Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction

Nikos Vasilakis
CSAIL, MIT
nikos@vasilak.is

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
staicu@cispa.de

Grigoris Ntousakis
TU Crete
gntousakis@isc.tuc.gr

Konstantinos Kallas
University of Pennsylvania
kallas@seas.upenn.edu

Ben Karel
Aarno Labs
bkarel@aarno-labs.com

André DeHon
University of Pennsylvania
andre@acm.org

Michael Pradel
University of Stuttgart
michael@binaervarianz.de

ABSTRACT

Third-party libraries ease the development of large-scale software systems. However, libraries often execute with significantly more privilege than needed to complete their task. This additional privilege is sometimes exploited at runtime via inputs passed to a library, even when the library itself is not actively malicious. Mir addresses such dynamic compromise by introducing a fine-grained read-write-execute (RWX) permission model at the boundaries of libraries. Every field of every free variable name in the context of an imported library is governed by a permission set. To help specifying the permissions given to existing code, Mir's automated inference generates default permissions by analyzing how libraries are used by their clients. Applied to over 1,000 libraries, Mir shows practical security (61/63 attacks mitigated), performance (2.1s for static analysis and +1.93% for dynamic enforcement), and compatibility (99.09%) characteristics—and enables a novel quantification of privilege reduction.

1 INTRODUCTION

Modern software development relies heavily on third-party libraries. Applications use several dozens or even hundreds of libraries, created by many different authors and accessed via public repositories. The heavy use of libraries is particularly common in JavaScript applications [38, 51], and especially in those running on the Node.js platform [80], where developers have millions of libraries at their fingertips through the node package manager (npm).[1]

Reliance on libraries introduces the risk of *dynamic compromise*, *i.e.*, the runtime exploitation of a benign library via its inputs, affecting the security of the entire application and its broader operating environment. For example, consider a (de)serialization library that uses JavaScript's built-in `eval` function to parse a string into a runtime object. While the library itself is benign, accessing no other external API apart from `eval`, an attacker may pass a malicious serialized object to the deserialization function, which in turn will pass it to `eval`. As a result, the library may be subverted into malicious behavior, *e.g.*, accessing the file system or the network, that goes far beyond what a (de)serialization library is supposed to do.

[1]This paper uses the terms library, module, and package interchangeably.

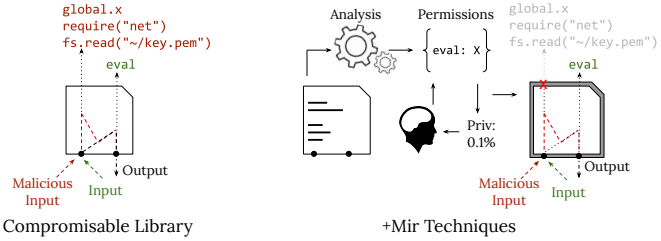


Fig. 1: Mir analyzes a library, possibly compromisable by malicious inputs, to infer a set of permissions. It then enforces this set at runtime, to lower the library's privilege over the application and its surrounding environment. It also computes a privilege reduction score for this set and, if needed, allows the set to be inspected or changed (leading to a new score).

The underlying problem is that each library running on Node.js has all privileges offered by the JavaScript language and its runtime environment. In particular, each library is allowed to access any built-in API, global variables, APIs of other imported libraries, and even import additional libraries. The left-hand side of Fig. 1 illustrates this current default situation.

This paper reduces the risk of dynamic compromise on Node.js through a system called Mir, which allows specifying, enforcing, inferring, and quantifying the privilege available to libraries. The goal of Mir is to permit a library to access only the functionality that it really needs, preventing attackers from subverting it into behavior that goes beyond its intended behavior. Our key insight to reach this goal 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.

More specifically, Mir introduces a fine-grained read-write-execute (RWX) permission model at the boundaries of libraries. To aid in specifying these permissions, Mir provides a program analysis component that infers permissions automatically. The permission model is first-order—less powerful than membranes [49], higher-order contracts [25], or information-flow monitoring [13]—and the analysis aims at a level of simplicity comparable to linting and minification,

tools commonly used by Node.js developers today. Combined, the permission model and associated analysis, aim at reducing the risk of attacks while maintaining practical performance and automation characteristics to enable adoption. By coupling default-deny semantics with explicit and statically-inferrable whitelisting, Mir minimizes the effects of dynamic compromise, as illustrated on the right-hand side of Fig. 1. Our key contributions can be summarized as follows:

Permission model & language: Our first contribution is a permission model and associated domain-specific language to express *read-write-execute permissions* (RWX). Permissions guard access paths in individual libraries. Access paths correspond to free variable names and their fields in the top-level scope of the library. They point to functionality imported from other libraries or available through built-in language features such as globals, process arguments, **require** capabilities. The aforementioned (de)serialization function would only be given an execute permission on `eval`, restricting accesses to all other APIs.

Permission inference: Our second contribution is a program analysis for automatically inferring permissions. This automation is critical for dealing with (1) continuous codebase evolution, which requires updating the specification for every code change, (2) naming issues, such as variable aliasing, and (3) library-internal code, possibly not intended for humans. The analysis identifies name usages within a library to infer the permissions that a library requires. We designed the analysis to be scalable, conservative, and make no assumptions about the existence of tests. The majority of the analysis is static, augmented with a short phase of import-time analysis addressing runtime meta-programming patterns common in npm packages.

Quantification of privilege reduction: Our third contribution is a metric for quantifying the privilege reduction achieved by Mir. This metric characterizes the remaining privilege that a library can exercise at runtime and is a direct byproduct of the permission model's design. The quantification is achieved by comparing the permissions granted by Mir to those a library would have by default—*i.e.*, statically counting all the access paths in the lexical scope of a library. The metric is designed to operate in harmony with the permission model and analysis, functioning as a proxy of the attack surface that remains available to runtime subversion after applying the chosen permissions.

Implementation & evaluation: Our last contribution is an open-source implementation and evaluation of Mir. We implement the majority of the analysis in Java, and the runtime enforcement in JavaScript. While Mir's permission model can be encoded in systems that implement more powerful protection models [5, 20, 29, 72], we chose a lightweight wrapping strategy that matches the permission model's first-order nature. We evaluate Mir on over one thousand libraries, and find that Mir (1) defends against 61/63 attacks on real-world vulnerable packages, (2) reduces a library's attack surface by an average of 143.48×, (3) avoids breaking compatibility for more than 99% of field accesses and in over 99.3% of test cases, and (4) averages 2.1*s* per library for its permission inference and 1.93% for its runtime enforcement overhead.

```
let srv = (req, res) => {
 let srl, obj;
 srl = require("serial");
 obj = srl.dec(req.body);
 route(obj, res);
}

let route = (…) => {…}
```

main

```
let lg = require("log");
lg.LVL = lg.levels.WARN;
module.export = {
 dec: (str) => {
  let obj;
  lg.info("srl:dec");
  obj = eval(str);
  return obj
 },
 enc: (obj) => {…}
}
```

serial

Fig. 2: Use of third-party modules. The `main` module (left) requires off-the-shelf serialization implemented by the `serial` *third-party* module (right), vulnerable to remote code execution (*Cf.*§2).

The paper is structured as follows. It starts with an example illustrating dynamic library compromise and how Mir's techniques address it (§2), followed by a discussion of Mir's threat model (§3). It then presents Mir's permission model and its specification language (§4), Mir's automated permission inference (§5), Mir's privilege reduction quantification (§6), and Mir's runtime enforcement component (§7). It then discusses Mir's security, compatibility, and performance evaluation (§8), and compares to prior work (§9). It finally concludes (§10) that Mir's automation and performance characteristics make it an important addition to a developer's modern toolkit—similar to a minifier or a linter—in many circumstances working in tandem with defenses that focus on other threats.

The Appendix contains additional evaluation results. The URL below contains an anonymized version of accompanying material, including the source code, benchmarks, and exploits, all of which will be made available with the camera-ready version of the paper: anonymous.4open.science/r/58f1be03-3d2d-44a3-8f20-41b13327c908

2 BACKGROUND AND OVERVIEW

This section uses a server-side JavaScript application to present dynamic application compromise due to third-party code (§2.1), and then to exemplify how Mir addresses these issues (§2.2).

2.1 Example: A De-serialization Library

Consider the Node.js scenario mentioned earlier (§1) 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. RCE problems due to serialization have been identified in widely used libraries [2, 3, 9] as well as high-impact websites, such as PayPal [69] and LinkedIn [70]. Injection and insecure de-serialization are respectively ranked number one and eight in OWASP's ten most critical web application security risks [56].

Fig. 2 zooms into the two fragments of this Node.js (de)serialization scenario. The **main** module (left) **require**s off-the-shelf serialization functionality through the **serial** module, whose **dec** method de-serializes strings using **eval**, a native language primitive. The **serial** module **require**s **log** and assigns it to the `lg` variable.[2]

[2]Naming is important in this paper: we differentiate between the module **log** and the variable `lg`. Mir tracks permissions at the level of modules, irrespective of the variables they are assigned to. To aid the reader, modules, and more broadly, contexts,

Although **serial** is not actively malicious, it is subvertible by attackers at runtime, who can use the input `str` for several attacks: (1) overwrite **info**, affecting all (already loaded) uses of **log**.**info** across the entire program; (2) inspect or rewrite top-level **Object** handlers, built-in libraries, such as **crypto**, and the **cache** of loaded modules; (3) access global or pseudo-global variables such as **process** to reach into environment variables; and (4) load other modules, such as **fs** and **net**, to exfiltrate files over the network.

2.2 Overview: Applying Mir on Serial

Our work address these security problems by first developing a permission model at the boundaries between and around libraries. The model specifies access to functionality that is defined outside a library with `RWX` permissions. A part of this functionality comes from imported libraries; for example, among other permissions, **serial** needs to be able to execute **info** from module **log**—*i.e.*, **log**.**info** is `X`. Another part of this functionality comes directly from the programming language itself; for example, **serial** clearly needs `X` permissions for **require** and **eval**. In all these three cases of `X` permissions, it is not **serial** that provides all this functionality but rather the language and its runtime environment.

These three permissions are a part of the total nine required for **serial**'s normal operation. To aid developers in identifying the remaining permissions, Mir comes with a static inference component that analyzes how libraries use the available names. The figure on the right exemplifies a small fragment of this analysis: **levels** and **WARN** are read, thus are annotated as `R`; **LVL** is written, thus `W`; and **info** is executed, thus `X`. The analysis also infers permissions for **require**, **eval**, **module**, and **exports**. Names that do not show up—even if they are built-in language objects—get no permissions.

```
let lg = require("log");
lg.LVL = lg.levels.WARN;
lg.info("srl:dec");
```
Read(R) Write(W) Exec(X)

After extracting all necessary permissions, the developer can start the program using Mir's runtime enforcement component. Mir shadows all variable names that cross a boundary with variables that point to modified values. When accessing a modified value, Mir checks the permissions before forwarding access to the original value. If a module does not have permission to access a value, Mir throws a special exception that prevents the access.

The attacks described at the end of the previous section (§2.1) are now impossible: (1) overwriting **info** from **serial** will throw a `W` violation, (2) inspecting **Object** handlers, built-in libraries, such as **crypto**, and the **cache** of loaded modules will all throw `R` violations, and (3) accessing global or pseudo-global variables, such as **process**, to reach into the environment will also throw `R` violations. Moreover, a refinement of the base `RWX` model (§4) shields against loading unexpected libraries by allowing **require** to be executed only with argument `log`.

3 THREAT MODEL

Focus: Mir focuses on the dynamic compromise of possibly buggy or vulnerable libraries, and does not target actively malicious or obfuscated libraries. Attacks are performed by passing malicious payloads to libraries through web interfaces, programmatic APIs, or shared interfaces such as **global** variables. 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 inputs (see §8).

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 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 executing—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, the module cache, and `require` ability; (3) interfaces to the standard library, such ones for the file-system or network; and (4) interfaces to other third-party libraries part of the same program. Tab. 7 in Appendix B offers many more real-world examples of such vulnerabilities Mir defends against.

Assumptions: Mir's static analysis is assumed to be performed prior to execution, otherwise a malicious library can rewrite the code of a benign library upon load. For the same reason, Mir's runtime enforcement component is assumed to be loaded prior to any other library. At the time of loading, Mir places trust in the language runtime and built-in modules, such as `fs`: a minimum of trusted functionality is needed from the module system to locate and load permissions.

Non-threats: Mir 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 for two reasons: (i) they cannot be analyzed by Mir's static analysis, which operates on source code, and (ii) they can bypass Mir's runtime protection, which depends on memory safety. In the context of JavaScript, these can be addressed by complementary techniques [33, 75] (see also §9).

Mir blocks runtime access 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. If a library already uses that name, however, 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 Mir's threat model and are handled by complementary techniques [14, 67]. Notably, if a library does invoke 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.

4 PERMISSION MODEL AND LANGUAGE

Mir's goal is to reduce the privilege that libraries possess. At the core of our approach is a model that can express the set of first-order permissions that should be granted to each library. The model is instantiated per-library using a domain-specific language (DSL, Fig. 3) that focuses on read (`R`), write (`W`), execute (`X`), and import (`I`) permissions.

Core Permission Model: The core of Mir's permission model and associated DSL is a per-library permission set: ModPermSet maps

are typeset in **purple sans serif**, fields in **olive teletype**, and plain variables in uncolored teletype fonts.

$$
\begin{array}{rll}
s, m & \in \mathsf{String} & \\
r := & \mathtt{R} \mid \epsilon & \mathsf{ReadPerm} \\
w := & \mathtt{W} \mid \epsilon & \mathsf{WritePerm} \\
x := & \mathtt{X} \mid \epsilon & \mathsf{ExecPerm} \\
i := & \mathtt{I} \mid \epsilon & \mathsf{ImpoPerm} \\
\mu := & [r\ w\ x\ i] & \mathsf{Mode} \\
f := & f.s \mid *.f \mid f.* \mid s & \mathsf{AccessPath} \\
p := & f\colon \mu \mid f\colon \mu, p & \mathsf{ModPermSet} \\
\omega := & m : \{p\} \mid m : \{p\}, \omega & \mathsf{FullPermSet}
\end{array}
$$

Fig. 3: Mir's permission language. The DSL captures the full set of specifications for modeling permissions across libraries (*Cf.*§4).

names accessible within the library context to a Mode, *i.e.*, a set of access rights encoded as RWX permissions. Names represent access paths within the object graph reachable from within the scope of the library—*e.g.*, **String**.**toUpperCase**. Access paths start from a few different points that can be grouped into two broad classes. The first class contains a closed set of known root points that are provided by the language, summarized in the first four rows of Tab. 1. These names are available by default through (and shared with) the library's outer context, *i.e.*, resolving to a scope outside that of a library and pervasively accessible from any point in the code. Examples include top-level objects and functions, such as **process**.**args** and **eval**, functions to perform I/O, such as **console**.**log**, ability to **require** other libraries.

The second class contains access paths that start from explicitly importing a new library into the current scope. Such an import results in multiple names available through the imported library's (equivalent of) **export** statement. Examples of such paths from Fig. 2 include **log**.**info** and **srl**.**dec** (§2.1).

Mir's model can thus be thought as an access-path protection service: access rights are expressed as permissions associated with paths starting from a set of variable names that are free at the top-level scope of the library. Names in this set are bound to values outside the scope of the library, pointing to functionality that is not implemented by the library. These values often contain fields, defining recursive maps from names to values. Names or values created within the scope of a library are *not* part of this model: Mir does not allow specifying or enforcing access restrictions on, say, arbitrary objects or function return values.

Semantics: The semantics behind the core set of permissions can be summarized as follows:

- A read permission (R) grants clients the ability to read a value, including assigning it to variables and passing it around to other modules.
- A write permission (W) grants clients the ability to modify a value, and includes the ability to delete it. The modification will be visible by all modules that have read permissions over the original value.
- An execute permission (X) grants clients the ability to execute a value, provided that it points to an executable language construct, such as a function or a method. It includes the ability to invoke the value as a constructor (typically prefixed by new).

Tab. 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),

RWX permissions are loosely based on the Unix permission model, with a few key differences. Reading a field of a composite value `x.f` requires R permissions on the value `x` *and* the field `f`—that is, an R permission allows only a single de-reference. Reading or copying a function only requires an R-permission, but performing introspection requires X permissions over its subfields due to introspection facilities being provided by auxiliary methods (*e.g.*, `toString` method). A W permission on the base pointer allows discarding the entire object. While a base write may look like it bypasses all permissions, modules holding pointers to fields of the original value will not see any changes.

Example: To illustrate the base permission model on the deserialization example (§2), consider `main`'s permissions:

```
main:
  require: RX
  require("serial"): I
  require("serial").dec: RX
```

The set of permissions for `serial` is:

```
serial:
  eval: RX
  require: RX
  module: R
  module.exports: W
  require("log"): I
  require("log").levels: R
  require("log").levels.WARN: R
  require("log").info: RX
  require("log").LVL: W
```

Importing: A simple X permission to the built-in `require` function gives libraries too much power. Thus, Mir needs to allow specifying which imports are permitted from a library.

This is achieved through an additional I permission. This permission is provided to an AccessPath that explicitly specifies the absolute file-system path of a library.[3] Using the absolute file-system path is a conscious decision: the same library name imported from different locations of a program may resolve to different libraries residing in different parts of the file system and possibly corresponding to different versions.

Using a separate permission I provides additional flexibility by distinguishing from R. Libraries are often imported by a program

[3]For portability, Mir prefixes records with a `__PWD_PREFIX__` variable that can be instantiated to different values across environments.

Tab. 2: Mir's analysis updates. Updates performed by the static analysis when visiting specific kinds of statements.

Kind of statement	Updates	Example
Assignment $lhs = rhs$ at location l:		
For each $a \in getAPIs(lhs)$	Add $(a, \texttt{W})$ to permission set C	`someModule.foo = 5`
For each $a \in getAPIs(rhs)$	Add $(a, \texttt{R})$ to permission set C	`x = require("someModule")`
	Add $lhs \; at \; l \mapsto a$ to $DefToAPI$	
Call of function f:		
For each $a \in getAPIs(f)$	Add $(a, \texttt{X})$ to permission set C	`someModule.foo()`
Any other statement that contains a reference r:		
For each $a \in getAPIs(r)$	Add $(a, \texttt{R})$ to permission set C	`foo(someModule.bar)`

only for their side-effects (*i.e.*, not for their interface). In these cases, their fields should not necessarily be accessible by client code. Typical examples include singleton configuration objects and stateful servers.

Wildcards: Mir's model offers *wildcards* to allow for *all* possible matches of a segment within a path to have a single permission mode. Wildcards match any string and are semantically similar to shell expansion (*i.e.*, the `*` in `cat *.txt`). The form *****.**f** assigns a mode to all fields named **f** reachable from any object, and **o**.***** assigns a mode to all fields of an object (or path) **o**. These forms may also be combined, as in **o**.*****.**f**.

Wildcards have many practical uses. The primary use case is when fields or objects are altered through runtime meta-programming. In such cases, the fields are not necessarily accessible from a single static name and might depend on dynamic information. Often, these fields (not just the paths) are constructed at runtime, means they are not available for introspection by Mir at library-load time.

Transitive & higher-order permissions: The default-deny semantics of Mir's permission model involves some nuanced characteristics. First, the absence of any permission to a function translates to absence of all permissions to all the functions (or, more generally access paths) that the first function points to. Conversely, holding a permission to a function or a library does not translate to holding permissions to all of its access paths—but only the paths to which that function or library has access to. By providing the ability to specify multiple layers of permission sets, Mir minimizes transitive permission leakage across multiple library levels.

The `RWX` model is first-order, in that it applies directly to access paths like **fs**.**read** and **lg**.**LVL**. It does not apply to arguments of functions, such as the names `req` and `res` found in function `srv`, nor to module-internal functions, such as `route`.

However, if a value provided to a function is itself an access path then its permissions are governed by the caller context. For example, if the path **fs**.**read** was provided as an argument to **lg**.**info**, then **read**'s permissions would be governed by the context providing it. More generally, a higher-order value passed to a module, such as a closure or an object, is governed by the permissions in the lexical scope of the module that created that value: if a module creating a closure f has no permission to access **fs**, then f will not be able to invoke **fs** in any context.

These transitivity and first-order properties are also related to permissions over an object's prototype chain. Mir's model—and associated analysis and enforcement components—is oblivious to where a property resolves in the prototype chain. For example, the invocation of a **toString** method on the return value of a different library is governed by the permissions of that library (or the library that created that value) to **toString** name.

5 PERMISSION INFERENCE

To aid users in expressing permissions, Mir automatically infers permissions that describe how a library uses its dependencies and built-in APIs. These permissions are inferred by an analysis that identifies and resolves accesses of functions and properties provided by third-party libraries. To be practical and to effectively reduce the risk of dynamic compromise, the analysis must fulfill three requirements. First, the analysis should work for arbitrary libraries, without assuming anything apart from access to the library's source code. In particular, the analysis should not rely on the existence of test suites or client code that uses the library. Second, the inferred permissions should be conservative, in the sense that the analysis should infer a permission only if there is evidence that the library indeed needs that permission. Third, the analysis should be efficient and scale well to complex libraries, as we want to apply it to real-world libraries. We are not aware of such an analysis in the literature; hence this section describes a permission analysis designed to fit these requirements.

With the first requirement in mind, the core of our permission inference is a static analysis of the library code (§5.1), augmented with a lightweight dynamic analysis that loads the library but does not rely on any client code (§5.2). Given the difficulties of statically analyzing JavaScript [8, 24], our static analysis aims neither at soundness nor completeness. Instead, it takes a pragmatic approach designed to work well for programming patterns common in real-world libraries, but not every conceivable corner-case. With the second requirement in mind, the static analysis grants a permission only if the analysis sees a possibly feasible path that uses the permission. Finally, with the third requirement in mind, the core of the static analysis is intra-procedural, i.e., it reasons about a function without analyzing all other functions called by it. While in principle, these decisions could lead to missing permissions, the evaluation shows this rarely to happen for real-world libraries. Moreover, if indeed a permission is missing, Mir will produce a runtime error that a user can address by refining the permissions.

5.1 Static Analysis of Required Permissions

The core of the analysis is an intra-procedural, flow-sensitive, forward data flow analysis. The analysis visits each statement of a module by traversing a control flow graph of each function. During

these visits, it updates two data structures. First, it updates the set C of (API, permission) pairs that eventually will be reported as the inferred permission set. The set C grows monotonically during the entire analysis, and the analysis adds permissions until reaching a fixed point. Second, the analysis updates a map $DefToAPI$, which maps definitions of variables and properties to the fully qualified API that the variable or property points to after the definition. For example, when visiting a definition `x = require("foo").bar`, the analysis updates $DefToAPI$ by mapping the definition of `x` to "foo.bar". The $DefToAPI$ map is a helper data structure discarded when the analysis completes analyzing a function.

Transfer Functions: Table 2 summarizes how the transfer functions of the analysis update C and $DefToAPI$ when visiting specific kinds of statements. The updates to C reflect the way that the analyzed module uses library-external names. Specifically, whenever a module reads, writes, or executes an API a, then the analysis adds to C a permission $(a, \texttt{R})$, $(a, \texttt{W})$, or $(a, \texttt{X})$, respectively. The updates to $DefToAPI$ propagate the information about which APIs a variable or property points to. For example, suppose that the analysis knows that variable `a` points to a module "foo" just before a statement `b = a.bar`; then it will update $DefToAPI$ with the fact that the definition of `b` now points to "foo.bar".

While traversing the control flow graph, the analysis performs the updates in Table 2 for every statement. On control flow branches, it propagates the current state along both branches. When the control flow merges again, then the analysis computes the union of the C sets and the union of the $DefToAPI$ maps of both incoming branches. Mir handles loops by unrolling each loop once, which is sufficient in practice for analyzing uses of third-party code, because loops typically do not re-assign references to third-party APIs.

Resolving Accesses to APIs: The transfer functions in Table 2 rely on a helper function $getAPIs$. Given a reference, *e.g.*, a variable or property access, this function returns the set of fully qualified APIs that the reference may point to. For example, after the statement `obj.x = require("foo").bar`, $getAPIs(obj.x)$ will return the set $\{\text{“}foo.bar''\}$. When queried with a variable that does not point to any API, $getAPIs$ simply returns the empty set. Algorithm 1 presents the $getAPIs$ function in more detail. We distinguish four cases, based on the kind of reference given to the function. Given a direct import of a module, $getAPIs$ simply returns the name of the module. Given a variable, the function queries pre-computed reaching-definitions information (see below) to obtain possible definitions of the variable, and then looks up the APIs these variables point to in $DefToAPI$. Given a property access, *e.g.*, `x.y`, the function recursively calls itself with the reference to the base object, *e.g.*, `x`, and then concatenates the returned APIs with the property name, *e.g.*, "y". Finally, for any other kind of reference, $getAPIs$ returns an empty set. The latter includes cases that we cannot handle with an intra-procedural analysis, *e.g.*, return values of function calls. In practice, these cases are negligible, because real-world code rarely passes around references to third-party APIs via function calls. We therefore have chosen an intra-procedural analysis, which ensures that the static permission set inference scales well to large code-bases.

To find the APIs a variable may point to, Algorithm 1 gets the reaching definitions of the variable. This part of the analysis builds upon a standard intra-procedural may-reach definitions analysis, which Mir pre-computes for all functions in the module. To handle nested scopes, *e.g.*, due to nested function definitions, Mir builds a stack of definition-use maps, where each scope has an associated set of definition-use pairs. To find the reaching definitions of a variable, the analysis first queries the inner-most scope, and then queries the surrounding scopes until the reaching definitions are found. To handle built-in APIs of JavaScript, *e.g.*, `console.log`, Mir creates an artificial outer-most scope that contains the built-in APIs available in the global scope.

Data: Reference r
Result: Set of APIs that r may point to
if r is an import of module "m" **then**
  return { "m" }
end
if r is a variable **then**
  $A \leftarrow \emptyset$
  $defs \leftarrow$ get reaching definitions of r
  for ***each*** d ***in*** $defs$ **do**
    $A \leftarrow A \cup DefToAPI(d)$
  end
  return A
end
if r is a property access $base.prop$ **then**
  $A_{base} \leftarrow getAPIs(base)$
  return $\{a + \text{“.”} + prop \mid a \in A_{base}\}$
end
return $\emptyset$

Algorithm 1: Helper function *getAPIs*.

Returning to the running example in Figure 2. For **main**, the static analysis results in the following permission set:

$$\{(\text{“}serial'', \texttt{R}), (\text{“}serial.dec'', \texttt{R}), (\text{“}serial.dec'', \texttt{X})\}$$

As illustrated by the example, the inferred permission set allows the intended behavior of the module, but prevents any other, unintended uses of third-party APIs. Our evaluation shows that the static analysis is effective also for larger, real-world modules (§8.4).

Limitations: In line with Mir's design goal of being conservative in granting permissions, the analysis infers a permission only if there exists a path that uses the permission. In contrast, the analysis may miss permissions that a module requires. For example, missed permissions may results from code that passed a reference to a module across functions:

```
x = require("foo");
bar(x);
```

In this example, the analysis misses any permissions on "foo" that `bar` relies on. Tracking object references across function boundaries would require an inter-procedural analysis, which is difficult to scale to a module and its potentially large number of transitive dependencies [80]. Another example of potentially missed permissions is code that dynamically computes property names:

```
x = require("foo");
p = "ba" + "r";
x[p] = ...;
```

In this example, the analysis misses the W permission for "foo.bar". Tracking such dynamically computed property names is known to be a hard problem in static analysis of JavaScript [66].

5.2 Dynamic, Import-time Permission Analysis

To augment the number of permissions inferred by static analysis, Mir adds a short phase of dynamic import-time analysis. This dynamic analysis is performed by simply importing the analyzed library, *i.e.*, 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 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 the ones inferred by the static analysis. The following code snippet demonstrates a simple but common pattern:

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

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. We evaluate the improvement of import-time analysis on permission inference (§8.3). 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.

6 QUANTIFYING PRIVILEGE REDUCTION

Any policy—whether created automatically or manually—on existing programs aims at striking a balance between compatibility and security: an ideal policy would allow only the necessary accesses but no more. Unfortunately, statically inferring such an ideal policy in the context of any language is known to be undecidable. However, some analyses are better than others, *i.e.*, they infer policies with fewer accesses, even if they do not infer the ideal policy. To be able to quantitatively evaluate the security benefits offered by such analyses, we propose a novel privilege reduction metric.

Privilege Reduction: Informally, the single-library privilege reduction is calculated as the ratio of disallowed permissions over the full set of permissions available by default within the lexical scope of the library. The default permission set is calculated by statically expanding all names available in scope; the disallowed set of permissions is calculated by subtracting the allowed permissions from the default permission set. Single-library privilege reductions across the full dependency tree are then combined into a single reduction metric for a program and its dependencies. The following paragraphs explain the details.

Informal Development: Before formalizing privilege reduction, we use the de-serialization example (Fig. 2) to build an intuition. We first need to identify two sets of objects: (i) the *subject modules* M_s, whose privilege we are interested in quantifying; and (ii) the *target critical resources* M_t that can be potentially accessed by the subject modules. Let's assume that from the two modules presented in Fig. 2, we are only interested in quantifying **main**'s privilege; thus, M_s = {**main**}. As implied earlier (§4), the set of critical resources contains many paths available to **main**. For simplicity, we now assume it only contains **globals**, **fs**, and **require**; thus, M_t = {**globals**, **fs**, **require**}. Module **main** needs an X permission on **require** to be able to load **serial**, and an X permission on **serial.dec** to be able to call the dec function. With this simple configuration, Mir disallows all accesses except for $P(M_s, M_t) = \{\langle$**require**$, X\rangle, \langle$**serial.dec**$, X\rangle\}$.

Mir's goal is to quantify this privilege with respect to the permissions available to a library by default. If **main** was executed without additional protection, its privilege would be $P_{base}(M_s, M_t) = \{\langle$**globals.***$, RWX\rangle, \langle$**fs.read**$, RWX\rangle, \ldots\}$.

Formal Development: More formally, by default at runtime any module has complete privilege on all exports of any other module. Thus, for any modules m_1, m_2 the baseline privilege that m_1 has on m_2 is:

$$P_{base}(m_1, m_2) = \{\langle a, \mu\rangle | a \in API_{m_2}, \mu \in \texttt{Mode}\}$$

where $\mu \in$ Mode is a set of orthogonal permissions on a resource, which for Mir is $\mathcal{P} = \{R, W, X\}$. Name a can be any field that lies arbitrarily deeply within the values exported by another module.

Mir reduces privilege by disallowing all but the white-listed permissions at module boundaries:

$$P_S(m_1, m_2) = \{\langle a, \mu\rangle | a \in API_{m_2}, S \text{ gives } m_1\ \mu \text{ on } a\}$$

To calculate the privilege reduction across a program that contains several different modules, we lift the privilege definition to a set of subject and target modules:

$$P(M_s, M_t) = \bigcup_{\substack{m_1 \in M_s \\ m_2 \in M_t}} P(m_1, m_2)$$

Based on this, we can define privilege reduction, a metric of the permissions restricted by a privilege-reducing system S, such as Mir:

$$PR(M_s, M_t) = \frac{|P_{base}(M_s, M_t)|}{|P_S(M_s, M_t)|}$$

A higher reduction factor implies a smaller attack surface since the subjects are given privilege to a smaller portion of the available resources. P_{base} is an under-approximation of base privileges, as a source module can, in principle, import and use any other module that is installed in the execution environment. Consequently, the measured privilege reduction is actually a lower bound of the privilege reduction that Mir achieves in practice.

Transitive Permissions: Fig. 2's **main** is not allowed to directly call **eval**; however, it can call **eval** indirectly by executing **serial.dec**. Accurately quantifying such transitive privilege requires tracking transitive calls across such boundaries, which requires heavyweight information flow analysis. Mir's privilege reduction quantification does not attempt such an analysis to keep runtime overheads low. As a result, Mir's estimate is necessarily conservative—*i.e.*, Mir reports a lower number than the one achieved in practice.

7 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

```
1let old_srl = srl;
2srl = {}
3srl.dec = (...args) => {
4  if (σ(srl.dec, perms.X)) {
5    return old_srl.dec(...args);
6  }
7}
```

(a) Object-wrapping fragment

```
1var CONTEXT = {
2  eval: mir.wrap(eval, {X}),
3  import: mir.wrap(require, ["log"]),
4  Number: mir.wrap(Number),   //denied
5  Array: mir.BT(Array),       //denied
6  toString: mir.BT(toString),//denied
7  // [another 150 entries denied]
8}
```

(b) Custom context creation

```
1function (cxt) {
2  var eval = cxt.eval;
3  var require = ctx.require;
4  var Number = ctx.Number;
5  var Array = ctx.Array;
6  var toString = ctx.toString;
7let lg = require("log");
8lg.LVL = lg.levels.WARN;
9exports = {
10  dec: (str) => {
11    log.info("[start]");
12    let obj = eval(str);
13    return obj
14  },
15  enc: (obj) => {...}
16}
17}
```

(c) Context rebinding

Fig. 4: Mir's runtime enforcement transformations. Mir's basic `wrapping` traverses objects and wraps fields with inline monitors (a, line 4). A new modified context is created by `wrapping` all values available in a module's original context (b). The modified context is bound to the module by enclosing the module source (half-visible code fragment) in a function that redefines all non-local variable names as function-local ones, pointing to values from the modified context (*Cf.*§7).

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 [5, 20, 29, 72, 75], employing program transformations to traverse and wrap selected values with interposition proxies. However, it differs in a few key points because of two characteristics related to the goals of Mir. The first characteristic is the first-order nature of Mir's permission model: Mir checks an `X` permission for every **srl**.**dec** in Fig. 2.1's `main`, but does not enforce permissions over its arguments—offering a potential for runtime performance gains. 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: **main** may need an `X` permission for **srl**.**dec**, but a different module might need an `R` for **srl**.**enc**. 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 yield into Mir rather than the built-in locate-and-load mechanism. For each module, the second phase creates a fresh copy of the runtime 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 re-defines 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, fields—it is allowed to access.

Base Transform: These transformations have a common structure that traverses objects recursively—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'. Every field f of O is wrapped with a method f' defined to enclose the permissions for f. Effectively, f' implements a security monitor—a level of indirection that oversees accesses to the field and ensures that they conform to the permissions corresponding to that field. At runtime, f' checks f's permission for the current access type: if the access is allowed, it forwards the call to f; otherwise, it throws a special exception, `AccessControlException`, that contains contextual information for diagnosing 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. 4a. The wrapper function uses a Mir-built-in function σ that checks f's `X` permission (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 will throw an exception and stop execution.

Context Creation: To prepare a new context to be bound to a library being loaded, Mir first creates an auxiliary hash table (Fig. 4*b*), mapping names to newly transformed values: names correspond to implicit modules—globals, language built-ins, module-locals, *etc.* (Tab. 1); transformed values are created by traversing individual values in the context using the `wrap` method to insert permission checks.

User-defined global variables 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 resolving well-known names hard-coded in a list. Using this list, Mir creates a list of pointers to unmodified values upon startup.

Care must be taken with module-local names—*e.g.*, the module's absolute filename, its `exported` 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 deferring these transformations for the context-binding phase (discussed next).

Fig. 4b shows the creation of `serial`'s modified context.

Context Binding: To bind the code whose context is being transformed with the freshly 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, pointing 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. Module-local variables (a challenge outlined earlier) are assigned the return value of a call to `wrap`, which will be applied only when the module is evaluated and the module-local value

becomes available. Mir evaluates the resulting closure, invokes it with the custom context as an argument, and applies further `wrap` transformations to its return value.

The result of such a source-to-source linking of `serial`'s context is shown in Fig. 4c.

8 EVALUATION

To evaluate Mir, we apply it to hundreds of real-world npm packages, investigating the following questions:

- **Q1** How effective is Mir at defending against attacks that exploit real-world vulnerabilities? (§8.1)
- **Q2** How much does Mir reduce the attack surface in terms of privilege reduction? (§8.2)
- **Q3** How compatible is Mir with existing code—*i.e.*, what is the danger of breaking legacy programs? (§8.3)
- **Q4** How efficient and scalable are Mir's inference and enforcement components? (§8.4)
- **Q5** How does Mir's techniques compare to other techniques, such as library debloating? (§8.5)

Implementation: Our implementation targets JavaScript packages in the Node.js ecosystem and is available via `npm i -g @blindorg/mir`. The static analysis component is implemented as a compiler pass in the Google Closure Compiler [26], amounting to about 2.1 KLoC. The runtime enforcement component is implemented in about 2.8 KLoC of JavaScript. The permissions inferred by the static analysis are provided to the runtime enforcement as human-readable JSON files.

Libraries and Workloads: We apply Mir to hundreds of real-world npm packages, using different sets of packages for different research questions, depending on what is required for a specific question. To address Q1, we apply Mir to real-world vulnerabilities obtained by systematically going through all publicly known vulnerabilities in npm packages [65].

For Q2 and Q3, we need extensive tests to be able to test compatibility. As not all of the Q1 libraries contain tests, and because some contain tests that require an elaborate setup, we augment the Q1 set with 50 popular packages [55] that provide comprehensive test suites executable via `npm test`. The additional libraries also answer Q2 and Q3 for modules that do not necessarily make use of security-critical APIs. The 50 libraries range between 1–3.2K lines of JavaScript code with extensive tests. For Q4, we apply the static analysis to an additional 986 npm packages gathered from the most depended-upon packages, which in total comprise 5,826,357 LoC.

Setup: Experiments were conducted 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. The JavaScript setup uses Node.js v12.19, bundled with V8 v7.8.279.23, LibUV v1.39.0, and `npm` version v6.14.8. Mir's static inference runs on Java SE 1.8.0_251, linked with Google Closure v20200927.

8.1 Real-world Vulnerability Defense (Q1)

This section evaluates Mir effectiveness at its ultimate goal—preventing attackers from exploiting vulnerabilities. To obtain a large, unbiased, and realistic set of vulnerabilities, we systematically go through *all* publicly known vulnerabilities in npm packages [65] and select those covered by our threat model (§3). Fig. 5 summarizes the results of our analysis, Tab. 3 highlights a few cases outlined later, and Tab. 7–8 in Appendix B show the full results.

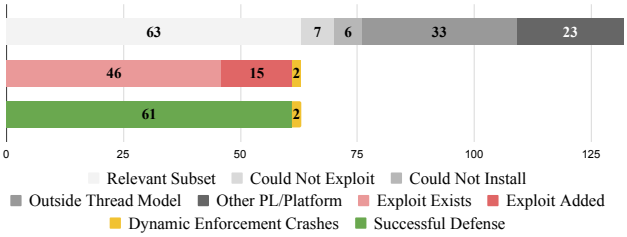


Fig. 5: Overview of real-world vulnerabilities used to evaluate Mir's effectiveness in preventing vulnerabilities (*Cf.*§8.1).

Relevant Subset: Starting from all Snyk [65] vulnerabilities, we first keep only vulnerabilities labeled as "arbitrary code execution", "remote code execution", and "sandbox escape" (Fig. 5, top bar: 132). We ignore other categories such as "malicious package", "denial of service", and "use after free", because they fall outside Mir's threat model. We then filter out 23 vulnerabilities for platforms other than Node.js (*e.g.*, Android, browser) or languages other than JavaScript (*e.g.*, PHP, Python), 33 misclassified vulnerabilities (*e.g.*, XSS, path traversal, sanitization), 6 modules we were not able to set up, and 7 modules we were not able to exploit. We note that we spent significant time on the last two classes, up to five hours per vulnerable module to (i) set up the module, and (ii) create or reproduce an exploit. Overall, we end up with 63 reproduced vulnerabilities: 46 with exploits provided in the vulnerability report and 15 with exploits that we newly created for vulnerabilities that do not come with an exploit.

We apply Mir to the 63 vulnerable modules and check whether the approach defends against the exploit. Mir's static analysis works on all of the libraries, but Mir's dynamic enforcement crashes on two libraries (even without applying the exploit): the *vm2* package applies itself complex runtime wrapping, not handled correctly by Mir's runtime wrapping, and the *typed-function* package affects the `Function` prototype chain in a way that is currently not supported by Mir. Mir successfully protects against all other 61 attacks to vulnerable modules.

Examples: We show a few of these successfully defended attacks on popular modules in Tab. 3, and proceed to highlight a couple of noteworthy examples. 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 prior to calling **eval**. Their PoCs either execute the `id` command to return the user identity or print `process.env`, both of which Mir defends. The case of *static-eval* is interesting because it accepts abstract syntax trees (ASTs) rather than strings; the PoC cleverly passes **process.env** crafted as AST through the Esprima parser—which Mir solves by disallowing 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**.

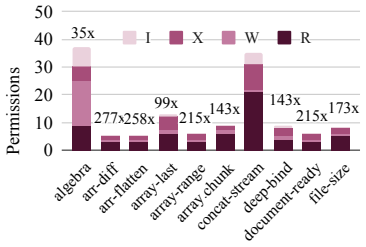
Tab. 3: Examples of vulnerabilities used in Q1, along with the permissions Mir infers for them, and whether Mir defends against the proof-of-concept (PoC) exploit (*Cf.*§8.1). *See Tab. 7–8 in Appendix B for the full list of vulnerabilities and their breakdown.*

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

Take away: Mir defends against 61/63 real-world attacks on vulnerable Node.js packages.

8.2 Privilege Reduction (Q2)

To quantify the extent to what Mir reduces the exploitable attack surface, we measure the privilege reduction (§6) achieved by the statically inferred permissions. We use all 31 libraries from the 63 Q1 attacks that we were able to set up for this experiment, excluding ten libraries that do not have tests; seven duplicate libraries, *i.e.*, cases where a second attack targets the same library; seven libraries for which we were not able to setup or run the test suite; two libraries for which Mir crashes (same as Q1); and six libraries for which Mir crashes on test cases (details in Tab. 7, Appendix B). We augment this set with another 50 libraries that have extensive test suites, to understand privilege reduction in modules that do not necessarily use security-critical APIs. The total is 81 libraries.



The total number of permissions per library ranges between 2–658 (avg: 42.2), spread unevenly between 1–341 `R` (avg: 22.1), 0–29 `W` (avg: 3.3), 0–209 `X` (avg: 12.5), and 0–87 `I` (avg: 4.1). Privilege is reduced by up to three orders of magnitude, ranging between 3.5×–706× (avg: 143.48×). The privilege reduction is high when developers use only a small fraction of the available APIs, and thus it is inversely correlated to the library size: smaller libraries see larger reduction in privilege. The figure on the right shows results for the first ten libraries (alphabetically), and Appendix C shows the full results for all libraries.

We manually inspect the inferred permissions to get a sense of their security criticality. We define as security-critical system-wide permissions the union of `X` permissions to `child_process.*`, `X` permissions `fs.{read,write}file[Sync]`, and `R` permissions to a subset of `process.env`. We also define security-concerning library-specific permissions the set of `RWX` permissions to unusual field names.[4] Our inspection indicates that after applying Mir, only one library maintains permissions to security-critical system-wide interfaces. Additionally, only five libraries maintain permissions to library-specific security-concerning interfaces.

[4]In one library, `react-dom`, Mir disallows access to fields of an object called `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED`.

Tab. 4: Compatibility across 81 libraries (*Cf.*§8.3).

	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%)

Take away: Mir reduces the attack surface by an average of 143.48×, usually blocking access to security-critical interfaces.

8.3 Compatibility Analysis (Q3)

To a large extent, backward-compatibility drives practical adoption of tools such as Mir. If a tool requires significant effort to address compatibility, chances are developers will avoid it despite any security benefits it provides. To investigate compatibility, we use the same 81 npm libraries from Q2 and their test suites. Tab. 4 summarizes the compatibility characteristics of testing these 81 libraries using two Mir configurations, and Tab. 5 and 6 in Appendix A break down compatibility results for the 31 Q1 libraries and the additional 50 Q2 libraries.

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. 4). The reason 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 permitted by Mir.

Influence of Import-Time Analysis: An important element of Mir's inference is its assistance of static analysis with a dynamic import-time analysis (§5.2). 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. 4, 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 wrappers for all 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`. Mir's 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's `package.json`. This loading is not inferable by Mir's combined static and import-time analyses, 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.

Take away: Mir correctly infers 99% of all accesses on a set of widely used packages with extensive test suites, indicating a small risk of breakage by applying Mir.

8.4 Efficiency and Scalability (Q4)

Static Analysis: To evaluate the efficiency and scalability of the static analysis, we run it on 1,036 npm packages that comprise 5,826,357 LoC. Mir's static analysis operates successfully on all packages without errors, except for malformed files on which the base Google Closure Compiler (*i.e.*, without our analysis) also fails—*e.g.*, files containing syntax errors.

	Min	1/4	2/4	3/4	Max
R	0	13	39	156	27,736
W	0	1	4	19	9,018
X	0	6	20	78	16,638
I	0	2	6	29	16,639
Σ:	0	25	73	295	66,559

Fig. 6: Number of inferred `RWXI` permissions: 1/4, 2/4 **and** 3/4 **represent the first quartile, the median, and the third quartile.**

The analysis takes 42 minutes in total, *i.e.*, an average of 2.5 seconds per npm package. To put these results into context, we also measure the performance of a popular linter—a lightweight static analysis pass flagging common human errors. We use `eslint` (v5.0.1) [79] and each library's default linting configuration, falling back to Google's style rules when no such configuration exists. `eslint` ranges between 0.94 and 6.017 seconds per package, with an average of 1.34 seconds. These results show that the static inference scales well to real-world packages and that its efficiency is comparable to other tools that developers use regularly. The resulting permissions of this large-scale analysis, averaging one permission per 10 LoC, are shown in Fig 6. Most permissions are `R` and `X` permissions (50.33% and 22.97%, respectively), showing that client packages rarely write to references outside of their boundaries, *e.g.*, to global variables or the API of the packages they use.

Take away: The static analysis scales to millions of lines of code, analyzing 1,036 of the most depended upon packages in 2.5 seconds, on average.

Dynamic Enforcement: We compare the performance of running the tests suite of the 81 libraries from Q2 and Q3 with Mir enforcement against that of the unmodified libraries. Mir's adds between 0.13–4.14*ms* of slowdown to executions that range between 324*ms* and 2.77*s*. Slowdowns average about 3.3*ms* per library, increasing the execution time by 1.93% on average. Based on these results, we do not anticipate a need for users to trade in runtime security to gain performance. The figure on the right shows results for the first 10 libraries (alphabetically, same sample as Q2 plot).

Mir applies an average of 346 wrappers per library, applying a total of 25,609 wrappers. The distribution of accesses at runtime 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.

Take away: The runtime enforcement imposes a runtime performance overhead of 3.3*ms* (1.93%), on average.

8.5 Comparison with Debloating (Q5)

We now compare against a possible alternative to Mir targeting a similar domain: Mininode [34] is a state-of-the-art static-analysis debloating tool for Node.js. We apply the latest Mininode (v.`f604d9e`) in its `--soft` (coarse-grained) and `--hard` (fine-grained) mode on a total of 88 libraries: 81 libraries from Q2 and Q3, and the seven from Q1 for which Mir crashes on tests (§8.2).

Mininode runs to completion for 81/88 libraries, which we check for compatibility using tests, as in Q2. Both `soft` and `hard` fail tests on three libraries: `soft` fails due to changes in whitespace (in `safe-eval`), `hard` eliminates a function (`hypenToCamel` in `ejs`), and both fail due to white-space differences (in `js-yaml`) and the removal of a critical file (`compile-dots.js` in `mol-proto`). Whitespace incompatibilities are due to Mininode's back-end, which uses `escodegen` to generate JavaScript from the debloated AST, affecting the resulting white-spaces and indentation. Mininode crashes on 7/88 libraries, failing to find the entry point (5 libs.), exiting due to dynamic imports (1 lib.), and running out of memory (1 lib.). Mininode's analysis takes 0.82–4.013 seconds, comparable to Mir, and incurs *no* overhead during library execution. All 63/63 PoC attacks succeed because (i) the debloated libraries still have access to functionality built into Node.js (*e.g.*, `Object`, `Array.of`, `Math`, `String`), and (ii) JavaScript's dynamic behavior, such runtime code evaluation, falls outside Mininode's focus.

Take away: Mininode's debloating achieves better compatibility than Mir and adds no runtime overhead, but does not protect against the specific dynamic threats that are the focus of Mir.

9 RELATED WORK

Mir's techniques touch upon a great deal of previous work in several distinct domains.

Privilege Reduction: A number of works have addressed privilege reduction [4, 10, 15, 16, 20, 27, 32, 42, 57, 59, 60, 78], often offering significant automation. This automation often comes at the cost of *lightweight annotations* on program objects—*e.g.*, configurations in Privman [32], `priv` directives in Privtrans [16], tags in Wedge [15], and compartmentalization hypotheses in SOAAP [27]. TRON [12] 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 altering programs manually to use its API, and SOAAP mostly checks rather than suggests policies. In comparison to these works, Mir (1) leverages existing boundaries, and (2) offers significantly more automation.

To ameliorate manual annotations on individual objects, more recent library-level compartmentalization [37, 43, 75] leverages 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 [41] 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.

Program Analysis: The static permission inference of Mir relates to work on statically inferring permissions that an application requires in the Java permission model [36]. Jamrozik et al. [31] describe a dynamic analysis to infer pairs of Android permissions and UI 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 to both above approaches is that Mir focuses on `RWX` permissions for specific access paths, instead of the more coarse-grained permissions supported by Java and Android. Pailoor et al. [54] also propose static inference of privilege reduction policies, but they focus on system calls accessible to C/C++ programs and describe a more heavyweight static analysis than this paper. 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. [61] for handling dynamically computed field names.

Chen [17] is an analyzer for privilege escalation attacks on browser extensions written in JavaScript. Chen's constraint-based analysis aims at detecting vulnerabilities, whereas Mir aims at preventing their exploitation.

JavaScript Protection: There is prior work on JavaScript protection [5, 29, 47, 48, 62, 68, 73] 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 [47], and offers automation via static and import-time analysis. ZigZag [77] 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. In contrast, Mir infers `RWX` permissions statically and uses load-time interposition to insert runtime checks. NodeSentry [74] proposes powerful server-side JavaScript protection—but its policies are Turing-complete and written manually. Akhawe et al. [7] 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 [28] specify a way for executing scripts in different global environments to avoid cross-contamination, while SES [71] advocates for a shared immutable global realm. These proposals drastically reduce privileges for JavaScript code, but aggressively prevent all accesses to powerful APIs such as `require`. Once access is granted to this API, 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 [39], we also note that much of the aforementioned work focuses on providing powerful security mechanisms [5, 28, 48, 68, 73], 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.

Software Debloating: Functionality elimination [58] and, more recently, software debloating [11, 30, 34, 35] have similar goals to Mir, but approach the problem differently. Rather than locking what functionality a piece of code can access at runtime, these techniques completely eliminate unused functionality altogether. A benefit of these techniques is that an attacker circumventing Mir's runtime enforcement would still not be able to call non-existent functionality in a de-bloated application. A case where Mir offers benefits over these techniques is when two program fragments use disjoint halves of functionality: while no half can be eliminated, Mir still restricts each fragment to half the permissions.

Language- & Capability-Based Isolation: Software fault isolation [76] modifies object code of modules written in *unsafe* languages to prevent them from writing or jumping to addresses outside their domains. Singularity's software-isolated processes [6] ensure isolation through verification. Leveraging memory safety, Mir supports environments with *runtime code evaluation*, for which verification and static transformation might not be an option. Capability systems [40, 64] and object-capability systems [22, 50] restrict the ability to *name* a resource. Joe-E for Java [46] and Caja for JavaScript [50] restrict languages to object-capability-safe subsets. Similar to capability systems, Mir augments a program's ability to name a resource with a permission check. Mir does not focus on a language subset, and its static analysis offers significant automation. Prior work has developed formal frameworks for stating and proving strong isolation properties in the context of new languages or subsets of existing languages [1, 21, 23, 44, 45]. Mir instead the full JavaScript language and quantifies privilege reduction.

Ecosystem Approaches: The challenges of third-party libraries [53, 63, 65] can be addressed by checking for known vulnerabilities in

a program's dependency chain or by freezing dependencies [52]. These approaches may cause users to forego valuable bug and vulnerability fixes, whereas Mir allows the permissions to evolve with the codebase. The Deno runtime offers a coarse-grained allow-deny permission model focusing on the file-system and the network [19], but it lacks Mir's automation and fine granularity.

10 CONCLUSION

Dynamic library compromise due to problems in benign libraries poses a serious security threat. Mir is a system for Node.js that addresses this problem by augmenting the module system with a fine-grained read-write-execute (RWX) permission model for specifying privilege at the boundaries of libraries. It infers these permissions automatically using static and import-time analysis, and introduces privilege reduction, a metric capturing Mir's effects on prevented permissions. Mir's evaluation shows that it prevents tens of attacks on real-world vulnerable modules without major functionality disruptions and while imposing modest performance overhead. We envision Mir's automation and performance characteristics to make it an important addition to a developer's modern toolkit, similar to a minifier or a linter—in many circumstances working in tandem with defenses that focus on other threats, such as command-injection or sanitization attacks. Mir will be available for installation via npm and its source code will be made available on GitHub, with the camera-ready version of the paper:
anonymous.4open.science/r/58f1be03-3d2d-44a3-8f20-41b13327c908

ACKNOWLEDGMENTS

We would like to thank Jürgen Cito, Petros Efstathopoulos, Sage Gerard, Cătălin Hriţcu, Dimitris Karnikis, Daniel Kats, CJ Silverio, and Isaac Z. Schlueter. We are indebted to our shepherd, Peter Snyder. This research was funded in part by DARPA contracts HR00112020013 and HR001120C0191, NSF awards CNS-1513687 and CCF-1763514, by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of DARPA or other agencies.

REFERENCES

[1] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1351–1368. https://doi.org/10.1145/3243734.3243745

[2] Ajin Abraham. 2017. Snyk: Arbitrary Code Execution in node-serialize. https://snyk.io/vuln/npm:node-serialize:20170208. https://snyk.io/vuln/npm:node-serialize:20170208 Accessed: 2020-03-19.

[3] Ajin Abraham. 2017. Snyk: Arbitrary Code Execution in serialize-to-js. https://snyk.io/vuln/npm:serialize-to-js:20170208. https://snyk.io/vuln/npm:serialize-to-js:20170208 Accessed: 2020-03-19.

[4] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*.

[5] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2420950.2420952

[6] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. 2006. Deconstructing Process Isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC '06)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/1178597.1178599

[7] Devdatta Akhawe, Prateek Saxena, and Dawn Song. 2012. Privilege Separation in HTML5 Applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 429–444. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/akhawe

[8] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* (2017).

[9] Unknown Author. 2020. Snyk: Arbitrary Code Injection in serialize-javascript. https://snyk.io/vuln/SNYK-JS-SERIALIZEJAVASCRIPT-570062. https://snyk.io/vuln/SNYK-JS-SERIALIZEJAVASCRIPT-570062 Accessed: 2020-03-19.

[10] Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens. 2013. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *International Conference on Security and Privacy in Communication Systems*. Springer, 252–269.

[11] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1697–1714.

[12] Andrew Berman, Virgil Bourassa, and Erik Selberg. 1995. TRON: Process-specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON'95)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=1267411.1267425

[13] Nataliia Bielova and Tamara Rezk. 2016. A taxonomy of information flow monitors. In *International Conference on Principles of Security and Trust*. Springer, 46–67.

[14] Prithvi Bisht and V. N. Venkatakrishnan. 2008. 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*. 23–43.

[15] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322. http://dl.acm.org/citation.cfm?id=1387589.1387611

[16] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1251375.1251380

[17] Stefano Calzavara, Michele Bugliesi, Silvia Crafa, and Enrico Steffinlongo. 2015. 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 (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 510–534. https://doi.org/10.1007/978-3-662-46669-8_21

[18] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C Myers. 2012. Automatic partitioning of database applications. *arXiv preprint arXiv:1208.0271* (2012).

[19] Ryan Dahl and the Deno Contributors. 2019. Deno. https://deno.land/manual/getting_started/permissions. https://deno.land/manual/getting_started/permissions Accessed: 2020-06-11.

[20] Willem De Groef, Fabio Massacci, and Frank Piessens. 2014. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 446–455. https://doi.org/10.1145/2664243.2664276

[21] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative policies for capability control. In *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE, 3–17.

[22] Sophia Drossopoulou and James Noble. 2013. The Need for Capability Policies. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs (FTfJP '13)*. ACM, New York, NY, USA, Article 6, 7 pages. https://doi.org/10.1145/2489804.2489811

[23] Sophia Drossopoulou, James Noble, Mark S. Miller, and Toby Murray. 2016. Permission and Authority Revisited, Towards a Formalisation. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs (FTfJP'16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 6 pages. https://doi.org/10.1145/2955811.2955821

[24] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*.

[25] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 48–59. https:

//doi.org/10.1145/581478.581484
[26] Inc Google. 2009. Closure. https://developers.google.com/closure/. https://developers.google.com/closure/ Accessed: 2019-06-11.
[27] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1016–1031.
[28] Jordan Harband and Kevin Smith. 2021. ECMAScript® 2020 Language Specification. https://262.ecma-international.org/11.0/#sec-code-realms. https://262.ecma-international.org/11.0/#sec-code-realms Accessed: 2021-04-14.
[29] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.
[30] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.
[31] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. 2016. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 37–48. https://doi.org/10.1145/2884781.2884782
[32] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*. 273–284.
[33] Yoonseok Ko, Tamara Rezk, and Manuel Serrano. [n. d.]. SecureJS Compiler: Portable Memory Isolation in JavaScript. In *SAC 2021-The 36th ACM/SIGAPP Symposium On Applied Computing*.
[34] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*.
[35] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.
[36] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. 2002. 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*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 359–372. https://doi.org/10.1145/582419.582452
[37] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*. ACM, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562
[38] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. (2017).
[39] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. 1975. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles (SOSP '75)*. ACM, New York, NY, USA, 132–140. https://doi.org/10.1145/800213.806531
[40] H. M. Levy. 1984. *Capability Based Computer Systems*. Digital Press. http://www.cs.washington.edu/homes/levy/capabook/
[41] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-mandering: Quantitative Privilege Separation. (2019).
[42] Marcela S Melara, Michael J Freedman, and Mic Bowman. 2019. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245* (2019).
[43] Marcela S Melara, David H Liu, and Michael J Freedman. 2019. Pyronia: Redesigning Least Privilege and Isolation for the Age of IoT. *arXiv preprint arXiv:1903.01950* (2019).
[44] Darya Melicher. [n. d.]. *Controlling Module Authority Using Programming Language Design*. Ph.D. Dissertation. Carnegie Mellon University.
[45] Darya Melicher, Yangqingwei Shi, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. 2018. Using Object Capabilities and Effects to Build an Authority-safe Module System: Poster. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security (HoTSoS '18)*. ACM, New York, NY, USA, Article 29, 1 pages. https://doi.org/10.1145/3190619.3191691
[46] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java.. In *Networked and Distributed Systems Security (NDSS'10)*, Vol. 10. 357–374.
[47] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 481–496.
[48] James Mickens. 2014. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 261–275.
[49] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Baltimore, MD, USA. Advisor(s) Shapiro, Jonathan S. AAI3245526.
[50] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2009. Caja: Safe active content in sanitized JavaScript, 2008. *Google white paper* (2009).
[51] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 736–747.
[52] npm, Inc. 2012. npm-shrinkwrap: Lock down dependency versions. https://docs.npmjs.com/cli/shrinkwrap. https://docs.npmjs.com/cli/shrinkwrap
[53] Erlend Oftedal et al. 2016. RetireJS. http://retirejs.github.io/retire.js/
[54] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 135:1–135:26. https://doi.org/10.1145/3428203
[55] Andrea Parodi. 2009. Awesome Micro npm Packages (latest commit: Oct 5, 2020; a302e14). https://git.io/JUpA4. https://git.io/JUpA4 Accessed: 2020-10-07.
[56] Open Web Application Security Project. 2018. OWASP Top Ten Project'17. https://www.owasp.org/index.php/Top_10-2017_Top_10. https://www.owasp.org/index.php/Top_10-2017_Top_10 Accessed: 2018-09-27.
[57] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251353.1251369
[58] Martin Rinard. 2011. Manipulating program functionality to eliminate security vulnerabilities. In *Moving target defense*. Springer, 109–115.
[59] J. M. Rushby. 1981. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/800216.806586
[60] Jerome H Saltzer. 1974. Protection and the control of information sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402.
[61] José Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt. 2015. Hybrid typing of secure information flow in a JavaScript-like language. In *Trustworthy Global Computing*. Springer, 63–78.
[62] José Fragoso Santos and Tamara Rezk. 2014. An information flow monitor-inlining compiler for securing a core of javascript. In *IFIP International Information Security Conference*. Springer, 278–292.
[63] Node Security. 2016. Continuous Security monitoring for your node apps. https://nodesecurity.io/
[64] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. 1999. *EROS: a fast capability system*. Vol. 33. ACM.
[65] Snyk. 2021. Snyk Vulnerability Database. https://snyk.io/vuln?type=npm
[66] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 435–458.
[67] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node. js. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23071
[68] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 131–146. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan
[69] Michael Stepankin. 2016. [demo.paypal.com] Node.js code injection (RCE). http://artsploit.blogspot.com/2016/08/pprce2.html. http://artsploit.blogspot.com/2016/08/pprce2.html Accessed: 2018-10-05.
[70] Michael Stepankin. 2016. Snyk: Code Injection in dustjs-linkedin. https://snyk.io/vuln/npm:dustjs-linkedin:20160819. https://snyk.io/vuln/npm:dustjs-linkedin:20160819 Accessed: 2019-03-19.
[71] TC39. 2021. Draft Proposal for SES (Secure EcmaScript). https://github.com/tc39/proposal-ses. https://github.com/tc39/proposal-ses Accessed: 2021-04-20.
[72] Mike Ter Louw, Phu H Phung, Rohini Krishnamurti, and Venkat N Venkatakrishnan. 2013. SafeScript: JavaScript transformation for policy enforcement. In *Nordic Conference on Secure IT Systems*. Springer, 67–83.
[73] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. 2012. JavaScript in JavaScript (js. js): sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*. 95–100.
[74] Neline van Ginkel, Willem De Groef, Fabio Massacci, and Frank Piessens. 2019. A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries. *Security and Communication Networks* 2019 (2019).
[75] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131
[76] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA,

Tab. 5: Compatibility results *only* for 31 Q1 libraries (*Cf.*§8.3).

	Mir without import-time analysis	Full Mir
Inferred permissions (avg.)	88.9	379.5
Compatibility:		
Field access locations (out of 2387)	1660 (69.54%)	2363 (98.9%)
Packages (out of 31)	16 (51.61%)	26 (83.87%)
Test cases (out of 1511)	1119 (74.06%)	1499 (99.2%)

Tab. 6: Compatibility results *only* for 50 additional Q2 libraries (*Cf.*§8.3).

	Mir without import-time analysis	Full Mir
Inferred permissions (avg.)	13.3	17.38
Compatibility:		
Field access locations (out of 1,044)	762 (72.9%)	1,037 (99.3%)
Packages (out of 50)	42 (84%)	47 (94%)
Test cases (out of 1,046)	1,032 (98.6%)	1,042 (99.6%)

203–216. https://doi.org/10.1145/168619.168635

[77] Michael Weissbacher, William K. Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2015. 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*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 737–752. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/weissbacher

[78] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. 2012. Codejail: Application-transparent isolation of libraries with tight program interactions. In *European Symposium on Research in Computer Security*. Springer, 859–876.

[79] Nicholas C. Zakas and ESLint contributors. 2013. ESLint—Pluggable JavaScript linter. https://eslint.org/. https://eslint.org/ Accessed: 2018-07-12.

[80] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 995–1010.

A COMPATIBILITY BREAKDOWN (Q3)

Tab 5 and 6 report the compatibility results for (Q3) for the Q1 library subset and the Q2 popular libraries independently.

B DETAILED SECURITY ANALYSIS (Q1)

The two tables below present the details of Mir's security evaluation. Tab. 7 presents the first half—vulnerabilities we can install, for which we have or have managed to create exploits, and which fall under Mir's threat model. Mir defends against 61/63 exploits and crashes on the other two libraries; Mir crashes on these libraries even when run without the exploit. The exploit used in Mir's evaluation was either the one was provided with the vulnerability report, or one manually developed by the authors when no exploit came with the vulnerability report. (The last column of Tab. 7 presents information related to Mir's compatibility analysis, and specifically which Q1 libraries were part of Mir's compatibility evaluation and why.) Tab. 8 contains vulnerable libraries on which we did not apply Mir and the reason why. Within a 5-hour human-effort timeout per library, 7 libraries could not be exploited and 6 libraries could not be installed; 33 libraries fall outside Mir's threat model; and 23 libraries were made for a different language or platform.

C DETAILED PRIVILEGE ANALYSIS (Q2)

Tab. 9 below contains the results for Mir's privilege analysis.

Tab. 7: Mir defends against 61/63 exploits against real-world libraries and crashes on the other two libraries; Mir crashes on these libraries even when run without the exploit. The exploit used in Mir's evaluation was either the one was provided with the vulnerability report, or one manually developed by the authors when no exploit came with the vulnerability report (noted with M). The last column of the table lists which libraries from the Q1 dataset where included in Q3 and which were excluded (and the reasons for their exclusion).

Name	CWE	Snyk Category	R	W	X	I	Total	Attack	Exploit?	Q1	Q3
access-policy	CWE-78	Arbitrary Code Execution	37	6	21	5	69	Print 123		yes	Included
angular-expressions	CWE-94	Remote Code Execution	15	7	8	1	31	Write file angular-expressions-success	M	yes	Included
cd-messenger	CWE-78	Arbitrary Code Execution	242	9	144	27	422	Print JHU		yes	Included
cryo	CWE-502	Arbitrary Code Execution	203	9	117	33	362	Print defconrussia		yes	Included
dns-sync	CWE-94	Remote Code Execution	44	3	21	10	78	Write file test		yes	Included
domokeeper	CWE-200,23,94	Arbitrary Code Execution	8	0	3	1	12	Write file domokeeper-success	M	yes	No tests
domokeeper	CWE-94	Arbitrary Code Execution	8	0	3	1	12	Write file domokeeper-success	M	yes	Duplicate
ejs	CWE-94	Arbitrary Code Execution	135	22	64	14	235	Write file ejs-success	M	yes	Included
eslint-utils	CWE-94	Arbitrary Code Execution	19	35	11	0	65	Write file eslint-utils-success	M	yes	Crash on test
front-matter	CWE-94	Arbitrary Code Execution	19	2	11	5	37	Print 1		yes	Included
fun-map	CWE-78	Arbitrary Code Execution	4	16	1	0	21	Print 123		yes	No tests
growl	CWE-94	Arbitrary Code Injection	25	3	12	5	45	Execute ls		yes	Included
grunt	CWE-94	Arbitrary Code Execution	192	25	101	22	340	Returns Date.now		yes	Included
heroku-exec-util	CWE-94	Remote Code Execution	13	9	1	2	25	Write file HACKED		yes	No tests
hot-formula-parser	CWE-94	Arbitrary Code Injection	183	77	62	31	353	Write file test		yes	Crash on test
is-my-json-valid	CWE-94	Arbitrary Code Execution	41	3	25	9	78	Execute cat /etc/passwd		yes	Included
jingo	CWE-94	Arbitrary Code Execution	341	71	173	116	701	Returns Date.now		yes	Cannot run suite
js-yaml	CWE-94	Arbitrary Code Execution	341	71	173	116	701	Returns Date.now		yes	Included
kmc	CWE-94	Arbitrary Code Injection	313	7	192	83	595	Write file kmc-success	M	yes	Included
m-log	CWE-94	Arbitrary Code Injection	20	1	17	2	40	Print injected		yes	No tests
marsdb	CWE-94	Arbitrary Code Injection	457	64	167	122	810	Write file marsdb-success	M	yes	Included
mathjs	CWE-94	Arbitrary Code Execution	5113	808	3093	1753	10767	Execute ps		yes	Crash on test
meta-git	CWE-94	Remote Code Execution	14	0	10	7	31	Execute ls		yes	Included
mixin-pro	CWE-94	Arbitrary Code Injection	28	1	18	4	51	Print hacked		yes	Included
mobile-icon-resizer	CWE-94	Arbitrary Code Injection	13	7	4	2	26	Print hacked		yes	No tests
mock2easy	CWE-94	Arbitrary Code Injection	382	49	188	136	755	Write mock2easy-success	M	yes	No tests
modjs	CWE-94	Arbitrary Code Injection	955	263	572	199	1989	Write modjs-succes	M	yes	Crash on test
modulify	CWE-94	Arbitrary Code Injection	16	2	8	3	29	Print hacked		yes	Included
mol-proto	CWE-94	Arbitrary Code Injection	57	2	37	8	104	Write file mol-proto-success	M	yes	Included
mongo-edit	CWE-94	Arbitrary Code Injection	239	17	110	62	428	Write file mongo-edit-success	M	yes	Cannot run suite
mongo-express	CWE-94	Remote Code Execution	331	48	164	53	596	Execute id		yes	Cannot run suite
mongo-parse	CWE-94	Arbitrary Code Injection	42	5	19	7	73	Write file hacked		yes	Crash on test
mongodb-query-parser	CWE-94	Remote Code Execution	4	2	1	1	8	Write file test-file		yes	Included
mongoosemask	CWE-94	Arbitrary Code Injection	35	7	23	9	74	Print evil code		yes	Included
mongui	CWE-94	Arbitrary Code Injection	158	28	85	32	303	Write file mongui-sucess	M	yes	No tests
morgan	CWE-94	Arbitrary Code Injection	24	4	17	5	50	Print hello		yes	Included
mosc	CWE-78	Arbitrary Code Execution	7	1	5	2	15	Write file Song		yes	Included
node-extend	CWE-78	Arbitrary Code Execution	13	2	7	2	24	Print 123		yes	No tests
node-import	CWE-78	Arbitrary Code Execution	26	5	11	1	43	Write file JHU		yes	Cannot run suite
node-rules	CWE-78	Arbitrary Code Execution	55	4	36	17	112	Print 123		yes	Included
node-serialize	CWE-502	Arbitrary Code Execution	15	3	10	3	31	Execute ls		yes	Included
notevil	CWE-693	Remote Code Execution	70	6	42	16	134	Return this context		yes	Crash on test
notevil	CWE-94	Remote Code Execution	70	6	42	16	134	Print pwned		yes	Duplicate
pg	CWE-94	Arbitrary Code Execution	105	9	41	22	177	Print process.env		yes	Cannot run suite
pixl-class	CWE-78	Arbitrary Code Execution	12	1	5	2	20	Print 123		yes	No tests
protojs	CWE-94	Arbitrary Code Injection	149	19	75	16	259	Write file protojs-success	M	yes	Duplicate
realms-shim	CWE-265	Sandbox Breakout	204	3	72	5	284	Messed with Object.toString	M	yes	Crash on test
reduce-css-calc	CWE-94	Arbitrary Code Injection	12	1	9	2	24	Read /etc/passwd		yes	Included
safe-eval	CWE-265	Sandbox Breakout	9	1	5	1	16	Return process		yes	Included
safe-eval	CWE-265	Sandbox Escape	9	1	5	1	16	Execute whoami		yes	Duplicate
safer-eval	CWE-94	Arbitrary Code Execution	24	4	14	3	45	Execute id		yes	Included
safer-eval	CWE-94	Arbitrary Code Execution	24	4	14	3	45	Print process.env		yes	Duplicate
safer-eval	CWE-94	Arbitrary Code Execution	24	4	14	3	45	Write file safer-eval-success	M	yes	Duplicate
sandbox	CWE-94	Arbitrary Code Execution	40	1	22	7	70	Print process.pid		yes	Included
serialize-to-js	CWE-502	Arbitrary Code Execution	38	17	23	7	85	Execute ls		yes	Included
shiba	CWE-94	Arbitrary Code Execution	341	71	173	116	701	Returns Date.now		yes	Cannot run suite
static-eval	CWE-94	Arbitrary Code Execution	39	1	25	14	79	Print process.env		yes	Included
static-eval	CWE-94	Arbitrary Code Execution	39	1	25	14	79	Print process.pid		yes	Duplicate
thenify	CWE-78	Arbitrary Code Execution	9	1	6	2	18	Write file Song		yes	Included
value-censorship	CWE-693	Arbitrary Code Execution	18	2	6	3	29	Access the Function constructor		yes	Included
typed-function	CWE-94	Arbitrary Code Execution	163	10	123	31	327	Execute whoami		Crash: Module alters proto	Crash on lib
vm2	CWE-265	Sandbox Breakout	10	2	4	1	17	Executes Error command		Crash: Module applies wrapping	Crash on lib

Tab. 8: This table contains vulnerable libraries on which we did not apply Mir and the reason why. Within a 5-hour human-effort timeout per library, 7 libraries could not be exploited and 6 libraries could not be installed; 33 libraries fall outside Mir's threat model; and 23 libraries were made for a different language or platform.

Name	CWE	Snyk categorization	Why not included	Details
addax	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
angular	CWE-78	Arbitrary Code Execution	Outside Mir's threat model	SVG sanitization
angular	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	Combines several vulns, incl. XSS
bunyan	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
cocos-utils	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	Command injection / sanitization
commit-msg	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
constantinople	CWE-264	Sandbox Breakout	Outside Mir's threat model	Typescript
discord-markdown	CWE-79	Remote Code Execution	Outside Mir's threat model	XSS / sanitization
express-cart	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	Path traversal
expressfs	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
git-lib	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
git-promise	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
gity	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
handlebars	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	HTML sanitizer
jsrsasign	CWE-94	Remote Code Execution	Outside Mir's threat model	Package specific problem
listening-processes	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
local-devices	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
locutus	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
mathjs	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	Unicode attack
mversion	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
node-os-utils	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
npm-git-publish	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
nuclide	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	XSS-like vulnerability in Atom
office-converter	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
open	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
pdf-image	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
pomelo-monitor	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
require-node	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	Path traversal
sanitize-html	CWE-94	Arbitrary Code Execution	Outside Mir's threat model	HTML sanitizer
strapi	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Attacker controls which package to install
tomato	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
wifiscanner	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
wiki-plugin-datalog	CWE-94	Arbitrary Code Injection	Outside Mir's threat model	Command injection / sanitization
windows-edge	CWE-94	Remote Code Execution	Outside Mir's threat model	Command injection / sanitization
sanitize-html	CWE-94	Remote Code Execution	Different platform	Raspbian module
react-dev-utils	CWE-94	Arbitrary Code Execution	Different platform	Windows Platform
jstree	CWE-94	Arbitrary Code Injection	Different platform	Client-side library (Web)
quill	CWE-94	Arbitrary Code Execution	Different platform	Client-side library (Web)
nd-validator	CWE-94	Arbitrary Code Injection	Different platform	Client-side library (Web)
xterm	CWE-94	Remote Code Execution	Different language	C/C++
electron	CWE-94	Arbitrary Code Execution	Different language	C/C++
haraka	CWE-94	Remote Code Execution	Different language	C, incomplete MIME
gifsicle	CWE-94	Arbitrary Code Execution	Different language	Native C add on
electron	CWE-228	Arbitrary Code Execution	Different language	C/C++
electron	CWE-264	Arbitrary Code Execution	Different language	C/C++
electron	CWE-94	Arbitrary Code Execution	Different language	C/C++
electron	CWE-453	Arbitrary Code Execution	Different language	C/C++
electron	CWE-1188	Arbitrary Code Execution	Different language	C/C++
cordova-android	CWE-264	Arbitrary Code Execution	Different language	Java/JDK
soletta-dev-app	CWE-94	Arbitrary Code Injection	Different language	C/C++
electron	CWE-284	Arbitrary Code Injection	Different language	C/C++
logkitty	CWE-94	Remote Code Execution	Different language	Typescript
infraserver	CWE-94	Arbitrary Code Execution	Different language	Python
gitlabhook	CWE-94	Arbitrary Code Execution	Different language	Python
jquery-file-upload	CWE-94	Arbitrary Code Execution	Different language	Php
blueimp-file-upload	CWE-434	Arbitrary Code Execution	Different language	Php
microservicebus.node	CWE-94	Arbitrary Code Injection	Could not install	
nameless-cli	CWE-94	Arbitrary Code Injection	Could not install	
gitlab-workflow	CWE-94	Arbitrary Code Execution	Could not install	
m2m-supervisor	CWE-94	Arbitrary Code Injection	Could not install	
wxchangba	CWE-94	Arbitrary Code Injection	Could not install	
pouchdb	CWE-94	Arbitrary Code Injection	Could not install	
nodebb	CWE-94	Arbitrary Code Execution	Could not exploit	
serialize-javascript	CWE-94	Arbitrary Code Injection	Could not exploit	
total.js	CWE-94	Remote Code Execution	Could not exploit	
cordova-plugin-inappbrowser	CWE-94	Arbitrary Code Execution	Could not exploit	
irc-framework	CWE-94	Remote Code Execution	Could not exploit	
mathjs	CWE-94	Arbitrary Code Execution	Could not exploit	
realms-shim	CWE-265	Sandbox Breakout	Could not exploit	

Tab. 9: This table presents the privilege analysis results across all libraries. *Full* is the default privilege; `R`, `W`, `X`, and `I` are the numbers of permissions, with `RWXI` being the sum; and PR is the privilege reduction that Mir achieves.

	Full	R	W	X	I	RWXI	PR
algebra	1288	9	16	5	7	37	34.8×
arr-diff	1384	3	1	1	0	5	276.8×
arr-flatten	1288	3	1	1	0	5	257.6×
array-last	1288	6	1	5	1	13	99.1×
array-range	1288	3	1	2	0	6	214.7×
array.chunk	1288	6	1	2	0	9	143.1×
concat-stream	1288	21	1	9	4	35	36.8×
deep-bind	1288	4	1	3	1	9	143.1×
document-ready	1288	3	1	2	0	6	214.7×
file-size	1384	5	1	2	0	8	173×
fs-promise	1288	9	0	2	4	15	85.9×
get-value	1288	6	1	4	1	12	107.3×
group-array	1288	13	1	11	6	31	41.5×
has-key-deep	1292	1	1	0	0	2	646×
has-value	1288	6	1	4	2	13	99.1×
he	1288	8	0	3	0	11	117.1×
identity-function	1288	1	1	0	0	2	644×
in-array	1296	1	1	0	0	2	648×
is-empty-object	1288	5	1	2	0	8	161×
is-generator	1288	2	2	0	0	4	322×
is-number	1288	4	1	2	0	7	184×
is-promise	1288	1	1	0	0	2	644×
is-sorted	1288	4	1	2	0	7	184×
left-pad	1296	1	1	0	0	2	648×
missing-deep-keys	1288	4	1	3	2	10	128.8×
ndarray	1288	18	1	10	2	31	41.5×
node-du	1400	10	1	7	3	21	66.7×
node-glob	1400	46	3	28	13	90	15.6×
node-slug	1400	5	1	2	1	9	155.6×
node-stream-spigot	1400	16	4	6	3	29	48.3×
not-defined	1400	4	1	2	0	7	200×
once	1400	9	2	4	1	16	87.5×
pad-left-simple	1400	4	1	2	0	7	200×
pad-left	1400	3	1	2	1	7	200×
parse-next-json-value	1400	3	1	2	1	7	200×
periods	1400	2	0	0	0	2	700×
property-validator	1400	8	1	2	6	17	82.4×
rimraf	1400	13	2	6	4	25	56×
rtrim	1400	1	1	0	0	2	700×
schema-inspector	1544	3	1	1	1	6	257.3×
set-value	1400	9	1	5	1	16	87.5×
static-props	1400	4	2	1	0	7	200×
syncthrough	1400	10	1	6	4	21	66.7×
through2-map	1292	6	4	3	2	15	86.1×
through2	1288	12	3	5	2	22	58.5×
time-stamp	1288	3	1	2	0	6	214.7×
unordered-array-remove	1412	1	1	0	0	2	706×
zipmap	1288	10	1	3	0	14	92×
access-policy	1642	37	6	21	5	69	23.8×
angular-expressions	659	44	9	25	1	79	8.3×
cd-messenger	706	32	2	15	7	56	12.6×
cryo	313	16	1	9	0	26	12×
dns-sync	664	21	2	12	5	40	16.6×
ejs	336	50	14	27	4	95	3.5×
front-matter	327	13	2	5	1	21	15.6×
growl	319	13	2	5	1	21	15.2×
grunt	2559	281	29	159	30	499	5.1×
is-my-json-valid	327	25	2	17	5	49	6.7×
js-yaml	3362	341	21	209	87	658	5.1×
kmc	1899	44	15	19	13	91	20.9×
marsdb	3638	143	8	112	31	294	12.4×
meta-git	325	32	5	19	7	63	5.2×
mixin-pro	328	14	1	9	1	25	13.1×
modulify	312	10	2	5	2	19	16.4×
mol-proto	4126	23	3	12	7	45	91.7×
mongodb-query-parser	1326	39	27	21	9	96	13.8×
mongoosemask	310	15	5	10	1	31	10×
morgan	245	24	4	17	5	50	4.9×
mosc	311	3	1	2	0	6	51.8×
node-rules	664	19	3	9	3	34	19.5×
node-serialize	313	7	1	4	0	12	26.1×
reduce-css-calc	3325	60	12	27	7	106	31.4×
safe-eval	330	9	1	5	1	16	20.6×
safer-eval	661	20	7	11	3	41	16.1×
sandbox	326	21	1	10	5	37	8.8×
serialize-to-js	961	32	3	19	3	57	16.9×
static-eval	331	8	1	4	1	14	23.6×
thenify	330	9	1	6	2	18	18.3×
value-censorship	330	14	2	6	5	27	12.2×
min	245	1	0	0	0	2	3.5×
max	4126	341	29	209	87	658	706×
average	1212.87	22.12	3.37	12.59	4.11	42.21	143.48×