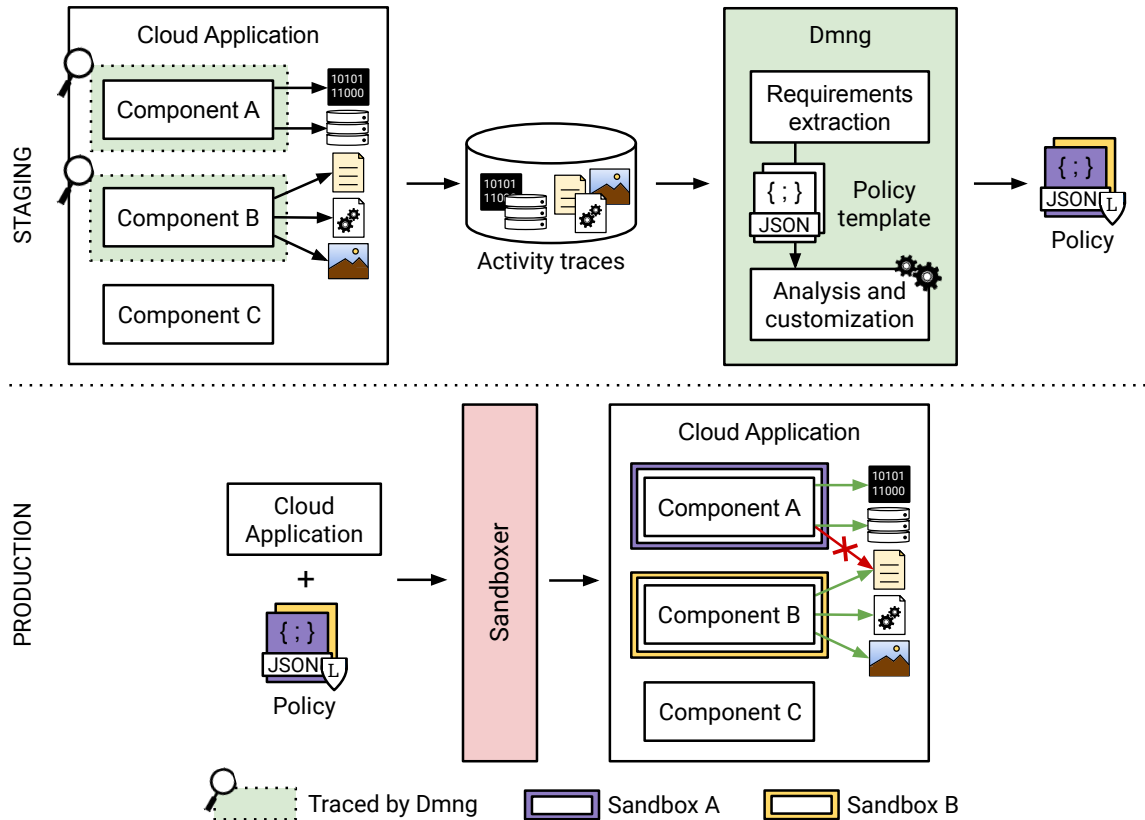


Figure 4.1: Approach overview: 1) Dmng uses probes to trace the application components A and B, 2) activity traces are saved into a SQLite DB, 3) requirements are extracted from the traces and used to build security policies, finally 4) policies are leveraged by the sandboxer to secure the application in production

### 4.3 Approach overview

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

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

shown in Figure 4.1. In detail, the tool leverages `ptrace` and `eBPF` to setup and activate temporary probes that register the actions performed by an application component, making it possible to track file-opening requests, reading and writing of data, use of shared memory and execution of native code via subprocesses and shared libraries. All these requests are automatically registered into a database and subsequently used to generate policy templates. Together with the file system path, each requirement is associated with permissions. Compatibly with Unix-like systems, three permissions are available: *read*, *write*, and *exec*. The policy templates generated by `Dmng` can then be interactively modified by the developer with the addition, modification or removal of policy rules. After the changes are committed, the policy is serialized into a JSON file and it is ready to be used to sandbox the application.

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

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

The following sections are organized as follows. Section 4.4 details the technologies used to support policy generation and implement monitoring. Section 4.5 clarifies the structure of the policy. Section 4.6 describes sandboxing through `Landlock`. Lastly, Section 4.7 showcases the mitigation capabilities and investigates the performance overhead.

## 4.4 Cloud application instrumentation

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

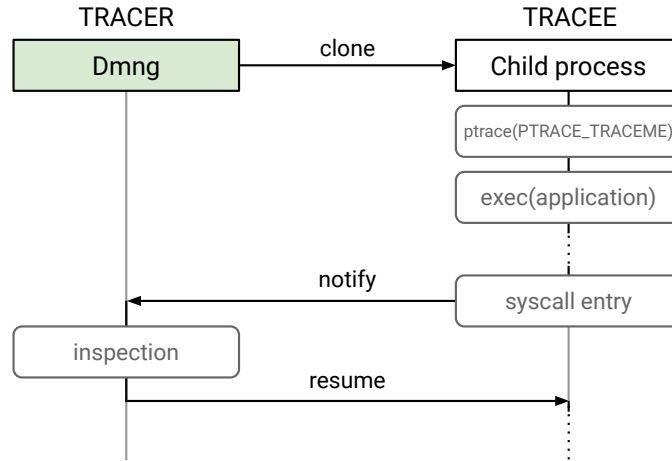
### 4.4.1 Ptrace-based instrumentation

`Ptrace` is a functionality implemented by the kernel aimed at debuggers and code analysis tools that permits a process, called the *tracer*, to control and observe the activity performed by another process, the *tracee*. In our implementation, `Dmng` acts as the tracer for any application component. To this end, it prepares a parent and a child process as shown in Figure 4.2, and then uses the `ptrace` system call to instruct the kernel that the child will be traced by the parent (through the `PTRACE_TRACEME` request). After this step is completed, `Dmng` injects into the child process the component to be run, and then starts it. While being traced, every time an event occurs, the tracee is stopped by the kernel and a notification is sent to the tracer, which has the possibility to inspect and perform changes before the execution of the tracee is resumed. `Dmng` leverages `ptrace` to capture all file system-related syscalls, effectively monitoring the requests issued to the kernel by the component. The syscalls and their arguments are recorded by the tracer and saved to the SQLite database mentioned in Section 4.3. The set of monitored syscalls includes interfaces such as `open`, `openat`, `creat`, `execve`, `link`, `linkat`, `mkdir`, and the related permission flags (e.g., `O_APPEND`, `O_CREAT`, `O_RDONLY`). It is important to point out that `Dmng` automatically captures and monitors the possible children spawned by the tracee, and it is also capable to identify the set of dynamic libraries they depend upon.

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

### 4.4.2 eBPF-based instrumentation

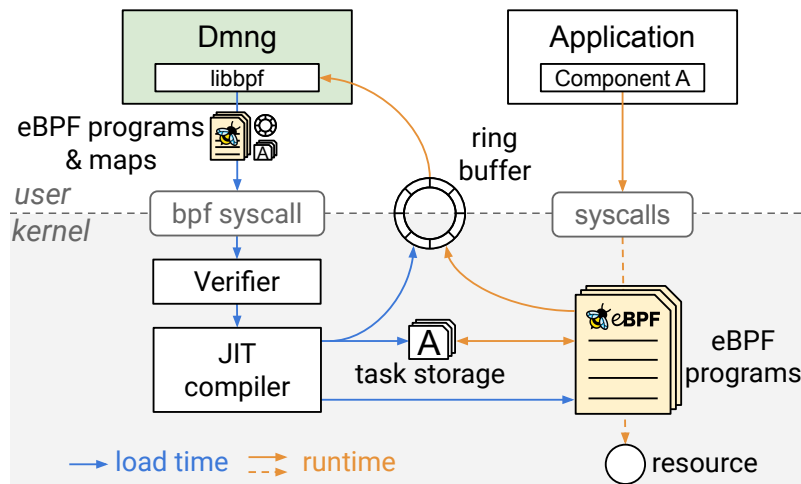
Similarly to other recent security and observability frameworks, like Cilium [183], and Tetragon [191], `Dmng` relies on the eBPF technology to implement continuous



**Figure 4.2: Dmng acts as a tracer for the application, inspecting the arguments of every syscall**

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

Dmng activates eBPF-based tracing on a given application component using its thread identifier. To this end, it leverages the *libbpf* [129] frontend to load into the kernel the eBPF programs and maps needed to perform tracing, and then starts collecting data. The process is shown in Figure 4.3. The set of eBPF programs comprises of: 1) dedicated programs to trace the application component lifetime, and 2) programs to monitor the file system-related events generated by the component. The former group of programs ensure that monitoring extends to tasks spawned through the clone system call by the component. Hence, they are attached to the *sched\_process\_fork* and *sched\_process\_exit* kernel tracepoints. Instead, the programs that record file system-related events are attached to hooks reported in Table 4.1. Whenever one of these hooks is triggered, the attached program writes the requirement path and the related permission to a ring buffer shared with the Dmng user



**Figure 4.3:** Dmng uses libbpf to load the eBPF tracing programs and maps, then it polls data from the shared ring buffer

space process. This permits Dmng to poll data regularly, and then to save it in the already mentioned SQLite database (Section 4.3).

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

## 4.5 Policy

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

**Policy structure** The policy obtained from Dmng is a JSON file structured as a list of objects as shown in Listing 4.1. For each of them, a field *policy\_name* identifies the application component the policy applies to, while the sections *read*, *write* and *exec* are used to configure the related permissions. The structure of the policy is flexible, and for each object only the field *policy\_name* is required. Since the policy implements a *default-deny* model, an object that does not list any section in its body has no runtime permission, hence the corresponding component cannot access any file system resource. Listing 4.1 shows an example in which a component called *filter* is granted execution access to the Awk program (together with its shared libraries) to process the read-only *users.csv* dataset.

**Table 4.1: List of file system traced hook points**

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

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

A useful feature implemented by Dmng is *permission pruning*. This function takes advantage of the structure of the Directory Tree [134] to help the developer lower the number of requirements in the policy by reducing the granularity of permissions. The reduction in granularity is based on a *pruning goal* set by the developer that represents the desirable maximum number of policy rules associated with an application component. To implement this feature, Dmng first uses the policy template to build a trie (or prefix tree), then starts pruning its branches iteratively following a best effort approach, until the pruning goal is achieved. The rationale is that there are areas of the file system in which fine granularity brings strong security guarantees (e.g., `/lib`), but there are also many other areas where fewer rules make



Listing 4.1: Example of JSON file with single policy

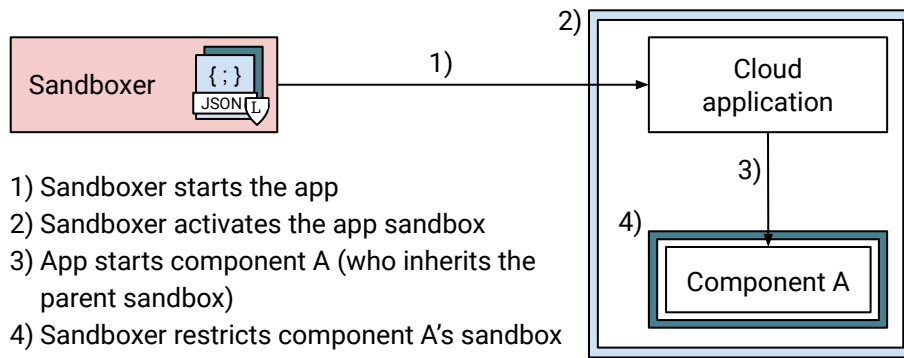
```

1 {
2   "policies": [{
3     "policy_name": "filter",
4     "read": [
5       "/lib/x86_64-linux-gnu/libsigsegv.so.2",
6       "/lib/x86_64-linux-gnu/libreadline.so.8",
7       "/lib/x86_64-linux-gnu/libmpfr.so.6",
8       "/lib/x86_64-linux-gnu/libgmp.so.10",
9       "/lib/x86_64-linux-gnu/libm.so.6",
10      "/lib/x86_64-linux-gnu/libc.so.6",
11      "/lib/x86_64-linux-gnu/libtinfo.so.6",
12      "/lib64/ld-linux-x86-64.so.2",
13      "/usr/bin/awk",
14      "users.csv"
15    ],
16    "exec": [
17      "/lib/x86_64-linux-gnu/libsigsegv.so.2",
18      "/lib/x86_64-linux-gnu/libreadline.so.8",
19      "/lib/x86_64-linux-gnu/libmpfr.so.6",
20      "/lib/x86_64-linux-gnu/libgmp.so.10",
21      "/lib/x86_64-linux-gnu/libm.so.6",
22      "/lib/x86_64-linux-gnu/libc.so.6",
23      "/lib/x86_64-linux-gnu/libtinfo.so.6",
24      "/lib64/ld-linux-x86-64.so.2",
25      "/usr/bin/awk"
26    ]
27  }]
28 }

```

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

**Coverage and effectiveness** To generate the policy templates, Dmng registers the activity performed by the application while a set of test cases is executed. This approach is similar to the one proposed by the Slim toolkit [172] to identify the dependencies of a container and minify its image, and to the one followed by the Google Sandbox2 utility [112] to retrieve the requirements of programs distributed as ELF files. However, test-based policy generation can be subject to coverage issues if the set of test cases is not exhaustive. Another aspect worth mentioning is that applications poorly structured may benefit less from the isolation properties pro-



**Figure 4.4: Landlock sandbox setup and inheritance**

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

## 4.6 Application sandboxing

In this chapter we implement the sandbox leveraging Landlock, an unprivileged sandboxing mechanism officially merged into the Linux kernel in 2021 (version 5.13), with the goal of mitigating the security impact of bugs and unintended or malicious behavior in user-space application. The main reasons why Landlock was preferred to alternative sandboxing solutions such as Google Sandbox2 [112] are: 1) it does not rely on a proxy to implement the restrictions, hence it ensures low overhead at runtime, and 2) it is directly implemented within the kernel, thus it provides strong security guarantees. This section clarifies how policies are enforced with Landlock. Furthermore, it explains how `rwX` policy rules are translated into the Landlock permission model, and how restrictions are inherited by new components dynamically spawned at runtime.

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

restrictions, a call to `landlock_restrict_self()` is performed. The process is illustrated in Figure 4.4.

An important property defined by Landlock is *policy inheritance*. Whenever a new component is dynamically spawned by the application in a child process, it automatically inherits the restrictions set on the parent. Moreover, after a ruleset has been activated, no new permissions can be granted to a component, as the ruleset can only be further restricted. Figure 4.4 shows the policy inheritance process for a generic application. The figure also shows how the inherited ruleset is further narrowed with a subsequent call to `landlock_restrict_self()` by the component.

## 4.7 Experiments

This section presents our experimental evaluation. In the first part (Section 4.7.1), we show the benefits coming from the introduction of sandboxing in cloud applications. In detail, we reproduce a sample of CVEs affecting open source software, and showcase how the sandbox mitigates the exploits. In the second part (Section 4.7.2), we analyze the performance overhead. Specifically, we implement an application that performs various operations on media resources, and then evaluate the degradation of latency when sandboxing and eBPF-based monitoring are activated. The tests have been executed on a workstation with Arch Linux, kernel version 6.4, an AMD Ryzen 5 7600X CPU, 32 GB RAM, and 1 TB SSD.

### 4.7.1 Mitigation of vulnerabilities

To demonstrate the importance of the introduction of sandboxing, we selected the sample of high severity vulnerabilities reported in Table 4.2. These vulnerabilities affect software extensively used in cloud application development such as FFmpeg, ImageMagick, OpenSSL and Exiftool. For each of them, we installed on the system a vulnerable version of the program or library, and then verified it was exploitable using public Proof of Concepts when the input is sent through programmatic APIs or web interfaces. Subsequently, we leveraged Dmng to generate least privilege policies as explained in Section 4.4. Finally, we repeated the execution of the previous tests starting each vulnerable component with our sandboxer. When benign input was submitted to the application, we were successfully able to conduct the tests without loss of functionality. When instead malicious input was used, Landlock correctly blocked the exploit, limiting access to only resources listed in the policy. We highlight that, in general, similar protections extend to a broader set of CVEs.

**Table 4.2: Sample of CVEs reproduced in our evaluation**

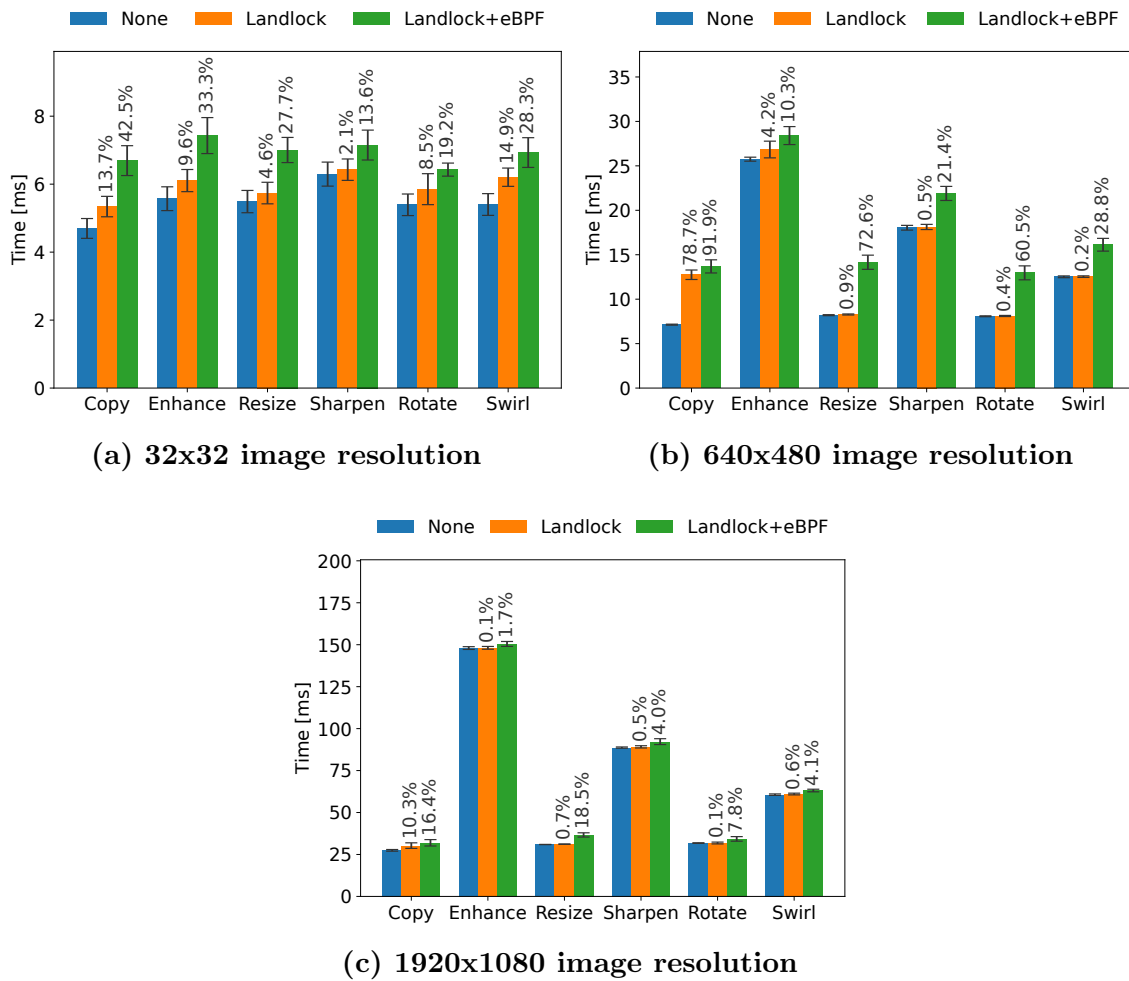
CVE	Software	Version	Description
CVE-2016-1897 CVE-2016-1898	FFmpeg	v2.x	A crafted AVI video is used to read arbitrary files
CVE-2020-29599	ImageMagick	v7.0.10-36	A bug in the PDF codec enables arbitrary code execution
CVE-2022-1292	OpenSSL	v3.0.2	Improper sanitisation allows command injection
CVE-2021-22204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

### 4.7.2 Overhead

As mentioned in Section 4.2, an important use case for cloud applications is represented by services that handle media resources such as videos, photos and audio. Therefore, to evaluate the overhead associated with our approach we implemented a Rust application that, upon receiving a request, leverages third-party software to apply several transformations on a media resource. Our goal is to measure the latency, or rather the time taken by the application to perform a given operation. Three test configurations are used: 1) no sandboxing is applied, hence no protection, 2) sandboxing is enabled leveraging our Landlock-based sandboxer, and 3) sandboxing is enabled plus eBPF-based continuous monitoring provided by Dmng is activated. Each operation is repeated 1000 times, and the measures are reported with 95% confidence intervals. Moreover, we focus on the server-side execution time, hence we do not consider the delay introduced by the network, which may make harder to visualize latency degradation for short-lived operations.

The first set of experiments focuses on image processing. In detail, the application leverages *convert* to copy, enhance, resize, sharpen, rotate and swirl images with 32x32, 640x480 and 1920x1080 resolutions. The results are shown in Figure 4.5. Inevitably, sandboxing introduces a slight degradation of latency compared to a scenario without protection. However, the overhead is non-negligible only for short-lived operations that last less than 10 ms, a duration that is considerably less than the average network delay. When instead eBPF-based monitoring is enabled, the data show worse latency degradation, especially for operations that last less than 50 ms. Remarkable is the case 640x480, in which the average operation overhead associated with eBPF-based monitoring is 47.6%.

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

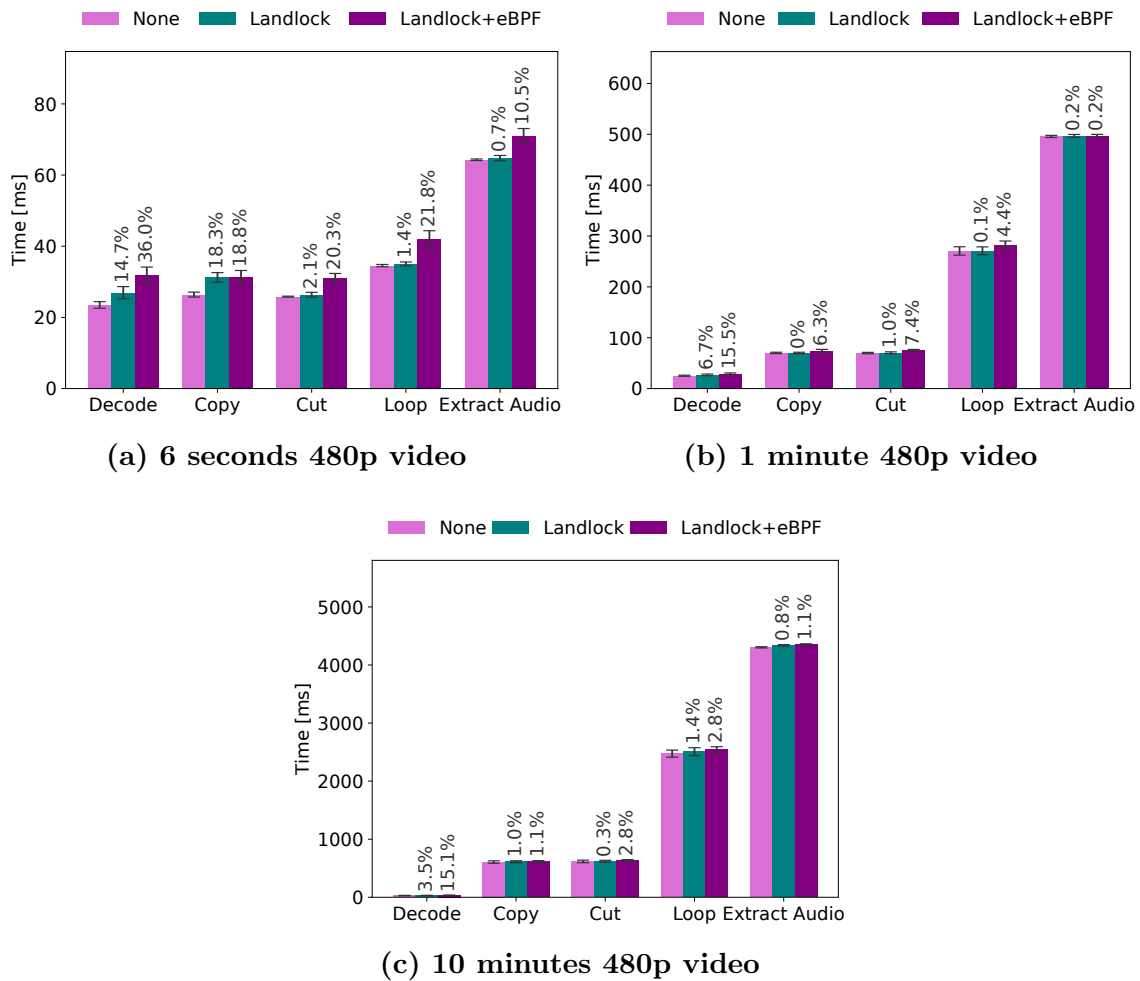


**Figure 4.5: Latency associated with various operations for an image processing application**

The results are shown in Figure 4.6. The overhead introduced by sandboxing is perceptible only for the decode and copy operations on the shortest video (6 seconds). However, it never exceeds 18.3%. As expected, the overhead becomes practically negligible for operations that take longer than 100 ms. The same considerations extend to the eBPF-based monitoring, which again confirms to be associated with more degradation compared to the Landlock-only solution.

## 4.8 Related work

Several research works have highlighted the importance of sandboxing and isolation techniques in modern software [126, 203, 55, 149, 167, 10]. Indeed, sandboxing plays a key role in many platforms (e.g., Linux, Windows, iOS, Android), and is integrated in widely used software such as browsers (e.g., Chrome [68], Firefox [145]), service



**Figure 4.6: Latency associated with various operations for a video processing application**

managers (e.g., Systemd [82]) and document viewers (e.g., Acrobat [8]).

With specific reference to the cloud scenario, many recent proposals have investigated the use of sandboxing to mitigate vulnerabilities [6, 5, 4, 11, 177, 67, 208, 102]. In *NatiSand* [6] and *Cage4Deno* [5] the authors modify the Deno runtime to control the permissions available to applications running native code. *BinWrap* [67] proposes similar measures to restrict the permissions available to Node.js native add-ons. *SandDriller* [11] describes an approach based on dynamic analysis for detecting sandbox escape vulnerabilities for Node.js applications. Zimmermann et al. [208] and Ferreira et al. [102] study the risks associated with vulnerable or malicious third-party dependencies and propose possible install (and update) time countermeasures. In general, all the previous proposals address the issues associated with a specific runtime ecosystem. Conversely, we aim to secure applications independently of their build toolchain or runtime.

Virtual machines and containers are two fundamental technologies in modern cloud architectures. Both permit to virtualize resources and execute applications in an isolated environment. Virtual machines ensure stronger security guarantees at the cost of higher resource utilization with respect to containers. The main reason is that applications executed in separate virtual machines have a distinct set of resources and do not share the same kernel [65, 64]. With specific reference to our scenario, both these technologies are associated with coarse granularity. Indeed, when working with them developers grant the application access to volumes rather than single resources. So, we provide a complementary approach to enable the introduction of fine-grained, per-resource access rules.

Modern industrial platforms like Cilium [182] and Falco [185] rely on eBPF as the primary means to enforce security policies in cloud applications. Cilium provides networking, observability, and security functions for container workloads, while Falco implements a threat detection engine for clusters. Both solutions are enterprise-oriented, hence the developer may find difficult to set up fine-grained policies leveraging them. Moreover, as already mentioned in the chapter, the performance of eBPF-based solutions is associated with large variability when fine-grained rules are used [186]. Therefore, we propose to complement these solutions by assisting the developer in the generation of least privilege security policies and using recent sandboxing technologies like Landlock, to reduce the overhead and strengthen the security boundary of the application.

## 4.9 Conclusions

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

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

## Availability

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



# 5. Enhancing the Sandbox of WebAssembly Runtimes

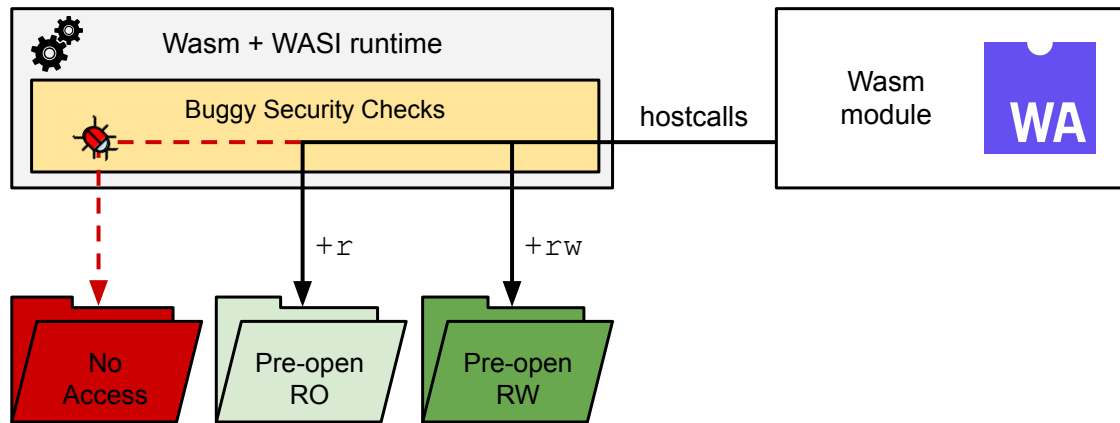
## 5.1 Introduction

While effective in restricting filesystem access to an entire microservice, the approach described in the previous chapter can be further improved by considering the specifics of emerging technologies that enable finer-grained compartmentalization of cloud applications.

WebAssembly (Wasm) [116] is a popular binary instruction format that enables the execution of untrusted code in a safe, isolated environment. Moreover, it is a portable compilation target for different languages, and can be executed efficiently on a wide range of platforms without the need of dedicated hardware. Wasm was originally meant to be run inside web browsers, but given the considerable advantages it brings, many runtimes that allow execution in standalone mode have been developed recently. Popular examples are Wasmtime, WasmEdge, Wasmer, and WAMR.

To answer the developers' need to access resources of the host system from within the runtime, a standardization effort called WebAssembly System Interface (WASI) [202] 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 [200], 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 [146]. 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 [124, 56, 128] proved the approach to be error-prone, leaving the system unprotected when a vulnerability was introduced in a hostcall wrapper (Figure 5.1). Moreover, this approach provides limited flexibility, as it is associated with directory-based granularity instead of file-based. Lastly, in



**Figure 5.1: 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)**

order to audit the policy regulating resource access, one must find the permissions by looking at the code. We claim that there is no practical advantage in having several implementations of the same access control checks for different runtimes. Our idea is to replace the user-space runtime-specific security checks with a single in-kernel implementation that leverages eBPF [187]. There are considerable advantages in doing so: (i) it permits to decouple the implementation of hostcall wrappers and the access control details, minimizing the risk of bugs [125, 167, 5], (ii) it enables the introduction of per-module policies with file-based granularity, and (iii) it fulfills Wasm’s promise of portability as eBPF programs are portable across different kernel versions [148] and also operating systems, thanks to Microsoft’s undergoing effort to port eBPF to Windows [143].

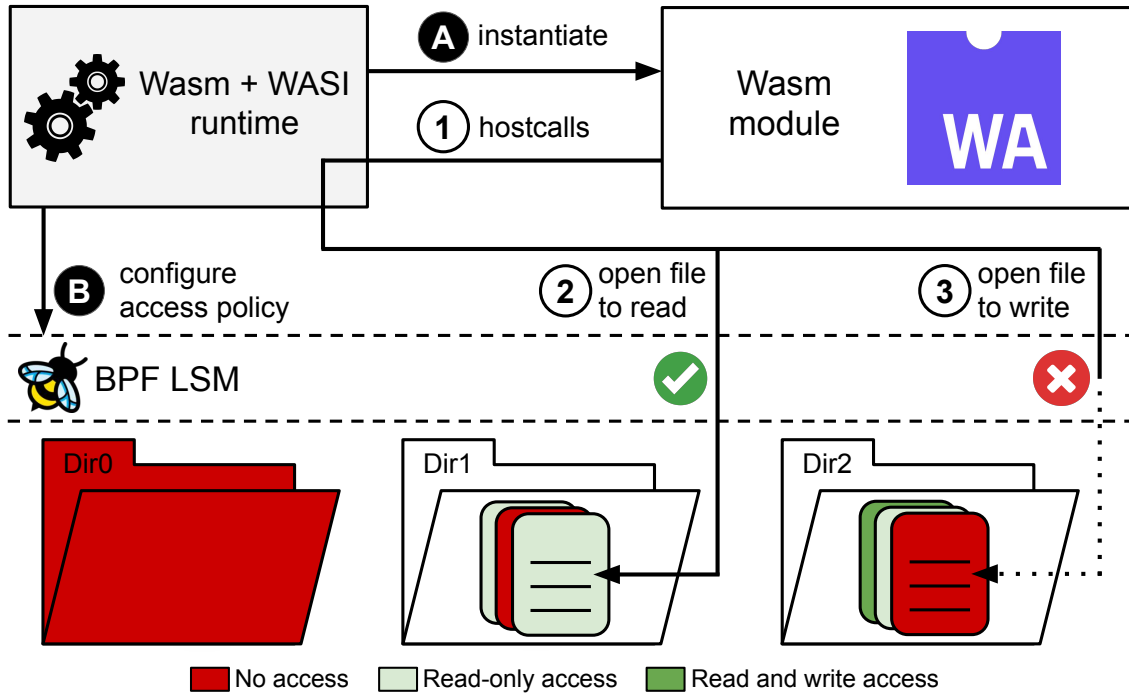
## 5.2 Threat model

Our assumptions reflect the threat model employed by Wasm runtimes. We assume that the code executed by the runtime is either untrusted or it is trusted but potentially affected by security vulnerabilities due to bugs. The goal of the attacker providing the code is to bypass the security checks enforced by the runtime to get access to the host file system. To fulfill this objective, the attacker can leverage the interface provided by WASI and send any argument. Runtime escapes caused by memory corruption or alteration of the program flow are out of scope of our work, since protection can be provided by other existing solutions (e.g., [56]).

## 5.3 Architecture

Our analysis starts from the scenario illustrated in Figure 5.1. Currently, WASI-compliant runtimes implement dedicated user-space wrappers to enforce the security boundaries of hostcalls. File system access is granted by the user on a set of pre-opened directories that are specified via CLI before the Wasm module is run (e.g., with the `--dir` option). We follow a similar approach, asking the user to state the permissions of each Wasm module in a JSON policy file. Contrary to existing runtimes, permissions can be granted with file-based granularity. Three permissions are available: (i) `read` to open and read a file, (ii) `write` to modify, truncate and append content to a file, and (iii) `delete` to remove the file. When permissions are related to a directory, `read` translates to listing its content, `write` allows to create and delete files within it. We extended the Wasmtime and WasmEdge runtimes to load the policy at startup, and, instead of pre-opening the directories available to the Wasm module, we enforce the policy with eBPF. eBPF code is split into programs attached to a kernel- or user-space function called *hook point* and executed whenever the hook is reached. Programs have visibility of function parameters, they can persist state and share it with user space using *maps*, and most of all they can enforce security checks based on this information. Once the policy is encoded inside the map and the eBPF programs are loaded, the runtime instantiates the Wasm module selected by the user (arrow **A** in Figure 5.2). At this stage, the modified runtime invokes a dedicated user probe specifying as a parameter the policy that confines the loaded Wasm module (**B**). The argument is captured by a dedicated eBPF program that also annotates the identifier of the thread running the Wasm interpreter in a tracing map. We highlight that the policy is activated before the runtime executes the module (i.e., before untrusted code is interpreted). The consequence is that, from this point on, all the hostcalls performed by the Wasm module are restricted by our eBPF programs (arrows ①, ②). The eBPF programs that make the security decisions are evaluated every time a file-related kernel security hook is reached (e.g., `security_file_open`), and any access decision is enforced at kernel level. When an unauthorized request is performed by the Wasm code (③), the related eBPF program detects the violation and denies the request, returning to the caller a permission denied error. When the execution of untrusted Wasm code terminates, another eBPF program is responsible for removing the access restriction from the thread executing the Wasm runtime. No further intervention from the runtime is required, as the maps and the eBPF programs are automatically removed from the kernel 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 [142], wrong



**Figure 5.2: eBPF-based restriction of Wasm modules.** The runtime instantiates the Wasm module (A), and configures the associated policy calling the traced user probe (B). After the Wasm module is run, all the hostcalls issued by the program (1) are restricted by eBPF (2, 3)

directory removal [69]). Then, by leveraging kernel hook points [187], our approach allows the runtime developer to focus on the interaction between Wasm code and the memory unsafe system call, leaving aside authorizations and policy-related issues. Lastly, access constraints can be audited by simply looking at the JSON policy, instead of inspecting the code.

## 5.4 Experiments

To investigate the overhead introduced by our solution we implemented it in WasmEdge and Wasmtime, two industrial state-of-the-art Wasm runtimes. The evaluation has been performed in the following test environment: an Ubuntu 22.04 LTS server powered by an AMD Ryzen 2950X CPU with 16 cores, 128 GB RAM, and 2 TB SSD. In order to assess the performance, we tested one of the most popular binaries that can be compiled to Wasm with support to WASI: `utils coreutils`, the porting of the `coreutils` in Rust [184]. First, we compiled the `coreutils` with the `wasm32-wasi` target, and applied runtime-specific optimization (with `wasmedgec` [199] for WasmEdge and with `wasmtime compile` [12] for Wasmtime) to further speed up the code. Then, we reproduced the benchmarks reported in the

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

coreutils repository, with the exception of those that are not portable to WASI due to temporary lack of support (e.g., the `dd` utility needs to spawn threads, a feature that is yet to be implemented [99]). Finally, we repeated the experiments with our protection in place. The Hyperfine benchmarking tool [75] was used to log measures, and 1000 runs were performed (with 100 warmups). As shown from the results in Table 5.1, our approach introduces a limited overhead, ranging from an additional 0.12% to 6.30% for WasmEdge, and from 0.28% to 14.29% for Wasmtime. As expected, the highest overhead is experienced by short-living utilities (e.g., `head`). We also observe that there are notable differences between the WasmEdge and Wasmtime test execution time for some utilities (e.g., `ls` and `cut`); from our analysis these differences are mostly caused by the specific post-compilation optimizations.

## 5.5 Related work

There are several successful solutions that leverage Wasm to sandbox untrusted code [149, 164, 108]. RLBox [149] is a framework that facilitates the isolation of third-party libraries in pre-existing software. eWASM [164] optimizes the execution of Wasm in embedded systems with constrained resources. Sledge [108] enables efficient Wasm-based serverless execution on the edge. The use of our approach for restricting access to the file system within these frameworks can strengthen their security assurance.

The memory safety guarantees of Wasm depend on the runtime implementation [128]. Hence, Bosamiya et al. [56] explore the problem of producing provably safe sandboxes. WaVe [124] explains that any interaction with the unsafe interfaces exposed by WASI can introduce security and safety violations. Thus, the

authors proposed a verified secure runtime system implementing WASI. However, both works require to redesign the runtime toolchain, while our solution can be directly integrated into existing runtimes.

The academic and industrial communities have investigated the use of eBPF for the isolation of software [104, 103, 183, 185]. *BPFBox* [104] and *BPFContain* [103] use an eBPF daemon to confine processes and services. *Cilium* [183] provides eBPF-based networking, observability and security for container workloads. *Falco* [185] enables lightweight threat detection in the cluster. These solutions highlight the potential of eBPF, and provide a simple and flexible confinement of system resources. However, they focus on containers or services, while our solution aims at enforcing fine-grained per-sandbox policies.

## 5.6 Conclusions

The results achieved by our approach are promising: not only it permits to introduce fine-grained policies to restrict file system access, it is also associated with a limited overhead which is aligned with the needs of a modern sandbox. The protection is currently applied only to the file system, but our approach has the potential to be extended also to network sockets, which are in the first stage of the standardization process [52]. We believe this could be an interesting line of research for future work.

# 6. Native Code Sandboxing for JavaScript Runtimes

## 6.1 Introduction

Nowadays the use of JavaScript and TypeScript for the development of cloud applications is quite established in the industry. In this chapter, we highlight its security features and limitations. Then, we propose NatiSand, a novel way to control, with fine granularity, access to system resources by native dependencies of cloud applications written in runtime-based, interpreted languages.

JavaScript (JS) and TypeScript (TS) are popular choices for the implementation of web applications. This success is motivated by their flexibility, since both are simple to use for the development of frontend and backend services, and by the vast ecosystem of open source packages that are available. For instance, the sole *npm registry* collects more than 1.3 million packages [152].

The execution of JS code on the server-side is enabled by a *JS runtime*. Since its introduction in 2009, Node.js [162] has been the de facto solution selected by developers, but recently Deno [87] and Bun [62] have received considerable attention by the community. While the three platforms provide distinctive features, they all depend on a key external component, namely the *JS engine*, V8 [195] in the case of Node.js and Deno, JavaScriptCore [44] in the case of Bun. The engine is a sophisticated software that securely renders the JS code in an isolated sandbox. Runtimes extend the engine providing components to access resources and functions that are not directly available to the web application from within the sandbox [159, 89]. Prominent examples are the functions to access the network and to read/write the filesystem. Runtimes also provide support for the execution of native code – i.e., running binary programs installed on the host operating system and calling functions from the available shared libraries.

The support provided by the runtime for the execution of native code greatly simplifies the work of the developer building the backend of a web application. However, the APIs enabling access to system resources and the execution of native code also raise security concerns, since they effectively break the isolation between the JS application and the host OS. The ability to control the resources accessible to a JS program was indeed one of the reasons that led to the creation of Deno in 2018 [166], and the solution identified by the community was to configure the resources available

to an application with simple permission flags [90]. This change also influenced the design of Node.js, which introduced a similar flag-based control model<sup>1</sup> two years later [160]. 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 [90].

Previous research [177, 102] has already shown that frequently JS modules depend on components written in native languages such as C or C++. The reuse of existing utilities permits to take advantage of popular high performance libraries and, in addition to performance, it minimizes the cost of development. Notable examples are: the `node-sqlite3` [192] and `deno-sqlite3` [92] database drivers; modules to perform image/video conversions, such as `sharp` [156], `fluent-ffmpeg` [153] and `gm` [154]; OCR engines like Tesseract [181]; and the cryptography modules relying on `bcrypt` [155]. The 2022 State of Open Source Security [175] claims that each open source JS project relies on an average of 174 third-party dependencies; also, each project is estimated to be affected by 40 vulnerabilities when its dependencies are taken into account. Taking into consideration that web applications in most cases process untrusted input, the risk of security incidents is high. For instance, we identified a sample of 32 high severity CVEs<sup>2</sup> that affect native code used by popular packages (with 2.6M downloads/week), and allow an adversary to corrupt the filesystem, perform privilege escalation, execute arbitrary code, open network connections to exfiltrate data, etc.

**Our contribution** We see a security gap in the way modern JS runtimes execute native code, as neither Node.js, nor Deno, nor Bun sandbox it. In this chapter we propose NatiSand, a framework to provide strong isolation guarantees against the execution of native code. In detail, NatiSand allows the developer to control on a native-component basis, access to filesystem, Inter-Process Communication, and network, effectively reducing the risks coming from the execution of binary programs and shared libraries. Our solution is characterized by a compact, generic architecture that fits nicely with modern runtimes. Internally, it leverages *Seccomp* [63] and Linux Security Modules (LSMs), such as *Landlock* [138] and *eBPF* [73] to restrict access to protected resources. In the design of our solution we paid attention to usability by developers; it is not necessary to have a full understanding of its advanced security features to use it. The developer is only required to provide a concise and readable JSON-formatted policy file, detailing the *ambient rights* – i.e., the access privileges available to the components of the web application that rely on native code. To this end, we provide the developer a comprehensive and interactive

---

<sup>1</sup>Node.js support for creating security policies is still experimental as of 2023.

<sup>2</sup>The list is reported in Table 6.3.



CLI tool to support policy generation, which, as best practice suggests, can also be integrated into CI/CD pipelines and run against a set of test cases [9, 172]. Another key advantage of our approach is that it permits to sandbox native code preserving backward compatibility, namely it does not require to change existing modules (including third-party dependencies) to leverage the new security features.

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

## 6.2 Background

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

### 6.2.1 JS runtimes

JS code rendering is a complex process, involving tasks such as code compilation, code optimization, memory allocation, runtime garbage collection of objects no longer needed, and many others. To perform these critical tasks, modern JS runtimes rely on *engines*, dedicated components implementing the ECMAScript specification that were originally developed for web browsers. As already mentioned in Section 6.1, Node.js and Deno embed Google’s V8 [195], while Bun relies on JavaScriptCore [44]. The interoperability between runtime and engine is achieved with specialized bindings, which are defined in the *node:v8* [161] module in the case of Node.js, in the *rusty\_v8* [91] library for Deno, and by *webcore* [163] in Bun.

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

implemented following this design pattern are: interaction with the filesystem (e.g., `fs`, `Deno.FsFile`), creation of UNIX sockets (e.g., `net`), and exposure of HTTP servers (e.g., `http`, `Deno.serveHttp`).

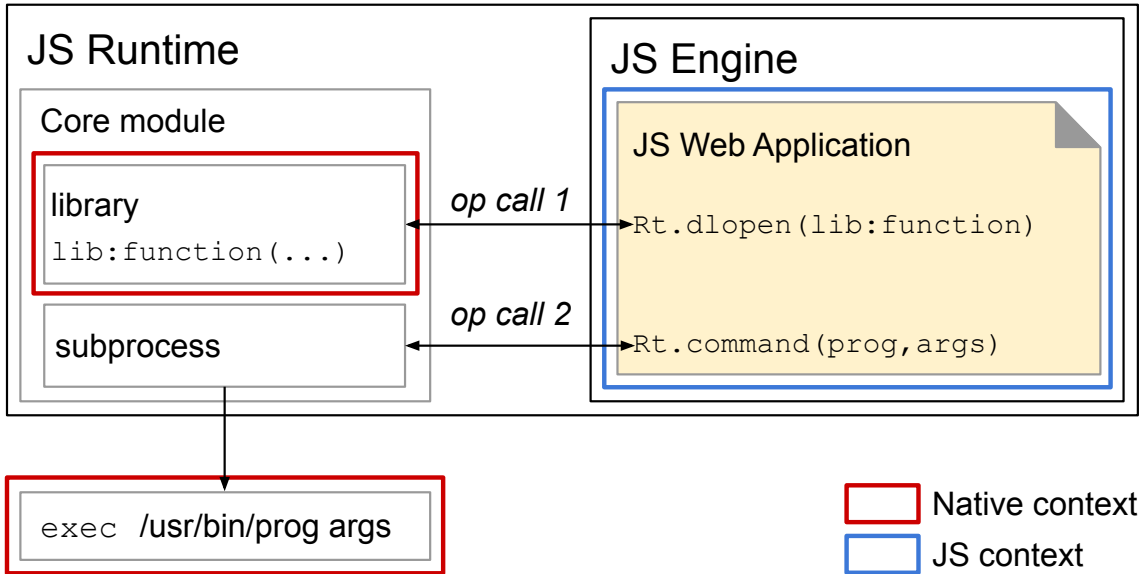
From the web application perspective there is no difference between the functions defined by the ECMAScript standard, and the ones provided by the runtime (and its extensions) [57, 94]. However, non-standard APIs are not served directly by the engine, but are redirected to the runtime leveraging the aforementioned bindings. Since these APIs deliberately permit to break the isolation between the environment controlled by the engine and the underlying system, JS runtimes allow developers to restrict them through the definition of permissions [160, 90]. Based on the runtime, permissions work with different granularities (e.g., single API vs set of APIs) and different default behavior. For example, Deno uses a default-deny model requiring the developer’s explicit consent to access system resources, with effect on multiple APIs [90].

Permissions are intuitive and effective, but they do not offer significant security guarantees when a module needs to run binary programs, or import shared libraries to leverage cross-language function calls [90]. To do so, the web application must be granted the permissions to call APIs like *command* or *dlopen* (e.g., with `allow-run` and `allow-ffi` flags in Deno). In the case of *dlopen*, native code is directly copied into one of the processes owned by the runtime itself before being executed, while with *command*, the runtime first delegates to the OS the creation of a process with the `clone` system call, then performs an `exec` to replace the process image and run the desired program. Independently of the runtime used, both APIs require to execute code outside of the isolated context managed by the engine, as shown in Figure 6.1, which means that this code runs with the same privileges of the user executing the entire JS application.

## 6.2.2 Components for resource protection

### Landlock

Landlock [138] is a Linux Security Module (LSM) introduced in the kernel starting from release 5.13. The goal of Landlock is to enable unprivileged applications to restrict their ambient rights in accordance with the least privilege principle. Ambient rights are specified by *rulesets* – i.e., simple structures that associate a set of permissible actions with a filesystem path (e.g., `read` and `exec` over the resources stored in `/tmp`). Several rulesets can be combined to determine the final set of actions available to an application. To make them effective, a call to the `landlock_restrict_self()` function is performed.



**Figure 6.1:** Execution of binary programs and shared library functions by the JS runtime

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

## BPF

Berkeley Packet Filter (BPF) was originally devised in 1992 [141]. The goal was to provide an in-kernel facility to filter and multiplex network packets, similarly to what was proposed by Mogul et al. [144]. This version of BPF, which is now commonly referred to as *classic BPF* (cBPF), was greatly revised in 2014 resulting in *extended BPF* (eBPF) [73]. The new framework provides an environment to execute programs inside the kernel [115, 125]. This permits to extend the kernel safely, without changing its source code nor loading new modules. eBPF has a wide variety of use cases, ranging from low overhead observability and tracing, to load-balancing, and container runtime security enforcement.

eBPF code is organized into compact units called programs. Each program is attached to a specific function named *hook point*, and is executed in a non-preemptable fashion every time the hook is reached. There are several types of hook point both in kernel space and in user space. Valid examples are [187]: system calls, kernel trapoints, network events, function entry/exit points, and LSM hooks. Specifically,

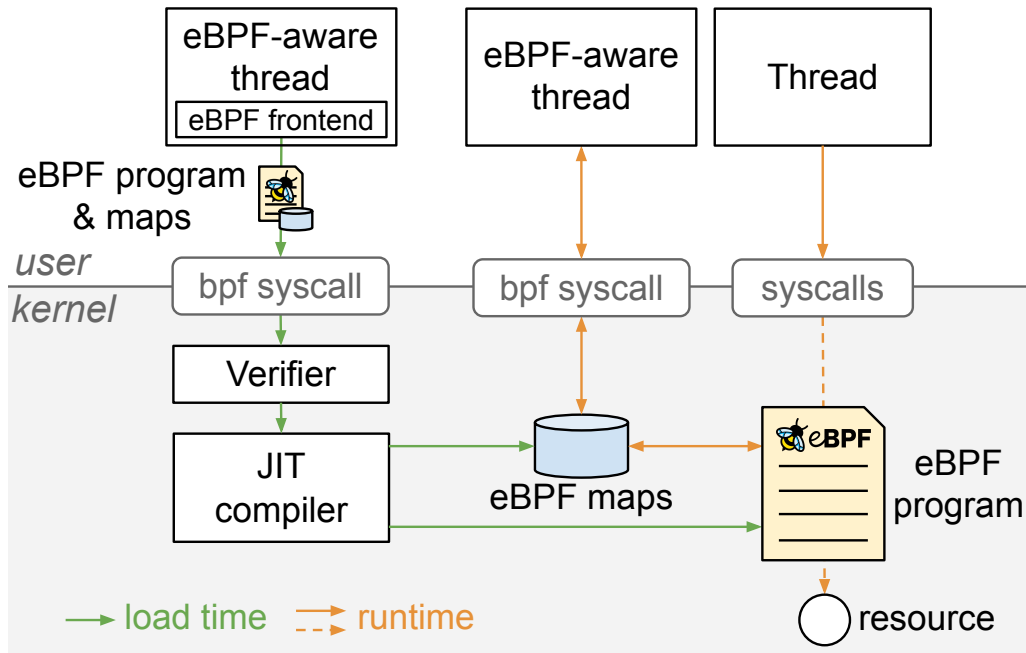


Figure 6.2: Overview of the eBPF architecture

LSM hooks correspond to the functions used by LSMs (e.g. SELinux) to perform security decisions and are characterized by operating entirely on arguments in kernel memory. To persist information between distinct invocations of the same program, data structures named *maps* are used. Maps also permit to share data among eBPF programs and user space applications.

eBPF programs are written in bytecode and are loaded into the kernel using the `bpf` syscall [130]. This is a privileged operation that requires a few capabilities, which vary with the nature of the program [1]. Briefly, `CAP_BPF` is always required, `CAP_PERFMON` is necessary to load tracing-related programs, while `CAP_NET_ADMIN` is used to load networking-related ones. After being loaded, each program undergoes a two-phase process comprising *program verification* and *JIT compilation*. The former is required to guarantee that the program is safe to execute by the kernel. The second phase instead ensures the bytecode is optimized, hence it can be run as efficiently as compiled kernel code on the underlying architecture. In case no errors are raised, the eBPF program is attached to the proper hook and it is ready to be executed.

Modern eBPF development is facilitated by the presence of *frontends*. These frameworks permit to write eBPF programs in a C dialect, and also assist the developer in automatically performing the steps needed to load and attach the programs to the intended hooks (see Figure 6.2). In the work presented in this chapter we rely on *libbpf* [129], a modern library leveraging the *Compile Once-Run Everywhere* (CO-RE) approach [13], which ensures that the bytecode produced at compile time works correctly across different kernel versions.

## Seccomp

Seccomp [63] is a mechanism provided by the Linux kernel to restrict the system calls available to a userspace application. The rationale is that the implementation of system calls may be affected by bugs or errors, therefore reducing the kernel surface exposed to an unprivileged application narrows the attack surface.

The initial implementation of Seccomp restricted the set of allowed system calls only to `exit`, `sigreturn`, `read` and `write` (on previously opened file descriptors) [121]. The implementation was greatly extended in 2012, and it now permits to intercept system calls and determine whether each of them is safe to execute. To this purpose a *filter* program written in the cBPF dialect must be provided. Unfortunately, a classic program has only access to the values of the arguments passed to the system calls (e.g., configuration *flags*), and pointers cannot be dereferenced to avoid TOCTOU issues [95].

## 6.3 Security motivation

JS runtimes let developers specify the set of access privileges given to a web application [90, 160]. This is possible through a set of configuration flags that, when specified, allow to constrain how JS code can access system resources. This is a significant security improvement compared to the past, where applications were able to access any underlying system resource [176, 208]. 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 [90].

There is a broad variety of applications that rely on the use of native code. One well-known example is the use of database drivers; low latency of queries is crucial to satisfy the constraints on response time of a web application and a pure JS implementation may not be able to match them. This led to the development of third-party modules that depend on the code of shared libraries corresponding to the required database driver (e.g., `libsqlite3.so` and `libmysqlclient.so`). To testify the wide adoption of this practice, popular modules for both Node.js (e.g., `node-sqlite3`) and Deno (e.g., `deno-sqlite3`) report more than 600 thousand downloads/week. Notably, the `deno-sqlite3` module was part of the official showcase of the performance of Deno when invoking the native code of a shared library [86]. Previous work [177] demonstrated how this module can be exploited with harmful effects for the web application and the underlying system.

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

**Filesystem compromise** Guaranteeing the integrity and confidentiality of the application host filesystem is crucial to mitigate risks of data corruption and exfiltration [55]. As a whole a web application often has access to many critical resources: databases, executables, private keys, user confidential files, etc. When native code is executed, it can use the same privileges of the web application. In line with the least privilege principle, the potentially vulnerable components should be able to access only the files needed to perform their duties. Authorizing access only to the needed portions of the filesystem restricts what can be read, written or run by an attacker, highly limiting the security risk associated with the presence of vulnerabilities.

**Escalation of privileges** Another relevant attack surface is the privilege of using the IPC channels provided by the operating system (e.g., pipes, message queues, unix sockets). By leveraging IPC, a compromised binary can establish a communication channel with system components and attempt a confused deputy attack to achieve privilege escalation on the host [171, 60]. Given the potential of this attack vector, it is important to limit the scope and set of IPC channels available to a specific native component only to those strictly necessary for its benign behavior.

**Malicious network channels** Network access is a precious resource that a malicious actor can leverage during an attack. A significant portion of malicious payloads open reverse shells to gain control of the victim system over the network [61]. In addition, attackers may open network channels to remotely recover data obtained on the vulnerable host [169]. Restrictions on how a single native component of the web application can access the network can greatly improve the overall security of the application. Network access should be forbidden or restricted only to domains defined by the developer, thus restricting the ability of adversaries to perform data exfiltration or fetch malicious payloads.

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

### 6.3.1 Threat model

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

## 6.4 Design and implementation

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

### 6.4.1 Objectives

We start with the definition of the design objectives.

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

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

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

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

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

## 6.4.2 High level architecture

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

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




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

Internally, NatiSand leverages dedicated Linux Security Modules to restrict access to each resource category. Filesystem-related permissions are enforced using Landlock, while the availability of IPC channels to interact with other processes or services already running on the host is controlled with Seccomp and eBPF. Finally, eBPF constrains the ability to open new connections and limits the devices reachable by a context. Three important characteristics are shared by the selected LSMs: (i) they are lightweight, (ii) they do not require to leverage root permissions while the application is running, and (iii) they operate in stacking mode [174], 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 6.3. Shortly after the JS runtime is executed, NatiSand parses the policy file input by the developer. Based on the policy, a set of sandboxing and tracing programs along with maps are initialized and loaded into the kernel. A pool of isolated contexts is also prepared by the sandbox. At runtime, NatiSand intercepts all the calls to native code performed by the application and executes them safely in the proper isolated context. A technical description of how our proposal is integrated into a modern JS runtime is given in Section 6.4.3, while details about its isolation features are reported in Section 6.4.4. The policy syntax used by NatiSand to configure permissions, along with the support to policy generation, are described in Section 6.5.

### 6.4.3 Integration with JS runtimes

NatiSand complements the architecture of the JS runtime with the addition of the *sandboxer*, a component that parses the policy and enforces the isolation of native code accordingly. In the following we explain the operations performed by the runtime to use it, referring to Figure 6.3.

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

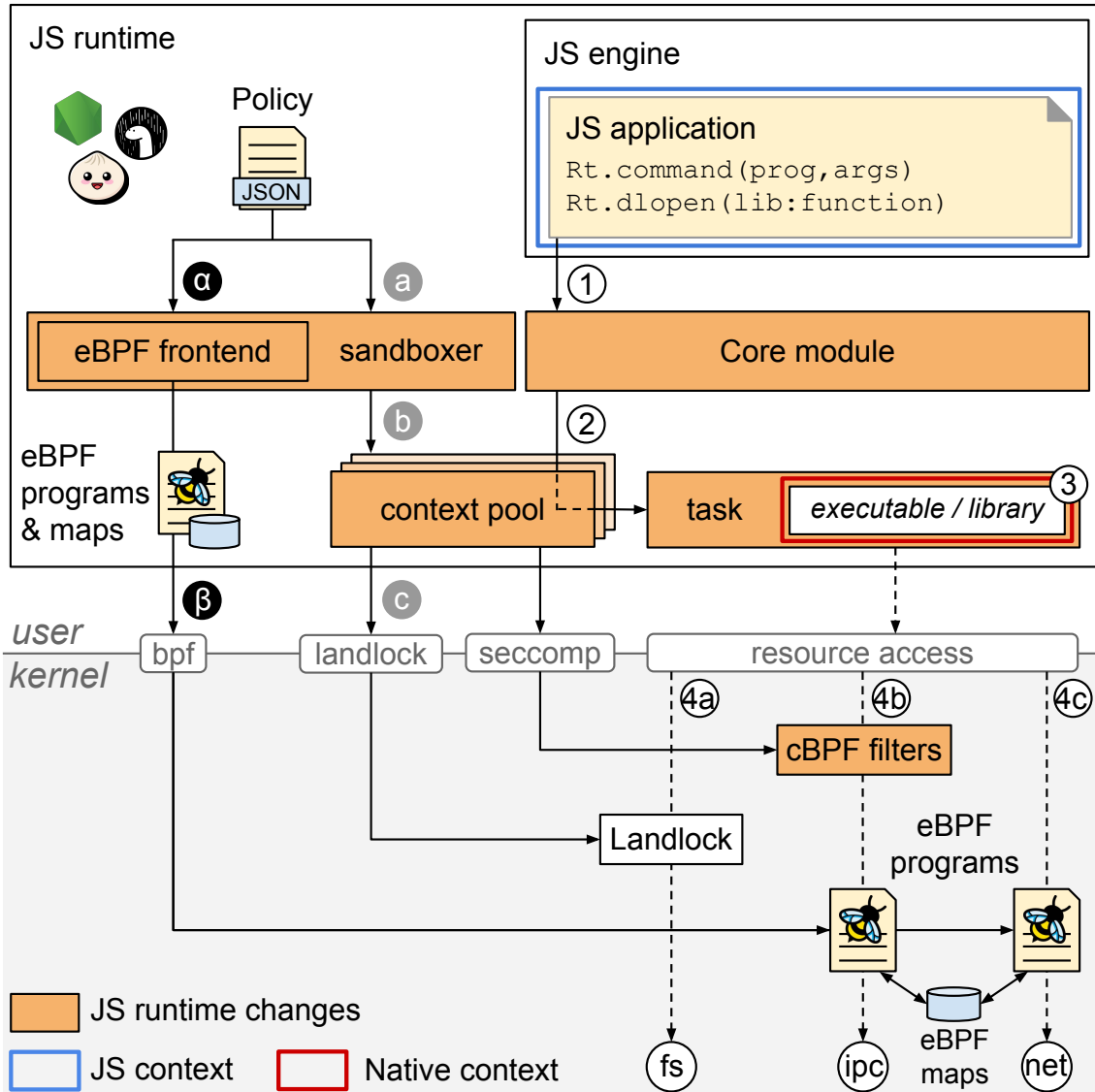


Figure 6.3: Integration of NatiSand in the JS runtime. **Bootstrap:** import security contexts ( $\alpha$ ,  $\beta$ ), creation of the context pool ( $a$ ,  $b$ ,  $c$ ). **Application runtime:** isolated execution of binary programs and shared library functions ( $1$ ,  $2$ ,  $3$ ,  $4a$ ,  $4b$ ,  $4c$ )

with each security context. Based on the policy, the sandboxer (i) configures the required Seccomp filters, and (ii) prepares and loads into the kernel the necessary eBPF programs and maps  $\beta$ . The eBPF programs are used to track the security contexts and enforce network-related and (part of) IPC-related restrictions (more details in Section 6.4.4). Maps instead associate each isolated compartment with a security policy, and store the ambient rights granted to them. To determine which policy is associated with a given isolated compartment we leverage its kernel `task_struct` identifier, which is used as the key in an eBPF map of type `TASK_STORAGE` to lookup the policy identifier. This information is used as an address within an eBPF map

of type `ARRAY_OF_MAPS`, and permits to retrieve an inner `HASH` map containing the ambient rights. Loading eBPF maps and programs is a privileged operation that requires the `CAP_BPF`, `CAP_PERFMON`, `CAP_NET_ADMIN` capabilities to be performed, thus we grant the JS runtime executable the corresponding Linux file capabilities [72]. After the completion of these steps, the capabilities are no longer necessary, hence they are dropped. This satisfies the requirement that root permissions at runtime are not needed for the activation of security contexts.

The sandboxer is also responsible for the creation of the security contexts where native code will be executed at runtime. Each context is an OS thread with permissions restricted by Landlock, Seccomp, and eBPF. To avoid paying the performance cost to instantiate each security context during the invocation of executables and shared libraries, we modified the JS runtime to allocate a thread per security context defined by the policy, restrict their permissions, and then, park them in a context pool (a, b, c). With Landlock and Seccomp, restriction of privileges is performed calling the corresponding syscalls from the specific context, while restriction of permissions based on eBPF is simply performed invoking the uprobe `attach_policy` reported in Table 6.1a. As a result, the `task_struct` identifier of a given context is annotated in the dedicated eBPF map along with the associated policy identifier. This design choice allows to reuse security contexts, thus minimizing latency, which, as highlighted in the objectives (Section 6.4.1), is a critical metric for web applications.

**Application runtime** After the web application is started, two operations can lead to the execution of native code: (i) the execution of a binary program in a subprocess, and (ii) the invocation of a shared library. NatiSand intercepts all the requests originating from the web application that require to execute native code ①, and leverages the sandboxer to assign them to the proper pre-allocated isolated context ②. Based on the type of request, a dedicated task inheriting the selected security context is launched and used to execute the native code ③. Specifically, when there is a request to run an executable, the JS runtime forks a process. On the other hand, when a shared library should be loaded, a thread is spawned. The consequence is that any request to access filesystem, IPC, and network resources will be subject to the restrictions imposed by the LSMs (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.

Context lifecycle	Access control				
uprobe/attach_policy tp_btf/sched_process_fork tp_btf/sched_process_exit	<table border="1" style="border-collapse: collapse;"> <tr> <td style="border-right: 1px solid black; text-align: center; vertical-align: middle;">IPC</td> <td>fentry/fifo_open lsm/socket_bind lsm/socket_connect</td> </tr> <tr> <td style="border-right: 1px solid black; text-align: center; vertical-align: middle;">Network</td> <td>lsm/socket_bind lsm/socket_create lsm/socket_connect</td> </tr> </table>	IPC	fentry/fifo_open lsm/socket_bind lsm/socket_connect	Network	lsm/socket_bind lsm/socket_create lsm/socket_connect
IPC	fentry/fifo_open lsm/socket_bind lsm/socket_connect				
Network	lsm/socket_bind lsm/socket_create lsm/socket_connect				
(a)	(b)				

Table 6.1: Hooks and tracepoints monitored by NatiSand

#### 6.4.4 Isolation features

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

**Policy inheritance** While Landlock and Seccomp guarantee policy inheritance after a `clone` syscall is performed, the eBPF map that tracks the restricted contexts must be updated explicitly. To this end, NatiSand relies on the eBPF tracing programs that are loaded into the kernel during the bootstrap phase and are attached to the `fork` and `exit` tracepoints reported in Table 6.1a. Whenever a security context allocates a new task with a fork operation, the tracing program registers a new entry into the map of restricted contexts. The entry maps the task identifier of the child to the policy identifier associated with the parent. When instead a context terminates its duties and issues an exit, its task identifier is deleted from the map. No intervention by the developer is required, as policy inheritance is transparently handled by our solution.

**Filesystem** NatiSand restricts access to the filesystem using Landlock. The sandbox enforces a straightforward `read`, `write`, `exec` (RWX) permission model, specified with three allow-list vectors (e.g., lines 5, 6, and 7 in Listing 6.1). After the security context has been activated, the available permissions can only be further restricted.

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

children with the necessary communication channels through the inheritance properties of the `clone` syscall [136]. 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 [137]. 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 [95]). Likewise, the `open` and `openat` syscalls do not represent the type of file to be opened through configuration flags, so Seccomp cannot handle the specific case properly. To solve these problems NatiSand relies on the eBPF programs attached to the hooks reported in Table 6.1b (which are not affected by TOCTOU issues, since they operate on arguments that were previously deep copied by the kernel). In particular, in the case of UNIX sockets the programs are attached to the `lsm/socket_bind` and `lsm/socket_connect` hooks, while for `fifo` files the kernel function `fifo_open` is used. A summary of the IPC mechanisms controlled by NatiSand, along with the LSMs leveraged to perform the security checks, is reported in Table 6.2.

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

connect only to a restricted list of hosts, and when needed, to use the network without restrictions. The sandboxer relies on eBPF programs to enforce permissions. The programs restrict the ability to `create`, `connect`, and `bind` sockets, and are thereby attached to the LSM hooks reported in Table 6.1b.

The creation of a socket opens a communication channel and returns a file descriptor as a result. By default, the sandboxer restricts the available communication domains to Internet Protocol (IP), denying applications the use of protocol families such as Bluetooth, Radio, VSOCK, and many more (this information is directly available from the arguments input to the `socket_create` interface). No restrictions are instead applied to UNIX domain sockets and the type of socket to be opened (e.g., stream, datagram). Opening a connection to a host is permitted only when the developer grants the isolated context to do so. The eBPF program that checks the opening of a connection first recovers the policy restricting the current security context, then uses it as a key to lookup the map of ambient rights from the `ARRAY_OF_MAPS` described in Section 6.4.3. The ambient rights map is an allow-list that stores the reachable (i.e., policy allowed) hosts, hence the security check is carried out with a lookup. Each network resource is uniquely identified by its IP address and port. Internally, we use the value zero for the port to represent the permission of opening a connection to a given host on every port.

Up to now, we have discussed the restrictions when the application connects as a client to a service. However, web applications frequently need to serve incoming requests. To do so, it is necessary to assign a “name” to a socket – i.e., configuring its address. This operation is done with the `bind` syscall, and we decided to permit it only when the policy gives the current security context access to the corresponding address and port pair. Again, the value zero for the port is used as a placeholder to allow binding on every port. On the other hand, no restrictions are applied to the `listen` and `accept` syscalls. Listen only marks a socket as passive, meaning that it will be used to accept incoming requests. However, no connection to a socket can happen if an address was not previously assigned to it [135]. The same applies to `accept`, which is used to extract the first connection request from the queue of pending connections [133].

**Limitations** While NatiSand significantly restricts the set of permissions and system resources associated with subprocesses and shared libraries, it provides strong memory isolation guarantees only when executables are run, as each subprocess is executed within its own address space. On the other hand, shared libraries are loaded within the hosting thread address space, hence a native library bug can impact the web application memory. Several research works have studied this problem and have proposed countermeasures [149, 127, 204, 53, 197, 67]. In general, these works are

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

**Table 6.2: LSMs used by NatiSand to restrict Linux IPC. The checkmark ✓\* indicates when Seccomp needs to evaluate the syscall configuration flags to make the access decision**

compatible with the design of our solution, therefore they could be used in conjunction with NatiSand to improve isolation of shared libraries. Among them, BreakApp [197] and BinWrap [67] propose approaches tailored for interpreted languages, but they require either the introduction of wrappers or the execution of remote procedure calls. RLBox [149] provides strong guarantees against memory corruption, but it demands the developer to manually retrofit existing code, a process that can take up to “few days” for each library according to the authors. Improved intra-process isolation can also be achieved leveraging dedicated hardware features like Intel Protection Keys for Userspace (PKU), but as demonstrated by PKU Pitfalls [70] these solutions can be bypassed using kernel functions that are agnostic of intra-process isolation (i.e., the attacks use the kernel as a confused deputy). Since NatiSand is completely aligned with the memory isolation assumptions made by the kernel, our solution is not affected by this issue.

## 6.5 Policy

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

### 6.5.1 Policy structure

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

of objects and each object details the permissions available to a security context. Within each object, a *name* is used to identify the context, a *type* indicates whether the context applies to an executable, a library, or a function of a library; the sections *fs*, *ipc* and *net* are used to configure the corresponding permissions. The structure of the objects is flexible, and only a *name* is required to configure a valid context. As the policy follows a *default-deny* model, a context that specifies only its name has no permissions at runtime. An excerpt from a policy file is shown in Listing 6.1 (the complete example is reported in Appendix B), while a summary of the most relevant policy features is described next.

### Name, Type

The name and type elements are used by NatiSand to determine which policy context must be enforced. The type element can be set to `executable` (the default value), `library` or `function`. At runtime NatiSand extracts the absolute path of the native program and function name, and based on the information available, it identifies the most selective entry in the policy. This gives the developer the flexibility to use different policies in case binaries and libraries have the same basename, or when different functions from the same library are invoked. Moreover, since absolute paths are used, this approach ensures symlinks cannot trick the lookup of the security context. Listing 6.1 shows a policy that is enforced every time the application runs the curl program.

### Fs

The `fs` element is used to configure filesystem-related permissions. `Fs` stores three optional arrays: *read*, *write* and *exec*. Filesystem paths are used as array values. As an example, the context detailed in Listing 6.1 can read and execute the curl binary, and write to `response.json` in the current working directory of the web application. In case the developer wants to operate with a coarser granularity, the value `true` can be used to replace any of the *fs*, *read*, *write* and *exec* arrays to grant access to the whole filesystem.

### Ipc

To restrict IPC access we decided to expose developers a simple interface where flags can be turned on and off based on their needs. Six optional flags are available in the policy: *fifo*, *message*, *semaphore*, *shmem*, *signal*, and *socket*. For example, in Listing 6.1, curl is allowed to use abstract, named, and unnamed Unix sockets. It is up to our sandboxer to abstract away the complexity of the underlying architecture



**Listing 6.1: Example of JSON policy file with single context**

```

1  [{
2    "name": "/usr/bin/curl",
3    "type": "executable",
4    "fs": {
5      "read": ["/usr/bin/curl", ...],
6      "write": ["response.json"],
7      "exec": ["/usr/bin/curl", ...]
8    },
9    "ipc": {
10     "socket": true,
11   },
12   "net": [{
13     "name": "https://www.example.com",
14     "ports": [443]
15   }]
16 }]

```

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

## Net

Web application developers are often interested in restricting the hosts an application can connect to. The policy permits to specify an array of reachable hosts. Each host is fully qualified by its URL/IP, and the sequence of permitted ports. As in the case of the filesystem, the policy permits to grant access to the network without limitations (setting *net* to `true`), enable all the ports for a specific host (setting *ports* to `true`), or completely remove access to the network (leveraging the default-deny behavior). In Listing 6.1, the process executing `curl` is only allowed to connect to `https://www.example.com` on port 443.

## 6.5.2 Policy generation

While designing NatiSand we opted for a minimal and easy to understand policy syntax to target a broad spectrum of users. However, writing a policy for large components may be a tedious and tricky task, since we do not expect all the developers to be aware of how binaries and external libraries used by their web application work internally. To assist the developer, we follow an approach similar to Slim-Toolkit [172], where a service is run against a test suite to generate the security profile. Specifically, we developed a CLI utility written in Go that provides generation of policy templates the developer can understand, modify, and audit. The utility persists policy-relevant information in a SQLite database, and exposes to the user many functions that permit to configure multiple contexts, merge the results collected from multiple tests, and refine policies interactively. In the following we provide details on the work we performed for each of the protected subsystems.

### Filesystem

The automatic retrieval of the dependencies of a binary is a well-known problem. Our utility is capable to discover the dependencies of programs installed as ELF files, and programs that are spawned by dedicated POSIX or shell wrappers. The utility first uses *ldd* [131] to discover the direct and transient dependencies, then, it relies on *strace* [132] to monitor the interaction between the kernel and a traced binary, so to complement the information previously found with additional filesystem permissions.

### IPC

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

### Network

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

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

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

## 6.6 Case study: Deno runtime

There are three well-known alternatives for the execution of JS code on the backend, namely Node.js, Deno, and Bun. As highlighted in Section 6.2.1, their architectures have strong similarities, and NatiSand is designed to be compatible with all of them (since no assumption is made on specific runtime components). Nevertheless the integration is not trivial, and it requires significant engineering effort, therefore we integrated NatiSand into only one of them to demonstrate the achievement of the set objectives (Section 6.4.1). In this section we explain our decision, then we highlight the main architectural changes.

### 6.6.1 Runtime selection

Considering (i) popularity among web developers, (ii) availability and support of third-party modules, and (iii) security-oriented features provided by the runtime, we selected Deno. Bun is the latest runtime released, and thus currently the least used by developers. Node.js is nowadays the most widely used platform, and it offers developers the largest collection of open source packages. However, Node.js by default does not prevent JS applications to access system resources, and although it recently introduced a module-based permission model, the feature is experimental [160]. Deno was instead designed with the protection of the host as one of its main goals [166], thus no access to privileged system resources is given to JS

applications unless the developer explicitly grants it. Deno provides the *Node Compatibility Mode* [88], a feature enabling the reuse of code and libraries originally built for Node.js. The availability of this function permits to import packages hosted by Deno on `deno.land/x`, as well as modules published to npm. To conclude, Node.js and Deno prevail over Bun on popularity and security-oriented features. Node.js wins over Deno on popularity (but Deno is quickly growing), they are comparable in terms of third-party modules, and Deno significantly outperforms Node.js on security oriented-features, leading us to choose Deno.

## 6.6.2 Deno integration

Deno has a modular architecture organized into components. Three of them are particularly important for NatiSand: (i) *rusty\_v8*, the package that bridges Deno and the V8 engine implementing the set of bindings to the V8's C++ API, (ii) *deno\_core*, which leverages *rusty\_v8* to expose the interfaces provided by Deno to the JS application, and (iii) *deno*, which defines the runtime executable together with the Command Line Interface.

**Bootstrap** Shortly after the Deno executable is run, the *deno* component is used to read the permissions granted by the developer via CLI. We extended this stage to read and parse the policy file specified with the new `native-sandbox` flag, then we added the permissions associated with each security context to the *global state* stored by the runtime. To complete the bootstrap phase, we also integrated the steps to load the necessary eBPF programs and to initialize the pool of isolated contexts, as explained in Section 6.4.3.

**Application runtime** After the JS application is started, the function calls that cannot be directly handled by V8 are routed to the Deno runtime through the bindings defined by *rusty\_v8*. Each of them is associated with the *op\_code*, a unique code identifying the operation to be performed. The *deno\_core* component receives such requests, it checks the permissions available from the global state, and serves them accordingly. We identified requests that require to execute native code (e.g., `command`, `dlopen`, `run`), and modified *deno\_core* so that they are restricted by NatiSand. The *op\_code* along with the arguments are used to select the proper security context.

## 6.6.3 Support to fast JS calls

In October 2020 V8 announced the support to fast JS calls [194]. The function allows V8 to directly invoke optimized native functions without leveraging the bindings that

**Listing 6.2: Code sandboxing with nativeCall()**

```

1 function query(db, stmt) {
2     const sqliteDB = new sqlite3.Database(db);
3     const query = sqliteDB.prepare(stmt);
4     const tuples = query.all();
5     sqlite_db.close();
6     return tuples;
7 }
8
9 const db = "database.db";
10 const stmt = "SELECT * FROM table";
11 const ts = Deno.nativeCall(query, [db, stmt]);
12 console.log(ts); // print tuples

```

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

Deno has introduced unstable support to fast JS calls in July 2022 [85]. The change affected the implementation of the *dlopen* API, which is now able to generate an optimized and a fallback (i.e., standard) execution path for native functions. The optimized path is triggered only when V8 is actually able to optimize a symbol, and it entails the execution of code leveraging the fast call interface. While the optimized path is associated with minimum overhead, from a security perspective it permits the web application to execute native code without the mediation of the JS runtime, invalidating the security reference monitor of Deno. In our prototype we address this Deno security issue offering developers two alternatives: (i) turn off the fast call support and safely rely on the execution of sandboxed native functions with NatiSand without any code change, and (ii) enable insecure fast calls but allow to select the JS functions that need to be isolated with minimal code changes. The second option permits to take advantage of fast calls when performance is critical and risks are limited (e.g., arithmetic operations), and at the same time benefit from the security features NatiSand provides. To this end, we introduced a new API named `Deno.nativeCall()`. The API receives, as first argument, the name of the function to be sandboxed, along with the list of its arguments. Listing 6.2 shows how to sandbox the functions from the native database driver *sqlite3*.

## 6.7 Experiments

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

experimental evaluation, we first show our solution is able to protect web applications relying on binary programs and shared libraries affected by high severity vulnerabilities (Section 6.7.1), then we investigate the performance of our approach (Section 6.7.2). Both tests use a server with Ubuntu 22.04 LTS, an AMD Ryzen 3900X CPU, 64 GB RAM, and 2 TB SSD.

### 6.7.1 Exploit mitigation

To conduct our analysis we built a representative sample of vulnerabilities targeting executables and libraries widely used in web applications. We identified the 32 CVEs reported in Table 6.3. The entries are separated into three classes: Arbitrary Code Execution (ACE), Arbitrary File Overwrite (AFO), and Local File Inclusion (LFI). The list of vulnerable utilities includes programs used to compress files (e.g., GNU Tar, RAR, Zip), to process multimedia (e.g., FFmpeg, GraphicsMagick, ImageMagick), database drivers (e.g., SQLite), and also Machine Learning libraries (e.g., Lightning, Sockeye, TensorFlow). We highlight that the vulnerabilities affect popular open source modules with 2.6M downloads/week available from the npm and `deno.land/x` archives. Concrete examples are `sharp` and `fluent-ffmpeg` from npm, or `flat` and `sqlite` from `deno.land/x`.

First, we checked that public Proofs of Concept of the CVEs in Table 6.3 successfully exploit the vulnerable version of the utilities. Then, we analyzed whether the vulnerabilities were exploitable sending the malicious payload through the JS module interface, and confirmed the feasibility of the attack. The *Node compatibility mode* was leveraged to execute in Deno the modules downloaded from npm. We finally repeated the experiment activating the security functions provided by NatiSand, and verified that the attack was no longer successful, while the application was still able to serve benign requests (i.e., no functionality loss). The only change we introduced in the experiment was the specification of a security policy through the `native-sandbox` CLI argument. The policy was generated using the tool described in Section 6.5.2. No modification to the web application, nor its dependencies, was required to benefit from the new sandboxing capabilities.

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

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

**Table 6.3: Sample of CVEs mitigated by NatiSand**

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

### 6.7.2 Performance evaluation

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

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

## Executables

In the first batch of experiments we analyze the overhead associated with executables. Compared to the default scenario where no protection is available, NatiSand spawns each program in a dedicated subprocess with its own set of constrained ambient rights. A handful of general purpose sandboxers can be adopted to achieve a comparable degree of protection by wrapping the execution of each subprocess with the chosen sandboxing utility. In our evaluation, we considered *Minijail* [111] and *Sandbox2* [112]. Minijail is a tool used in ChromeOS and Android to launch and sandbox other programs based on the set of arguments specified, while Sandbox2 is a C++ library written by Google that can be used to sandbox entire programs or portions of them. Both Minijail and Sandbox2 support multiple containment techniques, such as the introduction of dedicated user ids, restriction of the Linux capabilities, introduction of policy-based Seccomp filters, and isolation based on Linux namespaces.

**Benchmark I** In the first benchmark we implemented a JS application to test the execution of 17 common Linux utilities with four configurations: Deno, NatiSand, Minijail, and Sandbox2. The application uses `Deno.run()` to spawn each utility in a subprocess, and it leverages `Deno.bench()` to determine the duration of each request. The function ensures that each measure is statistically robust, as it automatically performs a dynamic number of rounds based on the duration of the test (i.e., the shorter the test duration, the higher the number of repetitions). The results are shown in Table 6.4 (tests are ordered by increasing execution time). As expected, the cost of activating the sandbox is amortized with the increase in the test duration. The tests also show that NatiSand suffers from a smaller performance degradation compared to Minijail and Sandbox2. This aspect is particularly evident for short-lived utilities. The reason is that our approach is integrated by design and, contrary to the other solutions, leverages lightweight technologies that introduce a smaller performance footprint.

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



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

Table 6.4: Average execution time for common Linux utilities

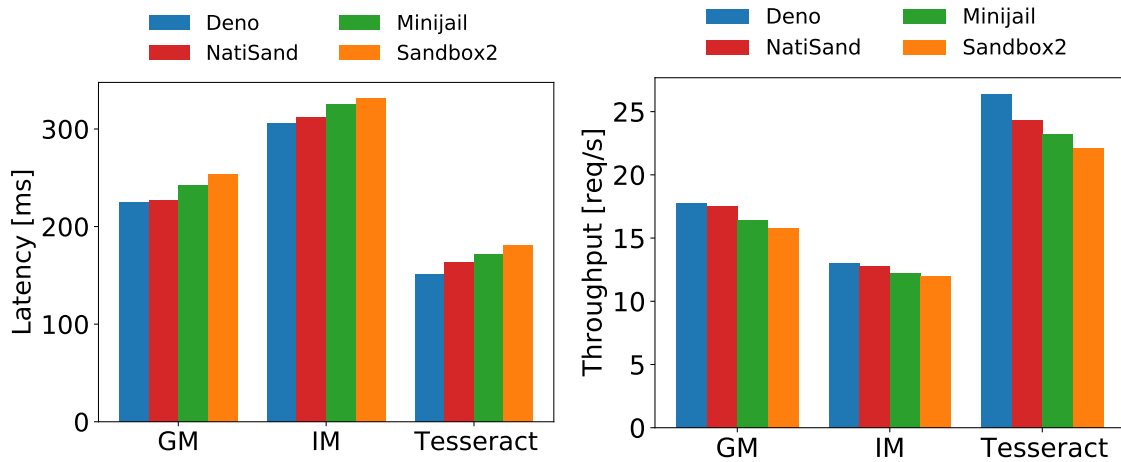


Figure 6.4: Average latency and throughput for microservices that execute subprocesses

GraphicsMagick and ImageMagick, to perform a *sharpen* operation on images input by the client, while the third microservice relies on Tesseract to perform Optical Character Recognition on a second sequence of images input by the client. Similarly to the previous case, the test was repeated for each of the four configurations: Deno, NatiSand, Minijail, and Sandbox2. This time the HTTP benchmarking tool *wrk* was used to measure the performance of each microservice. Network bandwidth and latency are 1 Gbps and 10 ms, respectively, while 100 warmup requests were carried out. Figure 6.4 shows the average latency and the throughput observed

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

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

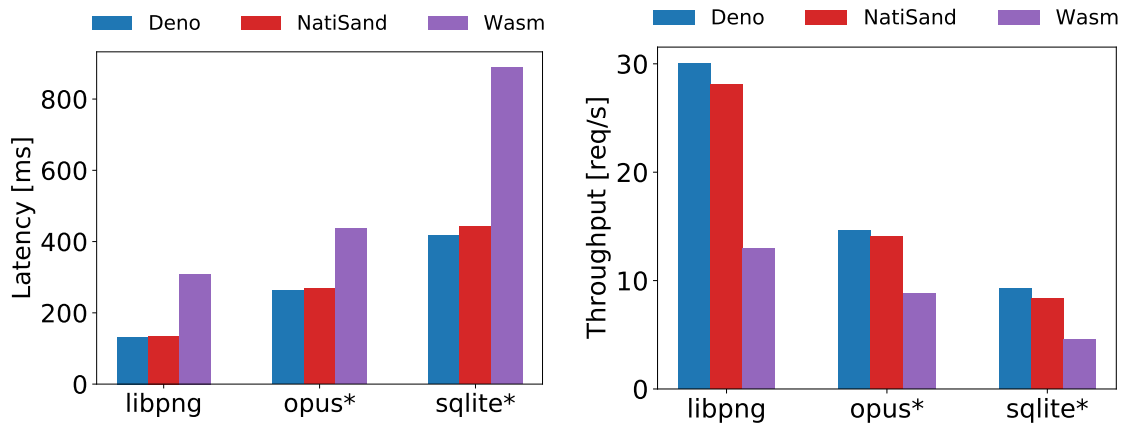
## Libraries

In the second batch of experiments we analyze the overhead associated with libraries. Contrary to the default JS runtime behavior, NatiSand transparently executes native library functions in dedicated contexts with limited ambient rights. A modern, valuable alternative approach to isolate libraries is to compile them to WebAssembly (Wasm), a standardized, portable binary instruction format executed in a memory safe, sandboxed environment. This approach has gained considerable attention recently, as browsers such as Firefox have used it to retrofit some of their components to safely interface with native libraries [149].

**Benchmark III** Similarly to Benchmark I, we implemented a JS application to highlight the overhead experienced on the server when native libraries are executed. In this case three configurations are evaluated: Deno, NatiSand, and Wasm. The application tests the operations provided by four popular libraries: (i) libxml2, to open and query XML data, (ii) libpng, to read metadata information and verify the signature of a png image, (iii) opus to encode and create an audio trace, and (iv) sqlite3, to open and query the Northwind database. Test durations were again measured with `Deno.bench()`, and the results are reported in Table 6.5. Deno exhibits a consistent performance advantage for operations that require up to 30 microseconds. However, NatiSand proves to be more efficient than Wasm, which in turn is affected by a substantial overhead in almost every test. This difference is due to the nature of Wasm; while there have been improvements, the just-in-time compiled language [196] remains slower than its native counterpart. Remarkable are the cases of opus and sqlite3, which used `nativeCall` and demonstrate its efficiency.

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

**Table 6.5:** Average execution time for common native libraries (\* marks the use of nativeCall)



**Figure 6.5:** Average latency and throughput for microservices that execute native functions (\* marks the use of nativeCall)

**Benchmark IV** To understand the slowdown perceived by a remote client, we exposed the functions of the libpng, opus, and sqlite3 libraries with microservices. For each of them, we configured the client to send the input to the server, and measured the latency and throughput using *wrk* (as explained in Benchmark II setup). The results are visualized in Figure 6.5. Once again the client observes a small degradation of latency and throughput when using NatiSand instead of Deno, but the overhead is far less noticeable compared to the results discussed in Benchmark III. Conversely, Wasm is affected by a significant degradation of latency. This is due to the just-in-time compilation of Wasm, and the additional memory management required to exchange data between the JS application and Wasm.

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

toolchain [96] and the WASI Sdk [201], respectively. Moreover, current implementations of the WebAssembly System Interface (WASI) can only restrict ambient rights programmatically, and filesystem privileges work at directory granularity. Lastly, Wasm requires the developer to explicitly allocate, write, and read bytes from the Wasm module linear memory.

## 6.8 Related work

Isolation of software has been widely investigated by both the academic and industrial communities [55, 126, 83, 111, 112, 150, 71, 167, 4]. *MBOX* [126] features an unprivileged sandboxing mechanism that prevents a process from modifying the host filesystem by layering the sandbox filesystem on top of it. The solution is implemented by interposing syscalls using *Seccomp* and *ptrace*. The use of *ptrace* required the authors careful attention to avoid the risk of TOCTOU attacks, moreover it suffers from non-negligible performance degradation. DeMarinis et al. [83] propose *Sysfilter*, a static analysis framework to reduce the attack surface of the kernel, by restricting with *Seccomp* the system call set available to processes. This approach proves to be effective in limiting the kernel APIs that can be abused by attackers, but whenever a system call is necessary for the benign behavior of a program, there is no way to control with *Seccomp* the specific instance of the resource used. *BPFBox* [104], *BPFContain* [103], and *Snappy* [54] are security frameworks that provide confinement of processes and containers with the use of eBPF. These solutions highlight the benefit of eBPF by providing simple, efficient, and flexible confinement of system resources, however these solutions either require a privileged daemon or require to load kernel modules to introduce a set of *dynamic helpers*. Often, industrial sandboxing solutions take advantage of multiple protection techniques to support process containment. This is the case for *Minijail* [111], and *Sandbox2* [112]. Some of the security mechanisms used are: introduction of new user ids, capabilities restriction, namespace isolation and policy-based *Seccomp* filtering. These tools expose a powerful interface meant to be used by security experts. Similar considerations are shared with other less mature tools such as *Firejail* [150] and *Bubblewrap* [71].

Multiple research efforts have studied the use of third-party components in software products [149, 127, 204, 53]. *PKRU-Safe* [127] proposes an automated method for preventing the memory corruption of memory-safe languages due to the interaction with unsafe code. It leverages the compiler infrastructure to provide hardware-backed memory protection requiring changes to build files, dependencies, and code (in the form of code annotations). *Codejail* [204] provides partial isolation of li-

libraries by spawning a new process and configuring the necessary communication channels to support tight memory interactions with the main program. To support the change in the interactions without modifications to the library code it is necessary to write a wrapper library. *Cali* [53] is a compiler-assisted library isolation system that compartmentalizes libraries into their own process, and automates the configuration of the necessary communication channels by tracking data flow between the program and the library at link time. *RLBox* [149] is a framework to isolate libraries in lightweight sandboxes – i.e., process, Wasm. It facilitates the retrofitting of applications employing static information flow enforcement and dynamic checks expressed in the C++ type system. These solutions were designed for compiled languages, so while some of the concepts are portable to JS runtimes, the solutions are not easily adapted to this domain.

A recent study by Staicu et al. [177] highlights how the possibility to invoke native code from scripting languages undermines the security assumption of applications. They discuss a methodology to detect misuses of the native extension API and show how the exploit of these vulnerabilities in npm packages can lead to web applications compromise. Previous proposals [197, 205, 67] tackle this problem by providing solutions to isolate the execution of third-party modules. *Wolf at the Door* [205] reduces the risk associated with the installation of npm packages by mediating their install-time capabilities. It enforces complex user-defined policies by leveraging AppArmor, hence prohibiting unauthorized access to confidential files and connections using an LSM that currently cannot coexist with SELinux and SMACK. *BreakApp* [197] takes advantage of module boundaries to compartmentalize npm modules in accordance with a set of code annotations. Modules are isolated with software, process, or container isolation, and it is possible to configure the visibility of the application context available to external modules. Process and container isolation enable the protection of native code, however the specification of their permissions are beyond the scope of the proposal. *Cage4deno* [5] protects filesystem resources from subprocesses executed by JavaScript runtimes. *BinWrap* [67] separates the execution of third-party components from the rest of the application using distinct execution threads for different domains of trust. The main focus of the proposal is prohibiting arbitrary accesses to sensitive data stored in the memory of the JS runtime by leveraging Intel’s MPK/PKU. NatiSand is complementary to the above solutions since our goal is to specify and enforce permissions on native code dependencies of web applications, rather than providing memory isolation for untrusted components.

Protecting JS code from being compromised is out of the scope of NatiSand, nonetheless, since proposals in this domain and ours both target the web development audience, our proposal shares some ideas with previous works in this do-

main [198, 9, 180, 157]. For instance, Ferreira et al. [102] propose a lightweight permission system providing per-package on/off switches that limit access to Node.js core modules (e.g., `child_process`, `fs`, `http`). By doing so, it can prohibit access to subprocess, filesystem, and network resources for the JS code. Similarly, NatiSand takes care of protecting filesystem, IPC, and network resources, targeting native code. *Mir* [198] is a system preventing the compromise of the application by third-party modules with the enforcement of fine-grained RWX permissions on every field of every variable in the JS context. NatiSand adopts an equivalent permission model to contain native code when accessing filesystem resources. Another research work enforcing security boundaries stated in a policy is *SandTrap* [9]. The approach enforces fine-grained access control policies on cross-domain interactions between application code and the third-party modules. The creation of policy files described by the authors consists in running test suites to create a policy with acceptable static cross-domain interaction coverage. We adopt a similar approach in the policy generation of NatiSand. Note that, differently from our proposal, solutions protecting JS code can run in user space, thus they do not limit the portability of the JS runtime to Linux systems.

## 6.9 Conclusions

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

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

## Availability

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

## 7. Conclusions and future work

In this thesis, we presented novel fine-grained access control techniques to protect resources in mobile and cloud applications.

We opened our dissertation by describing SEApp, a technique for mitigating security threats in the Android mobile operating system. SEApp empowers developers with the capability of isolating the internal components of Android apps and regulate their permissions on a per-component basis. This crucial step not only limits the impact a vulnerability has on the app resources, but it also allows to provide strong user privacy guarantees and meet data privacy regulations despite the use of third-party code.

The dissertation proceeded considering the importance to also secure cloud applications. Specifically, we presented an approach to support the introduction of security policies to restrict the file system resources available to an application. Compared to virtualization technologies such as VMs and containers, which are associated with coarse granularity, we demonstrate that our proposal enables the introduction of fine-grained, per-resource access rules. Then, we further explore the topic in the context of WebAssembly runtimes. Here, not only the approach permits to introduce fine-grained policies to restrict file system access, it also replaces error-prone userspace implementations of the security checks (and the security issues stemming from them) with a unified eBPF implementation. Finally, we consider the specific use of JavaScript runtimes for the creation of cloud applications, and how developers commonly rely on native code to speed up development and execution of the application. Since JS runtimes do not provide solutions for the isolation of native code, we propose NatiSand, a runtime-agnostic component to control the filesystem, Inter-Process Communication (IPC), and network resources available to binary programs and shared libraries.

During our research work, considerable attention was dedicated to the performance and usability of our proposals. The work presented in this thesis represent the final result of multiple improving iterations to ensure small performance footprint and high usability by developers. Indeed, these are very important aspects since the lower the performance side-effects and the effort on behalf of the developer to integrate our solutions with existing applications, the broader will be the adoption of our security mechanisms. With this regard, to facilitate the integration of our solutions with existing real systems and foster the reproducibility of our experimental evaluations, our prototype implementations are all available open source.

We believe the approaches proposed in this thesis contribute to improving the state of the art in this domain and support the evolution toward more secure mobile and cloud software platforms.

## 7.1 Future work

This section concludes the thesis with a discussion on the future work that can be done in the area of fine-grained access control technologies to protect resources in mobile and cloud applications.

**Mobile applications** – Chapter 3 describes SEApp, a novel proposal that provides developers with a mechanism to isolate the internal components of Android apps and regulate their permissions on a per-component basis. This is achieved by first executing components in dedicated processes, and then restricting access to the app and system resources with ad hoc SELinux policies. While effective, the decision to use SELinux to constraint the access of application components impose significant limitations. Resource-wise, SELinux implies the use of process isolation to isolate different components of the application; this increases CPU and memory utilization which negatively affect responsiveness and battery life of the mobile device. Usability-wise, despite our efforts, SELinux policies are hard to audit, author and maintain. Therefore, we consider the possibility of replacing SELinux and process isolation with novel Linux Security Modules and memory protection techniques an interesting and promising evolution of our work with the potential to solve both these limitations and, thus, further promote adoption.

**Cloud applications** – Chapter 4, 5, and 6 highlight limitations of the cloud technologies available at the time of writing with regard to fine-grained access control of system resources. Each chapter provides its own take on a specific aspect of the problem. Chapter 4 relies on instrumentation to collect and audit the activity traces generated by microservices, and then uses this information to create fine-grained access control policies and strengthen the security boundary of the cloud application. Chapter 5 focuses on the security implications of enabling access to system resources through the WebAssembly System Interface (WASI), and proposes improvements in the controlling access to file system resources. Chapter 6 considers the use of JS runtimes for the implementation of cloud applications, and proposes NatiSand, a component to control the filesystem, Inter-Process Communication (IPC), and network resources available to binary programs and shared libraries. Interesting future work could extend the observability and protection to a wider set of system resources. For example, the use of memory protection techniques could restrict the area affected by an attack to the sole exploited thread without leading to potential



compromise of the entire JS runtime. Finally, given the flexibility and versatility of our designs another interesting future work would be the adoption of these techniques to secure other Wasm and JS runtimes, or even other interpreted languages (e.g., PHP, Python, and Ruby).



# Acknowledgments

The research described in the contents of this thesis was supervised by Prof. Stefano Paraboschi (Università degli Studi di Bergamo), and has received funding from: 2015 Google Faculty Research Award Program, Horizon 2020 research and innovation programme under grant agreement No. 825333 (MOSAICrOWN), Horizon Europe research and innovation programme under grant agreement No. 101070141 (GLACIATION), and NextGenerationEU programme under grant agreement No. PE00000018 (GRINS).

I would like to thank Prof. Stefano Paraboschi for the guidance during this journey. My gratitude also goes to my former and present colleagues of the Security Lab of Università degli Studi di Bergamo: Marco Abbadini, Enrico Bacis, Michele Beretta, Dario Facchinetti, Gianluca Oldani, and Marco Rosa, with whom I had the opportunity to grow by sharing this experience. 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.



# A. Practical showcase of SEApp capabilities

This chapter gives a technical demonstration of the security measures introduced by SEApp. The description is based on the showcase app presented in Section 3.3. We show that: (1) the showcase app can operate without a policy module; in this mode, its vulnerabilities can be exploited; (2) the showcase app can also operate with the policy module listed in Appendix A.4 and use the services offered by SEApp; in this mode, the internal vulnerabilities are no longer exploitable.

The showcase app has a minimal structure. Its entry point is the *MainActivity* (Figure A.1), which is associated with the *core\_logic* process. From the *MainActivity* it is possible to send a *startActivity* intent to one among *UseCase1Activity*, *UseCase2Activity* and *UseCase3Activity*; the entry points of use cases 1, 2 and 3, respectively. For each entry point *Zygote* starts a dedicated process and, according to the content of the *seapp\_contexts* (in Listing A.1), assigns its specific domain (*user\_logic\_d* to UC#1, *ads\_d* to UC#2, *media\_d* to UC#3). A dedicated description of each use case follows.

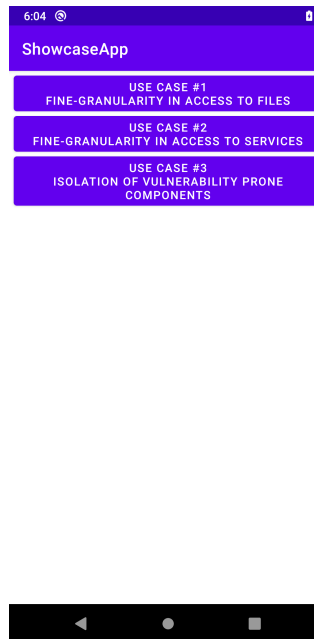


Figure A.1: Showcase app main view

## A.1 UC#1: fine-granularity in access to files

In this use case we demonstrate how an app could benefit from the fine-granularity access to files. In particular, we show how the *UseCase1Activity*, suffering of a path traversal vulnerability, cannot be exploited when the app is associated with a properly configured policy module. According to the Google Play Protect report on common application vulnerabilities [113], unsanitized path names that lead to path traversal are a primary source of problems in applications.

*UseCase1Activity* is quite simple: it displays the content of a file given its relative path through an intent (Figure A.2a). While this may be fine when the intent comes

from trusted components, the activity supports also implicit intents coming from untrusted sources. This makes the vulnerability easily exploitable by an attacker targeting the confidential files written by the *core.logic* components. In our setup phase, we leverage *MainActivity* to create an internal directory structure by using the `android.os.File` abstraction, which sets file and directory context upon its creation (see Section 3.6.2). Two directories are created: `user/` and `confidential/`; inside both folders a file `data` is saved. To test this use case, we first start *UseCase1Activity*, then we send an intent to “confuse” *UseCase1Activity* into showing us the content of `confidential/data`. This can be done via ADB with the command:

```
adb shell am start \
  -n com.example.showcaseapp/.UseCase1Activity \
  -a "com.example.showcaseapp.intent.action.SHOW" \
  --es "com.example.showcaseapp.intent.extra.PATH" \
  "../confidential/data"
```

When the policy module is not available, all app internal files are flagged with `app_data_file` and every app component executes within the `untrusted_app` domain, which holds read access to `app_data_file`. As a consequence the vulnerability is successfully exploited and *UseCase1Activity* shows the content of the `confidential/data` file (Figure A.2b).

Instead, when the policy module is available, the file `confidential/data` is flagged with `confidential_t`, as indicated in line 2 in `file_contexts` (see Listing A.2). Since no permission is granted on `confidential_t` in the `sepolicy.cil` to `user_logic_d`, any access to the file `confidential/data` by *UseCase1Activity* is blocked by SELinux (Figure A.2c). The following denial is written to the system log: *denied search to user\_logic\_d domain on confidential\_t type* (Figure A.3). The `confidential` directory cannot then be accessed despite the exploitation of the path traversal vulnerability.

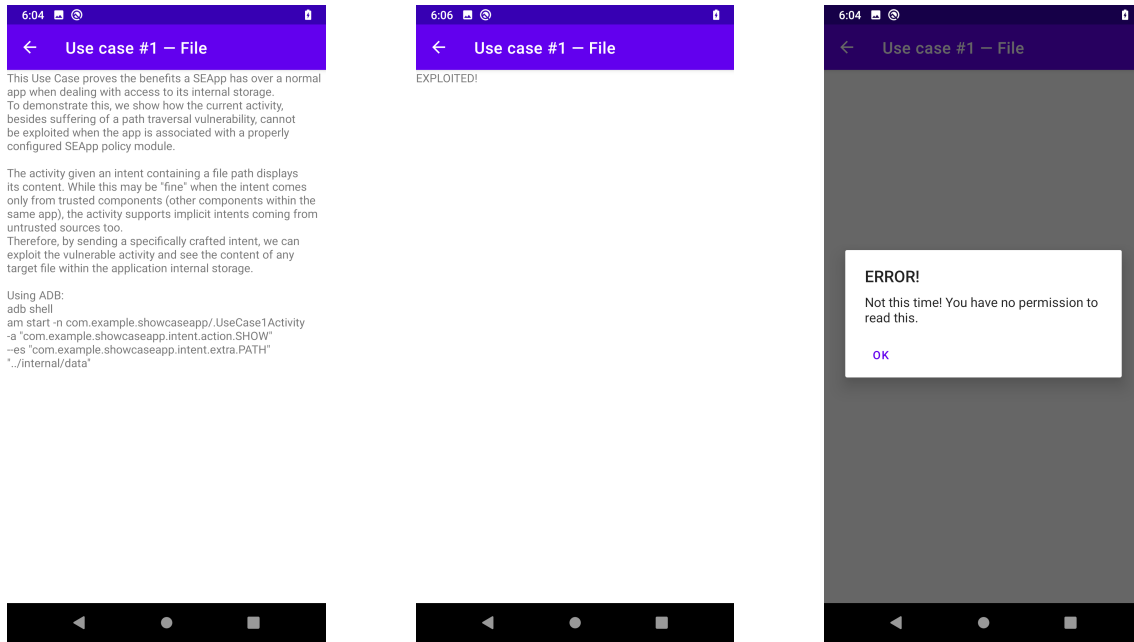


Figure A.2: UC#1 view (initiation, exploitation, and mitigation)

```
01-16 16:39:50.563 4201 4201 W eapp:user logic: type=1400 audit(0.0:29): avc: denied { search } for name="confidential" dev="sda13" i
no=1968083 scontext=u:r:com_example_showcaseapp.user_logic_d:s0:c153,c256,c512,c768 tcontext=u:object_r:com_example_showcaseapp.confide
ntial_t:s0:c153,c256,c512,c768 tclass=dir permissive=0 app=com.example.showcaseapp
```

Figure A.3: UC#1 SELinux denial message in the system log

## A.2 UC#2: fine-granularity in access to services

In this use case we show how to confine an Ad library into an ad-hoc process, with guarantees that it cannot abuse the access privileges granted to the whole application sandbox by the user. To do that, we deliberately inject, in the same process the library is executed, a malicious component (which is directly invoked by the library) that tries to capture the location when the permission *ACCESS\_FINE\_LOCATION* is granted to the app. The Ad library used is Unity Ads [193], which according to [179] in 2020 was used by 11% of apps that show ads.

In this case the library is invoked by *UseCase2Activity* (Figure A.4a), and according to line 3 of the *seapp\_contexts*, both the activity and the components created by the library are executed by *Zygote* in a process labeled with *ads\_d*. To interact with the Ad library, *UseCase2Activity* instances a *UnityAdsListener*. After the Ad initialization (including the registration of the listener) and displaying the Ad to the user (Figures A.4b-c), the Ad framework invokes the listener callback method *onUnityAdsFinish*, which executes the malicious routine *captureLocation*. The routine probes the app permissions; if *ACCESS\_FINE\_LOCATION* is granted to the app, the malicious component retrieves through the *servicemanager* a handle to the *LocationManager*, and registers to it an asynchronous listener to capture GPS

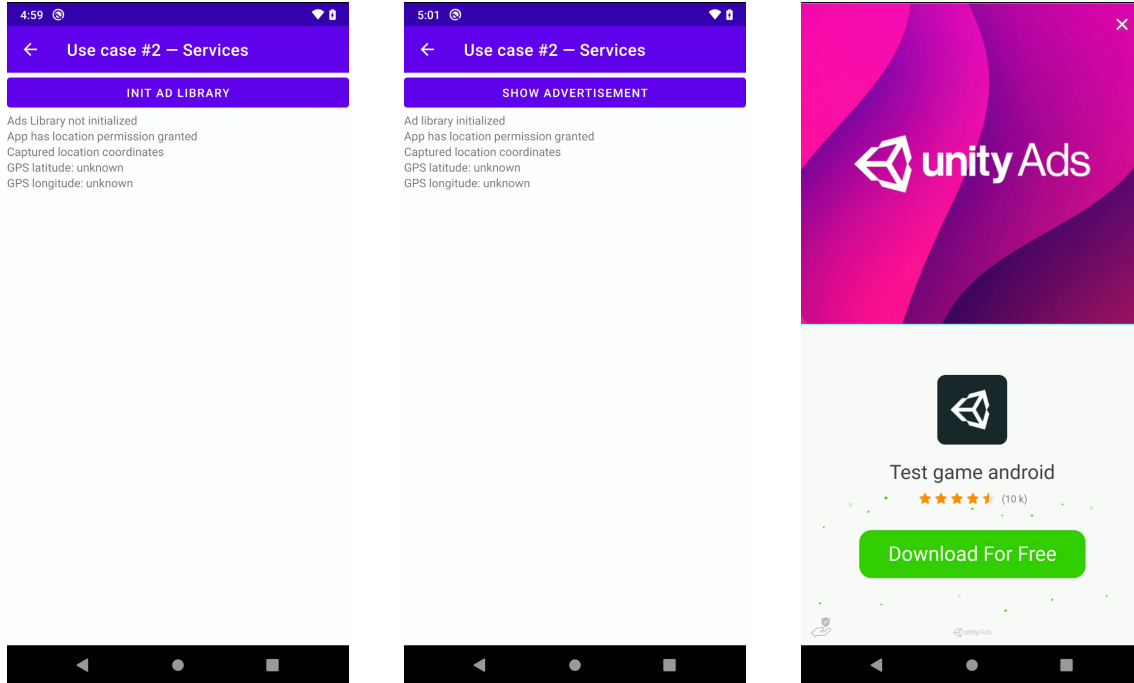


Figure A.4: UC#2 views

location (Figure A.5).

We show that when the policy module is enforced by SEApp, the malicious component cannot access the GPS coordinates. This is because the component is executed in the same process of the library, which is labeled with `ads_d`. If we look at the `sepolicy.cil` (lines 48-54), `ads_d` is not granted access to the SELinux type `location_service`, so the malicious routine cannot retrieve and therefore connect to the `location_service`. The following denial is written to the system log: *denied find on location\_service to the ads\_d domain* (Figure A.6). As a result, the malicious component is terminated by the `ActivityTaskManager` (Figure A.7).

The Ad library was included in the app as an `.aar` archive. To confine it, no modification was necessary, only the use of `AndroidManifest.xml` and `sepolicy.cil` was required.



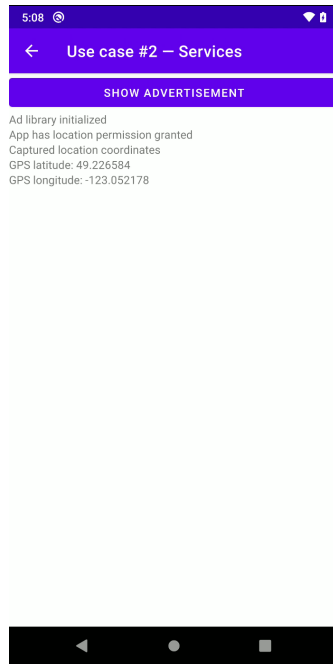


Figure A.5: UC#2 malicious gadget retrieves location data

```
01-15 17:01:57.788  824  824 I /vendor/bin/hw/android.hardware.health@2.0-service.wahoo: SRAM data: 3699000
01-15 17:01:58.325  6735 6735 D showcaseapp:ads d: Capturing location - setup location listener
01-15 17:01:58.325  6735 6735 D showcaseapp:ads d: Capturing location - trying to get a location manager handle
01-15 17:01:58.325  613  613 E SELinux : avc: denied { find } for service=location pid=6735 uid=10148 scontext=u:r:com_example_showcaseapp:ads d:s0:c148,c256,c512,c768 tcontext=u:object r:location service:s0 tclass=service manager permissive=0
```

Figure A.6: UC#2 SELinux denial message in the system log

```
01-15 17:01:58.334  1276 1498 W ActivityTaskManager: Force finishing activity com.example.showcaseapp/.UseCase2Activity
01-15 17:01:58.334  1276 6894 I DropBoxManagerService: add tag=data app crash isTagEnabled=true flags=0x2
01-15 17:01:58.342  6735 6735 I Process : Sending signal. PID: 6735 SIG: 9
01-15 17:01:58.380  1276 1501 W InputDispatcher: channel '135415b com.example.showcaseapp/com.unity3d.services.ads.adunit.AdUnitActivity (server)' - Consumer closed input channel or an error occurred. events=0x9
01-15 17:01:58.380  1276 1501 E InputDispatcher: channel '135415b com.example.showcaseapp/com.unity3d.services.ads.adunit.AdUnitActivity (server)' - Channel is unrecoverably broken and will be disposed!
01-15 17:01:58.384  1276 1501 W InputDispatcher: channel 'ebf97cb com.example.showcaseapp/com.example.showcaseapp.UseCase2Activity (server)' - Consumer closed input channel or an error occurred. events=0x9
01-15 17:01:58.385  1276 1501 E InputDispatcher: channel 'ebf97cb com.example.showcaseapp/com.example.showcaseapp.UseCase2Activity (server)' - Channel is unrecoverably broken and will be disposed!
```

Figure A.7: UC#2 activity termination due to SELinux denial

## A.3 UC#3: isolation of vulnerability-prone components

In this use case we show how to confine a set of components, which rely on a high performance native library written in C to perform some task. Our goal is to demonstrate that the context running the native library code is prevented to access the network, even when the permissions *INTERNET* and *ACCESS\_NETWORK\_STATE* are granted to the app sandbox.

The native library is invoked by *UseCase3Activity* (Figure A.8a), which, according to line 4 in the *seapp\_contexts*, is executed in a process labeled with *media.d* by *Zygote*. The call to the library is performed via JNI. Its job is to connect to the *camera\_service* and take a picture. Since the app is granted the *CAMERA* permis-

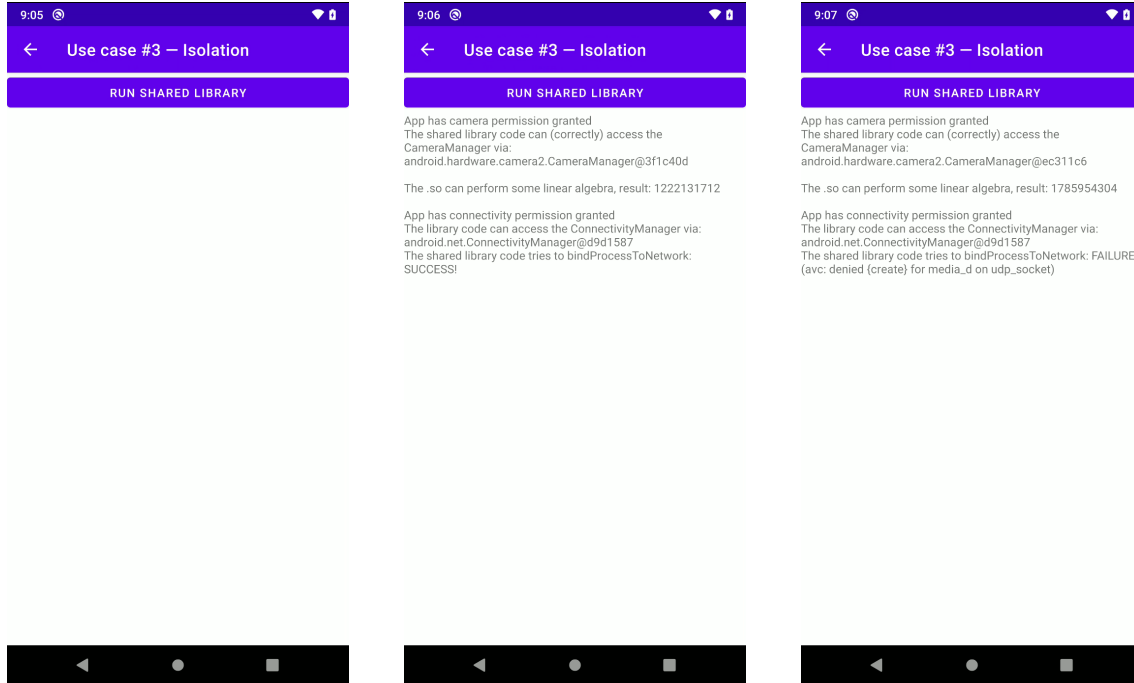


Figure A.8: UC#3 views (initiation, exploitation, and mitigation)

sion, the native library code (legitimately, line 60 in the `sepolicy.cil`) connects to the `CameraManager`.

Since the native library performs image processing, we do not want it to access the network. However, the permissions `INTERNET` and `ACCESS_NETWORK_STATE` are granted to the app, as they are required by the Ads framework. Thus, when the policy module is not available, the native library can connect to the `ConnectivityManager` and successfully bind the current process to the network (Figure A.8b). Instead, when the policy module is enforced by SEApp, since `media_d` was granted only the basic app permissions (line 11 in `sepolicy.cil`), the connection to the network is forbidden (Figure A.8c). This happens because binding a process to the network is associated with opening a network socket, an operation not permitted by SELinux without the required permissions. The following denial is written to the system log: *denied create on udp\_socket to media\_d domain* (Figure A.9).

```
01-15 21:30:04.752 11462 11462 W owcaseapp:media: type=1400 audit(0.0:111): avc: denied { create } for scontext=u:r:com.example.showcaseapp:media_d:s0:c152,c256,c512,c768 tcontext=u:r:com.example.showcaseapp:media_d:s0:c152,c256,c512,c768 tclass=udp_socket permissive=0 app=com.example.showcaseapp
```

Figure A.9: UC#3 SELinux denial message in the system log

This use case, besides showing how SEApp confines a native library, also demonstrates the power and simplicity of the macro, as adding the line (`call md_netdomain (media_d)`) to the policy module grants to `media_d` the needed permissions to access the network. The application developer is thus not required to

know or understand the internal SELinux policy in order to leverage this functionality.

The isolation properties introduced by SEApp applies also to other common security problems presented in [113]. Just to mention one, SEApp can mitigate the impact of incorrect sandboxing of a scripting language.

## A.4 Showcase app policy module

Here are reported the showcase app policy module files.

**Listing A.1: Showcase app seapp\_contexts**

```

1 user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.core_logic_d name=com.example.
  showcaseapp:core_logic levelFrom=all
2 user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.user_logic_d name=com.example.
  showcaseapp:user_logic levelFrom=all
3 user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.ads_d name=com.example.
  showcaseapp levelFrom=all
4 user=_app seinfo=showcase_app domain=
  com_example_showcaseapp.media_d name=com.example.
  showcaseapp:media levelFrom=all

```

**Listing A.2: Showcase app file\_contexts**

```

1 .* u:object_r:app_data_file:s0
2 files/confidential u:object_r:com_example_showcaseapp.
  confidential_t:s0
3 files/ads_cache u:object_r:com_example_showcaseapp.
  ads_t:s0

```

**Listing A.3: Showcase app mac\_permissions.xml**

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <policy>
3   <signer signature="SIGNATURE">
4     <package name="com.example.showcaseapp">
5       <seinfo value="showcase_app"/>
6     </package>
7   </signer>
8 </policy>

```

## Listing A.4: Showcase app sepolicy.cil

```
1 (block com_example_showcaseapp
2   ; creation of domain types
3   (type core_logic_d)
4   (call md_untrusteddomain (core_logic_d))
5   (type user_logic_d)
6   (call md_appdomain (user_logic_d))
7   (type ads_d)
8   (call md_appdomain (ads_d))
9   (call md_netdomain (ads_d))
10  (type media_d)
11  (call md_appdomain (media_d))
12  (typeattribute domains)
13  (typeattributeset domains (core_logic_d user_logic_d ads_d
14    media_d))
15  ; creation of file types
16  (type confidential_t)
17  (call mt_appdatafile (confidential_t))
18  (type ads_t)
19  (call mt_appdatafile (ads_t))
20
21  ; bounding the domains and types
22  (typebounds untrusted_app core_logic_d)
23  (typebounds untrusted_app user_logic_d)
24  (typebounds untrusted_app ads_d)
25  (typebounds untrusted_app media_d)
26  (typebounds app_data_file confidential_t)
27  (typebounds app_data_file ads_t)
28
29  ; grant core_logic_d access to confidential files
30  (allow core_logic_d confidential_t (dir (search write add_name)
31    ))
31  (allow core_logic_d confidential_t (file (create getattr open
32    read write)))
32  ; grant ads_d access to ads_cache files
33  (allow ads_d ads_t (dir (search write add_name)))
34  (allow ads_d ads_t (file (create getattr open read write)))
35
36  ; minimum app_api_service subset
37  (allow domains activity_service (service_manager (find)))
38  (allow domains activity_task_service (service_manager (find)))
39  (allow domains ashmem_device_service (service_manager (find)))
40  (allow domains audio_service (service_manager (find)))
41  (allow domains surfaceflinger_service (service_manager (find)))
42  (allow domains gpu_service (service_manager (find)))
```

```
43
44 ; grant core_logic_d access to the necessary services
45 (allow core_logic_d restorecon_service (service_manager (find))
46 )
47 (allow core_logic_d location_service (service_manager (find)))
48
49 ; grant ads_d access to unity3ads needed services
50 (allow ads_d radio_service (service_manager (find)))
51 (allow ads_d webviewupdate_service (service_manager (find)))
52 (allow ads_d autofill_service (service_manager (find)))
53 (allow ads_d clipboard_service (service_manager (find)))
54 (allow ads_d batterystats_service(service_manager (find)))
55 (allow ads_d batteryproperties_service (service_manager (find))
56 )
57 (allow ads_d audioserver_service (service_manager (find)))
58 (allow ads_d mediaserver_service (service_manager (find)))
59
60 ; grant media_d access to the necessary services
61 (allow media_d autofill_service (service_manager (find)))
62 (allow media_d cameraserer_service (service_manager (find)))
63 )
```



## B. Policy generated for the curl command

Listing B.1 reports the policy associated with the execution of the curl `https://www.example.com` command. The policy has been automatically generated using the utility described in Section 6.5.2.

**Listing B.1: Example of policy associated with curl**

```
1  [{
2    "name": "/usr/bin/curl",
3    "fs": {
4      "read": [
5        "/etc/gai.conf",
6        "/etc/host.conf",
7        "/etc/hosts",
8        "/etc/ld.so.cache",
9        "/etc/localtime",
10       "/etc/nsswitch.conf",
11       "/etc/passwd",
12       "/etc/resolv.conf",
13       "/etc/ssl/certs/ca-certificates.crt",
14       "/lib/x86_64-linux-gnu",
15       "/lib64/ld-linux-x86-64.so.2",
16       "/usr/bin/curl",
17       "/usr/lib/locale/locale-archive",
18       "/usr/lib/ssl/openssl.cnf"
19     ],
20    "exec": [
21      "/lib/x86_64-linux-gnu",
22      "/lib64/ld-linux-x86-64.so.2",
23      "/usr/bin/curl"
24    ]
25  },
26  "net": [{
27    "name": "https://www.example.com",
28    "ports": [443]
29  }]
30 }
```





# References

- [1] A. Starovoitov. CAP\_BPF. <https://lwn.net/Articles/820560/>, 2020.
- [2] M. Abbadini, M. Beretta, S. D. C. di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Supporting data owner control in IPFS networks. In *Proceeding of the IEEE International Conference on Communications (IEEE ICC 2024)*, 2024.
- [3] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Lightweight cloud application sandboxing. In *Proceeding of the 14th IEEE International Conference on Cloud Computing Technology and Science (IEEE CLOUDCOM 2023)*, 2023.
- [4] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Poster: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.
- [5] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. Cage4Deno: A fine-grained sandbox for Deno subprocesses. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.
- [6] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. NatiSand: Native code sandboxing for JavaScript runtimes. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023.
- [7] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons learned from Android security research for appified software platforms. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.
- [8] Adobe Inc. Sandbox protections. <https://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/sandboxprotections.html>, 2023.
- [9] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven trigger-action platforms. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

- [10] M. Albanese, A. De Benedictis, D. D. de Macedo, and F. Messina. Security and trust in cloud application life-cycle management. *Future Generation Computer Systems (FGCS)*, 111, October 2020.
- [11] A. AlHamdan and C. Staicu. SandDriller: A fully-automated approach for testing language-based JavaScript sandboxes. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2023.
- [12] B. Alliance. Cli options for wasmtime. <https://docs.wasmtime.dev/cli-options.html>, 2023.
- [13] N. Andrii. BPF portability and CO-RE. <https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html>, 2020.
- [14] Android. Google Play Protect. <https://www.android.com/play-protect/>, 2021.
- [15] Android Developers. adb install. <https://developer.android.com/studio/command-line/adb#move>, 2021.
- [16] Android Developers. Android App Bundles. <https://developer.android.com/platform/technology/app-bundle>, 2021.
- [17] Android Developers. Android Interface Definition Language. <https://developer.android.com/guide/components/aidl>, 2021.
- [18] Android Developers. android:isolatedProcess. <https://developer.android.com/guide/topics/manifest/service-element#isolated>, 2021.
- [19] Android Developers. Bound services overview. <https://developer.android.com/guide/components/bound-services#Creating>, 2021.
- [20] Android Developers. isolated\_app.te. [https://android.googlesource.com/platform/system/sepolicy/+refs/heads/master/private/isolated\\_app.te](https://android.googlesource.com/platform/system/sepolicy/+refs/heads/master/private/isolated_app.te), 2021.
- [21] Android Developers. Privacy Sandbox su Android. <https://developer.android.com/design-for-safety/privacy-sandbox>, 2023.
- [22] Android Open Source Project. Enable per-user isolation for normal apps. <https://android.googlesource.com/platform/external/sepolicy/+a833763ba04147e840fd054b613f759395bada35>, 2014.

- 
- [23] Android Open Source Project. SELinux for Android 8.0. [https://source.android.com/security/selinux/images/SELinux\\_Treble.pdf](https://source.android.com/security/selinux/images/SELinux_Treble.pdf), 2017.
- [24] Android Open Source Project. Android 9 release notes. [https://source.android.com/setup/start/p-release-notes#per-app\\_selinux\\_sandbox](https://source.android.com/setup/start/p-release-notes#per-app_selinux_sandbox), 2018.
- [25] Android Open Source Project. ActivityManagerService. <https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/am/ActivityManagerService.java>, 2021.
- [26] Android Open Source Project. Android Debug Bridge (adb). <https://developer.android.com/studio/command-line/adb>, 2021.
- [27] Android Open Source Project. Android Permissions. <https://developer.android.com/guide/topics/permissions/overview>, 2021.
- [28] Android Open Source Project. Android Runtime. <https://developer.android.com/guide/platform#art>, 2021.
- [29] Android Open Source Project. App manifest overview. <https://developer.android.com/guide/topics/manifest/manifest-intro>, 2021.
- [30] Android Open Source Project. Binder. <https://developer.android.com/reference/android/os/Binder>, 2021.
- [31] Android Open Source Project. Implementing SELinux. <https://source.android.com/security/selinux/implement>, 2021.
- [32] Android Open Source Project. init. <https://android.googlesource.com/platform/system/core/+/refs/heads/master/init/main.cpp>, 2021.
- [33] Android Open Source Project. installd. <https://android.googlesource.com/platform/frameworks/native/+/refs/heads/master/cmds/installd/>, 2021.
- [34] Android Open Source Project. Intent and intent filters. <https://developer.android.com/guide/components/intents-filters>, 2021.
- [35] Android Open Source Project. Mounting partitions early. <https://source.android.com/devices/architecture/kernel/mounting-partitions-early>, 2021.

- [36] Android Open Source Project. PackageManagerService. <https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/PackageManagerService.java>, 2021.
- [37] Android Open Source Project. PackageParser. <https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/pm/PackageParser.java>, 2021.
- [38] Android Open Source Project. Policy compatibility. <https://source.android.com/security/selinux/compatibility>, 2021.
- [39] Android Open Source Project. restorecond service. <https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/restorecond/restorecond.service>, 2021.
- [40] Android Open Source Project. secilc. <https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/secilc/>, 2021.
- [41] Android Open Source Project. SELinuxMMAC. <https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/SELinuxMMAC.java>, 2021.
- [42] Android Open Source Project. untrusted\_app\_all.te. [https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/untrusted\\_app\\_all.te](https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/untrusted_app_all.te), 2021.
- [43] Android Open Source Project. Zygote. <https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/com/android/internal/os/Zygote.java>, 2021.
- [44] Apple. JavaScriptCore. <https://developer.apple.com/documentation/javascriptcore>, 2023.
- [45] Ars Technica. The Android 11 interview. <https://arstechnica.com/gadgets/2020/09/the-android-11-interview-googlers-answer-our-burning-questions/>, 2020.
- [46] E. Bacis, D. Facchinetti, M. Guarnieri, M. Rosa, M. Rossi, and S. Paraboschi. I told you tomorrow: Practical time-locked secrets using smart contracts. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2021.

- 
- [47] E. Bacis, S. Mutti, and S. Paraboschi. AppPolicyModules: Mandatory access control for third-party apps. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2015.
- [48] E. Bacis, S. Mutti, and S. Paraboschi. Policy specialization to support domain isolation. In *Proceedings of the Workshop on Automated Decision Making for Active Cyber Defense (SafeConfig)*, 2015.
- [49] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in Android and its security applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.
- [50] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.
- [51] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. V. Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock Android. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2015.
- [52] D. Bakker. wasi-sockets. <https://github.com/WebAssembly/wasi-sockets>, 2023.
- [53] M. Bauer and C. Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2021.
- [54] M. Bélair, S. Laniece, and J. Menaud. SNAPPY: Programmable kernel-level policies for containers. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2021.
- [55] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 1995.
- [56] J. Bosamiya, W. S. Lim, and B. Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2022.
- [57] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in JavaScript bindings. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

- [58] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.
- [59] S. Bugiel, S. Heuser, and A. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2013.
- [60] T. Bui, S. P. Rao, M. Antikainen, V. M. Bojan, and T. Aura. Man-in-the-Machine: Exploiting ill-secured communication inside the computer. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018.
- [61] A. Bulekov, R. Jahanshahi, and M. Egele. Sapphire: Sandboxing PHP applications with tailored system call allowlists. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.
- [62] Bun. Bun is a fast all-in-one JavaScript runtime. <https://bun.sh/>, 2023.
- [63] C. Canella, M. Werner, D. Gruss, and M. Schwarz. Automating seccomp filter generation for Linux applications. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2021.
- [64] V. Casola, A. De Benedictis, M. Rak, and U. Villano. Monitoring data security in the cloud: A security sla-based approach. In *Security and Resilience in Intelligent Data-Centric Systems and Communication Networks*, Intelligent Data-Centric Systems. 2018.
- [65] V. Casola, A. De Benedictis, M. Rak, and U. Villano. Security-by-design in multi-cloud applications: An optimization approach. *Information Sciences*, 454-455, July 2018.
- [66] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid policies: Combining MAC and DAC in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.
- [67] G. Christou, G. Ntousakis, E. Lahtinen, S. Ioannidis, V. P. Kemerlis, and N. Vasilakis. BinWrap: Hybrid protection against native Node.js add-ons. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.
- [68] Chromium. Sandbox. <https://chromium.googlesource.com/chromium/src/+refs/heads/main/docs/design/sandbox.md>, 2023.

- 
- [69] B. Coenen. feat(wasi): add rename for a directory + fix remove\_dir. <https://github.com/wasmerio/wasmer/commit/e0e12f9d9ff41a512e44bd497324e>, 2021.
- [70] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard. PKU pitfalls: Attacks on PKU-based memory isolation systems. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.
- [71] containers. Bubblewrap. <https://github.com/containers/bubblewrap>, 2022.
- [72] J. Corbet. File-based capabilities. <https://lwn.net/Articles/211883/>, 2006.
- [73] J. Corbet. BPF: the universal in-kernel virtual machine. <https://lwn.net/Articles/599755/>, 2014.
- [74] CVE Mitre. Gitlab Exiftool vulnerability. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22205>, 2021.
- [75] P. David. hyperfine. <https://github.com/sharkdp/hyperfine>, 3 2023.
- [76] A. Dawoud and S. Bugiel. DroidCap: OS support for capability-based permissions in Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.
- [77] 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. *IEEE Transactions on Big Data*, 9(3), 2022.
- [78] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Multi-dimensional flat indexing for encrypted data. *Under submission*.
- [79] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Artifact: Scalable distributed data anonymization. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021.
- [80] 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 *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, 2021.

- [81] S. De Capitani di Vimercati, D. Facchinetti, S. Foresti, G. Oldani, S. Paraboschi, M. Rossi, and P. Samarati. Scalable distributed data anonymization. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021.
- [82] Debian. Service sandboxing. <https://wiki.debian.org/ServiceSandboxing>, 2023.
- [83] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. sys-filter: Automated system call filtering for commodity software. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.
- [84] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. Gunter. Free for all! Assessing user data exposure to advertising libraries on Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.
- [85] Deno Land. Deno 1.24 release notes – improved ffi call performance. <https://deno.com/blog/v1.24#improved-ffi-call-performance>, 2022.
- [86] Deno Land. Deno 1.25 release notes – FFI API improvements. <https://deno.com/blog/v1.25#ffi-api-improvements>, 2022.
- [87] Deno Land. Deno: JavaScript runtime. <https://deno.land/>, 2022.
- [88] Deno Land. Node compatibility mode. [https://deno.land/manual/node/compatibility\\_mode](https://deno.land/manual/node/compatibility_mode), 2022.
- [89] Deno Land. Deno API. <https://doc.deno.land/deno/stable/>, 2023.
- [90] Deno Land. Deno Permission Model. [https://deno.land/manual/getting\\_started/permissions](https://deno.land/manual/getting_started/permissions), 2023.
- [91] Deno Land. Rusty V8 bindings. [https://github.com/denoland/rusty\\_v8](https://github.com/denoland/rusty_v8), 2023.
- [92] Deno Land. sqlite3 bindings for Deno. <https://deno.land/x/sqlite3>, 2023.
- [93] M. Diamantaris, E. Papadopoulos, E. Markatos, S. Ioannidis, and J. Polakis. REAPER: Real-time app analysis for augmenting the Android permission system. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019.



- 
- [94] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.
- [95] J. Edge. Seccomp and deep argument inspection. <https://lwn.net/Articles/822256/>, 2020.
- [96] Emscripten Contributors. Emscripten toolchain. <https://emscripten.org/>, 2023.
- [97] W. Enck, D. Ocateau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2011.
- [98] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security & Privacy Magazine (IEEE S&P Magazine)*, 7(1), February 2009.
- [99] A. Ene, M. Kolny, and A. Brown. wasi-threads. <https://github.com/WebAssembly/wasi-threads>, 2023.
- [100] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions demystified. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2011.
- [101] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User attention, comprehension, and behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2012.
- [102] 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.
- [103] W. Findlay, D. Barrera, and A. Somayaji. BPFContain: Fixing the soft underbelly of container security. *arXiv*, 2021.
- [104] W. Findlay, A. Somayaji, and D. Barrera. bpfbox: Simple precise process confinement with eBPF. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2020.
- [105] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow considered harmful? The impact of copy paste on

- Android application security. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.
- [106] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna. On the security and engineering implications of finer-grained access controls for Android developers and users. In *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.
- [107] Free Software Foundation. GNU M4. <https://www.gnu.org/savannah-checkouts/gnu/m4/manual/m4-1.4.18/index.html>, 2016.
- [108] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. Sledge: A serverless-first, light-weight Wasm runtime for the edge. In *Proceedings of the International Middleware Conference (MIDDLEWARE)*, 2020.
- [109] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2006.
- [110] Google. Capsicum object-capabilities on Linux. <https://github.com/google/capsicum-linux>, 2017.
- [111] Google. Minijail. <https://google.github.io/minijail/>, 2022.
- [112] Google. sandbox2. <https://developers.google.com/code-sandboxing/sandbox2>, 2023.
- [113] Google Play Protect. Android app vulnerability classes: A whirlwind overview of common security and privacy problems in Android apps. [https://static.googleusercontent.com/media/www.google.com/en//about/appsecurity/play-rewards/Android\\_app\\_vulnerability\\_classes.pdf](https://static.googleusercontent.com/media/www.google.com/en//about/appsecurity/play-rewards/Android_app_vulnerability_classes.pdf), 2021.
- [114] Google Play Store. Android top apps. <https://play.google.com/store/apps/top>, 2021.
- [115] B. Gregg. BPF internals. 2021. USENIX Large Installation System Administration Conference (USENIX LISA).
- [116] 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 Web-Assembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

- 
- [117] HackerOne. External SSRF and Local File Read via video upload due to vulnerable FFmpeg HLS processing. <https://hackerone.com/reports/1062888>, 2021.
- [118] S. Heuser, A. Nadkarni, W. Enck, and A. Sadeghi. ASM: A programmable interface for extending Android security. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2014.
- [119] R. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An architecture for enforcing end-to-end access control over web applications. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2010.
- [120] J. Huang, O. Schranz, S. Bugiel, and M. Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock Android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.
- [121] J. Edge. A seccomp overview. <https://lwn.net/Articles/656307/>, 2015.
- [122] J. Vander Stoep. ioctl command whitelisting in SELinux. <http://kernsec.org/files/lss2015/vanderstoep.pdf>, 2015.
- [123] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu. Programmable system call security with ebpf, 2023.
- [124] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. WaVe: A verifiably secure WebAssembly sandboxing runtime. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2022.
- [125] M. Kehoe. eBPF: The next power tool of SREs. 2022. USENIX SREcon.
- [126] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.
- [127] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2022.

- [128] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of WebAssembly. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.
- [129] libbpf. libbpf. <https://libbpf.readthedocs.io/en/latest/index.html>, 2023.
- [130] Linux manual. bpf(2). <https://man7.org/linux/man-pages/man2/bpf.2.html>, 2022.
- [131] Linux manual. ldd(1). <https://man7.org/linux/man-pages/man1/ldd.1.html>, 2022.
- [132] Linux manual. strace(1). <https://man7.org/linux/man-pages/man1/strace.1.html>, 2022.
- [133] Linux manual. accept. <https://man7.org/linux/man-pages/man2/accept.2.html>, 2023.
- [134] Linux manual. hier. <https://man7.org/linux/man-pages/man7/hier.7.html>, 2023.
- [135] Linux manual. listen. <https://man7.org/linux/man-pages/man2/listen.2.html>, 2023.
- [136] Linux manual. pipe. <https://man7.org/linux/man-pages/man2/pipe.2.html>, 2023.
- [137] Linux manual. socketpair. <https://man7.org/linux/man-pages/man2/socketpair.2.html>, 2023.
- [138] M. Salaün. Landlock: unprivileged access control. <https://docs.kernel.org/userspace-api/landlock.html>, 2022.
- [139] K. MacMillan, C. Case, J. Brindle, and C. Sellers. SELinux Common Intermediate Language motivation and design. <https://github.com/SELinuxProject/cil/wiki>, 2020.
- [140] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kravlevich. The Android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3), April 2021.
- [141] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Conference (USENIX)*, 1993.

- 
- [142] M. McCaskey. Prevent parent directory from being opened without being preopened wasi. <https://github.com/wasmerio/wasmer/pull/463>, 2019.
- [143] Microsoft. ebpf for windows. <https://microsoft.github.io/ebpf-for-windows/>, 2023.
- [144] J. Mogul, R. Rashid, and M. Accetta. The packer filter: An efficient mechanism for user-level network code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1987.
- [145] MozillaWiki. Sandbox Architecture. [https://wiki.mozilla.org/Security/Sandbox/Process\\_model](https://wiki.mozilla.org/Security/Sandbox/Process_model), 2023.
- [146] MUSEC. libpreopen. <https://github.com/musec/libpreopen>, 2023.
- [147] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging “choice” to automate authorization hook placement. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.
- [148] A. Nakryiko. BPF CO-RE. <https://nakryiko.com/posts/bpf-core-reference-guide/>, 2021.
- [149] S. Narayan, C. Disselkoe, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.
- [150] netblue30. Firejail. <https://firejail.wordpress.com/>, 2022.
- [151] NIST. Application Container Security Guide. <https://csrc.nist.gov/publications/detail/sp/800-190/final>, 2023.
- [152] npm. Npm packages. <https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages.html>, 2020.
- [153] npm. fluent-ffmpeg. <https://www.npmjs.com/package/fluent-ffmpeg>, 2022.
- [154] npm. gm. <https://www.npmjs.com/package/gm>, 2022.
- [155] npm. bcrypt. <https://www.npmjs.com/package/bcrypt>, 2023.
- [156] npm. sharp. <https://www.npmjs.com/package/sharp>, 2023.
- [157] G. Ntousakis, S. Ioannidis, and N. Vasilakis. Detecting third-party library problems with combined program analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

- [158] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [159] OpenJS Foundation. Node.js API. <https://nodejs.org/docs/latest/api/>, 2023.
- [160] OpenJS Foundation. Node Permissions. <https://nodejs.org/api/permissions.html>, 2023.
- [161] OpenJS Foundation. Node.js V8 APIs. <https://nodejs.org/api/v8.html>, 2023.
- [162] OpenJS Foundation and Joyent. Node.js. <https://nodejs.org>, 2022.
- [163] oven sh. Webcore bindings. <https://github.com/oven-sh/bun/tree/main/src/bun.js/bindings/webcore>, 2023.
- [164] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova. eWASM: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11), October 2020.
- [165] P. Pearce, A. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2012.
- [166] R. Dahl. 10 things i regret about Node.js. 2018. JSConf EU.
- [167] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi. SEApp: Bringing mandatory access control to Android apps. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.
- [168] R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys (CSUR)*, 28(1), March 1996.
- [169] F. Schwarz and C. Rossow. SENG, the SGX-Enforcing network gateway: Authorizing communication from shielded clients. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.
- [170] U. Shankar, T. Jaeger, and R. Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2006.

- 
- [171] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The misuse of Android Unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.
- [172] Slim.AI. SlimToolKit. <https://github.com/slimtoolkit/slim>, 2023.
- [173] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.
- [174] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 2001.
- [175] Snyk. State of Open Source Security 2022. <https://snyk.io/reports/open-source-security/>, 2022.
- [176] C. Staicu, M. Pradel, and B. Livshits. Synode: Understanding and automatically preventing injection attacks on Node.js. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [177] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2023.
- [178] StatCounter Global Stats. Mobile Operating System Market Share Worldwide. <https://gs.statcounter.com/os-market-share/mobile/worldwide>, 2024.
- [179] Statista. Most popular installed ad network software development kits (SDKs) across Android apps worldwide as of September 2020. <https://www.statista.com/statistics/1035623/leading-mobile-app-ad-network-sdks-android/>, 2020.
- [180] J. Terrace, S. R. Beard, and N. P. K. Katta. JavaScript in JavaScript(js.js): Sandboxing third-party scripts. In *Proceedings of the USENIX Conference on Web Application Development (USENIX WebApps)*, 2012.
- [181] tesseract-ocr. Tesseract. <https://github.com/tesseract-ocr/tesseract>, 2023.
- [182] The Cilium Authors. Cilium. <https://cilium.io>, 2023.
- [183] The Cilium Authors. Cilium GitHub repository. <https://cilium.io>, 2023.

- [184] The coreutils Authors. `utils coreutils`. <https://github.com/uutils/coreutils>, 2023.
- [185] The Falco Authors. `Falco`. <https://falco.org>, 2023.
- [186] The Falco Authors. What is the performance overhead or resource utilization of Falco? <https://falco.org/about/faq/#what-is-the-performance-overhead-or-resource-utilization-of-falco>, 2023.
- [187] The kernel development community. LSM eBPF Programs. [https://docs.kernel.org/bpf/prog\\_lsm.html](https://docs.kernel.org/bpf/prog_lsm.html), 2023.
- [188] The Kubernetes Authors. Restrict a Container’s Syscalls with seccomp. <https://kubernetes.io/docs/tutorials/security/seccomp/>, 2023.
- [189] The SELinux Project. Type Enforcement. [https://selinuxproject.org/page/NB\\_TE](https://selinuxproject.org/page/NB_TE), 2015.
- [190] The SELinux Project. `libselinux`. <https://github.com/SELinuxProject/selinux/tree/master/libselinux>, 2021.
- [191] The Tetragon authors. `Tetragon`. <https://tetragon.cilium.io/>, 2023.
- [192] TryGhost. Asynchronous, non-blocking SQLite3 bindings for Node.js. <https://www.npmjs.com/package/sqlite3>, 2023.
- [193] Unity. Unity Ads. <https://unity.com/solutions/unity-ads>, 2021.
- [194] V8 project. Unsafe fast JS calls. <https://v8.dev/blog/v8-release-87#unsafe-fast-js-calls>, 2020.
- [195] V8 project. What is v8? <https://v8.dev/>, 2022.
- [196] V8 project. Webassembly compilation pipeline. <https://v8.dev/docs/wasm-compilation-pipeline>, 2023.
- [197] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, flexible application compartmentalization. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [198] N. Vasilakis, C. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel. Preventing dynamic library compromise on Node.js via RWX-based privilege reduction. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.



- 
- [199] WasmEdge. wasmedgec AOT compiler. <https://wasmedge.org/book/en/cli/wasmedgec.html>, 2023.
- [200] WebAssembly. WASI Libc. <https://github.com/WebAssembly/wasi-libc>, 2023.
- [201] WebAssembly. WASI SDK. <https://github.com/WebAssembly/wasi-sdk>, 2023.
- [202] WebAssembly. The webassembly system interface. <https://wasi.dev>, 2023.
- [203] C. Wright, C. Cowan, J. Morris, James, S. Smalley, and G. Kroah-Hartman. Linux Security Module framework. In *Ottawa Linux Symposium*, 2002.
- [204] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2012.
- [205] E. Wyss, A. Wittman, D. Davidson, and L. De Carli. Wolf at the door: Preventing install-time attacks in npm with latch. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2022.
- [206] Z. Xiao, A. Amit, and D. Wenliang. AFrame: Isolating advertisements from mobile applications in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.
- [207] Zerodium. Zerodium - The leading exploit acquisition platform. <https://zerodium.com>, 2021.
- [208] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.