NR: Automated and Quantifiable Privilege Reduction Against Dynamic Library Compromise

Nikos Vasilakis* Cristian-Alexandru Staicu† Greg Ntousakis° Konstantinos Kallas‡
Ben Karel◇ André DeHon‡ Michael Pradel††
*MIT, CSAIL †TU Darmstadt & CISPA °TU Crete ◇Aarno Labs ††University of Stuttgart ‡University of Pennsylvania

***Abstract*—Third-party libraries ease the development of large-scale software systems, but often exercise significantly more privilege than they need to complete their task. This additional privilege is often exploited at runtime via *dynamic compromise*, even when these libraries are not actively malicious *per se*. NR addresses this problem by introducing a fine-grained read-write-execute (`RWX`) permission model at the boundaries of modules. Each property of an imported module is governed by a set of permissions, which developers can express and inspect when importing modules. To enforce these permissions during program execution, NR transforms modules and their context to inline runtime checks. As permissions can overwhelm developers, NR provides a permission inference engine that generates default permissions by statically analyzing how modules are used by their consumers. A prototype for JavaScript demonstrates that the `RWX` permission model combines simplicity with power: it is simple enough to be well-understood by developers and inferred by static analysis, expressive enough to protect real modules from practical threats, performant enough to be usable in practice, and enables a novel quantification of privilege reduction by identifying the ratio of disallowed interfaces.**

I. Introduction

Modern software development relies heavily on third-party libraries.[1] Such reliance has led to an explosion of attacks [28], [23], [27], [13], [45], [56]: overprivileged code in imported libraries provides an attack vector that is exploitable long after libraries reach their end-users. Even when libraries are created and authored with the best possible intentions, their privilege can be exploited at runtime to compromise the entire application—or worse, the broader system on which the application is executing.

Such *dynamic compromise* is possible due to several compounding factors. Libraries offer a great deal of functionality, but only a small fraction may be used by any one particular consumer [24]. Default-allow semantics give any library unrestricted access to all of the features in a programming language—*e.g.*, naming global variables, introspecting and rewriting core functionality, and importing other libraries [10]. Testing focuses on confirming a library's developed functionality (a relatively small space, well-understood by the library's developers) rather than catching pathological use cases (a much larger space, typically understood by the library's consumers) [12]. For example, only the consumer of a de-serialization library knows whether it will be fed non-sanitized input coming directly from the network; the library's developers cannot make such assumptions about its use.

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

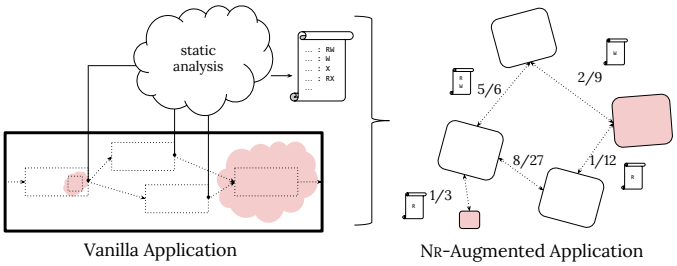


Fig. 1: NR overview. NR statically analyzes individual third-party libraries, many of which may be subvertible, to generate permissions. It then enforces these permissions at runtime in order to (quantifiably) lower the privielge of these libraries over the rest of the application and its surrounding environment.

To address dynamic compromise, NR[2] augments a module system with a model for specifying, enforcing, inferring, and quantifying the privilege available to libraries in a backward-compabible fashion (Fig. 1). NR's key insight is that libraries cannot be subverted at runtime to exploit functionality to which they do not already have access. Coupling default-deny semantics with explicit and statically-inferrable whitelisting, NR minimizes the effects of runtime subversion—regardless of the behavior exercised by a library and its clients.

NR's permission model allows specifying fine-grained *read-write-execute permissions* (`RWX`) that guard access to individual object fields available within the scope of a library. Such objects and fields are available through both explicit library imports and built-in language features—*e.g.*, globals, process arguments, `import` capabilities. In the aforementioned de-serialization library, two example permissions would be (1) `R` only on `process.env.PWD`, allowing read-access to no environment variable other than `PWD`, and (2) `X` only on `serialize`, disallowing execute-access to de-serialization.

Unfortunately, manually specifying permissions is challenging enough to overwhelm even security-paranoid developers. Typical challenges include (1) naming, such as variable aliasing, poor variable names, and non-local use, (2) library-internal code, possibly not intended for humans, and (3) continuous codebase evolution, which requires repeating the specification for every code change. To address these problems, NR offers an inference engine that generates default permissions automatically. By statically analyzing name uses within a library, NR extracts a likely set of permissions. In cases where precise extraction is impossible—*e.g.*, due to runtime meta-programming—NR can be configured to allow

[2]System name changed for double-blinding purposes.

matching multiple names, effectively giving developers more control between compatibility and security.

NR's permission model has multiple benefits. It is simple enough to be (1) understood and employed by developers without significant effort, and (2) inferred relatively accurately by static analysis. It also enables a novel quantification and reporting of the *privilege reduction* achieved by NR, especially useful in light of the aforementioned compatibility-security trade-offs. This is achieved by identifying the ratio of disallowed interfaces over the ones available to a library by default. This quantification is a direct byproduct of the permission model's discrete and finite nature—coarsely, a value's access is governed by a permission drawn from `RWX`, largely orthogonally to its semantics.

To enforce these permissions, NR transforms libraries upon import to inline security monitors. NR's transformations are enabled by the fact that modern dynamic programming languages feature a library-import mechanism that loads code at runtime as a string. Such lightweight load-time code transformations operate on the string representation of the module, as well as the context to which it is about to be bound, to insert permission-checking code into the library before it is loaded.

NR's implementation targets the JavaScript language, primarily because it powers the largest library ecosystem ($> 1.3M$ libraries). Results show that NR offers 98% compatibility, reduces privilege by up to three orders of magnitude, performs within 2% of the vanilla runtime, mitigates real threats that fall within its threat model, requires minimal-to-zero manual effort, and generally follows developer intuition. Sections IV–VIII present our key contributions:

- *Permission Model and Language*: a core permission model for controling the functionality available to modules, and associated extensions targeting an effective point in the usability-power spectrum (§IV),
- *Static Permission Analysis*: a permission inference engine that aids developers by statically analyzing library boundaries to generate default permission sets for individual libraries (§V),
- *Privilege Reduction Quantification*: a conceptual framework for quantifying privilege reduction, which allows reasoning about the privilege achieved by applying NR over various configurations (§VI),
- *Backward-compatible Runtime Enforcement*: a series of program transformations applied upon library load to insert permission checks, interposing on interactions with other libraries and core language structures (§VII),
- *Implementation and Evaluation*: an open-source implementation as an easily pluggable, backward-compatible library, along with an extensive performance, compatibility, and security evaluation (§VIII).

Other sections include an example illustrating runtime subversion and how NR addresses it (§II-B), a discussion of the threat model (§III), NR's relation to prior work (§IX).

```
function (req, res) {
 let srl, obj;
 srl = import("serial");
 obj = srl.dec(req.body);
 dispatch(obj, res);
}
```
main

```
let lg = import("log");
lg.LVL = lg.levels.WARN;
module.exports = {
 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) `imports` off-the-shelf serialization implemented by the `serial` *third-party* module (right), vulnerable to remote code execution (*Cf.*§II-A).

II. Background and Overview

This section uses a server-side JavaScript application to present (1) security issues related to third-party code (§II-A), and (2) an overview of how NR addresses these issues (§II-B).

A. Running Example: Using A De-serialization Library

Fig. 1 presents a simplified schematic of a multi-library program. Dotted boxes correspond to the context of different third-party modules of varying trust, one of which is used for (de-)serialization. This module is fed client-genarated strings, often without proper sanitization [46], which is one of several ways to cause remote code execution (RCE) attacks. RCE problems due to serialization have been identified in widely used libraries [6], [1], [2] as well as high-impact websites such as PayPal [48] and LinkedIn [49]. Injection and insecure de-serialization are respectively ranked number one and eight in OWASP's ten most critical web application security risks [38].

For clarity of exposition, Fig. 2 zooms into only two fragments of the program. The **main** module (left) **import**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 **import**s **log** and assigns it to the `lg` variable.[3]

Although **serial** is not actively malicious, it is subvertible by attackers at runtime, which 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.

B. Overview: Applying NR on the De-serialization Library

A developer can use NR to address these secrutiy problems. The first aspect they need to understand is that NR augments

[3]Naming is important in this paper: we differentiate between the module **log** and the variable `lg`. NR tracks permissions at the level of modules, irrespective of the variables they are assigned to. To aid the reader, modules (and more broadly, contexts) are typeset in **purple sans serif**, fields in **olive teletype**, and plain variables in `uncolored teletype` fonts.

library boundaries, treating boundaries as enclosing the entire library rather than just the interface (API) it exposes.

NR's permission model annotates functionality that is not part of the current 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** needs X. Another part of this functionality comes directly from the programming language; for example, **serial** clearly needs X for **import** and **eval**. It is not **serial** that provides 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, NR comes with a static inference component that analyzes how libraries use the available names. Fig. 3 shows a small example of this analysis: **levels** and **WARN** are read, thus are annotated as R; **LEVEL** is written, thus W; and **info** is executed, thus X. The analysis also infers permissions for **import**, **eval**, **module**, and **exports**. Names that do not show up—even if they are built-in language objects—get no permissions.

After extracting all necessary permissions, the developer can start the program using NR's runtime enforcement component. NR essentially shadows all variable names that cross a boundary with ones that point to modified values. Modified values for names that have persmissions check the corresponding permission before forwarding access to the original value. Modiefied values for names that do not have permissions essentially block all accesses; any access will throw a special exception to help the developer in diagnosing its cause.

The attacks described at the end of the previous section (§II-A) 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. Shielding against (4) module loading depends on a refinement on the base RWX model (§IV-B); this refinement does not make automated static inference intractable, as NR just notes that **import** is only executable with argument log.

Upon loading **serial**, NR also informs the developer of **serial**'s achieved privilege (among other libraries). This is helpful for a few different reasons, including the fact that NR comes with multiple configurations (necessarily ommitted for clarity) to support a brad compatibility-security trade-off space. For this configuration, NR reports that **serial**'s privilege has been reduced to 1.5% of the default one.

Sections IV–VIII discuss the details, after reviewing NR's threat model.

III. Threat Model

NR protects against against *dynamic* compromise of non-malicious libraries. The focus is runtime subversion of possibly buggy or vulnerable libraries, *i.e.*, ones that are not actively

```
var lg = import("log");
lg.LEVEL = lg.levels.WARN;
lg.info("srl:dec");
```

Read(R) Write(W) Execute(X)

Fig. 3: NR Inference. NR uses static analysis to infer module permissions. In this code snippet, the different fields of the **log** module are given distinct permissions based on how they are used.

malicious or hiding their intentions. Such libraries are subvertible by attackers providing payloads through web interfaces, programmatic APIs, or shared interfaces. Of particular interest are libraries that offer some form of object de-serialization or runtime code evaluation, where attackers can inject or execute arbitrary code by passing carefully constructed payloads. This is because these libraries implement their features using runtime interpretation, and thus are subvertible even when these payloads are expressed in a safe subset of the language or as Abstract Syntax Trees (ASTs) (see §VIII).

Focus: NR focuses on the confidentiality (*e.g.*, read global state, load other libraries, exfiltrate data) and integrity (*e.g.*, write global state or tamper with the library cache) of data and code. These concerns extend to the broader environment within which a program is executing—including environment variables, the file system, or the network. Such ambient over-privilege stems from: (1) common features in programming languages (*e.g.*, call stack inspection, reflection capabilities, monkey patching); (2) unusual language features or deficiencies (*e.g.*, in JavaScript: default-is-global, mutability attacks); (3) implementation concerns (*e.g.*, library cache, import capabilities); and (4) authority confusions, where all parts of a program have equal access rights (*e.g.*, read process.env or process.args, write to the file-system or network).

NR aims to mitigate the aforementioned attack vectors by allowing users to control ambient over-privilege at a fine granularity—that of individual names, functions, and fields in the scope of a module. NR does not limit itself to the interface exposed explicitly by a module, but rather captures the full observable behavior around it: any name that resolves to a value defined outside the module is protected—including language features, built-in names, and environment constructs accessible programmatically—as long as accessed with explicit naming.

Assumptions: NR'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, NR's runtime enforcement component is assumed to be loaded prior to any other library. NR 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.

NR does not consider native libraries written in lower-level languages such as C/C++ or available as binaries. These libraries are out of scope for two reasons. First, they cannot be analyzed by NR's static analysis, which fundamentally operates on source code. Second, they can bypass NR's language-based runtime protection, which fundamentally depends on

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

Fig. 4: NR's permission DSL. The DSL captures the set of annotations used for specifying permissions across libraries (*Cf.*§IV).

memory safety. Any operation violating memory safety could access arbitrary memory regions. NR also does not consider availability, denial-of-service, and side-channel attacks.

IV. Library Permission Model and Language

NR's goal is to reduce the privilege libraries possess. As a first step, it arms developers with the ability to explicitly specify a subset of the rights granted to modules by default upon import. This specification is expressible on a per-library using a domain-specific language (DSL, Fig. 4).

A. Core Permission Model

The core of NR's permission model and associated DSL is a per-library permission set: ModPermSet maps names accessible within the library context to a Mode, *i.e.*, a set of access rights encoded as `RWX` permissions. Ignoring `⋆`-constructs for now, names represent paths within the object graph reachable from within the scope of the library—*e.g.*, **String**.`toUpperCase`.

Paths: These 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. I. 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`, and the library-**import** ability *itself*.

The second class contains paths that start from explicitly importing a new library into the current scope. This 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` (§II-A).

NR's model can thus be thought as an object-path protection service: access rights are expressed as permissions associated with a path from the program's context roots to the field currently accessed. Values created within the scope of a library or a function are *not* part of this model: NR 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:

Tab. I: Object path roots. Objects resolving to a scope outside that of a module can be reached through a few starting points: (1) core built-in objects, (2) the standard library, (3) implementation-specific objsects, (4) module-locals, and (5) global variables.

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

- A read permission (`R`) grants consumers the ability to read a value, including assigning it to variables and passing it around to other modules.
- A write permission (`W`) grants consumers 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 consumers 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`).

`RWX` permissions are loosely based on the Unix permission model, with a few key differences. Reading a field of a composite value requires `R` permissions on the value *and* the field—that is, an `R` permission allows only a single dereference. A subtle case is `R`-permission functions, which can be read and communicated between compartments, but whose 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:
  serial.dec: X
```

The set of permissions for `serial` is more interesting:

```
serial:
  eval: X
  import: X
  module: R
  module.exports: W
  import("log").levels: R
  import("log").levels.WARN: R
  import("log").info: X
  import("log").LVL: W
```

B. Refinements and Subtleties

Having seen the base model, we now discuss a few refinements that target both simplicity and expressiveness.

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

Tab. II: NR'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 contract C	`someModule.foo = 5`
For each $a \in getAPIs(rhs)$	Add $(a, \texttt{R})$ to contract C	`x = import("someModule")`
	Add lhs *at* $l \mapsto a$ to $DefToAPI$	
Call of function f:		
For each $a \in getAPIs(f)$	Add $(a, \texttt{X})$ to contract C	`someModule.foo()`
Any other statement that contains a reference r:		
For each $a \in getAPIs(r)$	Add $(a, \texttt{R})$ to contract C	`foo(someModule.bar)`

This is achieved through an additional `I` permission. This permission is provided to an ObjPath that explicitly specifies the absolute file-system path of a library. Using the absolute file-system path is a concious 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 part of the program only for their side-effects (*i.e.*, not for their interface). In these cases, their fields should not necessarily be accessible by consumer code. Typical examples include singleton configuration objects and stateful servers.

Star Operators: It is often helpful or necessary to provide a single permission mode to *all* possible matches of a segment within a path. To achieve this, NR offers ⋆-operators: `⋆.f` assigns a mode to all fields named `f` reachable from any object, and `o.⋆` assigns a mode to all fields of a object (or path) `o`. Combinations of ⋆-operators are also possible—*e.g.*, `o.⋆.⋆.f`.

These operators 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, which means that they are not available for introspection by NR at library-load time.

This feature is also useful for NR's static analysis component. When NR cannot precisely infer the accessed path, often due to runtime meta-programming, it can be configured to output ⋆ operators in places of imprecise information.

V. Static Permission Inference

To aid users in expressing permissions, NR bundles a static analysis component that automatically infers permissions for a library and its dependencies. Intuitively, the static analysis extracts a partial specification that permits all the observed uses of module-external names in the analyzed module.

The core of the analysis is an intra-procedural, flow-sensitive forward analysis, that builds on a standard reaching-definitions analysis that NR queries to resolve references to their definitions. The analysis is neither sound nor complete, as these properties are hard to achieve for practical analyses of dynamic languages. Instead, the analysis is focused on correctly handling common cases of using third-party code, while being conservative in the granted permissions.

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 is growing monotonically during the entire analysis, *i.e.*, the analysis adds permissions until all uses of third-party code have been analyzed. 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 = import("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.

Table II summarizes how the analysis updates 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 II 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. NR 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. In line with NR's design goal of being conservative in granting permissions, the analysis will provide too few, instead of too many, permissions for code that does not meet these assumptions.

The update functions in Table II 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 = import("foo").bar`, $getAPIs(obj.x)$ will return the set $\{\text{“}foo.bar''\}$. When queried with a variable that does

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 $\{a + "." + prop \mid a \in A_{base}\}$
end
return $\emptyset$

Algorithm 1: Helper function $getAPIs$.

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 the pre-computed reaching-definitions information 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. This case 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 contract 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 reaching definitions analysis, which NR pre-computes for all functions in the module. To handle nested scopes, *e.g.*, due to nested function definitions, NR 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`, NR creates an artificial outer-most scope that contains the built-in APIs available in the global scope.

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

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

As illustrated by the example, the inferred contract 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 (§VIII-B).

VI. Quantifying Privilege Reduction

At runtime, NR estimates the achieved reduction on the privilege of libraries and the entire program as an aid to developers. As NR's static analysis is neither sound nor complete (§V), it offers no clear cut between compatibly and security; privilege quantification fills this gap, severing as a comparable index for NR's various configurations.

A. Privilege Reduction

The quantitative definition of privilege reduction depends on a few key notions. The first notion is a set of target critical resources M_t that are used to restrict a library's (or program's) access. The second notion is the set of subjects M_s that can access these resources through specific means (the permissions). NR's quantitative definition of privilege reduction is general in that it can be applied to any scenario with a finite number of subjects and critical resources.

Informal Development Before formalizing privilege reduction, we use the de-serialization example to build our intuition. From the two modules presented in Fig. 2, module **main** is the only subject module; thus, $M_s = \{\texttt{main}\}$. As implied earlier (§IV), the set of critical resources contains many paths available to **main**. For simplicity, we now assume it only contains **globals**, **fs**, and **import**; thus, $M_t = \{\texttt{globals, fs, import}\}$. Module **main** needs an `X` permission on **import** to be able to load **serial**, and `X` permission on **serial.dec** to be able to call the `dec` function. With this simple configuration, NR disallows all accesses except for $P(M_s, M_t) = \{\langle \texttt{import}, X\rangle, \langle \texttt{serial.dec}, X\rangle\}$.

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

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:

$$P_{base}(m_1, m_2) = \{\langle a, \rho\rangle | a \in API_{m_2}, \rho \in P\}$$

where $\rho \in \mathcal{P}$ is a set of orthogonal permissions on a resource, which for NR is $\mathcal{P} = \{R, W, X\}$. Name a can be an field that lies arbitrarily deeply within the export values of another module.

NR reduces privilege by disallowing permissions at module boundaries:

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

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:

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

where M_s is the set of subject modules that we are interested in (usually the untrusted ones), and M_t is the set of target resources (usually all the base critical modules in the system).

Based on this, we can define privilege reduction—*i.e.*, a metric of the permissions restricted by a privilege-reducing system S such as NR. Privilege reduction is defined on a set of subjects M_s and a set of target resources M_t.

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

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 malicious module that is installed in the execution environment. Consequently, the measured privilege reduction is actually a lower bound of the privilege reduction that NR achieves in practice.

B. Discussion

The discussion of privilege reduction leaves a few points unaddressed—most notably, the relative importance of different base permissions, the measurement of transitive permissions, and module-internal permissions.

Permission importance An important observation is that different permissions $\rho \in R$ may not be equally important. In principle, `X` may be considered more dangerous than `W`, which in turn may be considered more dangerous than `R`. Similarly, in terms of "base" APIs, **console**.`log` may be considered less important that **fs**.`write`. For these reasons, NR's model abstracts these values as constants, which developers can configure according to their preferences. Meaningful comparisons between different configurations should use the same constants.

Transitive Permissions Fig. 2's **main** is not allowed to directly call **eval**; however, it can call **eval** indirectly by executing **serial**.`dec`. This is a limitation of attempting to calculate the privilege reduction dynamically within an environment that specifies opaque abstraction boundaries such as libraries. Accurately quantifying such transitive privilege requires tracking transitive calls across such boundaries, which requires heavyweight information flow analysis. NR's privilege reduction quantification does not attempt such an analysis in order to keep runtime overheads low. As a result, NR's estimate is necessarily conservative—*i.e.*, NR reports a lower number than the one achieved in practice.

```
wrap (e: Value) : Value := match e with
| {(s, v) :: vs}  → {(s, wrap v) :: wrap vs}
| [v :: vs]       → [(wrap v) :: wrap xs]
| λ(v, ...).f     → λ(v, ...).{ (R, W, X)? f(args) : () }
| __              → interpose(e)
end
```

Fig. 5: Base transformation. The algorithm (simplified) is presented in functional style to simplify variable binding; types, whose structure is used for pattern matching, are shown in light *turquoise* (*Cf.*§VII).

VII. Permission Runtime Enforcement

During the execution of a program, NR's runtime component enforces the permission set that was generated statically. NR's runtime enforcement approach is enabled by the fact that modern dynamic languages such as JavaScript feature a module-import mechanism that loads code at runtime as a string. Lightweight load-time code transformations operate on the string representation of the module, as well as the context to which it is about to be bound, to insert enforcement-specific wrappers into the module before it is loaded.

NR's transformations can be grouped into four phases. The first phase simply modifies `import`, such that calls yield into NR 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 binds them to the modified values. After interpreting the module, the fourth phase further transforms the module's interface so that its consumer can only access the names—*e.g.*, methods, fields—it is allowed to access.

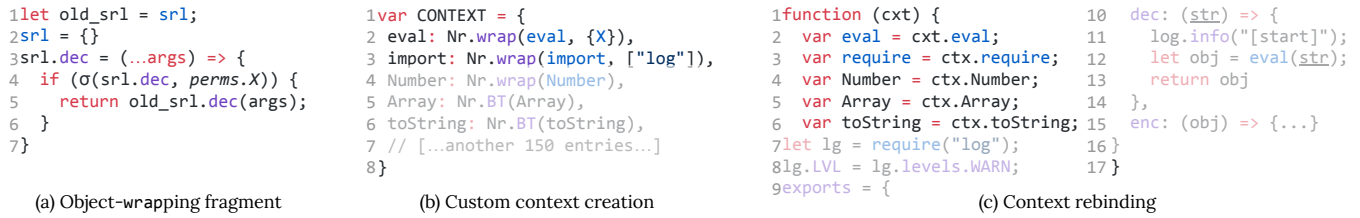
These transformations have a common structure that traverses objects recursively—a base transform `wrap` which we review first. Each round of `wrap` encloses the current field with a new security monitor—a level of indirection that oversees accesses to the field and ensures that they conform to the permissions corresponding to that field. If a violation is detected, NR 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.

Transforming Values NR's transformations boil down to a base form `wrap` that traverses objects and wraps their fields with runtime security monitors, outlined in Fig. 5. 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. 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 raises an exception.

The result of applying the `wrap` transformation to the object (returned by) `serial` is shown in Fig. 6a.

Context Creation To prepare a new context to be bound to a library being loaded, NR first creates an auxiliary hash table (Fig. 6*c*), mapping names to (newly transformed) values: names correspond to implicit modules—globals, language built-ins, module-locals, *etc.* (Tab. I); 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,



(a) Object-wrapping fragment (b) Custom context creation (c) Context rebinding

Fig. 6: NR enforcement transformation and context rebinding. NR's basic wrapping traverses objects and wraps fields with inline monitors (a, line 4). All values available in the module context are wrapped, Finally, the modified context is linked with the module, by wrapping the module source with a function closure that redefines all globally-available variable names as function-local ones and populating them with values from the modified context (*Cf.*§VII).

NR collects such values by resolving well-known names hard-coded in a list. Using this list, NR 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 NR's scope will fail subtly, as they will end up resolving to module-local values of NR *itself*—and specifically, the module within NR applying the transformation. NR solves this issue deferring these transformations for the linking phase (*i.e.*, from within the module).

The result of modifying `serial`'s context is shown in Fig. 6b.

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

NR's closure starts by redefining and enclosing names default-available in the current scope as module-local ones, pointing to transformed values that exist in the newly-created context. The closure 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. NR 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. 6c.

VIII. Evaluation

We built a prototype of NR for JavaScript, targeting the Node.js ecosystem, and available via `npm -i @blind/nr`. Permissions (§IV) are specified as auxiliary JSON files. NR's runtime enforcement component is about 2.8 KLoC. NR's static analysis component (§V) is implemented as a compiler pass in the Google Closure Compiler [17] in about 2.1 KLoC. To evaluate NR, we investigate the following questions:

- **Q1** How compatible is NR with existing code-bases—*i.e.*, what is the danger of breaking legacy applications? Out of 50 libraries, 13 (but only 2% of tests) encountered an invalid access; the number of invalidly blocked accesses is less than 10 for all but two libraries. ⋆-operators improve by 85.1% over this baseline, particularly for libraries with many invalid accesses.
- **Q2** How effective is NR in theory, in terms of potential for privilege reduction? NR provides dozens-to-hundreds-fold reduction in terms of permissions over the default execution. The effects due to ⋆-operators is relatively minor (*i.e.*, same orders of magnitude are observed). This is consistent with the intuition that most code uses a small subset of the API surface of the libraries it imports.
- **Q3** How effective is NR in practice, against *in vivo* and *in vitro* vulnerabilities? Applied on multiple real and synthesized vulnerabilities, NR mitigates all attacks that fall under its threat model. In all scenarios, NR applied default permissions inferred automatically, except for one library.
- **Q4** What are NR's performance characteristics for both its inference and enforcement components? NR's static analysis averages 1.258s per library, comparably to a widely-used linter (`eslint`), and its runtime overhead remains under 2% with the majority of the overhead coming from tracking global variables.

The remainder of the evaluation section presents the supporting evidence and details underlying these findings.

Libraries To answer Q1, Q2, and Q4 we perform a large-scale evaluation on 50 JavaScript packages from the npm ecosystem. These libraries are highly popular, averaging 4.8M weekly downloads (total: 227M) and are depended upon by about 656 other packages or applications on average (total: 30K). These libraries are relatively small—adhering to known ecosystem trends [56]—simplifying compatibility inspection.

To answer Q3, we use a different set of benchmarks that contain confirmed vulnerabilities. Half of the vulnerabilities were found *in vivo* [45]; the rest were generated *in vitro* to test the feasibility of certain attacks.

Workloads Except noted otherwise, each library was run against the test suite provided by its nominal developers—usually via `npm test`. As these libraries are quite popular, development investment in their testing infrastructure has been significant, having a variety of different tests that stress

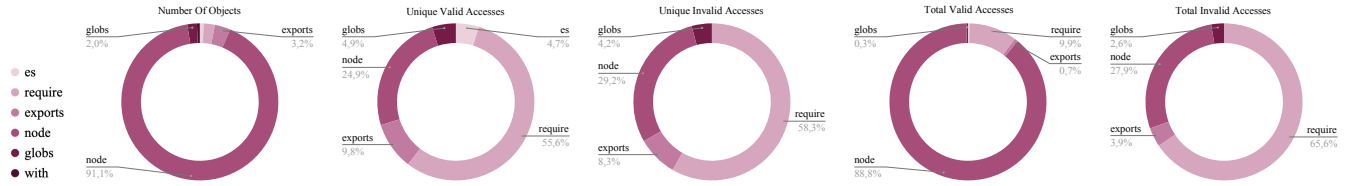


Fig. 7: Context vs. compatibility. From the left: (a) number of NR wrappers, (b) unique valid accesses, (c) unique invalid accesses, (d) total valid accesses, and (e) total invalid accesses—all as a function of context (*Cf.*§VIII-A)

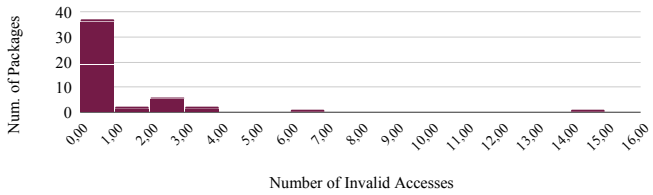


Fig. 8: Invalid accesses. Of the 50 libraries on which we applied NR, only 13 showed some incompatibility. Of these 13, 11 diverged on nine permissions or less (*i.e.*, $> 95\%$ of all); the remaining two diverged on 14 (shown) and 135 (not shown, for clarity) permissions respectively (*Cf.*§VIII-A).

different parts of the library, corresponding to different runtime accesses. In contrast to most applications that only use a fraction of the provided functionality, tests cover the majority of the provided functionality—*e.g.*, we observed test suites that were $10\times$ the size of the corresponding library.

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 v8.9.4, bundled with V8 v6.1.534.50, LibUV v1.15.0, and `npm` version v6.4.1. NR's static inference run on Java SE 1.8.0_251, linked with Google Closure v20180910. Libraries were tested on their latest versions at the time of this writing.

Except when noted otherwise, NR's runtime enforcement is configured with depth 3, focuses only on the target module, includes full context, and excludes fields with names in the following set: `valueOf`, `toString`, `Promise`, `prototype`. This is NR's default configuration; at times, we show results for other configurations.

A. Compatibility Analysis (Q1)

To a large extent, backward-compatibility drives practical adoption of tools such as NR—if a tool requires significant effort to address compatibility, chances are developers will avoid it despite of any security benefits it provides. This is especially true for NR which follows a static analysis approach to meet its threat model of dynamic subversion, rather than one based on dynamic analysis.

Big Picture NR applies an average of 346 transformations per library, wrapping a total of 25,609 fields. Not all fields are accessed at runtime—on average only 20.88 (6.06%) of fields are accessed per library. The ones that do, are accessed multiple times: 794.9 times per field, on average.

Fig. 8 shows a histogram of the number of packages and the number of unique invalid accesses, *i.e.*, the number of benign accesses that happen in the tests but were not found by NR's static analysis. From the 50 libraries on which we applied NR, only 13 showed some divergence between static and runtime behavior. Of these 13, 11 diverged on nine accesses or less (*i.e.*, $> 95\%$ of all); the remaining two diverged on 14 (shown) and 135 (not shown, for clarity) accesses respectively, due to their focus (explained below).

These libraries were tested under multiple workloads each triggering different accesses. Counting with respect to tests, rather than libraries, about 98% of all tests performed no invalid accesses, and 99.9% performed less than 10 invalid accesses. From 1044 total unique accesses, only 158 (15.1%) were invalid; if repetitions are taken into account, 310 out of 16,598 (1.86%) were invalid—meaning that cases not inferred correctly by the static analysis are indeed rare in practice.

Context Context refers to the broad source of names that are available in the current scope (§IV). This includes names defined by the EcmaScript standard (**es**), through an explicit import (**exports**), by the Node.js runtime (**node**), or via global variables (**globs**). A few names seem globally available but are in fact module-locals (**require**). User-defined global variables are not prefixed with `global` thus requiring special interposition.

Fig. 7 shows several statistics across all 50 libraries as a function of context. In terms of the number of objects wrapped, the vast majority comes from **node** which accounts for 91.1% of all wrappers. In terms of unique number of accesses, for both valid and invalid ones the majority comes from **require**. Taking the number of accesses into account, the majority of accesses comes from **es** and **node** names, whereas the majority of invalid ones comes from **exports**.

These results highlight that NR gathers most of its correct results around **node** and **es** names. In retrospect, this makes sense: there is no dynamic resolution required to see, for example, that **Math** refers to the standard math library—and this is the most common to refer to it. The majority of invalid accesses come from module exports. This is expected, as a few meta-programming libraries use introspection and reflection capabilities to rewrite entire interfaces and re-export them dynamically.

Depth Depth refers to how far NR traverses references starting from the names of objects in the contexts. For example, the access **global**.`obj`.**x** is two levels deep and **fs**.`readFile`

is one level deep.

Fig. 9 shows the same statistics as before but relative to depth. There are a few highlights worth noting. First, while the number of objects which NR's runtime component keeps track of start growing exponentially, it quickly reaches an average upper bound of about 400 (depending on the exact configuration). Second, accesses grow quickly for the first couple of levels—objects for some of these libraries are hit several thousand times—but then remain stable. This is usually because many interfaces follow a mostly-flat format where all methods are defined in the top level. Lastly, the majority of invalid accesses remain constant across several levels of depth. This is mostly because these libraries exhibited dynamic behavior at exactly one point—*e.g.*, the `export` object or a dynamically-computed property (see Tab. III). (This is also true for **fs-promisify** which these plots do not include).

B. Permissions and Privilege Reduction (Q2)

To understand the privilege reduction achieved by NR's different configurations, we compare the resulting sets of permissions with the ones available to modules by default.

Permissions and Privilege Reduction Fig. 10 shows NR's base number of statically inferred `R`, `W`, `X`, and `I` permissions. The total number of permissions ranges between 3–54 (avg: 11.9),

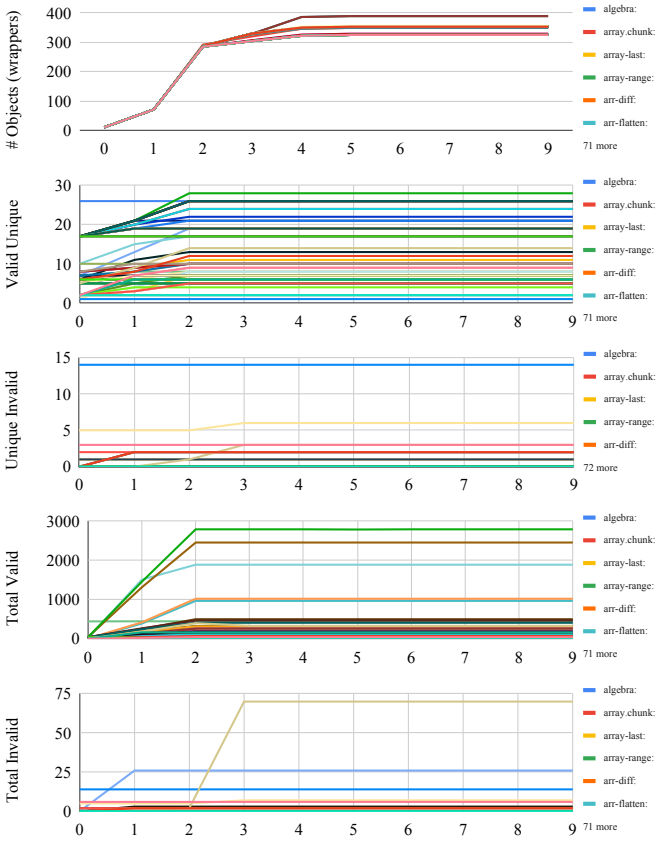


Fig. 9: Depth vs. compatibility. From the top: (a) number of NR wrappers, (b) unique valid accesses, (c) unique invalid accesses, (d) total valid accesses, and (e) total invalid accesses—all as a function of depth (*Cf.*§VIII-A).

spread unevenly between `R` (6.85), `W` (1.54), `X` (3.42), and `I` (1.52). The use of ⋆-operators (not shown, for clarity), affects primarily the upper bound of this range: adding **o.⋆** pushes the max to 159 (avg: 18.94), **⋆.f** to 54 (avg: 12.02), and the combined ⋆.⋆ to 168 (avg: 19.42). NR infers a small number of permissions because developers use only a small number of APIs.

Fig. 10 also shows the privilege reduction achieved by NR—*i.e.*, the ratio of allowed permissions over a library's default set of permissions (§VI). In this configuration, the default set of permissions includes all the names that a library can access except fields used for implicit type coercions (see evaluation setup in §VIII); adding these would lead to higher reduction, but would also affect compatibility.

For the base set of permissions, privilege is reduced by up to three orders of magnitude, ranging between 15.6×–706× (avg: 224.5×). ⋆-operators have small effects on this: 5.6–644× (avg: 133×) for **o.⋆**, 15.6–706× (avg: 222.98×) for **⋆.f**, and 5.3–644× (avg: 131.24×) for ⋆.⋆. NR's privilege reduction is high because developers use only a small fraction of the available APIs.

Incompatibility We now turn our attention to the reasons for incompatibility (Fig. 8) and the benefits provided by NR's star operators. The two are interlinked, as these operators were designed explicitly to tackle cases where the static analysis fails to capture the program's runtime accesses due to dynamic behaviors.

Tab. III zooms into modules that show some incompatibility. It shows the number of inferred permissions (col. 2, 3, 4, 5), the number of invalid permissions (col. 6, 7, 8, 9), and the primary reasons for each incompatibility, under four configurations: base model, **o.⋆**, **⋆.f**, and ⋆.⋆. The ⋆ configurations improve compatibility significantly by lowering invalid accesses by 76% with **o.⋆**, 6.9% with **⋆.f**, and 85.1% with ⋆.⋆ over the baseline, on average. The next few paragraphs focus on a few interesting cases.

Modules `zipmap` and `concat-stream` create a local **`toString`** that they get via the **Object** prototype. NR's runtime enforcement is configured to not wrap the prototype chain by default, missing the `R`. Moreover, the runtime reports **`toString`** as global (capturing a `R` on `global` and `W` on **`toString`**), because it is defined via `var`. (Changing `var` to `let` leads to base incompatibility of 1.) Modules `static-props`, `set-value`, `fs-promise` `is-generator`, and `he` compute properties they access dynamically. Module `fs-promise` is a good example of this behavior: it traverses the built-in `fs` interface to generate wrappers that return a promise.

These cases highlight a fortunate correspondence between the size of the interface manipulated programmatically, the possible incompatibility, and ⋆-operators. When developers use many or all of the properties of an object, they are likely to access them via reflection, and thus manipulate them programmatically. Conversely, if they use reflection, they are likely to be accessing many (or all of the) properties. In both

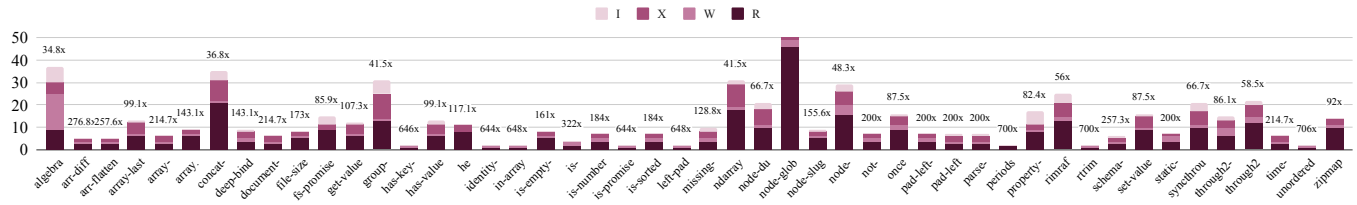


Fig. 10: Statically Inferred Permissions. Number of statically inferred permissions. Stacked bars show the number of `R`, `W`, `X`, and `I` permissions; the number above the bar shows the privilege reduction over their default permissions (*Cf.*§VIII-B).

Tab. III: Invalid accesses and reasons for incompatibility. The majority of incompatibility are due to dynamically computed properties—*e.g.*, on objects such as **export** and **process.env**. The runtime enforcement also miss-characterizes a few property accesses (*Cf.*§VIII-B).

	Inferred Permissions				Invalid Permissions				
	o.f	o.⋆	⋆.f	⋆.⋆	o.f	o.⋆	⋆.f	⋆.⋆	Reason for incompatibility
zipmap	15	19	15	19	3	3	3	3	Accesses global `toString` via `Object.prototype`
through2	18	28	18	28	2	2	2	2	Gets (inherits) properties through a testing library
syncthrough	15	44	15	44	2	2	2	2	Access to `Function` object through prototype / instance check
static-props	10	10	10	10	2	2	0	0	Dynamic field access to `exports.*`
set-value	14	21	14	22	3	3	2	1	Dynamic access to `process.env.npm_package_keywords_15`
periods	6	9	6	9	14	14	14	14	Static analysis does not see `RWX` (function call in `export` index)
node-slug	9	33	9	38	1	1	1	0	Dynamic property access to `module.exports.*`
missing-deep-keys	9	9	9	9	2	2	2	2	Invalid accesses arise from `npm run coverage` (not `test`)
is-generator	7	11	7	11	2	2	2	0	`fn.constructor` does not see `fn` as a parameter
he	12	32	12	32	6	6	1	0	Dynamically computed fields `exports.version` *etc.*
fs-promise	14	14	16	16	135	2	131	2	Dynamically computed across all methods of `require("fs")`
file-size	9	21	9	21	1	1	1	1	Assignment at return: `return exports = plugin()`
concat-stream	26	44	26	45	2	2	2	1	Dynamic analysis see `isArray` as `R`
total	164	295	166	304	175	42	163	26	85.1% due to dynamically computed fields

Tab. IV: Vulnerable Modules *In vivo* (top) and *in vitro* (bottom) attacks defended by NR. The last four columns show permissions under four NR configurations; for the *in vitro* cases, only `eval` is allowed (*Cf.*§VIII-C).

Module	Version	Attack	CVE	CWE	base	o.⋆	⋆.f	⋆.⋆
Ser/pt	< 1.0	Arbitrary Code Execution	2017-5941	502	13	30	13	30
Nod/ze	0.0.4	Arbitrary Code Execution	2017-5954	502	7	18	7	19
Ser/js	< 3.1	Arbitrary Code Injection	2020-7660	94	58	113	59	115
Saf/al ×4	< 0.4	Sandbox Escaping	(multiple)	265	10	11	10	11
Stat/al ×2	< 2.0	Arbitrary Code Execution	2017-16226	94	9	53	9	60
Fast-redact	< 1.5	Arbitrary Code Execution	—	94	32	52	32	52
Mathjs ×5	< 1.5	Arbitrary Code Execution	(multiple)	94	1	1	1	1
Morgan	< 1.9	Arbitrary Code Injection	2018-3784	502	27	65	27	68
Glob.js	—	write `x`, `global.x`	—	—	2	2	2	2
Mod.js	—	read `require.cache`	—	—	2	2	2	2
Arg.js	—	execute `process.argv`	—	—	2	2	2	2
Env.js	—	read `process.env`	—	—	2	2	2	2
Fs.js	—	`fs.read /etc/hosts`	—	—	2	2	2	2
Cp.js	—	`chi/ocess.spawn`	—	—	2	2	2	2
Es.js	—	call `Math.log`	—	—	2	2	2	2
Arr.js	—	call `Array`	—	—	2	2	2	2
Eval.js	—	call `eval`	—	—	2	2	2	2
Os.js	—	read `os.EOL`	—	—	2	2	2	2

cases, static analysis is unlikely to be extract these behaviors, which is exactly where ⋆-operators shine.

C. Practical Security Impact (Q3)

This section complements the analysis of NR's privilege reduction with an investigation of its effectiveness on vulnerable code. Table IV shows the results of applying NR on a combination of *in vivo* and *in vitro* vulnerabilities. The former focus on packages found in the wild [45], for which the table includes the vulnerable versions and MITRE's common enumeration identifiers [33], [34]. The latter are attacks that we were not able to find in the wild by scanning the JavaScript vulnerability databases, but which we know are possible under certain scenarios. NR defends against all these vulnerabilities; apart from *Mathjs*, no permissions required manual intervention.

In Vivo For these vulnerabilities, we use the proof-of-concept exploit (PoC) attached to the original vulnerability report; many of them include multiple CVEs or PoCs, all of which we attempt. *Ser/pt*, *Nod/ze*, and *Ser/js* all perform some form of unsafe serialization; their 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, NR disallows access to these APIs.

Saf/al and *Stat/al* sanitize input prior to calling **eval**. The PoC payloads are similar to the ones described earlier, against which NR protects. The case of *Stat/al* is interesting because it accepts ASTs rather than strings; the PoC passes **process.env** through the Esprima parser making it vulnerable—which NR solves by disallowing access to **process.env**.

The *Fast-redact* PoC uses a **constructor** to reach into the **Function** prototype and invoke **open** via **process** bindings. NR allows the first two steps, crucial for compatibility, but blocks the call to both **bindings** and **open**.

The popular *Mathjs* module includes a math evaluator, which the PoC exploits to **console**.`log` a message. As *Mathjs* is compiled into a universal-module-definition by webpack (*i.e.*, features no imports and thus is a pure function), we provided its permissions manually. Despite the fact that *Mathjs* does not import any modules, the PoC calls into Node's APIs

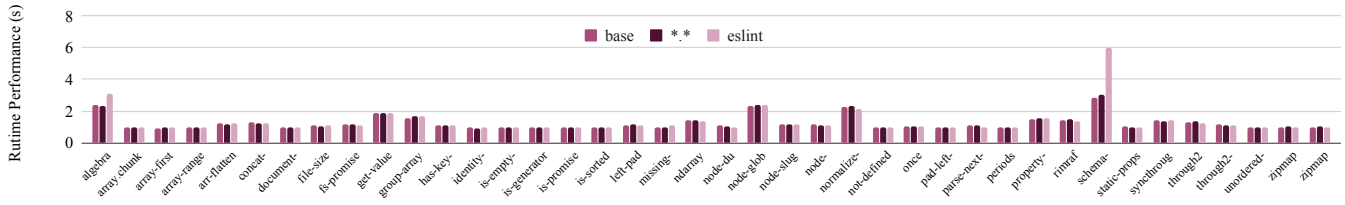


Fig. 11: Performance of NR's static analysis Comparison between two different configurations of NR's static analysis; for reference, they are compared with linting performance for the same codebases (*Cf.*§VIII-D).

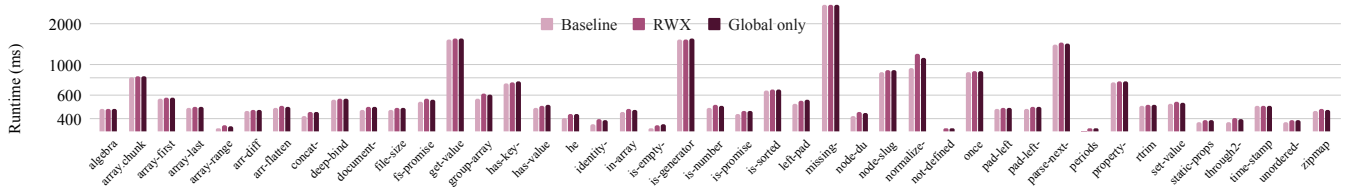


Fig. 12: Performance of NR's runtime enforcement. Comparison between vanilla runtime performance with (1) full `RWX` with 3-level depth and across all contexts, and (2) 3-level depth across global-only variables. On average, +3.3ms per library (1.93% slowdown) and +3.2ms (1.62% slowdown) (*Cf.*§VIII-D).

via **cos**.`constructor` (à la *fast-redact*). While NR does not block the `constructor` call, it does block **process**.`env` and **fs**.`read` from within *Mathjs*.

For most of these benchmarks, NR blocks the PoC at multiple levels—*e.g.*, even if *Nod/ze*'s import of **child_process** was allowed, NR would still block `exec`.

In Vitro We construct a benign-but-vulnerable module `e` that only exposes an **eval** function. NR infers two permissions: `{e: {eval: RX, exports: W}}`.

We pass a large amount of code as string from NR's extensive testing infrastructure—over 800 tests (>11K possible accesses), targeting built-in JavaScript, Node, and Common.js interfaces—of which we only highlight 10. NR allows primitive operations within `e`, such as arithmetic on primitive values and basic string manipulation, the use of Array and Object literals, as well as code evaluation through the **Function** (and via `constructor`) and **eval** functions (the latter shown in Tab. IV). However, it disallows operations that have broader side-effects outside `e`—such as the ones in Tab. IV.

D. Performance Analysis (Q4)

Static Analysis Fig. 11 shows the runtime performance of NR's static analysis under two static analysis configurations: the base model and star operators. To put these results into context, it also includes the performance of running benchmarks against a lint—a lightweight static analysis flagging common human errors. We use `eslint` (v5.0.1) [55] and each library's default linting configuration, falling back to Google's style rules when no such configuration exists.

Linting aside, NR takes an average of 1.258s (base) and 1.272s (star) per library. As libraries are relatively small, fixed (per run) startup costs seem to dominate these overheads: running NR across *all* benchmarks in a single pass takes 13.52s and 13.79s for base and stars—both averaging to about 0.27s per library.

Despite its fixed startup costs, NR performs comparably to `eslint`, which averages 1.343s per package—that is, NR is faster by about 70ms, with the exact number depending on the configuration details. NR can thus be used as part of modern development workflows, typically involving linting and minification, without being detrimental to their performance.

Dynamic Enforcement Fig. 12 shows the performance of NR's runtime enforcement component under two configurations: full `RWX` enforcement across all contexts and `RWX`-light focusing only on global variables (both at the default 3-level depth). These are compared with the performance of the unmodified runtime.

On average, NR's full `RWX` enforcement adds about 3.3ms per library causing 1.93% slowdown. Global-only enforcement is slightly faster, adding a 3.2ms causing a 1.62% slowdown. Based on these results, we do not anticipate a need for users to trade-in runtime security to gain performance.

To understand the sources of these overheads, we perform a series of micro-benchmarks with tight loops calling several ES-internal libraries without any I/O. The primary source of overhead seems to be the use of JavaScript's `with` construct (§VII), The `with` construct dominates overheads, as it (1) interposes on too many accesses, only a fraction of which are relevant, and (2) remains significantly unoptimized, since its use is strongly discouraged by the JavaScript standards [35].

IX. Related Work

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

Privilege Reduction A number of works have addressed privilege reduction [42], [41], [3], [39], [20], [11], [10], [54], [7], [15], [18], often offering significant automation. This automation often comes at the cost of *lightweight annotations* on program objects—*e.g.*, configurations in Privman [20], `priv` directives in Privtrans [11], tags in Wedge [10], and

compartmentalization hypotheses in SOAAP [18]. TRON [9] introduced a permission model similar to NR, but at the level of processes rather than libraries.

Wedge and SOAAP stand out as offering significant automation via dynamic and static analysis, respectively. Despite such automation, Wedge requires altering programs to use its API and SOAAP mostly checks rather than suggests policies. In comparison to these works, NR (1) leverages existing boundaries, and (2) offers significantly more automation.

To ameliorate manual annotations on individual objects, more recent library-level compartmentalization [52], [22], [29] exploits runtime information about module boundaries to guide compartment boundaries. NR is different from these works in both the threat model and techniques: (1) it focuses on benign-but-buggy libraries, rather than actively malicious ones, (2) leverages language-based techniques both for the analysis and enforcement, and (3) offers a simple `RWX` permission model rather than more expressive (often Turing-complete) policies.

Pyxis [14] and PM [26] reduced the problem of boundary inference to an integer programming problem by defining several performance and security metrics. These systems can be seen as complementary to NR, as they focus on separating the application code to a sensitive and insensitive compartment in order to minimize these metrics, while NR tries to automatically infer and restrict the permissions between different libraries.

Language-based Isolation NR draws inspiration from language-based isolation techniques [53], [5]: rather depending on the operating system for protection, NR enforces access control from with the language runtime—even for libraries that access the file-system and the network. Software fault isolation [53] 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 [5] ensure isolation through software verification. Leveraging memory safety, NR can be applied in environments with *runtime code evaluation*, for which software verification and static transformation might not be an option.

JavaScript Isolation There is a significant amount of work focusing on JavaScript protection [31], [32], [50], [4], [47]. This work is primarily focused on the web browser, often requiring modifications to the browser's runtime engine. NR is unusual in its model, inference, and enforcement: (1) it only allows simple `RWX` permissions rather than Turing-complete functions; (2) its inference is static rather than dynamic; (3) it leverages the library-import mechanism to apply both source- and context-transformations, thus wrapping a library's full observable behavior rather than just its interface.

More recently, NodeSentry [51] proposed powerful policies mediating between libraries. Different from NR, these policies are Turing-complete, are applied manually, and focus on inter-library APIs rather that a library's full observable behavior.

Capabilities Capability systems [25], [44] and object-capability systems [16], [32] place access restrictions by restricting the ability to *name* a resource, essentially treating the object reference graph as an access graph. To make capabilities benefits more widespread, efforts such as Joe-E for Java [30] and Caja for JavaScript [32] restrict popular languages to object-capability-safe subsets. Similar to capability systems, NR interferes with the program's ability to name a resource—but rather than disabling naming, it augments it with a permission check. Moreover NR does not focus on a language subset, and its static-analysis offers significant automation.

Software De-bloating Functionality elimination [40] and, more recently, software de-bloating [19], [21], [8] lower potential vulnerabilities by eliminating unused code in a program. These techniques focus on making subvertible, dormant code inactive—thus are similar in terms of goal to NR, but approach the problem differently. Rather than eliminating code, NR makes the code inaccessible at runtime.

Ecosystem-focused Non-academic response to the challenges of third-party libraries [43], [37], [45] focuses on tools that check the program's dependency chain for known vulnerabilities. Support for locking dependencies between deployments [36] necessarily rule out security problems; on the contrary, users forego valuable bug and vulnerability fixes, while experiencing a more convoluted dependency management.

The Deno JavaScript/TypeScript runtime recently introduced a coarse-grained allow-deny permission model focusing on the file-system and the network. While Deno's recent addition is a testament that practitioners are still lacking a solution, it lacks NR's advanced automation and fine granularity.

X. Conclusion

Third-party libraries ease the development of large-scale software systems, but often exercise significantly more privilege than they need to complete their task. This additional privilege is often exploited at runtime via *dynamic compromise*, even when these libraries are not actively malicious *per se*.

NR addresses this problem by introducing a fine-grained read-write-execute (`RWX`) permission model at the boundaries of libraries. Each property of an imported library is governed by a set of permissions, which developers can express and inspect when importing libraries. To enforce these permissions during program execution, NR transforms libraries and their context to inline runtime checks. As permissions can overwhelm developers, NR provides a permission inference engine that generates default permissions by statically analyzing how libraries are used by their consumers.

A prototype for JavaScript demonstrates that the `RWX` permission model combines simplicity with power: it is simple enough to be well-understood by developers and inferred by static analysis, expressive enough to protect real libraries from practical threats, and enables a novel quantification of privilege reduction by identifying the ratio of disallowed interfaces.

Acknowledgments

We would like to thank J urgen Cito, Petros Efstathopoulos, Daniel Kats, Isaac Z. Schlueter, and CJ Silverio. This research

was funded in part by National Science Foundation grant CNS-1513687 and DARPA contract HR00112020013. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF or DARPA.

References

[1] A. Abraham. (2017) Snyk: Arbitrary code execution in node-serialize. https://snyk.io/vuln/npm:node-serialize:20170208. Accessed: 2020-03-19. [Online]. Available: https://snyk.io/vuln/npm:node-serialize:20170208

[2] ——. (2017) Snyk: Arbitrary code execution in serialize-to-js. https://snyk.io/vuln/npm:serialize-to-js:20170208. Accessed: 2020-03-19. [Online]. Available: https://snyk.io/vuln/npm:serialize-to-js:20170208

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

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

[5] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, "Deconstructing process isolation," in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, ser. MSPC '06. New York, NY, USA: ACM, 2006, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1178597.1178599

[6] U. Author. (2020) Snyk: Arbitrary code injection in serialize-javascript. https://snyk.io/vuln/SNYK-JS-SERIALIZEJAVASCRIPT-570062. Accessed: 2020-03-19. [Online]. Available: https://snyk.io/vuln/SNYK-JS-SERIALIZEJAVASCRIPT-570062

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

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

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

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

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

[12] J. N. Buxton and B. Randell, *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[13] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 516–519.

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

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

[16] S. Drossopoulou and J. Noble, "The need for capability policies," in *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, ser. FTfJP '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:7. [Online]. Available: http://doi.acm.org/10.1145/2489804.2489811

[17] I. Google. (2009) Closure. https://developers.google.com/closure/. Accessed: 2019-06-11. [Online]. Available: https://developers.google.com/closure/

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

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

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

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

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

[23] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," 2017.

[24] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 723–735. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831189

[25] H. M. Levy, *Capability Based Computer Systems*. Digital Press, 1984. [Online]. Available: http://www.cs.washington.edu/homes/levy/capabook/

[26] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative privilege separation," 2019.

[27] S. J. Long, "Owasp dependency check," 2015.

[28] M. Maass, "A theory and tools for applying sandboxes effectively," Ph.D. dissertation, Carnegie Mellon University, 2016.

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

[30] A. Mettler, D. Wagner, and T. Close, "Joe-e: A security-oriented subset of java." in *Networked and Distributed Systems Security*, ser. NDSS'10, vol. 10, 2010, pp. 357–374.

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

[32] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja: Safe active content in sanitized javascript, 2008," *Google white paper*, 2009.

[33] C. MITRE, "Common vulnerability enumeration," 2006, accessed: 2018-11-18. [Online]. Available: http://cbe.mitre.org

[34] ——, "Common weakness enumeration," 2006, accessed: 2018-11-18. [Online]. Available: http://cwe.mitre.org

[35] M. D. Network. (2020) with statement. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with. Accessed: 2020-03-19. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with

[36] npm, Inc. (2012) npm-shrinkwrap: Lock down dependency versions. https://docs.npmjs.com/cli/shrinkwrap. [Online]. Available: https://docs.npmjs.com/cli/shrinkwrap

[37] E. Oftedal *et al.* (2016) Retirejs. [Online]. Available: http://retirejs.github.io/retire.js/

[38] O. W. A. S. Project. (2018) Owasp top ten project'17. https://www.owasp.org/index.php/Top_10-2017_Top_10. Accessed: 2018-09-27. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10

[39] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA:

USENIX Association, 2003, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251369

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

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

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

[43] N. Security. (2016) Continuous security monitoring for your node apps. [Online]. Available: https://nodesecurity.io/

[44] J. S. Shapiro, J. M. Smith, and D. J. Farber, *EROS: a fast capability system*. ACM, 1999, vol. 33, no. 5.

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

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

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

[48] M. Stepankin. (2016) [demo.paypal.com] node.js code injection (rce). http://artsploit.blogspot.com/2016/08/pprce2.html. Accessed: 2018-10-05. [Online]. Available: http://artsploit.blogspot.com/2016/08/pprce2.html

[49] ——. (2016) Snyk: Code injection in dustjs-linkedin. https://snyk.io/vuln/npm:dustjs-linkedin:20160819. Accessed: 2019-03-19. [Online]. Available: https://snyk.io/vuln/npm:dustjs-linkedin:20160819

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

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

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

[53] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.acm.org/10.1145/168619.168635

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

[55] N. C. Zakas and E. contributors. (2013) Eslint—pluggable javascript linter. https://eslint.org/. Accessed: 2018-07-12. [Online]. Available: https://eslint.org/

[56] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Smallworld with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 995–1010.