Extracting Taint Specifications for JavaScript Libraries

Cristian-Alexandru Staicu
TU Darmstadt
cris.staicu@gmail.com

Martin Toldam Torp
Aarhus University
torp@cs.au.dk

Max Schäfer
GitHub
max-schaefer@github.com

Anders Møller
Aarhus University
amoeller@cs.au.dk

Michael Pradel
University of Stuttgart
michael@binaervarianz.de

ABSTRACT

Modern JavaScript applications extensively depend on third-party libraries. Especially for the Node.js platform, vulnerabilities can have severe consequences to the security of applications, resulting in, e.g., cross-site scripting and command injection attacks. Existing static analysis tools that have been developed to automatically detect such issues are either too coarse-grained, looking only at package dependency structure while ignoring dataflow, or rely on manually written taint specifications for the most popular libraries to ensure analysis scalability.

In this work, we propose a technique for automatically extracting taint specifications for JavaScript libraries, based on a dynamic analysis that leverages the existing test suites of the libraries and their available clients in the npm repository. Due to the dynamic nature of JavaScript, mapping observations from dynamic analysis to taint specifications that fit into a static analysis is non-trivial. Our main insight is that this challenge can be addressed by a combination of an access path mechanism that identifies entry and exit points, and the use of membranes around the libraries of interest.

We show that our approach is effective at inferring useful taint specifications at scale. Our prototype tool automatically extracts 146 additional taint sinks and 7 840 propagation summaries spanning 1 393 npm modules. By integrating the extracted specifications into a commercial, state-of-the-art static analysis, 136 new alerts are produced, many of which correspond to likely security vulnerabilities. Moreover, many important specifications that were originally manually written are among the ones that our tool can now extract automatically.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

taint analysis, static analysis, dynamic analysis

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380390

ACM Reference Format:
Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting Taint Specifications for JavaScript Libraries. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380390

1 INTRODUCTION

JavaScript is powering a wide variety of web applications, both client-side and server-side. Many of these applications are security-critical, such as PayPal, Netflix, or Uber, which handle massive amounts of privacy-sensitive user data and other assets. An important characteristic of modern JavaScript-based applications is the extensive use of third-party libraries. On the npm platform more than 1 million packages (mostly libraries) are available,[1] and only a few of them have been screened intensively for security vulnerabilities. A challenge when analyzing the security of npm packages is that they are often not self-contained, but they in turn depend on other npm packages for providing lower-level functionality. Recent work shows that, on average, every npm package depends on 79 other packages and on code published by 39 maintainers [47]. To correctly understand an application that uses npm packages, one needs to consider all these dependencies.

Two main directions are being pursued for automatically securing npm packages. First, there are tools that aggregate known security vulnerabilities in specific versions of individual libraries and report them to the developer directly. For example, `npm audit` analyzes all the dependencies of a Node.js application and warns the developer about any known vulnerabilities in the dependent-upon code. GitHub, Snyk, and other companies offer similar services, and related work [27] advertises such security controls. The main limitation of this approach is the high number of false positives. Often the critical part of the library is not used by the application, or it is used in a way that is completely harmless. For example, an application may use an npm module vulnerable to command injection attacks, but it passes only string constants provided by the developer as input to this module. We believe it is important to make the distinction between merely relying on a library that contains a potential known vulnerability and using that library in an insecure way. Another problem with these tools is that libraries that use insecure features of the JavaScript language or of the Node.js framework are often not registered as having "known vulnerabilities" if their documentation indicates that these features are being used internally. An example of such a library is the package `jsonfile` that provides functionality for easily accessing JSON

[1] http://www.modulecounts.com/

files. Even though such a library is not considered vulnerable by itself, it may be used in an insecure manner, e.g., by propagating attacker-controlled data into file system paths.

A more precise approach for securing JavaScript applications, pursued both by academia and by industry practitioners, is static program analysis. In taint analysis, which is a kind of program analysis that can in principle detect most common forms of security issues, security properties are expressed as direct information flows from *sources* to *sinks*: either from untrusted sources to sensitive sinks (integrity) or conversely from sensitive sources to untrusted sinks (confidentiality). We focus on integrity because it covers the vast majority of security vulnerabilities reported by the community,[2] and we ignore indirect flows, also called implicit flows, because they have been shown to appear seldom in real-world npm vulnerabilities [43].

Modularity is the key to scalable static analysis. For example, GitHub's LGTM[3] platform includes a state-of-the-art taint analysis for JavaScript (and other languages), which achieves high scalability by analyzing modularly. When analyzing one module of an application, other modules are either ignored or treated according to manually written specifications that describe essential taint flows where available. Ignoring modules leads to inaccurate analysis results, while manually constructing specifications is a demanding and error-prone task, so only a limited number of npm modules are considered. An important question hence is how to obtain specifications of modules in an automated way.

Inspired by Modelgen for Android [9], we present a technique that dynamically infers explicit taint flow summaries for npm modules, to be utilized in a static analysis, such as LGTM. Besides being designed for JavaScript, our technique is more general than Modelgen, allowing for complex summaries to be extracted. For example, we are the first to support summaries involving callback arguments and instantiated exported classes. Moreover, our technique considers the large amount of transitive dependencies in npm and thus allows the extraction of summaries for multiple npm packages in the same execution.

Another source of inspiration is the NoRegrets tool [29] that leverages the vast number of open source packages available in the npm repository to obtain information about how the most important libraries are being used. Many of those packages have test suites, and by running the test suite of a package we can gain information about the taint flows in all the packages it depends on, both directly and transitively.

A central technical challenge for adapting the Modelgen idea to our setting is that JavaScript is a highly dynamic language, which makes it non-trivial to map observations from a dynamic analysis to taint specifications that fit into a static analysis. To this end, we adopt the notion of dynamic access paths from NoRegrets, allowing us to identify entry and exit points of taint flow in the libraries. Our dynamic analysis uses a variant of membranes [11, 17, 25, 30] for tracking the taint flow between libraries and clients. It identifies flows between entry and exit points (propagations), between entry points and existing sinks (additional sinks) and between existing sources and exit points (additional sources). Finally, we propose

[2] https://www.npmjs.com/advisories
[3] https://lgtm.com

```
1  let userInput = {
2          tempDir: "./path/to/dir",
3          cacheDir: "./path/to/cache"
4  }
5  const _ = require("lodash");
6  const rimraf = require("rimraf")
7
8  let obj = _.forIn(userInput, function(value) {
9   rimraf(value, function(err) {
10       if (err)
11           console.log(err)
12   })
13  })
```

Figure 1: A typical example of JavaScript code that uses npm modules. With dotted/blue we mark exit points from the client code and with solid/orange the entry points from the library code.

deploying one membrane per npm module and hence extracting summaries for multiple modules at once.

We show that our approach is highly scalable by successfully running our dynamic analysis on 15 892 clients of 751 packages. The dynamic analysis is efficient, spending, on average, only 112 seconds per successfully analyzed client or 302 seconds per inferred specification. In total, it extracts 146 additional taint sinks and 7 840 propagation summaries spanning 1 393 modules. 35% of the summaries contain complex taint flows, such as between an argument of an exported method and a parameter passed by the library to a callback. The evaluation also shows that the extracted summaries can improve static analyses by enabling it to reveal otherwise missed vulnerabilities: 136 new alerts are produced, many of which correspond to likely vulnerabilities.

In summary, our contributions are:

- We present a novel, highly-scalable specification extraction technique for JavaScript libraries that builds on a dynamic taint analysis and leverages existing test suites.
- We report our results from an extensive experimental evaluation of the approach. The results show that the dynamic analysis is able to infer non-trivial and accurate taint flow models in widely used npm modules.
- We demonstrate that the inferred taint specifications can be integrated into an existing static analysis tool, thereby enabling discovery of previously unknown security vulnerabilities.

2 MOTIVATING EXAMPLE

Let us consider the example in Figure 1. This code fragment uses two of the most popular npm packages: `lodash`, a general-purpose utility library, and `rimraf`, a simple library for recursively deleting directories on the disk. In the presented example, the `forIn` method from `lodash` is used to iterate through the values of each property on the user input object. Each of these values is then passed to the `rimraf` module.

A human or an automated tool that aims at analyzing the code fragment in Figure 1 must first understand the essential semantics of the two modules. For example, one needs to understand that if some user input is passed to `rimraf` without sanitization, then it exposes a directory traversal vulnerability. However, this style of code can hinder understandability, both for unexperienced users

and for static analysis tools. Specifically, it may not be clear that by invoking the `forIn` method with two parameters – an object to be traversed and a callback function – the second parameter will be invoked with the property values of the first parameter as arguments.

One way to address this problem is to analyze the library code together with the client code using a whole-program dataflow analyzer. However, that approach suffers from serious scalability issues. For example, the implementation of the apparently trivial `forIn` method spans across 32 files. In Figure 2 we show a subset of the code that needs to be analyzed. Statically analyzing such a large amount of highly dynamic code is extremely expensive and tends to give prohibitively imprecise results [2].

When trying to analyze the source code of the `rimraf` module, one is faced with even greater challenges, as illustrated by Figure 3. To reveal the directory traversal problem discussed earlier, one needs to show that there is an unsanitized flow from the first parameter of the `rimraf` function to one of the file system access methods, e.g., `fs.rmdir`. However, as shown by the example, the call to this method is dispatched using dynamically attached methods on the `options` object. Once again, to the best of our knowledge, existing static analysis tools for JavaScript are unable to successfully analyze such highly-dynamic code at scale and with a precision that is practically useful.

Modular analysis, as exemplified by LGTM, addresses this challenge by analyzing each package in isolation. If a package depends on other packages, those are either ignored or modeled using manually-written specifications that capture the essential dataflows. Relying on simple generic specifications, e.g., saying that whenever a parameter is tainted then so is the return value, would be too imprecise for this example and lead to the static analysis missing important flows. Since creating useful specifications manually is difficult and not scalable, efficient automated alternatives are needed.

Our approach leverages the information in the npm repository about packages and their dependencies, together with the package source code available on GitHub. For this specific example, both `lodash` and `rimraf` have numerous open-source clients, many with test suites. By dynamically analyzing the executions of those test suites, we can automatically learn useful specifications.

```
14 /* In the file forIn.js */
15 var baseFor = require('./_baseFor'),
16     castFunction = require('./_castFunction'),
17     keysIn = require('./keysIn');
18 function forIn(object, iteratee) {
19   return object == null
20     ? object
21     : baseFor(object, castFunction(iteratee), keysIn);
22 }
23 module.exports = forIn;
24 /* In the file _baseFor.js */
25 var createBaseFor = require('./_createBaseFor');
26 var baseFor = createBaseFor();
27 module.exports = baseFor;
28 /* In the file _createBaseFor.js */
29 function createBaseFor(fromRight) {
30   return function(object, iteratee, keysFunc) {
31     var index = -1,
32         iterable = Object(object),
33         props = keysFunc(object),
34         length = props.length;
35
36     while (length--) {
37       var key = props[fromRight ? length : ++index];
38       if (iteratee(iterable[key], key, iterable) === false) {
39         break;
40       }
41     }
42     return object;
43   };
44 }
45 module.exports = createBaseFor;
46 /* ... skipped the other transitive dependencies ... */
```

Figure 2: The implementation of `lodash`'s `forIn` method. For space reasons, only two of the 31 dependent files are shown. With dotted/blue we mark entry points to the library code and with solid/orange the exit points from the library code.

```
47 var fs = require("fs")
48 function defaults (options) {
49   var methods = [
50     'unlink', 'chmod', 'stat', 'lstat', 'rmdir', 'readdir'
51   ]
52   methods.forEach(function(m) {
53     options[m] = options[m] || fs[m]
54     m = m + 'Sync'
55     options[m] = options[m] || fs[m]
56   })
57 }
58 function rmdir (p, options, originalEr, cb) {
59   defaults(options)
60   options.rmdir(p, function (er) {
61     cb(er)
62   });
63 }
64 module.exports = function rimraf(p, options, cb) {
65   options.lstat(p, function (er, st) {
66     return rmdir(p, options, er, cb)
67   });
68 }
```

Figure 3: Simplified source code for the `rimraf` module.

3 TAINT SPECIFICATIONS FOR MODULES

The specifications we are interested in summarize the taint-relevant information for entry and exit points of JavaScript libraries. For example, one can specify that the information from entry point A may flow into exit point B or that values passed to an entry point eventually reach a potentially dangerous operation.

The careful reader may have observed that there is a duality between the exit points of the client code, e.g., in Figure 1, and the entry points of the library, e.g., in Figure 2. For example, the `userInput` argument in line 8 corresponds to the `object` parameter in line 18. We will refer to both an entry point and its corresponding exit point by using the term *contact point*. We also introduce an access path mechanism to uniquely identify each contact point.

The specifications described in the remainder of this section can in principle be produced in multiple ways: either manually or by using a static or dynamic analysis. Section 4 presents an automatic inference process based on dynamic analysis.

3.1 Specifying Contact Points

Inspired by previous work to detect breaking changes in npm package updates [29, 31], we propose using an access path mechanism

for specifying contact points. An access path, or short *ap*, can be described as an S-expression, which is read from the innermost expression outwards. Each type of symbol corresponds to an operation in the JavaScript language.

```
ap ::= (root <uri>)
     | (member <name> <ap>)
     | (parameter <i> <ap>)
     | (return <ap>)
     | (instance <ap>)
```

The innermost subexpression of a path always contains a `root` symbol, which holds an URI that refers to the module. For space reasons, we use package names instead of package URIs. For example, `(root dotenv)` refers to the module that is loaded when calling `require('dotenv')`. The other symbols are `member` to refer to properties of objects, `parameter` that refers to the *i*-th parameter of a function, `return` that refers to the return value of a call, and `instance` that refers to constructed values. For example, the path `(parameter 0 (member forIn (root lodash)))` represents both the exit point in line 8 of Figure 1 and the first entry point in line 18 of Figure 2. In the remainder of this section we show how access paths can express different kinds of taint specifications.

We assume that a collection of so-called known sources and sinks is provided. For example, values obtained from network communication via the Node.js standard library are commonly treated as sources, and arguments to `exec` and `eval` are sinks.

We are interested in three kinds of specifications: *additional sinks* when we observe a flow from an entry point to a known sink, *additional sources* when there is flow from a known source to an exit point, and *propagation summaries* when there is a flow from an entry point to an exit point. We will now proceed to describe each of them in detail.

3.2 Propagation Summaries

The propagation summaries, or propagations for short, specify how taint may flow in and out of a library's functions. For example, a propagation summary can specify that if a tainted value enters the library as a specific argument to a function, then specific exit points of the library, e.g., properties on the return value, should also be considered tainted. Having such information available allows program analyses to reason about the potential taint flows without needing to reanalyze the source code of the library for every client.

The most basic form of flow is from an argument of a function to its return value, either because the argument is returned directly, or because the argument is used in the computation of the return value. Other more complicated forms of flow may also occur. For example, if an argument is written to some internal state of the library, and this state is then returned from another function, then we have a taint flow from the argument of one function to the return value of another function, which can also be captured as a propagation summary.

A propagation summary consists of two access paths: one that represents the point in the library API where the tainted value enters, and one that represents the point where the tainted value exits. Consider Example 1 where a function `f` has a parameter `x` and returns an object that has a property `p` with a value obtained from the `p` property of `x`.

Example 1

```
69 //module m
70 function f(x) {
71   return { p : x.a };
72 }
73 module.exports.f = f;
```

Taint Specification

```
(member a (parameter 0 (member f (root m))))
                    ↓
(member p (return (member f (root m))))
```

The interesting taint flow for this code is modeled by the taint specification shown next to the example, which indicates that taint flows from `x.a` to the `p` property of `f`'s return value.

This way of expressing taint flows is sometimes inconvenient. For example, a common JavaScript pattern is to iterate through all the properties of an object, which means that the accessed property names differ from client to client. An example of this reflective pattern is seen in Example 2. With the current notion of propagation summaries, we can only express flows involving specific properties, but in this case the relevant property names depend on the clients. For this purpose we introduce a wildcard notation for referring to every property of an object: `(member * <ap>)`. For example, one may refer to all the properties of the `obj` parameter in the program example with `(member * (parameter 0 (root sum))` as shown in the taint specification of Example 2.

Example 2

```
74 //module sum
75 function f(obj) {
76   let sum = 0;
77   for (prop in obj) {
78         sum += obj[prop];
79   }
80   return sum;
81 }
82 module.exports.f = f;
```

Taint Specification

```
(member * (parameter 0 (member f (root sum))))
                    ↓
(return (member f (root sum))
```

As mentioned earlier, callbacks are common contact points in npm modules. Our specifications refer to callbacks by treating a parameter as a function. The following specification summarizes the part of the `lodash` library presented in Figure 2, using a callback parameter exit point:

```
(member * (parameter 0 (member forIn (root lodash))))
                    ↓
(parameter 0 (parameter 1 (member forIn (root lodash))))
```

This propagation says that the value of every property of the object passed as the first argument of the `forIn` function may flow into the first parameter of the callback passed as the second argument.

A final propagation pattern worth discussing is one that involves contact points with return values. In Example 3, the `padder` module exports a single anonymous function in line 84. However, this function in turn creates an object with an `lpad` property pointing to an internal anonymous function. This case corresponds to the factory method design pattern from object-oriented literature. After invoking the main exported method of the module, a client obtains a reference to the internal object declared in line 85, which in turn allows the client to invoke the internal anonymous function from line 86. Thus, there are two exit points of the `padder` library in the presented example: one that returns an object with an `lpad` method, in line 89, and one corresponding to that method itself, in line 87. The latter depends on the former, because an object with the `lpad` method is only exposed to the client through the first exit point, which in turn creates more entry and exit points for the `lpad` method.

Example 3

```
83 //module padder
84 module.exports = function() {
85   let res = {};
86   res.lpad = function(s) {
87         return " " + s;
88   }
89   return res;
90 }
```

Taint Specification

```
(parameter 0 (member lpad (return (root padder))))
↓
(return (member lpad (return (root padder))))
```

3.3 Additional Sinks and Sources

If a value passed into a library reaches a known sink, we say that the entry point through which the value entered is an additional sink. Intuitively, passing the value to that contact point or to the sink itself has the same security implications for the client of the library, hence a program analysis can treat them the same way.

Revisiting the source code of the `rimraf` library in Figure 3, we can observe that the value passed as first argument to the main library function ends up in `fs.rmdir()`, which is a known sink for directory traversal vulnerabilities. This method allows recursively removing any folder on the disk, hence if an attacker can control the value passed into it, she can cause serious harm on the system. Therefore, it makes sense to specify the contact point `(parameter 0 (root rimraf))` as an additional sink.

Conversely, if inside the library a tainted value is created which then escapes into the client code through an exit point, we say that the exit point is an additional source. Example 4 shows a simple module that performs a TCP request and invokes a callback whenever data is received from the target server. This data should be considered tainted since it comes from untrusted third-party computers, so it is reasonable to specify the contact point `(parameter 0 (parameter 2 (root my-tcp))` as an additional source.

Example 4

```
91 //module my-tcp
92 module.exports = function (host, port, cb) {
93   const net = require('net');
94   const client = new net.Socket();
95   client.connect(port, host, function() {});
96   client.on('data', function(data) {
97     cb(data);
98   });
99 }
```

Even though our dynamic analysis presented in Section 4 can in theory extract all the three kinds of specifications presented so far, our prototype implementation introduced in Section 6 only supports the extraction of propagations and additional sinks. The main reason for omitting extraction of additional sources is that existing security vulnerability reports for npm packages often involve additional sinks, for example CVE-2017-1000219 or CVE-2018-3772, but vulnerabilities caused by additional sources are less common.

4 INFERRING TAINT SPECIFICATIONS VIA DYNAMIC ANALYSIS

We now present a technique for dynamically inferring taint specifications, i.e., propagation summaries and additional sinks, of the form described in Section 3. The goal is to find relations between entry points and exit points, between entry points and existing sinks, and between existing sources and exit points.

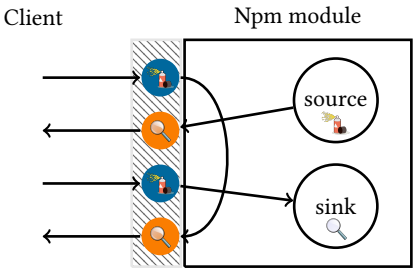

Figure 4: Inferring specifications for a single module: taint values at entry points and sources, and then check for taints at sinks and exit points. The arrows show information flows and the shaded gray area represents the membrane.

Figure 4 illustrates how our technique works for a single npm module. The arrows represent information flow, possibly spanning multiple methods and modules. When the test suites are executed, values are intercepted at entry points and tainted with a unique identifier per entry point. The taint inside the module is then propagated using a dynamic taint analysis. Whenever a tainted value reaches a sink or an exit point, an additional sink or a propagation summary, respectively, is generated. Similarly, if a value that is tainted by an internal source is observed at an exit point, an additional source is generated for that exit point. All taints are removed at exit points, so we only infer specifications for the library code and not for the client code.

Previous work [9] considers arguments of methods in the public API as entry points and return values as exit points, but as the motivating example shows, this is insufficient for many npm modules. JavaScript libraries interact with their clients in complex ways, e.g., through callbacks like the ones in Figure 1, or by allowing plugins to be configured inside the library. Therefore, it is non-trivial to determine where the library code starts and where the client code ends. One way to refer to this point of contact between components and thus to generalize the idea of entry and exit points is by using the concept of *membranes* [11, 30].

4.1 Membrane-Based Analysis

The main idea of a membrane is to interpose analysis behavior on every interaction between the client and the library. Moreover, every reference that passes through the membrane becomes part of it. Existing work describes how to implement membranes and how to use them for implementing generic policies such as "the library should never use the native module `fs`". However, to be useful in our setting, we need a way to distinguish between entry and exit points of the library and to uniquely refer to every such point in the membrane.

To rigorously define membranes, we first introduce a way of intercepting operations on a given value. To this end, we rely on proxies, a concept introduced in ECMAScript 6. A proxy $P(v)$ for a value v is a wrapper object that attaches traps to the wrapped value. Every operation applied to the proxy results in an invocation on the corresponding trap. For example, property reads, property writes, function applications, and constructor applications all result in their

Table 1: Creation of contact points inside the membrane and the corresponding taint operations executed before and after the proxied operation. The direction indicates whether the proxy corresponds to an entry or an exit point. The taint(*v, ap*) action associates a taint corresponding to the access path *ap* to runtime value *v*. The checkTaint(*v, ap*) action recursively searches for tainted values in *v*, where the taint has the same root package as the access path *ap*. Finally, untaint(*v, ap*) recursively declassifies all the values in *v* that have a taint with the same root as the access path *ap*.

Operation	Existing contact point		New contact point(s)		Pre action	Post action
	Access path	Direction	Access path	Direction		
`require("foo");`	-	-	`ap = (root foo)`	ENTRY	-	-
`x.prop`	ap_x	ENTRY	`ap = (member prop` ap_x`)`	ENTRY	-	-
	ap_x	EXIT	`ap = (member prop` ap_x`)`	EXIT	-	taint(x.prop, ap)
`res = x(arg)`	ap_x	ENTRY	ap_{par}`=(parameter <i>` ap_x`)`	EXIT	taint(arg, ap_{par})	checkTaint(res, ap_x)
			ap_{ret}`=(return` ap_x`)`	ENTRY		untaint(res, ap_x)
	ap_x	EXIT	ap_{par}`=(parameter <i>` ap_x`)`	ENTRY	checkTaint(arg, ap_x)	taint(res, ap_{ret})
			ap_{ret}`=(return` ap_x`)`	EXIT	untaint(arg, ap_x)	
`res = new x(arg)`	ap_x	ENTRY	ap_{par}`=(parameter <i>` ap_x`)`	EXIT	taint(arg, ap_{par})	checkTaint(res, ap_x)
			ap_{ret}`=(instance` ap_x`)`	ENTRY		untaint(res, ap_x)
	ap_x	EXIT	ap_{par}`=(parameter <i>` ap_x`)`	ENTRY	checkTaint(arg, ap_x)	taint(res, ap_{ret})
			ap_{ret}`=(instance` ap_x`)`	EXIT	untaint(arg, ap_x)	

corresponding traps firing. The traps can modify the behavior of the operation or just perform observing operations, such as logging.

A proxy can therefore observe operations applied to the wrapped value, and even decide to modify these operations. Our analysis uses proxies to perform taint-relevant operations before and after the proxied operation is executed. We also need a way to associate a unique address, i.e., an access path, to each proxy and to specify whether the proxy corresponds to an exit or an entry point:

Definition 1. *A contact point, denoted $\langle v, ap, d\rangle$, is a tuple consisting of a proxy $P(v)$ around a value v, an access path ap that uniquely identifies the contact point, and a direction flag d that specifies whether the contact point is an entry or an exit point.*

For simplicity, we abuse the notation for a contact point $\langle v, ap, d\rangle$ by using $\langle v\rangle$ whenever the access path and the direction are not relevant for the description. One can specify entry and exit points for a library by introducing proxies around exported API methods in the library source code. The challenge lies in automatically identifying *all* the values that need to be proxied for intercepting all the interactions between two npm modules. Membranes provide an elegant solution to this problem:

Definition 2. *A membrane $\mathcal{M}$ is a set of contact points interposed between a library ℓ and its clients. $\mathcal{M}$ is initialized with $\{\langle v_\ell, (\texttt{root}\ \ell), \text{ENTRY}\rangle\}$, i.e., the contact point that wraps the main value v_ℓ exported by the library. For every value v that is passed into or returned by an existing contact point in $\mathcal{M}$, a new contact point $\langle v, ap_v, d'\rangle$ is added to the membrane, i.e., $\mathcal{M} := \mathcal{M} \cup \{\langle v, ap_v, d'\rangle\}$.*

The new access point ap_v is derived from the existing ap by picking the grammar rule from Section 3.1 that corresponds to the JavaScript operation v passing through the exit point, e.g., a property access or a parameter to a function call. Similarly, the direction of the new contact point d' is derived from the direction of the original contact point d by using the following observation: the direction changes for all the values that are passed as arguments to a method in the membrane. Let us consider a function object that is passed into an entry point of a library as an argument. Once it reaches the other side of the membrane, i.e., in the library code, it should be considered as an exit point for the library. In Table 1 we summarize all the possible operations on a contact point and how to derive the access paths and direction flags for the new contact points. We also show the auxiliary operations necessary for tracking tainted values in the pre and post action columns. Note that both arguments and return values can be entry or exit points, depending on the direction flag.

To illustrate how contact points are created, consider the membrane between `lodash` and its client in Figure 1. The first contact point of the membrane is created when the library is required in line 5, i.e., $\mathcal{M} := \{\langle \_\rangle\}$. When the `forIn` property is accessed in line 8 a new contact point is added to the membrane, $\mathcal{M} := \mathcal{M} \cup \{\langle \_.\texttt{forIn}\rangle\}$. When the accessed property is invoked in the same line, three contact points are created, i.e., $\mathcal{M} := \mathcal{M} \cup \{\langle \texttt{userInput}\rangle, \langle \texttt{function...}\rangle, \langle \texttt{obj}\rangle\}$. Finally, when the callback is invoked, three more contact points are created, one for each parameter. The access paths for each of these contact points are shown in Figure 5; they correspond to a derivation tree of the grammar in Section 3.1. To obtain the access path of a given contact point, one should traverse the tree from the root and replace all the $\diamond$ symbols with the access path of the parent node. For example, the access path of $\langle \texttt{first}\rangle$ is:

```
(parameter 0 (parameter 1 (member forIn (root lodash))))
```

The dynamic taint analysis we use for propagating taint inside analyzed modules is fairly standard, with few idiosyncrasies. As noted earlier, we implement the taint-relevant operations described in the last column of Table 1 inside each module's membrane. These operations are in fact additional sources and sinks from the taint analysis' perspective since they either attach taint or check/remove taint. Once a property `p` is accessed on a value having a taint `t`, instead of directly propagating the taint, we create a new tainted value `(member p t)`. If the property `p` itself is also tainted then we propagate the taint `(member * t)`. The intuition is that the tainted property comes from outside the module or from iterating through a tainted object, hence it should be considered as a generic access.

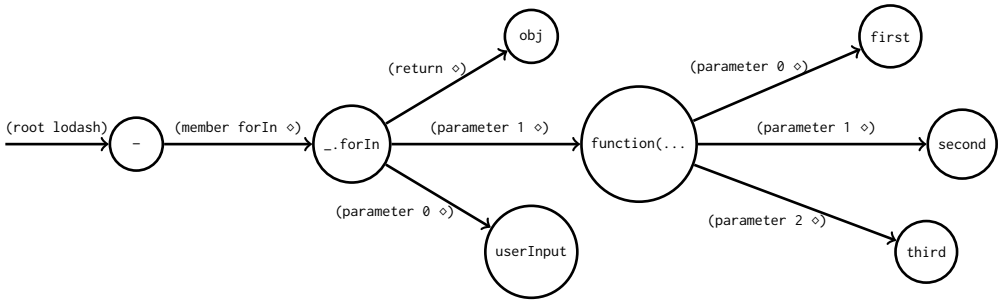

Figure 5: Contact points in the membrane between `lodash` and the client code in Figure 1.

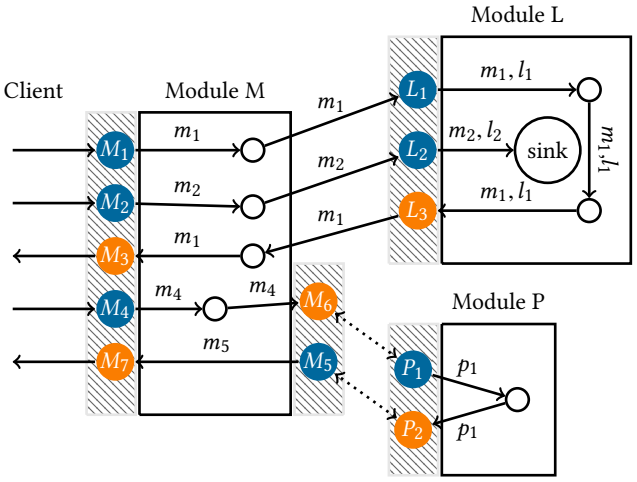

Figure 6: Inferring specifications for multiple modules at once: every entry point adds a unique taint and every corresponding exit point declassifies it. The semantics of shapes and colors are the same as in Figure 4. The dotted arrows depict equivalences between contact points.

4.2 Multi-Module Analysis

Since an npm module can in turn use other npm modules, we propose deploying a membrane around each module to maximize the number of extracted specifications. We present this setup in Figure 6 in which module M interacts with two other modules: a direct dependency L and a plugin P. For now let us consider the relation between module M and its dependency L. Every entry point attaches a taint that uniquely identifies that entry point to each value that passes through it, e.g., entry point M_1 sets taint m_1. When a value passes through an exit point of a module, the analysis removes all the taints corresponding to that particular module. As a result, tainted values for module M can only live inside M or inside M's transitive dependencies, such as L. This behavior can be observed when following the information flow between entry point M_1 and exit point M_3. The taint m_1 is carried by the value all the way through module L until the exit point M_3. Our analysis infers two propagation specifications: $M_1 \rightarrow M_3$ and $L_1 \rightarrow L_3$.

Similarly, the value that enters through M_2 inside module M gets attached the taint m_2, and further enters through L_2 inside module L, where it gets attached taint l_2. Thus, when the value finally reaches the sink inside module L, it has two taints, m_2 and l_2. The analysis generates two additional sinks, for M_2 and for L_2, since from the client's perspective a value may flow from one of these entry points into an existing sink.

4.3 Handling Plugins

One may wonder whether or not a module's dependencies should be considered as part of the module's code as described in the previous section. We propose distinguishing between two types of dependencies: direct dependencies and plugins. A direct dependency is one that is required verbatim by the developer in the source code of the module, and a plugin is a dependency that is injected by the client code. For registering a plugin, a client needs to pass a reference to the plugin through the membrane. An example of this pattern can be observed in Example 5 that shows the popular `express` framework instantiated with the plugin `body-parser`. The client code loads the `body-parser` plugin and passes it to the `express` module through the `use` method that is part of `express`'s membrane.

Example 5

```
100 const express = require('express');
101 const bodyParser = require('body-parser');
102 const app = express();
103 app.use(bodyParser.json())
```

Treating plugins differently when extracting specifications is extremely important because we do not want to infer specifications that only apply when a certain plugin is loaded. Instead, we want the specifications to be as widely applicable as possible. Therefore, direct dependencies that are loaded inside the module are considered part of the code base of the module, while plugins are not.

Consider the relation between module M and its plugin P in Figure 6. When the value carrying the taint m_4 reaches the membrane that separates M from P the taint is removed; we say that the value is declassified. Overall, for the flow between M_4 and M_7 that passes through plugin P, our analysis infers three specifications: $M_4 \rightarrow M_6$, $M_5 \rightarrow M_7$, and $P_1 \rightarrow P_2$.

5 USING TAINT SPECIFICATIONS

The main use case of the extracted taint specifications is for improving existing program analyses. Most importantly, taint specifications can be consumed by static analyses. The benefits of hybrid analyses, i.e., static plus dynamic, are thoroughly explored in the literature. Typically, a static analysis uses the results from a dynamic analysis, either to get a more precise result or to get coverage of code that is otherwise difficult to analyze statically. As mentioned in the introduction, industrial static analyses sometimes do not even try to analyze Node.js modules, but instead rely on manually written taint specifications or coarse-grained assumptions about taint flow in modules. However, such specifications are both error-prone and hard to maintain. In contrast, our specification generation analysis is fully automatic, and can therefore easily be re-run whenever modules are updated. Moreover, we show that there is a significant overlap between the specifications our analysis generates and existing manually-written models used by the commercial LGTM taint analysis, demonstrating that our analysis can infer precise module specifications that resemble and improve upon hand-written specifications.

Another use case for the extracted specifications is to serve as a form of documentation for the module they were extracted from. Effectively, they can act as a contract between module developers and module users that specifies, for example, who is responsible for sanitizing end user input. We observe that many security vulnerabilities reported by the community or by researchers [42] are actually additional sinks. For example, a typical vulnerability occurs when a user-supplied value is involved in constructing some string that is then executed by the `eval` function. In some unfortunate situations attackers can compose the user-supplied value in ways that enables executing malicious code. To warn users of modules about potential vulnerabilities, inferred specifications could be shown to developers. For example, an additional sink could inform the client that an argument passed to a specific method should be sanitized to prevent malicious code injection attacks.

Finally, we propose using the generated taint specifications for regression analysis. When a previously unobserved taint specification is suddenly generated for a new version of a library, e.g., a new additional sink appears, both the developer of the library and its clients should be alerted. Essentially, a change in a taint specification should be treated as a change to the API. Automatically inferred specifications could help automate this kind of regression analysis.

6 EVALUATION

Implementation. We implement our specification extraction technique in a tool called Taser[4]. The dynamic analysis component is built on top of NodeProf [44], an instrumentation framework for Node.js. As a starting point for finding additional sinks, we mark 40 methods of the built-in JavaScript APIs as sinks. These methods cover five well-known security issues: command injection, code injection, directory traversal, regular expression injection, and NoSQL injection. We implement limited support for sanitizers by declassifying any information flow that passes through a function and an npm module whose name or dynamic access path contains specific strings, e.g., "escape" or "sanitize".

Benchmarks. We apply Taser to 751 npm packages, all from the top-1000 most depended upon packages. Because some packages contain multiple modules and because Taser performs a multi-module analysis we analyze a total of 2 300 modules. For each analyzed npm package, we consider the 200 highest rated clients, according to npm stars, and execute their test suites to analyze the execution with Taser. We stop a test suite after a timeout of 10 minutes. If available, we also use the test suite of the npm package itself for driving the dynamic analysis. Ignoring some clients that we currently cannot analyze, e.g., due to test frameworks Taser does not support, or due to limitations of our implementation, the evaluation covers 15 892 clients, out of which 5 707 clients trigger at least one creation of a tainted value.

Research questions. Our evaluation focuses on the following research questions:

- **RQ1** How many taint specifications does Taser extract?
- **RQ2** How efficient is the analysis?
- **RQ3** Are the extracted specifications useful for statically analyzing the security of npm modules?
- **RQ4** How do the extracted specifications compare to manually created models of npm modules?

The implementation of Taser and experimental data are available at `http://brics.dk/taser/`.

RQ1: Extracted Taint Specifications

For the 2 300 analyzed modules, Taser extracts 7 840 propagation summaries and 146 additional sinks. For 457 packages, the tool extracts at least one propagation summary, and for 118 packages, it extracts at least one additional sink. The overall amount of specifications shows that manually writing taint specifications for thousands of packages is highly impractical. Instead, Taser enables extracting specifications automatically and updating them regularly with little effort.

We also check whether the specifications Taser extracts contain advanced language constructs not supported by previous work [3, 9]. To that end, we count every propagation summary that involves (i) instantiated objects, i.e., an `instance` symbol in one of its access paths, (ii) callbacks, i.e., two or more `parameter` symbols in one of its access paths, or (iii) nested API calls, i.e., two or more `return` symbols in one of its access paths. We find 595 propagation summaries with instantiated objects, 1 467 with callbacks and 1 578 with nested API calls. In total, at least 2 838 specifications, i.e., 35% of the total, could not have been extracted by those previous approaches (even if re-implemented for JavaScript).

RQ2: Efficiency of the Dynamic Analysis

Generating specifications is not something that should be done often, so having a relatively large one-time cost is acceptable in practice. However, over time new libraries are created and existing libraries are updated, so new taint specifications naturally have to be generated for those libraries. Therefore it is interesting to consider the computational cost of generating taint specifications. On average, it takes 112 seconds to run the test suite of one client

[4]It is an abbreviation of the longer *TAint Spec ExtractoR*.

Rule ID	New alerts
js/command-line-injection	2
js/file-access-to-http	64
js/path-injection	29
js/reflected-xss	5
js/regex-injection	13
js/remote-property-injection	20
js/user-controlled-bypass	2
js/xss	1
Total	136

Figure 7: Improvements to LGTM's standard analysis; rule IDs are hyperlinked to their documentation.

with the dynamic analysis enabled. This number depends on many factors, such as the size of the test suite, and how many JavaScript statements are being executed. Regenerating specifications for updated libraries and generating specifications for new libraries can be done in only a few hours per library, which we consider acceptable for specifications that can be reused repeatedly by a static analysis and other applications.

RQ3: Usefulness for Static Analysis

We evaluate the usefulness of Taser-extracted taint specifications by integrating them into LGTM, a state-of-the-art, industrial static analysis platform. A free instance hosted at `https://lgtm.com` continuously checks more than 130 000 open-source projects (including thousands of npm modules) for security problems. Without the specifications, LGTM reasons about third-party npm modules based on a limited number of manually created taint specifications. We add the extracted specifications into the static analysis and measure how many additional security alerts the analysis reports.

Figure 7 shows the improvements gained from enhancing LGTM's standard security analysis suite with the additional sinks and propagation summaries extracted by Taser. The first column lists the LGTM rule ID; for instance, `js/path-injection` flags potential directory-traversal vulnerabilities. The second column shows the number of new alerts found by incorporating our additional sinks and propagation summaries. In total, Taser enables LGTM to find 136 otherwise missed potential security problems.

To better understand the quality of the added alerts, we randomly sample 30 of the new alerts (five for rules with five or more results, and all results for the other rules). We find that 24 of them are true positives in the sense that they exhibit flow from a source to a sink.[5] Of the six false positives, five are due to imprecision of the static analysis (and hence unrelated to Taser), and one is due to a spurious additional sink extracted by Taser.

Figure 8 shows a simple example of a newly identified alert for the `js/path-injection` rule, which originates from the `FineUploader/server-examples` project from GitHub. The `req` argument contains an HTTP request object, so the LGTM security analysis considers `req.params.uuid` to be untrusted data since it might originate from a malicious attacker. After being concatenated with another string, it is passed to the `rimraf` function, which

[5]How many of these new results correspond to exploitable security vulnerabilities is a different question, which we do not consider here.

```
104 var rimraf = require('rimraf');
105 /* omitted */
106 function onDeleteFile(req, res) {
107     var uuid = req.params.uuid,
108         dirToDelete = uploadedFilesPath + uuid;
109     rimraf(dirToDelete, function(error) {
110         /* omitted */
111     });
112 }
```

Figure 8: Example of a new alert found based on a Taser-inferred specification.

(recursively) deletes the file system path denoted by this string if it exists. The value of `req.params.uuid` is not checked, so in particular it could contain "`..`" components, allowing an attacker to delete arbitrary files on the file system.

Even though the flow from source to sink is very simple, LGTM does not flag this out-of-the-box, since it does not have a model of the `rimraf` package, and its implementation is too complicated for the static analysis to model as explained above. Our additional sinks, however, identify the first parameter of `rimraf` as a taint sink for `js/path-injection`, allowing LGTM to flag this code.

As an example of the use of propagation summaries, we notice that four of the five new alerts for `js/remote-property-injection` we examined make use of the propagation summaries for `_.forEach`, a `lodash` function similar in style to `_.forIn`. These propagation summaries describe flow through a callback parameter, underscoring the importance of supporting such summaries.

RQ4: Comparison with Manually Created Specifications

The standard LGTM security analysis suite already includes manually written models of many popular npm packages, including sinks and taint propagation rules. By examining our automatically extracted taint specifications for overlap with these manually written models, we find that 12 of our additional sinks and 40 of our propagation summaries correspond to existing models. On the one hand, this confirms that the specifications we extract are practically relevant. On the other hand, it also shows that the vast majority of the Taser-extracted specifications are not yet covered by manual models.

As one example, our dynamic analysis correctly identifies the first parameter of the single function exported by the `cross-spawn` package as a sink for `js/command-line-injection`. LGTM includes a manual model for this. Additionally, Taser also identifies an analogous sink for the `win-spawn` package, a by now deprecated predecessor of `cross-spawn`. LGTM does *not* include a model for this, presumably because `win-spawn` is less popular than `cross-spawn`, and the LGTM analysis authors focused on popular packages in writing their models. Our automated approach is not limited by such considerations and can hence provide a much broader coverage.

7 DISCUSSION

In this section we present limitations of our work, and we discuss how automatically inferred taint specification can improve the current security practices in the JavaScript community.

```
var printer = require("printer");
var benignInput = "printerName";
printer.printDirect({
    data: "Test",
    printer: benignInput,
    success: function (jobID) {
        console.log("sent to printer with ID: " + jobID);
    },
    error: function (err) {
        console.log(err);
    }
});
```

Figure 9: Benign input for the vulnerability described in npm advisory number 27.

7.1 Limitations

Taser is affected by the well-known limitations of dynamic analysis, i.e., one can analyze only code that is executed. Therefore, adequate test coverage is essential for effectively extracting taint specifications. Even though in our evaluation we do not directly measure or aim to increase coverage for the used test suites, by analyzing several clients of a given library, we increase the chance of observing multiple realistic use cases of the library. Our hypothesis is that these inputs are representative for most of the library usages in the wild. Related work employs similar assumptions [29, 31].

In the current work, we do not consider implicit flows which were shown to have limited value for detecting integrity issues in server-side JavaScript [43]. However, future work should evaluate whether this assumption also holds for extracting taint specifications.

In our evaluation, we judge the usefulness of the extracted summaries by showing that they improve an existing static analysis. Similarly to the work of Clapp et al. [9], future work should perform a more extensive set of experiments in which the quality of the extracted specifications is directly evaluated, e.g., by extensively comparing with manually written specifications.

7.2 Comparison with Coarse-Grained Warnings

The current security practice in the npm community, as implemented, e.g., in the `npm audit` tool, is to warn users whenever they are relying on a module with a known vulnerability. This approach suffers from two limitations. First, it is limited to previously known and reported vulnerabilities. Second, it often causes spurious warnings, as a warning is issued for every package that depends on a vulnerable module, independently of whether the first module's use of the second module is affected by the vulnerability.

We show that our approach can help address both these limitations. First, one can use Taser to automatically find vulnerabilities, i.e., unsanitized, undocumented additional sinks. To evaluate the effectiveness of this approach, we run Taser using benign inputs for 24 vulnerable packages aggregated by related work [42]. Our approach finds additional sinks in 11 of the 24 packages. Limitations of the existing policy, i.e., missing sources, and insufficient modeling of arrays are the reasons why Taser does not find the remaining sinks.

Second, Taser-extracted specifications can help identify the problematic entry point of a vulnerable library. This can reduce the false positive rate of the `npm audit` solution by only reporting an alarm when user-controlled values can reach that entry point. While implementing a more precise replacement for `npm audit` based on Taser-extracted specifications is out of the scope of this work, we illustrate its potential effectiveness with the vulnerability in Figure 9. The example shows benign inputs passed to a module that suffers from a known vulnerability.[6] Taser infers the following additional sink for the vulnerable module:

```
(member printer (parameter 0 (member printDirect
                (root printer))))
```

Instead of alerting all users of the `printer` module, as `npm audit` would do, the extracted specification could help raise an alarm only for users that call the vulnerable entry point with a non-constant string value. Similarly to Synode [42], an improvement over the current `npm audit` tool could check whether the value passed at the entry point is statically computable, and raise an alarm only if that is not the case. As illustrated by this example, Taser can help reduce the false positives of the existing technique by only alerting developers when necessary. In addition to an `npm audit`-like tool, IDEs could also alert developers that specific entry points should be treated as sinks.

8 RELATED WORK

Specifications of libraries and frameworks. The idea of using pregenerated specifications to aid static analysis of library and framework code has been pursued previously [3, 5, 9, 21, 34]. The only other work that uses a dynamic analysis to infer taint specifications is the technique by Clapp et al. [9], which infers specifications for the Android SDK. Our work differs in multiple ways. First, we introduce the idea of membrane-based, multi-module analysis, allowing Taser to infer specifications for all modules used directly or indirectly by a client. In contrast, Clapp et al. [9] infer specifications from a client's usage of a single framework. Second, we use a fine-grained specification mechanism that can track flows at the level of individual properties and can express flows via callbacks, while their specifications are coarse-grained, i.e., only tracking flows between parameters and return values. Finally, our analysis accounts for the dynamic nature of JavaScript, e.g., using the star expression (⋆) as described in Section 3.2.

Taint analysis. Taint analysis [10] has been used for checking security properties [4, 18, 32, 45] and other analysis problems [15, 22]. In particular, there are both static [33] and dynamic [24, 28] taint analyses for JavaScript. Taint specifications inferred with a Taser-like approach could in principle be plugged into any static taint analysis that involves third-party modules. To the best of our knowledge, we are the first to present such an approach for static taint analysis for JavaScript. To facilitate the use of taint analysis for checking security properties, some work proposes to infer which functions to consider as sources, sinks, and sanitizers [7, 36]. In contrast, Taser infers specifications that summarize flows through entire third-party modules.

JavaScript security. Previous work has shown that there is a wide range of vulnerabilities in JavaScript software in general and for the Node.js platform in particular, e.g., injection vulnerabilities [42], regular expression-based denial of service vulnerabilities [12, 41],

[6] `https://www.npmjs.com/advisories/27`

and implementation issues in Node.js [6]. Many existing mitigation techniques rely on some form of dynamic enforcement [1, 13, 17, 42, 46]. Since even a small runtime overhead is often unacceptable, especially for server-side applications, our work instead aims at improving the static detection of vulnerabilities, e.g., via the LGTM analysis tool used in our evaluation. Zimmermann et al. [47] have shown that npm modules depend on many (79, on average) other npm modules, which become part of a module's attack surface. The Taser-inferred taint specifications enable a static analysis to consider such third-party modules without relying on manually created specifications or whole-program analysis.

Membranes. The membrane pattern, introduced by Miller [30], has been applied in several settings [11, 17, 25, 29–31]. The idea is to separate two object graphs, such that operations taking place on the boundary between the graphs can be captured and potentially modified. Taser uses membranes at the boundary between a module and a client, and between different modules, to capture taint flows between them.

Coarse-grained alerts. Some tools, most prominently `npm audit`[7] and Snyk[8], warn developers about known vulnerabilities in any of their dependencies. As discussed by Lauinger et al. [27], an important limitation is that such tools do not analyze how dependencies are used, and will warn even about vulnerabilities in code that a client does not use, or not use in a vulnerable way. A more precise analysis, e.g., based on specifications inferred by Taser, avoids the inevitable false positives caused by coarse-grained alerts.

JavaScript program analysis. The dynamic and reflective nature of JavaScript makes it difficult to construct sound, whole-program static analyses that scale to large real-world applications [2, 14, 23, 26, 33, 40, 41]. For that reason, much research has been devoted to constructing more pragmatic bug-detection tools [1, 8, 16, 19, 20, 24, 35, 37, 38, 43]. Some frameworks facilitate the implementation of dynamic JavaScript analyses [39, 44], including NodeProf [44] that Taser builds upon.

9 CONCLUSION

The massive use of third-party libraries in modern JavaScript web development calls for new techniques to discover security vulnerabilities. Modular static taint analysis is a powerful approach, as demonstrated by the successful commercial tool LGTM, but it critically relies on taint specifications of the libraries being used. Writing such specifications manually is demanding and error-prone, so automated solutions are needed. This work presents such a solution. It combines and adapts a number of ideas from previous work, in particular the idea of inferring information flow specifications using dynamic analysis [9], the membrane mechanism [11, 30], the use of test suites of open-source library clients, and the notion of dynamic access paths [29].

Our implementation and experiments demonstrate that this design is able to automatically detect non-trivial and accurate taint flow specifications in widely used Node.js modules, which enables an existing static analyzer, LGTM, to discover many previously unknown security vulnerabilities. We believe this approach is a promising alternative to the current coarse-grained security tools like `npm audit` that only consider the package dependency structure but completely ignore the dataflow. Our next step is to extend the implementation with support for more testing frameworks, and then apply the approach in production. Thereby we can gain experience with its use in practice and possibly refine the expressiveness of the taint specifications to further increase the ability to detect vulnerabilities in real-world JavaScript applications.

[7] `https://docs.npmjs.com/cli/audit`
[8] `https://snyk.io/`

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreements No 647544 and 851895). It was also supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] 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. *ACM Comput. Surv.* 50, 5 (2017), 66:1–66:36. `https://doi.org/10.1145/3106739`

[2] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 17–31. `https://doi.org/10.1145/2660193.2660214`

[3] Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 725–735. `https://doi.org/10.1145/2884781.2884816`

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 259–269.

[5] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 678–692. `https://doi.org/10.1145/3192366.3192383`

[6] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 559–578. `https://doi.org/10.1109/SP.2017.68`

[7] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 760–774. `https://doi.org/10.1145/3314221.3314648`

[8] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 629–643. `https://doi.org/10.1145/2810103.2813684`

[9] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: mining explicit information flow specifications from concrete executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, 129–140. `https://doi.org/10.1145/2771783.2771810`

[10] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. ACM, 196–206. `https://doi.org/10.1145/1273463.1273490`

[11] Tom Van Cutsem and Mark S. Miller. 2010. Proxies: design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*. 59–72. `https://doi.org/10.1145/1869631.1869638`

[12] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice:

an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 246–256. https://doi.org/10.1145/3236024.3236027

[13] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 343–359.

[14] 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*. 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[15] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[16] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, 94–105. https://doi.org/10.1145/2771783.2771809

[17] 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 2014, New Orleans, LA, USA, December 8-12, 2014*. 446–455. https://doi.org/10.1145/2664243.2664276

[18] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. ACM, 177–187. https://doi.org/10.1145/2001420.2001442

[19] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. 1663–1671. https://doi.org/10.1145/2554850.2554909

[20] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. 2017. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 49–70. https://doi.org/10.1007/978-3-662-54455-6_3

[21] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 710–720. https://doi.org/10.1145/2786805.2786875

[22] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 720–725. https://doi.org/10.1145/2970276.2970321

[23] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*. 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

[24] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. 2018. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* (2018).

[25] Matthias Keil and Peter Thiemann. 2015. TreatJS: Higher-Order Contracts for JavaScripts. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 28–51. https://doi.org/10.4230/LIPIcs.ECOOP.2015.28

[26] Erik Krogh Kristensen and Anders Møller. 2019. Reasonably-most-general clients for JavaScript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 83–93.

[27] 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. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.

[28] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *ACM Conference on Computer and Communications Security*. ACM, 1193–1204.

[29] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs)*, Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 7:1–7:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.7

[30] Mark Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University.

[31] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node.js libraries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 409–419. https://doi.org/10.1145/3338906.3338940

[32] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society.

[33] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. 455–465. https://doi.org/10.1145/3338906.3338933

[34] Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. 2019. Automatic Modeling of Opaque Code for JavaScript Static Analysis. In *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Vol. 11424. Springer, 43–60. https://doi.org/10.1007/978-3-030-16722-6_3

[35] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 314–324. https://doi.org/10.1109/ICSE.2015.51

[36] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society.

[37] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 165–174. https://doi.org/10.1145/2491956.2462168

[38] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test generation for higher-order functions in dynamic languages. *PACMPL* 2, OOPSLA (2018), 161:1–161:27. https://doi.org/10.1145/3276531

[39] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. ACM, 615–618. https://doi.org/10.1145/2491411.2494598

[40] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2019.8

[41] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 361–376.

[42] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

[43] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. In *Workshop on Programming Languages and Analysis for Security (PLAS)*.

[44] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient dynamic analysis for Node.js. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 196–206. https://doi.org/10.1145/3178372.3179527

[45] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. ACM, 87–97.

[46] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[47] 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 *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 995–1010.