Proceedings of

The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023)

October 16-18, 2023 | Hong Kong, China

ISBN: 979-8-4007-0765-0

RAiD 2023

The Association for Computing Machinery
1601 Broadway, 10th Floor
New York, New York 10019, USA

ACM ISBN: 979-8-4007-0765-0

Message from the RAID 2023 Program Co-Chairs

Welcome to the Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023)!

We are grateful for the immense community services that helped shape the program of RAID 2023. The proceedings are the result of thousands of hours of hard review work. The RAID 2023 program committee carefully reviewed 213 submissions, out of which 50 were accepted to appear in the program (23.5% acceptance rate). The RAID 2023 conference proceedings present a comprehensive compilation of cutting-edge research papers in the field of cybersecurity. The papers included here were carefully selected by the Program Committee, guaranteeing their significance and originality. We believe that this collection will serve as a valuable resource for researchers, practitioners, and industry professionals working in the domain of cybersecurity.

We express our profound gratitude to the 67 program committee members (full list see below) who provided insightful comments and evaluations for the submitted papers. Your commitment to maintaining the academic rigor and quality of RAID 2023 is highly commendable. Although we significantly increased the PC size compared to the previous year, PC members still had a higher reviewer load than expected due to a ~50% increase in submissions. We owe our deepest gratitude to the PC members that made up for this extra load. Special thanks go to those PC members that took over a shepherding task to guide conditionally-accepted papers, all of which were accepted ultimately: Adrian Coleșa (twice), Alessandro Sorniotti, Amin Kharraz, Antonio Bianchi, Babak Amin Azad, Daoyuan Wu, Ding Wang, Diogo Barradas, Fabian Schwarz, Georgios Smaragdakis, Imtiaz Karim, Kangjie Lu, Kevin Borgolte, Kimberly Ruth, Konrad Rieck, Marco Cova, Meng Xu, Mridula Singh, Mulong Luo, Nathan Burow, Omar Alrawi, Parjanya Vyas, Ruimin Sun, Sanghyun Hong, Shuai Hao, Simone Aonzo, Yue Zhang, and Zachary Weinberg. We also would like to thank the ten junior PC members and the large number of high-quality nominations we received for additional PC members. It was a pleasure working with all of you and seeing your intense motivation and detailed, timely, on-spot reviews.

We extend our sincerest appreciation to all external reviewers who contributed to the review process with their detailed paper assessments: Amit Ahlawat, Antonio Ruggia, Biagio Montaruli, Brian Kondracki, Guanhong Tao, Jinjin Liang, Johnny So, Ju Chen, Lucjan Hanzlik, Minghui Li, Minxin Du, Muhammad Umar, Nils Hendrik Lukas, Sajad Meisami, Samin Y. Chowdhury, Urs Hengartner, Yuefeng Du, Yufei Han, and Yunpeng Luo.

We thank the members of the organizing committee and the corporate sponsors for making the conference possible. This year, RAID offers two awards for papers that deserve to be particularly recognized; the Best Practical Paper award introduced last year, and the Best Paper award introduced in 2018. The Best Practical Paper award recognizes a paper that would be impactful and easy to adopt in practice. The Best Paper award recognizes the best overall research from among the accepted papers. A subcommittee, composed of Diogo

Barradas, Hamed Okhravi, Konrad Rieck, Marcus Botacin, Martin Henze, and Yueqi Chen selected candidate papers for both awards, debated their merits, and finally voted the best papers. For the Best Practical Paper, the subcommittee initially agreed on a set of concrete criteria for judging the practicality of research papers. For instance, they could present defenses that are ready for a Transition to Practice, or measurements with actionable implications; the subcommittee considered out of scope, e.g., attacks that need a lot of prerequisites or papers describing tools that are not released. Our hope is that this award will continue to raise awareness of research advances among security professionals.

RAID 2023 would not have been possible without the generous support and contributions from our sponsors. We acknowledge their dedication to advancing the field of security research and fostering an atmosphere of collaboration and knowledge exchange. Our sincere appreciation goes out to our diamond sponsor Blocksec Inc and our platinum sponsor Resilient Computing and Cybersecurity Center (RC3), KAUST. Furthermore, we would like to thank RC3, KAUST for also sponsoring the Best Practical Paper award with a prize of 2,000 EUR, continuing the success of this additional award introduced 2022.

We extend our heartfelt gratitude to the organizing committee for their unwavering dedication and support in preparing the RAID 2023 proceedings. Above all, we express our deepest appreciation to Xiapu Luo and Man Ho Au, the general chairs, for masterfully organizing RAID 2023 in Hong Kong and ensuring the seamless operation of the conference. Special thanks go to Wei Meng, the publication chair, Michael Schwarz and Haoyu Wang, the publicity chairs, Qingchuan Zhao and Chenxiong Qian, the local organization chair, and Yangyang Liu and Edith Yiu, the website organization chairs, for their invaluable contributions.

As we conclude, we would like to convey our profound thanks to the entire security research community, without whom these proceedings would not have come to fruition. We hope you enjoy RAID 2023 and find it intellectually stimulating and professionally enriching. Thank you for your participation and support, and we look forward to seeing you at the conference.

Christian Rossow and Yousra Aafer
RAID 2023 PC Co-Chairs

Conference Committees

General Chairs

Xiapu Luo, The Hong Kong Polytechnic University

Man Ho Au, The Hong Kong Polytechnic University

Program Committee Chair

Christian Rossow, CISPA Helmholtz Center for Information Security

Program Committee Co-Chair

Yousra Aafer, University of Waterloo

Publication Chair

Wei Meng, The Chinese University of Hong Kong

Publicity Chairs

Michael Schwarz, CISPA Helmholtz Center for Information Security

Haoyu Wang, Huazhong University of Science and Technology

Local Organization Chairs

Qingchuan Zhao, City University of Hong Kong

Chenxiong Qian, The University of Hong Kong

Website Organization Chairs

Yangyang Liu, The Hong Kong Polytechnic University

Edith Yiu, The Hong Kong Polytechnic University

Steering Committee

Johanna Amann, International Computer Science Institute

Davide Balzarotti, EURECOM Graduate School and Research Center

Marc Dacier, King Abdullah University of Science and Technology

Zhiqiang Lin, The Ohio State University

Mathias Payer, EPFL

Michalis Polychronakis, Stony Brook University

Michael Bailey, Georgia Institute of Technology

Angelos Stavrou, Virginia Tech

Leyla Bilge, NortonLifeLock Research Group

Program Committee

Adrian Coleșa, Technical University of Cluj-Napoca

Alessandro Sorniotti, IBM Research Europe

Amin Kharraz, Florida International University

Antonio Bianchi, Purdue University

Aurélien Francillon, EURECOM

Babak Amin Azad, Cloudflare

Balakrishnan Chandrasekaran, Vrije Universiteit Amsterdam

Biswabandan Panda, IIT Bombay

BumJun Kwon, Samsung Research

Carrie Gates, Bank of America

Christian Rossow, CISPA Helmholtz Center for Information Security

Daoyuan Wu, The Chinese University of Hong Kong

Ding Wang, Nankai University

Diogo Barradas, University of Waterloo

Doowon Kim, University of Tennessee, Knoxville

Fabian Schwarz, CISPA Helmholtz Center for Information Security

Gautam Akiwate, Stanford University

Georgios Smaragdakis, TU Delft

Ghada Dessouky, Google

Gururaj Saileshwar, University of Toronto / NVIDIA Research

Haining Wang, Virginia Tech

Haixin Duan, Tsinghua University; Zhongguancun Laboratory

Hamed Okhravi, MIT Lincoln Laboratory

Heng Yin, UC Riverside

Hongjun Choi, DGIST

Hyungsub Kim, Purdue University

Imtiaz Karim, Purdue University

Jiyong Jang, IBM Research

Kangjie Lu, University of Minnesota

Kevin Borgolte, Ruhr University Bochum

Khaled Serag, Purdue University

Kimberly Ruth, Stanford University

Konrad Rieck, TU Berlin

Liz Izhikevich, Stanford University

Lorenzo Cavallaro, University College London

Marco Cova, VMware

Marcus Botacin, Texas A&M University

Martin Henze, RWTH Aachen University & Fraunhofer FKIE

Matteo Grosse-Kampmann, Institute for Internet-Security and AWARE7 GmbH

Meng Xu, University of Waterloo

Merve Sahin, SAP Security Research

Mridula Singh, CISPA Helmholtz Center for Information Security

Mu Zhang, University of Utah

Mulong Luo, Cornell University

Nathan Burow, MIT Lincoln Laboratory

Nick Nikiforakis, Stony Brook University

Omar Alrawi, Georgia Institute of Technology

Parjanya Vyas, University of Waterloo

Peisen Yao, Zhejiang University

Qingchuan Zhao, City University of Hong Kong

Ruimin Sun, Florida International University

Sanghyun Hong, Oregon State University

Sherman S. M. Chow, The Chinese University of Hong Kong

Shiqing Ma, UMass Amherst

Shuai Hao, Old Dominion University

Simone Aonzo, EURECOM

Srdjan Matic, IMDEA Software Institute

Stefano Zanero, Politecnico di Milano

Thijs van Ede, University of Twente

Wei You, Renmin University of China

Yousra Aafer, University of Waterloo

Yuan Hong, University of Connecticut

Yue Duan, Illinois Institute of Technology

Yue Zhang, Jinan University

Yueqi Chen, University of Colorado Boulder

Zachary Weinberg, Carnegie Mellon University

Zhenpeng Lin, Northwestern University

Table of Contents

IoT / Firmware / Binaries

IDS and Applied Crypto

Deep into Systems and Formats

ML (I): Inference and Toxicity

ML (II): Adversarial, Robust and Explainable AI

Network and Cloud Security

Malware and Fuzzing

Software Security I

Software Security II

Web Security and Authentication

NatiSand: Native Code Sandboxing for JavaScript Runtimes

Marco Abbadini
marco.abbadini@unibg.it
Università degli studi di Bergamo
Bergamo, Italy

Dario Facchinetti
dario.facchinetti@unibg.it
Università degli studi di Bergamo
Bergamo, Italy

Gianluca Oldani
gianluca.oldani@unibg.it
Università degli studi di Bergamo
Bergamo, Italy

Matthew Rossi
matthew.rossi@unibg.it
Università degli studi di Bergamo
Bergamo, Italy

Stefano Paraboschi
stefano.paraboschi@unibg.it
Università degli studi di Bergamo
Bergamo, Italy

ABSTRACT

Modern runtimes render JavaScript code in a secure and isolated environment, but when they execute binary programs and shared libraries, no isolation guarantees are provided. This is an important limitation, and it affects many popular runtimes including Node.js, Deno, and Bun [20, 61].

In this paper we propose 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. NatiSand does not require changes to the application code and offers to the user an easy interface. To demonstrate the effectiveness and efficiency of our approach we implemented NatiSand and integrated it into Deno, a modern, security-oriented JavaScript runtime. We reproduced a number of vulnerabilities affecting third-party code, showing how they are mitigated by NatiSand. We also conducted an extensive experimental evaluation to assess the performance, proving that our approach is competitive with state of the art code sandboxing solutions. The implementation is available open source.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Access control**; **Web application security**.

KEYWORDS

Sandboxing, Access Control, Web Application Security, JavaScript Runtime, Deno, Native Code Isolation

ACM Reference Format:
Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. NatiSand: Native Code Sandboxing for JavaScript Runtimes. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3607199.3607233

RAID '23, October 16–18, 2023, Hong Kong, China
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0765-0/23/10.
https://doi.org/10.1145/3607199.3607233

1 INTRODUCTION

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 [54].

The execution of JS code on the server-side is enabled by a *JS runtime*. Since its introduction in 2009, Node.js [62] has been the de facto solution selected by developers, but recently Deno [37] and Bun [13] 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 [81] in the case of Node.js and Deno, JavaScriptCore [6] 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 [38, 60]. 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 [68], and the solution identified by the community was to configure the resources available to an application with simple permission flags [20]. This change also influenced the design of Node.js, which introduced a similar flag-based control model[1] two years later [61]. 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 [20].

Previous research [25, 74] 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,

[1] Node.js support for creating security policies is still experimental as of 2023.

it minimizes the cost of development. Notable examples are: the node-sqlite3 [79] and deno-sqlite3 [40] database drivers; modules to perform image/video conversions, such as sharp [58], fluent-ffmpeg [56] and gm [57]; OCR engines like Tesseract [76]; and the cryptography modules relying on bcrypt [55]. The 2022 State of Open Source Security [72] claims that each open source JS project relies on an average of 174 third-party dependencies; also, each project is estimated to be affected by 40 vulnerabilities when its dependencies are taken into account. Taking into consideration that web applications in most cases process untrusted input, the risk of security incidents is high. For instance, we identified a sample of 32 high severity CVEs[2] that affect native code used by popular packages (with 2.6M downloads/week), and allow an adversary to corrupt the filesystem, perform privilege escalation, execute arbitrary code, open network connections to exfiltrate data, etc.

Our contribution. We see a security gap in the way modern JS runtimes execute native code, as neither Node.js, nor Deno, nor Bun sandbox it. In this work 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* [78] and Linux Security Modules (LSMs), such as *Landlock* [50] and *eBPF* [18] to restrict access to protected resources. In the design of our solution we paid attention to usability by developers; it is not necessary to have a full understanding of its advanced security features to use it. The developer is only required to provide a concise and readable JSON-formatted policy file, detailing the *ambient rights* – i.e., the access privileges available to the components of the web application that rely on native code. To this end, we provide the developer a comprehensive and interactive CLI tool to support policy generation, which, as best practice suggests, can also be integrated into CI/CD pipelines and run against a set of test cases [3, 66]. 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.

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.

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 1, Node.js and Deno embed Google's V8 [81], while Bun relies on JavaScriptCore [6]. The interoperability between runtime and engine is achieved with specialized bindings, which are defined in the *node:v8* [63] module in the case of Node.js, in the *rusty_v8* [39] library for Deno, and by *webcore* [64] 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) [10, 22]. 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 [20, 61]. 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 [20].

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 [20]. 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 1, which means that this code runs with the same privileges of the user executing the entire JS application.

[2] The list is reported in Table 3.

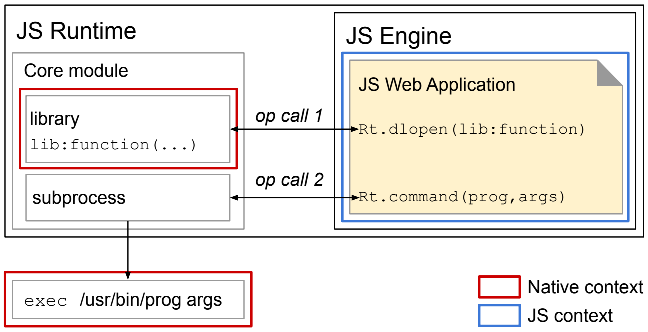

Figure 1: Execution of binary programs and shared library functions by the JS runtime

2.2 Components for resource protection

Landlock. Landlock [50] is a Linux Security Module (LSM) introduced in the kernel starting from release 5.13. The goal of Landlock is to enable unprivileged applications to restrict their ambient rights in accordance with the least privilege principle. Ambient rights are specified by *rulesets* – i.e., simple structures that associate a set of permissible actions with a filesystem path (e.g., read and exec over the resources stored in /tmp). Several rulesets can be combined to determine the final set of actions available to an application. To make them effective, a call to the landlock_restrict_self() function is performed.

The ambient rights granted by Landlock are thread-based, and are automatically 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 [49]. The goal was to provide an in-kernel facility to filter and multiplex network packets, similarly to what was proposed by Mogul et al. [51]. This version of BPF, which is now commonly referred to as *classic BPF* (cBPF), was greatly revised in 2014 resulting in *extended BPF* (eBPF) [18]. The new framework provides an environment to execute programs inside the kernel [30, 32]. 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 [77]: system calls, kernel tracepoints, network events, function entry/exit points, and LSM hooks. Specifically, LSM hooks correspond to the functions used by LSMs (e.g. SELinux) to perform security decisions and are characterized by operating entirely on arguments in kernel memory.

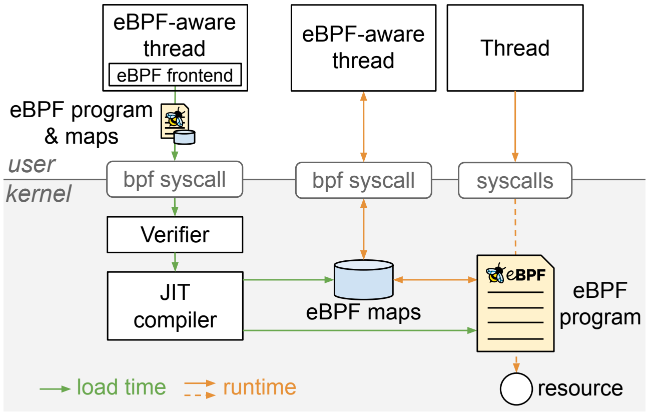

Figure 2: Overview of the eBPF architecture

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 [43]. This is a privileged operation that requires a few capabilities, which vary with the nature of the program [4]. 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 2). In our work we rely on *libbpf* [41], a modern library leveraging the *Compile Once-Run Everywhere* (CO-RE) approach [5], which ensures that the bytecode produced at compile time works correctly across different kernel versions.

Seccomp. Seccomp [78] 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) [31]. 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 [23].

3 SECURITY MOTIVATION

JS runtimes let developers specify the set of access privileges given to a web application [20, 61]. 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 [73, 87]. 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 [20].

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 [36]. Previous work [74] 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 [9]. 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 [11, 70]. 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 [12]. In addition, attackers may open network channels to remotely recover data obtained on the vulnerable host [69]. 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.

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.

4 DESIGN AND IMPLEMENTATION

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

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.

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 [71], 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.

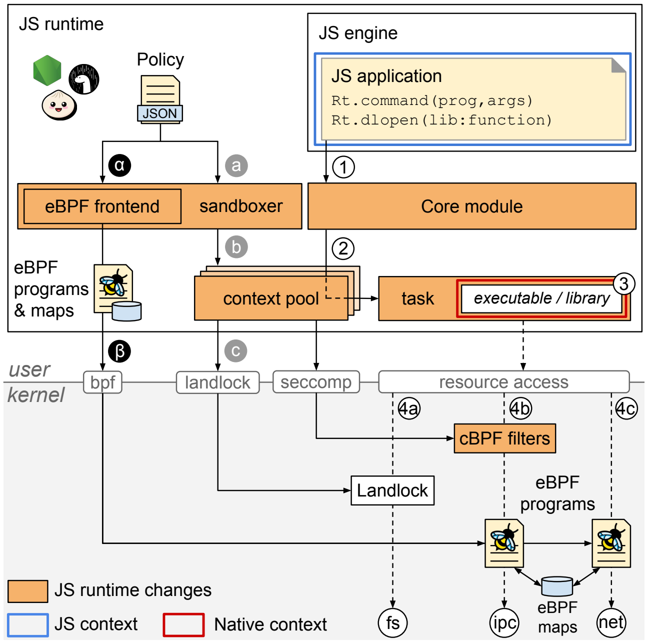

Figure 3: Integration of NatiSand in the JS runtime. Bootstrap: import security contexts (α, β), creation of the context pool (a, b, c). Application runtime: isolated execution of binary programs and shared library functions (1, 2, 3, 4a, 4b, 4c)

The architecture of our solution is shown in Figure 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 4.3, while details about its isolation features are reported in Section 4.4. The policy syntax used by NatiSand to configure permissions, along with the support to policy generation, are described in Section 5.

4.3 Integration with JS runtimes

NatiSand complements the architecture of the JS runtime with the addition of the *sandboxer*, a component that parses the policy and enforces the isolation of native code accordingly. In the following we explain the operations performed by the runtime to use it, referring to Figure 3.

Bootstrap. The JS runtime boot procedure is modified to read the JSON policy file, which is then parsed by the sandboxer α to retrieve the information associated with each security context. Based on the policy, the sandboxer (i) configures the required Seccomp filters, and (ii) prepares and loads into the kernel the necessary eBPF programs and maps β. The eBPF programs are used to track the security contexts and enforce network-related and (part of) IPC-related restrictions (more details in Section 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 [17]. 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 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 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
`uprobe/attach_policy`
`tp_btf/sched_process_fork`
`tp_btf/sched_process_exit`

(a)

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

(b)

Table 1: Hooks and tracepoints monitored by NatiSand

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 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 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 [46]. 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 [47]. 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 [23]). 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 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 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 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 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.

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

Table 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

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 [45]. The same applies to accept, which is used to extract the first connection request from the queue of pending connections [42].

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 [7, 14, 34, 52, 82, 85]. In general, these works are compatible with the design of our solution, therefore they could be used in conjuction with NatiSand to improve isolation of shared libraries. Among them, BreakApp [82] and BinWrap [14] propose approaches tailored for interpreted languages, but they require either the introduction of wrappers or the execution of remote procedure calls. RLBox [52] 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 [15] these solutions can be bypassed using kernel functions that are agnostic of intra-process isolation (i.e., the attacks use the kernel as a confused deputy). Since NatiSand is completely aligned with the memory isolation assumptions made by the kernel, our solution is not affected by this issue.

5 POLICY

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

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 1 (the complete example is reported in Appendix A), 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 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 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.

Listing 1: Example of JSON policy file with single context

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

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 1, curl is allowed to use abstract, named, and unnamed Unix sockets. It is up to our sandboxer to abstract away the complexity of the underlying architecture and enforce the policy when IPC is performed between groups of threads associated with separate contexts. No understanding of the standards available (e.g., System V, POSIX) is required by 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 1, the process executing curl is only allowed to connect to `https://www.example.com` on port 443.

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 SlimToolkit [66], 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* [44] to discover the direct and transient dependencies, then, it relies on *strace* [48] to monitor the interaction between the kernel and a traced binary, so to complement the information previously found with additional filesystem permissions.

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

Network. Network rules are relatively easy to write. However, we do not assume developers to be necessarily aware of every network connection needed by the native code. So, we automate the generation of the policy by observing the execution of the binary with eBPF programs. In fact, these provide a list of the domain names resolved, their IPs, and those hardcoded IPs the utility connects to without performing name resolution. To track domain name resolutions we attach a `uprobe` to the `getaddrinfo` function of the libc library, for IPs we observe network socket connections using `kprobes` on `socket_bind` and `socket_connect` LSM hooks. To handle IP address migrations that may occur at runtime, we similarly propose to capture the list of IPs returned by the `getaddrinfo` with a dedicated eBPF program, which also updates the eBPF map of allowed hosts accordingly. By doing so we make sure that the security checks reflect the policy. This approach can also be extended monitorng DNS traffic on port 53, hence providing support for native components that do not rely on libc functions.

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

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 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 4.1). In this section we explain our decision, then we highlight the main architectural changes.

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 still in beta version (v0.5.8), so it is 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 [61]. Deno was instead designed with the protection of the host as one of its main goals [68], thus no access to privileged system resources is given to JS applications unless the developer explicitly grants it. Deno provides the *Node Compatibility Mode* [21], a feature enabling the reuse of code and libraries originally built for Node.js. The availability of this function permits to import packages hosted by Deno on `deno.land/x`, as well as modules published to npm. To conclude, Node.js and Deno prevail over Bun on all three dimensions. Node.js wins over Deno on popularity (but Deno is quickly growing), they are comparable in terms of third-party modules, and Deno significantly outperforms Node.js on security oriented-features, leading us to choose Deno.

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 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.

Listing 2: Code sandboxing with `nativeCall()`

```
function query(db, stmt) {
    const sqliteDB = new sqlite3.Database(db);
    const query = sqliteDB.prepare(stmt);
    const tuples = query.all();
    sqlite_db.close();
    return tuples;
}
const db = "database.db";
const stmt = "SELECT * FROM table";
const ts = Deno.nativeCall(query, [db, stmt]);
console.log(ts); // print tuples
```

6.3 Support to fast JS calls

In October 2020 V8 announced the support to fast JS calls [80]. The function allows V8 to directly invoke optimized native functions without leveraging the bindings that 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 [35]. 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 2 shows how to sandbox the functions from the native database driver *sqlite3*.

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 7.1), then we investigate the performance of our approach (Section 7.2). Both tests use a server with Ubuntu 22.04 LTS, an AMD Ryzen 3900X CPU, 64 GB RAM, and 2 TB SSD.

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 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 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

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 3: Sample of CVEs mitigated by NatiSand

introduced in the experiment was the specification of a security policy through the `native-sandbox` CLI argument. The policy was generated using the tool described in Section 5.2. 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, nor it can make infeasible to use them in an exploit chain.

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.

7.2.1 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* [28] and *Sandbox2* [29]. 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 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.

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

Table 4: Average execution time for common Linux utilities

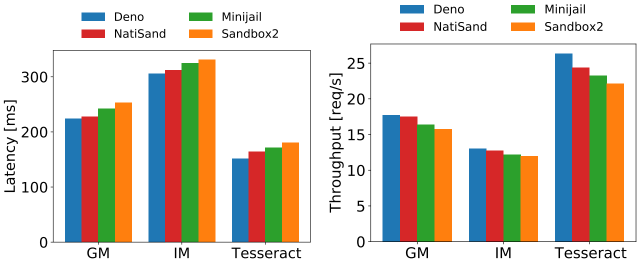

Figure 4: Average latency and throughput for microservices that execute subprocesses

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 GraphicsMagick and ImageMagick, to perform a *sharpen* operation on images input by the client, while the third microservice relies on Tesseract to perform Optical Character Recognition on a second sequence of images input by the client. Similarly to the previous case, the test was repeated for each of the four configurations: Deno, NatiSand, Minijail, and Sandbox2. This time the HTTP benchmarking tool *wrk* was used to measure the performance of each microservice. Network bandwidth and latency are 1 Gbps and 10 ms, respectively, while 100 warmup requests were carried out. Figure 4 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.

7.2.2 *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 [52].

Benchmark III. Similarly to Benchmark I, we implemented a JS application to highlight the overhead experienced on the server when native libraries are executed. In this case three configurations are evaluated: Deno, NatiSand, and Wasm. The application tests the operations provided by four popular libraries: (i) libxml2, to open and query XML data, (ii) libpng, to read metadata information and verify the signature of a png image, (iii) opus to encode and create an audio trace, and (iv) sqlite3, to open and query the Northwind database. Test durations were again measured with `Deno.bench()`, and the results are reported in Table 5. 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 [65] remains slower than its native counterpart. Remarkable are the cases of opus and sqlite3, which used `nativeCall` and demonstrate its efficiency.

Benchmark IV. To understand the slowdown perceived by a remote client, we exposed the functions of the libpng, opus, and sqlite3 libraries with microservices. For each of them, we configured the client to send the input to the server, and measured the latency and throughput using *wrk* (as explained in Benchmark II setup). The results are visualized in Figure 5. 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.

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

Table 5: Average execution time for common native libraries (* marks the use of `nativeCall`)

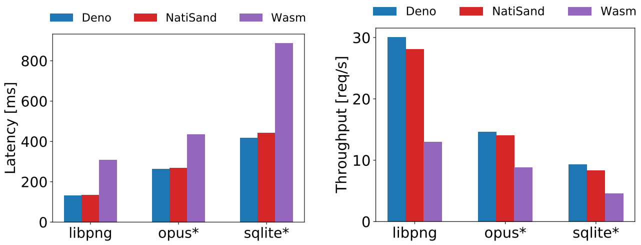

Figure 5: Average latency and throughput for microservices that execute native functions (* marks the use of `nativeCall`)

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 [24] and the WASI Sdk [84], 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.

8 RELATED WORK

Isolation of software has been widely investigated by both the academic and industrial communities [1, 9, 16, 19, 28, 29, 33, 53, 67]. *MBOX* [33] 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. [19] 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* [27], *BPFContain* [26], and *Snappy* [8] 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* [28], and *Sandbox2* [29]. 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* [53] and *Bubblewrap* [16].

Multiple research efforts have studied the use of third-party components in software products [7, 34, 52, 85]. *PKRU-Safe* [34] 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* [85] provides partial isolation of libraries by spawning a new process and configuring the necessary communication channels to support tight memory interactions with the main program. To support the change in the interactions without modifications to the library code it is necessary to write a wrapper library. *Cali* [7] 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* [52] 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. [74] 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 [14, 82, 86] tackle this problem by providing solutions to isolate the execution of third-party modules. *Wolf at the Door* [86] 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* [82] 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* [2] protects filesystem resources from subprocesses executed by JavaScript runtimes. *BinWrap* [14] separates the execution of third-party components from the rest of the application using distinct execution threads for different domains of trust. The main focus of the proposal is prohibiting arbitrary accesses to sensitive data stored in the memory of the JS runtime by leveraging Intel's MPK/PKU. NatiSand is complementary to the above solutions since our goal is to specify and enforce permissions on native code dependencies of web applications, rather than providing memory isolation for untrusted components.

Protecting JS code from being compromised is out of the scope of NatiSand, nonetheless, since proposals in this domain and ours both target the web development audience, our proposal shares some ideas with previous works in this domain [3, 59, 75, 83]. For instance, Ferreira et al. [25] 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* [83] 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* [3]. 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.

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 paper 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 https://github.com/unibg-seclab/natisand

ACKNOWLEDGMENTS

The work was supported by the European Commission within the GLACIATION project (No 101070141) and by project GRINS (PE00000018) under the MUR NRRP funded by the EU - NextGenerationEU.

REFERENCES

[1] Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In *Proceeding of the 18th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2023).*

[2] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses. In *Proceeding of the 18th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2023).*

[3] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars E. Olsson, and Andrei Sabelfeld. 2021. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *Proceeding of the USENIX Security Symposium (USENIX Security).*

[4] Alexei Starovoitov. 2020. Introduce CAP_BPF. https://lwn.net/Articles/820560/

[5] Nakryiko Andrii. 2020. BPF Portability and CO-RE. https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html

[6] Apple. 2023. JavaScriptCore. https://developer.apple.com/documentation/javascriptcore

[7] Markus Bauer and Christian Rossow. 2021. Cali: Compiler-Assisted Library Isolation. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS).*

[8] Maxime Bélair, Sylvie Laniepce, and Jean-Marc Menaud. 2021. SNAPPY: Programmable Kernel-Level Policies for Containers. In *Proceedings of the ACM Symposium on Applied Computing (SAC).*

[9] Andrew Berman, Virgil Bourassa, and Erik Selberg. 1995. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC).*

[10] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and preventing bugs in javascript bindings. In *Proceeding of the IEEE Symposium on Security and Privacy (IEEE S&P).*

[11] Thanh Bui, Siddharth Prakash Rao, Markku Antikainen, Viswanathan Manihatty Bojan, and Tuomas Aura. 2018. Man-in-the-Machine: Exploiting Ill-Secured Communication Inside the Computer. In *Proceeding of the USENIX Security Symposium (USENIX Security).*

[12] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. 2021. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *Proceeding of the USENIX Security Symposium (USENIX Security).*

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

[14] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P. Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection Against Native Node.js Add-ons. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS).*

[15] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proceedings of the USENIX Security Symposium (USENIX Security).*

[16] containers. 2023. Bubblewrap. https://github.com/containers/bubblewrap

[17] Jonathan Corbet. 2006. File-based capabilities. https://lwn.net/Articles/211883/

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

[19] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*

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

[21] Deno Land. 2023. Node compatibility mode. https://deno.land/manual/node/compatibility_mode.

[22] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[23] Jake Edge. 2020. Seccomp and deep argument inspection. https://lwn.net/Articles/822256/

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

[25] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing malicious package updates in npm with a lightweight permission system. In *Proceedings of the International Conference on Software Engineering (ICSE).*

[26] William Findlay, David Barrera, and Anil Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. *ArXiv preprint* (2021).

[27] William Findlay, Anil Somayaji, and David Barrera. 2020. bpfbox: Simple Precise Process Confinement with eBPF. In *Proceedings of the ACM Conference on Cloud Computing Security Workshop (CCSW).*

[28] Google. 2023. Minijail. https://google.github.io/minijail/

[29] Google. 2023. Sandbox2. https://developers.google.com/code-sandboxing/sandbox2/

[30] Brendan Gregg. 2021. BPF Internals. https://www.usenix.org/conference/lisa21/presentation/gregg-bpf USENIX Large Installation Systems Administration Conference (LISA).

[31] Jake Edge. 2015. A seccomp overview. https://lwn.net/Articles/656307/

[32] Michael Kehoe. 2022. eBPF: The Next Power Tool of SREs. https://www.usenix.org/conference/srecon22americas/presentation/kehoe-ebpf USENIX SREcon Americas (SRECON).

[33] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC).*

[34] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the European Conference on Computer Systems (EuroSys).*

[35] Deno Land. 2022. Deno 1.24 Release Notes – Improved FFI call performance. https://deno.com/blog/v1.24#improved-ffi-call-performance

[36] Deno Land. 2022. Deno 1.25 Release Notes – FFI API improvements. https://deno.com/blog/v1.25#ffi-api-improvements

[37] Deno Land. 2023. Deno: A modern runtime for JavaScript and TypeScript. https://deno.land/

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

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

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

[41] libbpf. 2023. libbpf. https://libbpf.readthedocs.io/en/latest/index.html

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

[43] Linux manual. 2023. bpf. https://man7.org/linux/man-pages/man2/bpf.2.html

[44] Linux manual. 2023. ldd. https://man7.org/linux/man-pages/man1/ldd.1.html

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

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

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

[48] Linux manual. 2023. strace. https://man7.org/linux/man-pages/man1/strace.1.html

[49] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter Conference (USENIX).*

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

[51] Jeffrey Mogul, Richard Rashid, and Michael Accetta. 1987. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP).*

[52] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceeding of the USENIX Security Symposium (USENIX Security).*

[53] netblue30. 2023. Firejail. https://firejail.wordpress.com/

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

[55] npm. 2023. bcrypt. https://www.npmjs.com/package/bcrypt

[56] npm. 2023. fluent-ffmpeg. https://www.npmjs.com/package/fluent-ffmpeg.

[57] npm. 2023. gm. https://www.npmjs.com/package/gm.

[58] npm. 2023. sharp. https://www.npmjs.com/package/sharp

[59] Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. 2021. Detecting Third-Party Library Problems with Combined Program Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

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

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

[62] OpenJS Foundation. 2023. Node.js. https://nodejs.org

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

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

[65] V8 project. 2023. WebAssembly compilation pipeline. https://v8.dev/docs/wasm-compilation-pipeline

[66] Kyle Quest. 2023. SlimToolkit. https://github.com/slimtoolkit/slim

[67] Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, and Stefano Paraboschi. 2021. SEApp: Bringing Mandatory Access Control to Android Apps. In *Proceeding of the USENIX Security Symposium (USENIX Security).*

[68] Ryan Dahl. 2018. 10 Things I Regret About Node.js. https://youtu.be/M3BM9TB-8yA European JavaScript Community Conference (JSConf EU).

[69] Fabian Schwarz and Christian Rossow. 2020. SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients. In *Proceeding of the USENIX Security Symposium (USENIX Security).*

[70] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z. Morley Mao. 2016. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.*

[71] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* (2001).

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

[73] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[74] Cristian-Alexandru Staicu, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. 2023. Bilingual Problems: Studying the Security Risks Incurred by Native Extensions in Scripting Languages. In *Proceeding of the USENIX Security Symposium (USENIX Security)*.

[75] Jeff Terrace, Stephen R. Beard, and Naga P. K. Katta. 2012. JavaScript in JavaScript(js.js): Sandboxing Third-Party Scripts. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*.

[76] tesseract-ocr. 2023. Tesseract. https://github.com/tesseract-ocr/tesseract

[77] The kernel development community. 2023. LSM BPF Programs. https://docs.kernel.org/bpf/prog_lsm.html

[78] The kernel development community. 2023. Seccomp BPF (SECure COMPuting with filters). https://docs.kernel.org/userspace-api/seccomp_filter.html

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

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

[81] V8 project. 2023. What is V8? https://v8.dev/

[82] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[83] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[84] WebAssembly. 2023. Wasi SDK. https://github.com/WebAssembly/wasi-sdk

[85] Yongzheng Wu, Sai Sathyanarayan, Ronald H. C. Yap, and Zhenkai Liang. 2012. Codejail: Application-transparent isolation of libraries with tight program interactions. In *European Symposium on Research in Computer Security (ESORICS)*.

[86] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. 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)*.

[87] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceeding of the USENIX Security Symposium (USENIX Security)*.

A CURL POLICY

Listing 3 reports the policy associated with the execution of the `curl https://www.example.com` command. The policy has been automatically generated using the utility described in Section 5.2.

Listing 3: Example of policy associated with curl

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