Modular Call Graph Construction for Security Scanning of Node.js Applications

Benjamin Barslev Nielsen
Aarhus University
Denmark
barslev@cs.au.dk

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

Anders Møller
Aarhus University
Denmark
amoeller@cs.au.dk

ABSTRACT

Most of the code in typical Node.js applications comes from third-party libraries that consist of a large number of interdependent modules. Because of the dynamic features of JavaScript, it is difficult to obtain detailed information about the module dependencies, which is vital for reasoning about the potential consequences of security vulnerabilities in libraries, and for many other software development tasks. The underlying challenge is how to construct precise call graphs that capture the connectivity between functions in the modules.

In this work we present a novel approach to call graph construction for Node.js applications that is modular, taking into account the modular structure of Node.js applications, and sufficiently accurate and efficient to be practically useful. We demonstrate experimentally that the constructed call graphs are useful for security scanning, reducing the number of false positives by 81% compared to *npm audit* and with zero false negatives. Compared to *js-callgraph*, the call graph construction is significantly more accurate and efficient. The experiments also show that the analysis time is reduced substantially when reusing modular call graphs.

CCS CONCEPTS

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

KEYWORDS

static analysis, JavaScript, modularity

ACM Reference Format:
Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular Call Graph Construction for Security Scanning of Node.js Applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464836

1 INTRODUCTION

The npm package repository is the largest software repository in the world with more than one million JavaScript packages. These packages tend to depend heavily on each other: on average each package depends on more than 50 other packages when considering both direct and transitive dependencies [14, 36]. Packages are comprised of modules, which correspond to JavaScript files that are loaded individually by the module system. A typical Node.js application thus consists of hundreds or thousands of JavaScript files, with more than 90% of the code coming from third-party libraries [15].

As security vulnerabilities in libraries are frequently discovered [5, 29, 30, 34–36], to ensure maximal security of the applications it is important for the application developers to know the structure of dependencies within the applications. One of OWASP's top 10 categories of web application security risks is "Using Components with Known Vulnerabilities".[1] A study has shown that up to 40% of all npm packages depend on code with at least one publicly known vulnerability [36]. Another study has found that 12% of the available packages have a release that directly relies on a version of a package that contains a vulnerability listed in Snyk's security reports [5], and if taking transitive dependencies and more security reports into account the percentage is likely much higher. (A related study [17] shows similar numbers for JavaScript on web pages, but we here focus on the Node.js ecosystem.) This situation has motivated the development of *security scanners*, which are tools that warn developers if their programs either directly or transitively depend on a library with a known security vulnerability. Existing security scanners, such as *Dependabot*,[2] *npm audit*,[3] and *Snyk*,[4] only consider the package dependency structure that is specified in the `package.json` files, without looking at the program code. This means that they cannot tell whether the client actually uses the vulnerable part of the library, and consequently client developers are often overwhelmed with false-positive warnings. In a study of npm projects where such security scanners reported high-priority security warnings, 73% of the projects did not actually use the vulnerable parts of the libraries [34]. That study also concludes that mapping the usage of library code in client projects is difficult and that better automatic approaches are needed.

In this paper, we present an analysis that constructs call graphs for Node.js applications. A call graph has a node for each function in the application and an edge from a node F to a node G if F may call G [28]. It is well known that call graphs have many applications for a variety of development tools [6]. We demonstrate that it is possible to considerably improve the precision and usefulness of security scanning by using call graphs. For this purpose, the call graph analyzer ideally needs to be *sound*, *precise*, and *efficient* when applied to real-world applications. We do not require theoretical

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464836

[1] https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities
[2] https://dependabot.com/
[3] https://docs.npmjs.com/cli/audit
[4] https://snyk.io/

soundness guarantees, but if the constructed call graph misses many call edges that are possible in concrete executions then security issues may be overlooked. High precision is important, because if the call graph has too many edges then the technique is no better than the existing security scanners that only look at the package dependency structure. Efficiency is necessary such that the tool can be integrated into existing development processes.

Besides security scanning, other possible applications of the call graphs include change impact analysis [1, 8], which may be useful for finding out how breaking changes in library updates affect client code [4, 24]. Furthermore, precise knowledge of function-level dependencies across packages can also be useful for library developers to learn how the library features are being used, and in IDEs for code navigation, completion, and refactoring tools [6, 20].

Multiple approaches for constructing call graphs for JavaScript programs already exist (see Section 8), but none of them take advantage of the module structure of Node.js applications. The salient feature of the call graph analysis we present is its *modularity*. The analysis has two stages: First, each module is analyzed separately, resulting in a module summary. Second, the module summaries are composed for producing call graphs for collections of modules. This modular approach is an ideal match with the massive reuse of packages in the Node.js ecosystem. As a variant of the example above, assume both packages A_1 and A_2 depend on B, which in turn depends on C. Call graphs can then be built bottom-up in the package dependency graph. After creating module summaries for B and C, we can build a call graph $\mathcal{G}_{BC}$ for the collection $\{B, C\}$. Then later we can build call graphs for both $\{A_1, B, C\}$ and $\{A_2, B, C\}$ by reusing $\mathcal{G}_{BC}$ and only adding information from the module summaries for A_1 and A_2, respectively, thereby avoiding redundant work.

In summary, the main contributions of this paper are:

- We propose an analysis, Jam,[5] that constructs call graphs for JavaScript programs modularly, by first creating module summaries (Section 4) and then composing the summaries and building call graphs for collections of modules (Section 5).
- We present a proof-of-concept tool that leverages call graph construction for security scanning (Section 6).
- We demonstrate experimentally (Section 7) that on 12 Node.js applications, the call-graph-based security scanner finds the same 8 vulnerabilities as *npm audit* while reducing the number of false positives by 81% (from 26 to 5), and that the analysis time is reduced substantially when reusing modular call graphs. Moreover, compared to the state-of-the-art call graph contruction tool *js-callgraph*,[6] which is a further development of the tool by Feldthaus et al. [6], Jam achieves substantially better precision, accuracy, and analysis time.

2 MOTIVATING EXAMPLE

Consider the npm application *writex*[7] for converting markdown files into latex. For version 1.0.4 of *writex* (the most recent version as of August 2020), the *npm audit* security scanner reports that *writex* may be affected by up to 10 known vulnerabilities. They originate from 5 different security advisories, but *npm audit* reports an alarm for every occurrence of a vulnerable dependency, and some appear through several dependency chains. For example, a prototype pollution vulnerability[8] affecting *lodash* prior to version 4.17.19 is reported twice, because a vulnerable version of *lodash* is required through both *writex* → *lodash-template-stream* → *lodash* and *writex* → *gaze* → *globule* → *lodash*.

By manually examining the source code of *writex*, we find that only 1 of the 5 different advisories is a true positive: a regular expression vulnerability affecting the `minimatch(path, pattern)` function of the *minimatch* library for matching strings against glob patterns.[9] We classify an alarm as a true positive if the vulnerable library function is used by the application, disregarding whether an actual exploit is feasible. For the remaining 4 vulnerabilities (spanning 8 different alarms), the vulnerable function is not reachable from the *writex* application, and those alarms can therefore safely be ignored.

Using Jam to run a call-graph-based security scan of *writex*, only the true positive *minimatch* vulnerability is reported. Furthermore, the Jam call graph shows through which chain of function calls the vulnerable function is reachable, making it easier to determine whether the vulnerability is exploitable compared to the alarms reported by *npm audit*. For the true positive alarm in the *writex* client, the following fragment of a stack trace shows how the vulnerable function on line 114 of `minimatch.js` may be reached via the *globule* package.

```
writex/node_modules/minimatch/minimatch.js:114:0
writex/node_modules/minimatch/minimatch.js:74:9
writex/node_modules/globule/lib/globule.js:35:30
...
```

Two other functions in the *minimatch* API, `filter` and `match`, use the vulnerable `minimatch` function internally. This means that a client using those functions may also be vulnerable, however, this fact is unclear from the advisory description, so the client developer might be inclined to regard the alarm from *npm audit* as a false positive. A user of Jam is unlikely to make a similar mistake, because the call graph generated by Jam records the internal calls to `minimatch`.

The *writex* application transitively depends on 53 different packages consisting of a total of 187 JavaScript files (modules). The call graph generated by Jam shows that only 90 of the modules (spanning 42 packages) are reachable from the *writex* application. These numbers illustrate why the *npm audit* security scanner produces so much noise; if half of the files are dead code, it is unsurprising that most of the security scanner alarms are false positives.

Jam builds the call graph for *writex* and all its dependencies in 2.6 seconds, and it infers a unique caller to the vulnerable function in `minimatch.js`. In comparison, the existing tool *js-callgraph* takes 24 minutes to analyze that application, and the resulting call graph contains 1 379 call sites to the vulnerable function, so it cannot be used for providing a stack trace as the one shown above.

Furthermore, the modular analysis approach of Jam makes it possible to reuse the module summaries. For example, if we have

[5] **Ja**vaScript **m**odule analyzer
[6] https://github.com/Persper/js-callgraph
[7] https://www.npmjs.com/package/writex
[8] https://www.npmjs.com/advisories/1523
[9] https://www.npmjs.com/advisories/118

already produced modular call graphs for *writex*'s direct dependencies (which are all used also by many other applications), then the analysis time for *writex* is reduced from 2.6s to 0.2s.

3 KEY CHALLENGES

To understand some of the challenges with computing call graphs for JavaScript applications, we describe two examples.

Example 1 Consider the code below consisting of the two modules `lib1.js` and `client1.js`:

`lib1.js`:
```
1  module.exports.filter = (iteratee) => {
2    return (arr) => {
3      const res = [];
4      for (var x of arr) {
5        if (iteratee(x))
6          res.push(x);
7      }
8      return res;
9    };
10 }
```
`client1.js`:
```
11 const filter = require('./lib1.js').filter;
12 console.log(filter(x => x % 2 == 0)([1, 2, 3]));
```

The `lib1.js` module implements a curried filter function that takes a function argument, `iteratee`, and returns another function. This function then takes an array argument, `arr`, and iterates over all the elements of the array, passing each element to the `iteratee` function, and eventually returns an array containing all of the elements for which `iteratee` returned a truthy value.

To analyze this code, the first challenge we must address is that the code is split into modules. The public interface of a module is constructed dynamically by writing properties to the special object `module.exports`. For example, the `filter` method is exported by `lib1.js` as illustrated on line 1. When a module is loaded, an object containing exactly the properties written to `module.exports` is returned. The module loading happens by calling the `require` function, as demonstrated on line 11.[10] It is possible, and also quite common, to use dynamic property writes to create the `module.exports` object, and it is therefore in general difficult to statically compute the structure of a `module.exports` object [15]. As we explain in Sections 4 and 5, we approximate the module structure using a light-weight field-based static analysis that tracks what functions are written to which fields (also called properties in JavaScript) but without distinguishing individual objects. By combining the field-based observations with a heuristic for filtering irrelevant functions, we can statically compute the module structure with high precision.

The *js-callgraph* tool, which does not take a modular approach, loses precision in this example and confuses the `filter` function with `Array.prototype.filter` from JavaScript's standard library.

The second challenge is that a higher-order function is used; the `filter` function on line 1 takes a function as argument and also returns a function. The analysis should be able to determine that the call to `iteratee` on line 5 is really a call to the arrow function on line 12, and that the call to the value returned from `filter` on line 12 (blue parentheses) is really a call to the function on lines 2–9. Our call graph analysis keeps track of all these functions and where they are being called.

[10] This module system is known as CommonJS. The standardized ES6 module system is also supported by Jam but is rarely used in practice.

Example 2 Consider the following application consisting of `lib2.js` and `client2.js`:

`lib2.js`:
```
13 function Arit () { ... }
14 Arit.prototype.sum = (x, y) => x + y;
15 Arit.prototype.mul = (x, y) => x * y;
16 ...
17 module.exports.Arit = Arit;
```
`client2.js`:
```
18 const lib = require('./lib2');
19 const arit = new lib.Arit();
20 ... arit.sum(a, b) ...
```

The `lib2.js` module exports a constructor, `Arit`, which is used to construct objects with a set of methods for performing basic arithmetic. The `client2.js` module imports `lib2.js` and then constructs an `Arit` object and stores it in the constant `arit` on line 19. On line 20, the `sum` method is called on `arit`, resulting in an invocation of the function defined on line 14.

For the call graph analysis to resolve the call on line 20, a natural approach would be a form of dataflow analysis or pointer analysis that keeps track of what objects each expression may evaluate to. However, such an approach is extremely challenging for JavaScript, and no existing analysis of that kind is capable of scaling to real-world programs [16, 25, 31]. As we explain in Section 4, the field-based approach ensures that our analysis both scales well and remain precise for real-world programs. In particular, since the method call on line 20 involves a property named `sum`, it is connected to the write to the property named `sum` on line 14.

While the field-based approach could easily result in spurious call edges added to functions stored in properties with the same name but in unrelated objects, previous work has demonstrated that it works well in practice for client-side JavaScript applications that build on jQuery and related libraries [6]. The *js-callgraph* tool extends the tool by Feldthaus et al. [6] with support for newer JavaScript features, including ES6/CommonJS/AMD module loading, but the analysis itself is not modular, i.e., it does not take advantage of the module structure of the applications. As we demonstrate in Section 7, *js-callgraph* is not suited for analyzing Node.js applications since they often contain many modules.

4 MODULE SUMMARY CONSTRUCTION

The first phase of the analysis constructs a summary for each module, without considering the connections between the modules. Let *Loc*, *Prop*, *Var*, and *Exp* denote the sets of all possible source code locations,[11] property names, variable names (including parameters), and program expressions, respectively. Given a single JavaScript file f as input,[12] we compute a *module summary* β_f consisting of three separate pieces of information:

- $\beta_f.\mathsf{calls} : \mathit{Loc} \hookrightarrow \mathcal{P}(\mathit{AccessPath})$ is a *function call summary*, which for each function definition (represented by its source location) describes all the functions that are called within its body, using a special access path mechanism introduced below.

[11] Source locations consist of file name, begin line, and begin column, and are therefore always unique for every function definition. However, for brevity we omit the column information and only write ⟨file name, begin line⟩.

[12] We use the terms module and file interchangeably since a module is always stored in a single file in Node.js.

$$
\begin{array}{rcl}
\mathit{AccessPath} & ::= & < \mathit{ImportPath} > \\
& | & \mathsf{Fun}\langle f, l\rangle \\
& | & \mathsf{Fun}\langle f, l\rangle.\mathsf{Param}[i] \\
& | & \mathit{AccessPath}\ .\ \mathit{Prop} \\
& | & \mathit{AccessPath}\ (\mathcal{P}(\mathit{AccessPath}), \ldots) \\
& | & \mathsf{U}
\end{array}
$$

Figure 1: Grammar for access paths.

- $\beta_f.\mathsf{returns}\colon \mathit{Loc} \hookrightarrow \mathcal{P}(\mathit{AccessPath})$ is a *function return summary*, which for each function definition similarly describes its possible return values.
- $\beta_f.\mathsf{props}\colon \mathit{Prop} \hookrightarrow \mathcal{P}(\mathit{AccessPath})$ is an *object property summary*, which for each property name describes the values that may be assigned to object properties of that name.

We use a light-weight static analysis to compute the three components, as explained next.

Access Paths. The static analysis uses an access path mechanism (inspired by Mezzetti et al. [22] and Møller et al. [23]) to describe values of expressions in the program. The language of access paths is defined by the grammar in Figure 1.

- $< m >$ denotes the loading of a module m, as in, e.g., `require('m')`.
- $\mathsf{Fun}\langle f, l\rangle$ denotes a function definition in file f at line l.
- $\mathsf{Fun}\langle f, l\rangle.\mathsf{Param}[i]$ denotes the i'th parameter of the function definition in file f at line l.
- $ap.P$ denotes accesses to properties named P of objects denoted by the access path ap.
- $ap(S_1, \ldots, S_n)$ denotes calls to functions denoted by ap where the i'th argument is denoted by an access path in S_i.
- U is used for expressions where the static analysis is unable to assign any other access path, as explained later.

As an example, the access path $\mathsf{Fun}\langle \mathsf{lib1}, 1\rangle.\mathsf{Param}[0]$ describes the `iteratee` parameter of the `filter` function on line 1 in Example 1.

Alias Analysis and Access Path Analysis. The module summary construction computes access paths for each expression in the analyzed file using an access path analysis. This analysis uses a simple field-based alias analysis to compute a map

$$\mathit{Alias}_f\colon (\mathit{Var} \cup \mathit{Prop}) \to \mathcal{P}(\mathit{Exp})$$

for the file f, such that if the value of an expression E is written to a variable or property X, then $E \in \mathit{Alias}_f(X)$.[13]

The alias analysis constructs the map through a single traversal of f's AST. At each assignment $X = E$ or $E'.P = E$, the expression E is added to $\mathit{Alias}_f(X)$ or $\mathit{Alias}_f(P)$, respectively. Transitive dataflow is taken into account later when the alias information is being used.

Based on the alias analysis result, a map is computed that assigns a set of access paths to each expression in f:

$$\mathit{AccPaths}_f\colon \mathit{Exp} \to \mathcal{P}(\mathit{AccessPath})$$

The map is computed by $\mathit{AccPaths}_f(E) = \mathit{AP}_\emptyset(E)$ for each expression E in f, where AP is defined in Figure 2. The subscript V in AP and in the *lookup* auxiliary function ensures termination for recurrences of expressions. For module loads, such as, `require('lodash')`, the access path corresponding to the module load string is returned. For a property read $E.P$, AP computes the access paths both by recursively computing the access paths for the sub-expression E and appending $.P$, and by using the *lookup* function to compute the access paths of the expressions that the alias analysis has determined to be aliased by $.P$.[14] For a call, AP computes the receiver and argument access paths recursively, and then creates a call access path for each receiver access path. For a read of a variable that is not a parameter, AP uses *lookup* to recursively compute the access paths for the expressions aliased by the variable. A parameter is treated similarly to a variable read but also adds a parameter access path. For a function definition expression, AP creates the corresponding Fun access path. For conditional and logical expressions, the access paths are computed as the union of the access paths for the sub-expressions. In any other case (e.g., a `+` operation), the access path U is assigned to the expression.

Notice that this analysis design combines field-based analysis [6] and the use of access paths [22, 23], which enables the analysis to reason about individual modules.

Summary Construction. The function call summary $\beta_f.\mathsf{calls}$ is formed by grouping the access paths $\mathit{AccPaths}_f(E)$ for each expression E according to the function definition containing E. (For an expression in a nested function, we here only consider the innermost function.) Every Node.js module is wrapped in a function upon load of the module. We use the special access path $\mathsf{Fun}\langle f, \mathit{Main}\rangle$ to refer to the function that wraps the analyzed file f. Similarly, $\langle f, \mathit{Main}\rangle$ denotes the location of that function.

The function return summary $\beta_f.\mathsf{returns}$ is similarly computed by grouping the access paths assigned to the expressions of return statements in the function.

Finally, the object property summary $\beta_f.\mathsf{props}$ is constructed as $\beta_f.\mathsf{props}(P) = \bigcup_{E \in \mathit{Alias}_f(P)} \mathit{AccPaths}_f(E)$ for each property P.

Example 3 Continuing Example 1, we obtain the module summaries β_{client1} and β_{lib1}. Since the `filter` function is called in the outermost scope of the `client1.js` file, the call of `filter` is recorded as follows.

$$\beta_{\mathsf{client1}}.\mathsf{calls}(\langle \mathsf{client1}, \mathit{Main}\rangle) = \{<\mathsf{lib1}>.\mathsf{filter}(\ldots), \ldots\}$$

Furthermore, the return summary of the `filter` function records the access path of the function returned:

$$\beta_{\mathsf{lib1}}.\mathsf{returns}(\langle \mathsf{lib1}, 1\rangle) = \{\mathsf{Fun}\langle \mathsf{lib1}, 2\rangle\}$$

Example 4 Continuing Example 2, the function defined on line 14 is written to the property `sum`, and the function defined on line 15 is written to the property `mul`. Therefore the object property summary for the module lib2 contains the following entries:

$$\beta_{\mathsf{lib2}}.\mathsf{props}(\mathsf{sum}) = \{\mathsf{Fun}\langle \mathsf{lib2}, 14\rangle\}$$
$$\beta_{\mathsf{lib2}}.\mathsf{props}(\mathsf{mul}) = \{\mathsf{Fun}\langle \mathsf{lib2}, 15\rangle\}$$

[13] In the implementation, we extend *Prop* to also include special pre- and postfix forms of property names. For example, for a write `x["foo"+ x]= y`, the analysis records that `y` is written to a property that has `"foo"` as a prefix: $\mathit{Alias}_f(\texttt{foo*}) = \{\texttt{y}\}$. When analyzing, for example, `z.fooBar()`, the analysis will then predict that the function `y` is among the possible callees. This extension ensures that the analysis can handle some dynamic property reads, but in a way that does not lead to a major loss of precision.

[14] The analysis ignores dynamic property reads that are not of the form described in footnote 13, but since it is field-based this has little effect on its recall (see Section 5).

$$AP_V(E) := \begin{cases} \{\texttt{<}m\texttt{>}\} & \text{if } E = \texttt{require}(m) \text{ or } \texttt{import} \ldots \texttt{from}\ m \\ \{ap.P \mid ap \in AP_V(E')\} \cup lookup_V(P) & \text{if } E = E'.P \\ \{ap(AP_V(E_1), \ldots, AP_V(E_n)) \mid ap \in AP_V(E')\} & \text{if } E = E'(E_1, \ldots, E_n) \text{ or } E = \texttt{new}\ E'(E_1, \ldots, E_n) \\ lookup_V(X) & \text{if } E = X \text{ where } X \text{ is a non-parameter variable} \\ \{\mathsf{Fun}\langle f, l\rangle.\mathsf{Param}[n]\} \cup lookup_V(X) & \text{if } E = X \text{ where } X \text{ is the } n\text{'th parameter in a function created at line } l \text{ in file } f \\ \{\mathsf{Fun}\langle f, l\rangle\} & \text{if } E \text{ is a function definition at line } l \text{ in file } f \\ AP_V(E_1) \cup AP_V(E_2) & \text{if } E = E'\ \texttt{?}\ E_1\ \texttt{:}\ E_2 \text{ or } E = E_1\ \texttt{||}\ E_2 \text{ or } E = E_1\ \texttt{\&\&}\ E_2 \\ \{\mathtt{U}\} & \text{otherwise} \end{cases}$$

$$lookup_V(Z) := \begin{cases} \bigcup_{E \in Alias_f(Z)} AP_{V \cup \{Z\}}(E) & \text{if } Z \notin V \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 2: Access path computation.

5 CALL GRAPH CONSTRUCTION

Before constructing the call graph for a Node.js application, we combine the module summaries for all its modules. For example, β.calls (omitting the module name) denotes the combined call summary and is computed by $\beta.\mathsf{calls}(loc) = \bigcup_{f \in M} \beta_f.\mathsf{calls}(loc)$ for all $loc \in Loc$ where M is the set of modules, and similarly for the other components.

The call graph needs to span across multiple modules, so in the call graph construction phase, we combine the module summaries from each file into a call graph $\mathcal{G} = (V, E, \beta, \alpha)$ with nodes $V \subseteq Loc$ corresponding to function definitions and edges $E \subseteq Loc \times Loc \times AccessPath$ that represent the call edges, β is the combined module summary, and α is explained below. Each edge in E is annotated with the access path of a function call between the two functions. We use these annotations when resolving calls to higher-order function parameters.

Computing the call graph amounts to solving the constraints generated by the rules of Figure 3. The constraints involve two relations: E, which contains the call graph edges, and $\alpha \subseteq AccessPath \times AccessPath \times AccessPath$, which is used for resolving function calls during the analysis. We say that an expression E in a file f is *represented* by an access path ap if $ap \in AccPaths_f(E)$. The notation $n \overset{ap}{\rightsquigarrow} n'$ is a shorthand for $(n, n', ap) \in E$, meaning that the function at n may call the function at n' and ap is an access path that represents such a call. Similarly, $ap \overset{ap'}{\sim} ap''$ means $(ap, ap', ap'') \in \alpha$, which intuitively means that expressions represented by ap may obtain their function values from expressions represented by ap'', and ap' represents calls to such functions.

Example 5 The call to `filter` on line 12 marked with red parentheses in Example 1 gives rise to the following entries in α and E.

$$\texttt{<lib1>.filter} \overset{\texttt{<lib1>.filter}(\ldots)}{\sim} \mathsf{Fun}\langle \mathsf{lib1}, 1\rangle$$

$$\langle \mathsf{client1}, Main\rangle \overset{\texttt{<lib1>.filter}(\ldots)}{\rightsquigarrow} \langle \mathsf{lib1}, 1\rangle$$

We explain in Examples 6 and 7 how these entries are produced.

The call graph computation works by iteratively extending E and α according to the constraint rules until a fixed point is reached. Such a fixed point is guaranteed to exist since E or α always increases in size and there are finitely many access paths in the module summaries.

The first two rules in Figure 3 only depend on the function call and object property summaries (β.calls and β.props), and not on E or α, so they can be resolved in the first iteration of the algorithm.

- *MODULE-CALL* connects the access path $\texttt{<}m\texttt{>} \ldots g$, representing the callee of a call access path that is in the function call summary, to an access path ap' if the module m resolves[15] to a file f' and the access path ap' is in the object property summary for a file f'' and property g, where the package that contains f' (denoted $package(f')$) is the same as or depends directly or transitively on the package that contains f''. The reason for considering only the object property summaries for those files is explained by the following scenario: If a package A depends on a package B, which in turn depends on package C, then functions in A typically do not affect the interface of B,[16] whereas the functions of C may be re-exported through modules in B. We here use the package dependency structure, because it can be extracted soundly, directly from the `package.json` files. If the call is to the module object directly, i.e., $ap = \texttt{<}m\texttt{>}()$, then the default exported function is extracted from the object property summary using the property name `exports`.
- *OTHER-CALL* connects the callee represented by an access path ap' to itself if $ap'(\ldots)$ appears in the function call summary, provided that the *MODULE-CALL* rule does not apply. The remaining analysis constraints will resolve ap' to the functions it represents.

Example 6 Based on the module summaries presented in Example 3, the analysis has recorded that $\mathsf{Fun}\langle \mathsf{lib1}, 1\rangle$ is written to the `filter` property. The rule *MODULE-CALL* then applies to the call to `filter` on line 12, which results in the α entry shown in Example 5. Since no other functions are written to a property named `filter` in lib1 or its dependencies, that entry is the only one added to α for this call. We describe in Example 7 how the corresponding call edge is produced from α.

[15] The *resolve* function in this rule is similar to the `require.resolve` function from Node.js.

[16] It is possible to construct a scenario where functions of A become part of B's interface, but we have not observed this behavior in practice, which is why we resort to this heuristic even though it is theoretically unsound.

$$[\textit{MODULE-CALL}]\quad \frac{ap = \texttt{<}m\texttt{>}\ldots g(\ldots) \in \beta.\mathsf{calls}(\langle \_, \_ \rangle) \qquad f' = \mathit{resolve}(m) \qquad ap' \in \beta_{f''}.\mathsf{props}(g) \qquad \mathit{package}(f') \text{ is or depends on } \mathit{package}(f'')}{\texttt{<}m\texttt{>}\ldots g \overset{ap}{\sim} ap'}$$

$$[\textit{OTHER-CALL}]\quad \frac{ap = ap'(\ldots) \in \beta.\mathsf{calls}(\langle \_, \_ \rangle) \qquad \textit{MODULE-CALL} \text{ does not apply}}{ap' \overset{ap}{\sim} ap'}$$

$$[\textit{RETURN-CALL}]\quad \frac{\cdots \overset{ap''}{\sim} ap(\ldots) \qquad ap \overset{\cdots}{\sim} \mathsf{Fun}\langle f, l \rangle \vee ap = \mathsf{Fun}\langle f, l \rangle \qquad ap' \in \beta_f.\mathsf{returns}(\langle f, l \rangle)}{ap(\ldots) \overset{ap''}{\sim} ap'}$$

$$[\textit{PARAM-CALL}]\quad \frac{\cdots \overset{ap''}{\sim} \mathsf{Fun}\langle f, l \rangle.\mathsf{Param}[n] \qquad \cdots \overset{ap}{\rightsquigarrow} \langle f, l \rangle \qquad ap' \in \mathit{arg}(ap, n)}{\mathsf{Fun}\langle f, l \rangle.\mathsf{Param}[n] \overset{ap''}{\sim} ap'}$$

$$[\textit{PROP-CALL}]\quad \frac{\cdots \overset{ap''}{\sim} ap.q \qquad ap' \in \beta.\mathsf{props}(q)}{ap.q \overset{ap''}{\sim} ap'}$$

$$[\textit{TRANSITIVE}]\quad \frac{ap \overset{ap'''}{\sim} ap' \qquad ap' \overset{ap'''}{\sim} ap''}{ap \overset{ap'''}{\sim} ap''}$$

$$[\textit{EDGE}]\quad \frac{\cdots \overset{ap}{\sim} \mathsf{Fun}\langle f', l' \rangle \qquad ap \in \beta.\mathsf{calls}(\langle f, l \rangle)}{\langle f, l \rangle \overset{ap}{\rightsquigarrow} \langle f', l' \rangle}$$

Figure 3: Analysis constraint rules.

The next three constraint rules presented in Figure 3 model calls to functions returned by other functions (*RETURN-CALL*), calls to function parameters (*PARAM-CALL*), and calls to functions stored in object properties (*PROP-CALL*). These rules are applied iteratively since, for example, for the expression `f()()`, resolving the second call depends on the result of the first call.

- *RETURN-CALL* ensures that if ap'' represents a call to a function value that is returned from a function call $ap(\ldots)$, i.e. $\cdots \overset{ap''}{\sim} ap(\ldots)$, then $ap(\ldots)$ may obtain its function values from the return values of ap functions. Those functions are retrived using $ap \overset{\cdots}{\sim} \mathsf{Fun}\langle f, l \rangle$. In the special case where ap denotes a function definition ($ap = \mathsf{Fun}\langle f, l \rangle$), we use the return values of that function directly. We retrieve the access paths ap' representing return values using the function return summary ($ap' \in \beta_f.\mathsf{returns}(\langle f, l \rangle)$).
- *PARAM-CALL* ensures that if ap'' represents a call to a function value that comes from the n'th parameter of a function $\mathsf{Fun}\langle f, l \rangle$, i.e. $\cdots \overset{ap''}{\sim} \mathsf{Fun}\langle f, l \rangle.\mathsf{Param}[n]$, then the function values of that parameter may come from the corresponding arguments at call sites. The access paths ap of the call sites are retrieved from the annotations of the call edges that point to $\langle f, l \rangle$, i.e. $\cdots \overset{ap}{\rightsquigarrow} \langle f, l \rangle$. We use $\mathit{arg}(ap, n)$ to denote the set of access paths at the n'th argument of the call access path ap.
- *PROP-CALL* ensures that if ap'' represents a call to a function value that comes from an object property q, i.e. $\cdots \overset{ap''}{\sim} ap.q$, then the possible function values ap' include those found in the object property summary for q ($ap' \in \beta.\mathsf{props}(q)$).

The remaining two rules model transitivity and construction of call edges.

- *TRANSITIVE* models the fact that, for calls represented by some access path ap''', if an expression E represented by ap may obtain its values from an expression E' represented by ap', and ap' may obtain its values from an expression E'' represented by ap'', then E may obtain its values from E''. We require the access path ap''' to be the same in the two premises to avoid mixing together calls from different call sites to the same function.
- *EDGE* ensures that a call edge is added from $\langle f, l \rangle$ to $\langle f', l' \rangle$ whenever there is an entry $\cdots \overset{ap}{\sim} \mathsf{Fun}\langle f', l' \rangle$ in α where the access path ap is in the function call summary of some function $\langle f, l \rangle$. Intuitively, such an entry in α tells us that $\langle f', l' \rangle$ may be called from a call site represented by access path ap, and if $ap \in \beta.\mathsf{calls}(\langle f, l \rangle)$ then the call site is in the function $\langle f, l \rangle$.

Example 7 Continuing Examples 5 and 6, the call edge entry in E is created by the rule *EDGE* from the entry in α since <lib1>.filter(...) $\in$ β.calls($\langle$client1, *Main*$\rangle$).

Example 8 Continuing Example 1, let us now consider how the call in blue parentheses on line 12 to `filter`'s return value is resolved. From β_{client1}.calls we get the access path of this call as <lib1>.filter(...)(...). Since *MODULE-CALL* does not match, and the access path ends with (...), the rule *OTHER-CALL* applies:

$$\texttt{<lib1>.filter}(\ldots) \overset{\texttt{<lib1>.filter}(\ldots)(\ldots)}{\sim} \texttt{<lib1>.filter}(\ldots)$$

Furthermore, we saw in Example 3 that the return summary contains the fact that `filter` returns the function $\mathsf{Fun}\langle \mathsf{lib1}, 2 \rangle$. By the *MODULE-CALL* rule, we also have

$$\texttt{<lib1>.filter} \overset{\texttt{<lib1>.filter}(\ldots)}{\sim} \mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle$$

so by *RETURN-CALL* we have

$$\texttt{<lib1>.filter}(\ldots) \overset{\texttt{<lib1>.filter}(\ldots)(\ldots)}{\sim} \mathsf{Fun}\langle \mathsf{lib1}, 2 \rangle$$

which finally by *EDGE* ensures that an edge is added in E between the caller and the callee:

$$\langle \mathsf{client1}, \mathit{Main} \rangle \overset{\texttt{<lib1>.filter}(\ldots)(\ldots)}{\rightsquigarrow} \langle \mathsf{lib1}, 2 \rangle$$

Example 9 Consider the call to `iteratee` colored brown on line 5 in Example 1. The access path is $\mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle.\mathsf{Param}[0](\ldots)$ for this call. Again, by *OTHER-CALL*:

$$\mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle.\mathsf{Param}[0] \overset{\mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle.\mathsf{Param}[0](\ldots)}{\sim} \mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle.\mathsf{Param}[0]$$

The *PARAM-CALL* rule says that the parameter call access path is related to the access paths of the 0'th argument of calls flowing into $\mathsf{Fun}\langle \mathsf{lib}, 1 \rangle$. The only such call is the call to `filter` on line 12 with access path <lib1>.filter({$\mathsf{Fun}\langle$client1, 12$\rangle$}) (we have omitted the arguments in the access path previously due to space constraints). So by the *PARAM-CALL* rule:

$$\mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle.\mathsf{Param}[0] \overset{\mathsf{Fun}\langle \mathsf{lib1}, 1 \rangle.\mathsf{Param}[0](\ldots)}{\sim} \mathsf{Fun}\langle \mathsf{client1}, 12 \rangle$$

which by *EDGE* results in the call edge:

$$\langle \text{lib1}, 1\rangle \overset{\text{Fun}\langle \text{lib1},1\rangle.\text{Param}[0](\dots)}{\rightsquigarrow} \langle \text{client1}, 12\rangle$$

Example 10 Consider the call to `sum` on line 20 in Example 2. From β_{client2}.calls we have that the access path of `sum` is <lib2>.Arit().sum(). Again, the rule *OTHER-CALL* applies:

$$\text{<lib2>.Arit().sum} \overset{\text{<lib2>.Arit().sum}(\dots)}{\sim} \text{<lib2>.Arit().sum}$$

This triggers the *PROP-CALL* rule, which says that <lib2>.Arit().sum is related to all functions in the object property summary for `sum`. From Example 4 we know that the only such function is the one defined on line 14.

$$\text{<lib2>.Arit().sum} \overset{\text{<lib2>.Arit().sum}(\dots)}{\sim} \text{Fun}\langle \text{lib2}, 14\rangle$$

By *EDGE* this results in the following call edge:

$$\langle \text{client2}, \mathit{Main}\rangle \overset{\text{<lib2>.Arit().sum}(\dots)}{\rightsquigarrow} \langle \text{lib2}, 14\rangle$$

Example 11 Let us consider an example program that requires multiple applications of *RETURN-CALL* and an application of *TRANSITIVE*, so that we can see how more complex calls are resolved.

```
lib3.js:
21 function e() {...}
22
23 function f() {
24   return e;
25 }
26
27 function g() {
28   return f;
29 }
30
31 module.exports.h = function() {
32   return g();
33 }


client3.js:
34 const lib = require('lib3');
35 const x = lib.h()();
36 x();
```

The function `h` (defined at line 31) returns the return value of `g` (defined at line 27), which is the function `f` (defined at line 23), and `f` returns the function `e` (defined at line 21). Hence, the expression on line 35 results in calls to the functions `f`, `g`, and `h`, and the expression on line 36 results in a call to the function `e`.

Let us first consider the call to `h` (`lib.h()`). Since $\text{Fun}\langle \text{lib3}, 31\rangle \in \beta_{\text{lib3}}.\text{props}(\text{h})$, we have by the *MODULE-CALL* rule:

$$\text{<lib3>.h} \overset{\text{<lib3>.h()}}{\sim} \text{Fun}\langle \text{lib3}, 31\rangle$$

By the *EDGE* rule, we then have:

$$\langle \text{client3}, \mathit{Main}\rangle \overset{\text{<lib3>.h()}}{\rightsquigarrow} \langle \text{lib3}, 31\rangle$$

We now consider how the analysis resolves `lib.h()()` to `f`. By the *OTHER-CALL* rule we have:

$$\text{<lib3>.h()} \overset{\text{<lib3>.h()()}}{\sim} \text{<lib3>.h()}$$

From the resolution of the `lib.h()` call above, we have

$$\text{<lib3>.h} \overset{\text{<lib3>.h()}}{\sim} \text{Fun}\langle \text{lib3}, 31\rangle$$

and during the module summary construction we have recorded access paths representing return values of `h`:

$$\text{Fun}\langle \text{lib3}, 27\rangle() \in \beta.\text{returns}(\langle \text{lib3}, 31\rangle)$$

The *RETURN-CALL* rule then applies:

$$\text{<lib3>.h()} \overset{\text{<lib3>.h()()}}{\sim} \text{Fun}\langle \text{lib3}, 27\rangle()$$

and by a second application of the *RETURN-CALL* rule:

$$\text{Fun}\langle \text{lib3}, 27\rangle() \overset{\text{<lib3>.h()()}}{\sim} \text{Fun}\langle \text{lib3}, 23\rangle$$

Finally, by the *EDGE* rule, an edge is added from the main function of client3 to `f` in lib3:

$$\langle \text{client3}, \mathit{Main}\rangle \overset{\text{<lib3>.h()()}}{\rightsquigarrow} \langle \text{lib3}, 23\rangle$$

The last call remaining is the call at line 36. For resolving that call we first apply the *OTHER-CALL* rule:

$$\text{<lib3>.h()()} \overset{\text{<lib3>.h()()()}}{\sim} \text{<lib3>.h()()}$$

Next, we apply the *TRANSITIVE* rule, based on the entries added to α by the two applications of the *RETURN-CALL* rule for the second call on line 35:

$$\text{<lib3>.h()} \overset{\text{<lib3>.h()()}}{\sim} \text{Fun}\langle \text{lib3}, 23\rangle$$

From the function return summary we have

$$\text{Fun}\langle \text{lib3}, 21\rangle \in \beta.\text{returns}(\langle \text{lib3}, 23\rangle)$$

and finally, by the *RETURN-CALL* rule:

$$\text{<lib3>.h()()} \overset{\text{<lib3>.h()()()}}{\sim} \text{Fun}\langle \text{lib3}, 21\rangle$$

The *EDGE* rule then adds an edge from the main function of client3 to `e` in lib3:

$$\langle \text{client3}, \mathit{Main}\rangle \overset{\text{<lib3>.h()()()}}{\rightsquigarrow} \langle \text{lib3}, 21\rangle$$

Modular Analysis. The call graph analysis is modular, as described in Section 1. Let us consider the example from the introduction. We assume a call graph $\mathcal{G}_{BC}$ has been built for the modules B and C, and we want to create a call graph for the application A_1 that depends on B and C. The analysis starts with the α, and E relations from $\mathcal{G}_{BC}$. The analysis then computes the module summaries for every module in A_1 and applies the first two rules of Figure 3 using these summaries. The remaining constraints are now solved using the combined module summaries from B, C, and A_1. Because $\mathcal{G}_{BC}$ represents a partial result of $\mathcal{G}_{A_1BC}$, we can compute $\mathcal{G}_{A_1BC}$ faster with $\mathcal{G}_{BC}$ precomputed, as demonstrated in Section 7. The resulting call graphs when using this bottom-up approach are the same as when combining all module summaries in a single step.

Analysis Extensions. The analysis described so far does not have support for built-in functions, getters, setters, and events. We now describe how the analysis is extended to handle these features. The source code of built-in functions is generally unavailable, so the analysis handles these by assuming that their function arguments are always called. We leverage the field-based analysis design, such that $ap \overset{ap'''}{\sim} ap'.q$ from Figure 3 also adds $ap'.q \overset{ap'''}{\sim} ap''$ for any access path ap'' that represents callbacks to a built-in function q. For example, if q is `map`, the analysis adds the above entry to α for each $ap'' \in x[0]$ since `Array.prototype.map` takes a callback function as the first argument.

JavaScript supports getter (and setter) properties that invoke a function when they are read (or written). For this reason, we extend the module summary with two maps from property names to sets of access paths describing the getter and setter functions similar to

how we handle field-based information. We refer to these additional maps as β.getters and β.setters. For each property q that is read or written, we add $\mathsf{Fun}\langle f, l\rangle \overset{\cdots}{\sim} ap$ to α for each $ap \in \beta.\mathsf{getters}(q)$ for getters and $ap \in \beta.\mathsf{setters}(q)$ for setters.

We also added a special mechanism for handling the Node.js event system where events are registered using an `on` method and emitted using an `emit` method. Each module summary is augmented with an *event summary* similar to the object property summary. An event summary is a map from event names to sets of access paths. At calls to a method named `on` with two arguments where the first argument is a string, the map is extended with the access paths of the second argument. At `emit` calls, the corresponding access paths for the emitted event is then looked up in the event summary, and call graph edges are added accordingly.

Restricting Object Properties to Adjacent Packages. The number of property writes in applications rises as the number of dependencies grow, so for large applications, it is possible that unrelated object properties from unrelated packages are mixed together. Since this blowup increases the risk of spurious edges added by the PROP-CALL rule, we have added a heuristic where the object property summary is only mixed between directly related packages. With this heuristic, the lookup $\beta.\mathsf{props}(q)$ in PROP-CALL only considers the object property summary from the packages that are direct dependencies or direct dependents to the package with the caller. While this heuristic theoretically makes the technique more unsound, it does not cause any vulnerabilities to be missed in our security scanning experiments (see Section 7).

Soundness Assumptions. The call graph analysis is not theoretically sound. There are four potential sources of unsoundness [19]: (1) The analysis ignores dynamic property reads/writes unless they are of the special pre- or postfix form mentioned in footnote 13, but since the analysis is field-based, the analysis results are not affected much by this [6]. (2) The adjacent packages heuristic can result in missing edges if values flow between packages that are not directly linked in the package dependency graph. The design choice for the MODULE-CALL rule may have a similar consequence as mentioned in footnote 16. However, as such flows occur rarely in practice, the analysis result remains sound for practical purposes. (3) The analysis ignores dynamic module loads and dynamic code generation. We have not found many usages of dynamic module loads, and dynamic code generation is typically also only used sparsely in Node.js programs. (4) The analysis does not model all ECMAScript features such as iterators and implicit calls. However, the parser used by Jam is compatible with all existing ECMAScript versions, so the analysis will still produce results in the presence of these features.

6 SECURITY SCANNING

To use the call graph for security scanning, the analysis has to know which nodes represent vulnerable functions. We describe known security vulnerabilities from the npm vulnerability database[17] using a simple pattern language, and use the function *findNodes* (see Figure 5) to convert these patterns to source locations.[18]

$$
\begin{array}{rcl}
\mathit{VulnDesc} & ::= & (\mathit{AdvisoryID}, \mathit{PackageName}, \\
 & & \mathit{VersionRange}, \mathit{API\text{-}Pattern}) \\
\mathit{API\text{-}Pattern} & ::= & \{\ \mathit{API\text{-}Pattern}, \ldots, \mathit{API\text{-}Pattern}\ \} \\
 & | & <\ \mathit{ImportPath}\ > \\
 & | & \mathit{API\text{-}Pattern}\ \texttt{.}\ \mathit{Prop} \\
 & | & \mathit{API\text{-}Pattern}\ \texttt{()}
\end{array}
$$

Figure 4: API patterns.

The grammar of the pattern language is shown in Figure 4. A vulnerability description (*VulnDesc*) consists of the advisory ID, the name of the package affected by the vulnerability, the range of affected versions, and an *API-Pattern* identifying the vulnerable parts of the library API. *AdvisoryID* is a number, and *ModuleName* and *VersionRange* are strings. An *API-Pattern* can express disjunctions ({*API-Pattern*, ..., *API-Pattern*}), values obtained from loading modules (< *ImportPath* >), values read from properties (*API-Pattern* . *Prop*), and return values of functions (*API-Pattern* ()). The language of API patterns resembles the language of access paths (Figure 1) but is designed for easily identifying API functions, whereas access paths are used only internally by the analysis.

If the application depends on some version of the vulnerable package in the vulnerable version range, then the function *findNodes* is used to find the source locations of the vulnerable functions. It uses the vulnerability descriptions and the module summaries to compute the source locations. The first case handles the disjunction pattern, $p = \{p_1, ..., p_n\}$, as the union of the results of calling *findNodes* for each subpattern. For property read sequences on a module object without any calls, $p = \texttt{<}m\texttt{>}\ldots g$, the function source locations are extracted similarly to the MODULE-CALL rule of Figure 3 (this rule also applies when the module object is read directly, i.e., $p = \texttt{<}m\texttt{>}$, in which case the special `exports` property is used). For a property read, $p = p'.q$, where p does not begin with a module load, we first extract the access paths of q from β.props and then the concrete source locations of these access paths from α. For calls to returned values, $p = p'()$, the source locations represented by p' are extracted by calling *findNodes* recursively. For each function at these locations, the access paths representing the return values of that function are extracted by a lookup in the return summary of the function. Finally, the actual function definitions are extracted from α (similar to the RETURN-CALL rule).

The security scanner can then check whether these functions are reachable in the call graph from the entry node of the application.[19] If any of the functions are reachable, the user is warned, and a link to the informal npm advisory description is presented together with the top of a stack trace leading to the vulnerable function as shown in Section 2. The stack trace is computed by traversing backwards in the call graph, from the vulnerable function.

[17] https://www.npmjs.com/advisories

[18] One might be tempted to simply describe the vulnerable functions as a set of source locations directly, but that would make the analysis sensitive to changes in source locations across different versions of the vulnerable dependency.

[19] While the call graph analysis works on both libraries and applications, the security scanner is limited to applications that have a single, well-defined entry point.

$$findNodes(p) := \begin{cases} \bigcup_{p' \in \{p_1,\ldots,p_n\}} findNodes(p') & \text{if } p = \{p_1, \ldots, p_n\} \\ \{\langle f, l\rangle \mid f' = resolve(m) \\ \quad \wedge\ package(f') \text{ is or depends on } package(f'') \\ \quad \wedge\ ap \in \beta_{f''}.\text{props}(g) \\ \quad \wedge\ ap \stackrel{\cdots}{\sim} \text{Fun}\langle f, l\rangle\} & \text{else if } p = \text{<m>}\ldots g \\ \{\langle f, l\rangle \mid ap \in \beta.\text{props}(q) \wedge ap \stackrel{\cdots}{\sim} \text{Fun}\langle f, l\rangle\} & \text{else if } p = p'.q \\ \bigcup_{\langle f,l\rangle \in findNodes(p')} \{\langle f', l'\rangle \mid ap \in \beta_f.\text{returns}(\langle f, l\rangle) \wedge ap \stackrel{\cdots}{\sim} \text{Fun}\langle f', l'\rangle\} & \text{else if } p = p'() \end{cases}$$

Figure 5: Algorithm for finding vulnerable functions from API patterns.

7 EVALUATION

We have implemented Jam (including the security scanner) in 3 000 lines of TypeScript code, using *acorn*[20] and *acorn-walk*[21] for parsing JavaScript files and traversing ASTs. We evaluate the approach by answering the following research questions.

RQ1: What are the precision and the recall of performing security scanning on Node.js applications based on call graphs constructed by Jam, compared to the *npm audit* approach that is based on package-level dependencies?

RQ2: What are the precision (measured by unique callees) and the recall (measured by comparing against dynamically created call graphs) for the call graphs constructed by Jam, and how do they compare to call graphs computed by the existing tool *js-callgraph*?

RQ3: How fast is the analysis? Is it faster if we, by taking advantage of the modularity of the application structure and the analysis, assume we have precomputed call graphs for the packages used by the applications?

7.1 Experimental Setup

To answer the research questions, we randomly selected 12 Node.js applications from the npm registry where *npm audit* reports one or more alarms (to get nontrivial data for RQ1). The benchmarks are listed in Table 1.

We run both *npm audit* and the Jam-based security scanner on each benchmark and manually classify the reported issues as true or false positives. Our security scanner is configured to use the same set of known library vulnerabilities as *npm audit*. As mentioned in Section 2, the security warnings generated by our approach provide reachability information at the level of functions, while the warnings from *npm audit* only contain coarse-grained information at the level of packages. For this experiment, we disregard this reachability information and only look at whether or not the given application is flagged as potentially affected by each of the known library vulnerabilities. We classify a security warning as a true positive if the vulnerable library function is reachable in some concrete execution of the application. (Note that reachability does not imply that the vulnerability is exploitable, which is a more subjective matter.) Since *npm audit* reports alarms for all the known library vulnerabilities in all transitive dependencies, a priori it has no false negatives, and the Jam-based security scanner by construction always reports the same or a subset of the issues reported by *npm audit*.

[20] https://www.npmjs.com/package/acorn
[21] https://www.npmjs.com/package/acorn-walk

For RQ2, to measure recall relative to dynamically generated call graphs, we use NodeProf [32] and different inputs to the applications to cover a variety of combinations of their configuration options. Since it is difficult to obtain high dynamic coverage we measure precision of the call graphs independently of dynamic executions, as the percentage of call sites that have a unique callee according to the analysis (Jam or *js-callgraph*) as in previous work (e.g. [26, 31]). For these precision and recall measurements we disregard call edges that are not reachable from the entry, since only the reachable edges are relevant for security scanning. We run *js-callgraph* in "optimistic" mode (strategy *DEMAND*), which gives the best results.

To answer RQ3, for each application we first compute call graphs for each of its direct dependencies. From these call graphs, we compute an aggregated call graph of all the dependencies (see the paragraph on modular analysis in Section 5). Finally, we compute the call graph for the entire application using the aggregated call graph of the dependencies.

Our experiments have been run on a machine with a 2.9GHz Intel core i7 CPU with 10GB RAM for the analysis.

7.2 Results for RQ1 (Security Scanning)

The results of the security scanning experiment are presented in Table 1, where "**Functions**" shows the total number of functions in the application and all its dependencies, and, in parentheses, the number of functions reachable from the application entry according to the call graph computed by Jam, "**Modules**" and "**Packages**" similarly show the numbers of modules and packages, the "**npm audit**" columns show the number of security alarms reported by *npm audit* security scanner, and the "**Jam**" columns show the alarms reported by Jam. The alarms are categorized into alarms about actual usage of a vulnerable library function (true positives, "**TP**"), alarms about a vulnerable library function that is never used by the application (false positives, "**FP**"), and usages of a vulnerable library function where no alarm was raised (false negatives, "**FN**").

We have manually classified the alarms by *npm audit* into true and false positives. As can be seen in Table 1, *npm audit* reports 34 alarms for the 12 benchmarks, where only 8 are true positives and 26 are false positives, yielding a precision of only 24%.

The Jam security scanner found all 8 vulnerabilities, resulting in a perfect 100% recall of the security warnings. For all the 7 applications where *npm audit* reports false positives, the call-graph-based security scanner reduces the number of false positives. For 5 of them, the call-graph-based security scanner even manages to remove all the false positives. In total, the call-graph-based security scanner reduced the number of false positives by 81% compared

Table 1: Experimental results for security scanning.

							npm audit		Jam		
Name	**Functions**		**Modules**		**Packages**		**TP**	**FP**	**TP**	**FP**	**FN**
makeappicon 1.2.2	6 165	(628)	1 393	(44)	13	(13)	0	3	0	2	0
toucht 0.0.1	6 479	(61)	1 560	(4)	25	(1)	0	3	0	0	0
spotify-terminal 0.1.2	8 259	(61)	783	(3)	106	(1)	0	4	0	0	0
ragan-module 1.3.0	839	(589)	85	(79)	61	(53)	1	0	1	0	0
npm-git-snapshot 0.1.1	898	(357)	120	(55)	41	(35)	1	0	1	0	0
nodetree 0.0.3	1 557	(143)	15	(7)	4	(4)	0	4	0	0	0
jwtnoneify 1.0.1	27 703	(3 762)	1 869	(201)	93	(55)	0	4	0	3	0
foxx-framework 0.3.6	4 334	(1 124)	261	(124)	68	(51)	1	0	1	0	0
npmgenerate 0.0.1	1 638	(530)	266	(31)	23	(19)	2	0	2	0	0
smrti 1.0.3	1 228	(732)	121	(116)	64	(56)	1	0	1	0	0
writex 1.0.4	4 177	(1 237)	187	(90)	53	(42)	1	4	1	0	0
openbadges-issuer 0.4.0	6 043	(670)	1 366	(133)	69	(37)	1	4	1	0	0
Total	**69 320**	**(9 894)**	**8 026**	**(887)**	**620**	**(367)**	**8**	**26**	**8**	**5**	**0**

Table 2: Experimental results for call graph construction.

	Jam				js-callgraph		
Name	**Precision**	**Recall**	**Time (full)**	**Time (modular)**	**Precision**	**Recall**	**Time (full)**
makeappicon 1.2.2	86.05%	100.00%	2.04s	0.07s	–	–	–
toucht 0.0.1	92.08%	100.00%	0.69s	0.01s	–	–	–
spotify-terminal 0.1.2	92.79%	100.00%	0.73s	0.01s	–	–	–
ragan-module 1.3.0	87.42%	98.95%	1.39s	0.04s	80.00%	1.39%	1.84s
npm-git-snapshot 0.1.1	82.45%	94.78%	0.99s	0.04s	43.86%	85.32%	10.22s
nodetree 0.0.3	70.65%	100.00%	0.87s	0.03s	–	0.00%	12.93s
jwtnoneify 1.0.1	71.43%	98.18%	23.01s	0.70s	–	–	–
foxx-framework 0.3.6	89.14%	99.41%	2.37s	0.27s	68.44%	62.24%	61.30s
npmgenerate 0.0.1	97.42%	100.00%	1.80s	0.08s	59.81%	56.55%	614.37s
smrti 1.0.3	80.80%	96.51%	1.65s	0.09s	66.20%	63.53%	3.77s
writex 1.0.4	86.07%	100.00%	2.56s	0.18s	52.14%	64.20%	1 450.70s
openbadges-issuer 0.4.0	75.85%	95.59%	2.63s	0.11s	40.08%	52.04%	147.54s
Average	**84.35%**	**98.62%**	**3.39s**	**0.14s**	**58.64%**	**48.16%**	**287.83s**

to *npm audit*, which means that the precision of the Jam security scanner is 61% compared to the 24% precision of *npm audit*.

The 5 false positives are caused by vulnerabilities in the *lodash* library. The reason for these false positives is not that the computed call graphs have too many edges, but that the vulnerable library function, which is not used by the applications, is mixed together with a function that is being used by the applications. This happens because those two functions are defined in the library via a higher-order function and originate from the same function definition, and they differ only because of their free variables. Since Jam uses the function definition source locations to identify the functions, it does not distinguish between the two functions. Improving this aspect is an interesting opportunity for future work.

Although Jam has no false negatives in the experiments, it is possible that it may miss some call edges, as discussed in Section 5. We have manually inspected the module connectivity in the call graphs for the three benchmarks with fewer than 10 reachable modules, and we find no inter-module edges missing. Also, we have checked for all the benchmarks that all modules that are being loaded in a concrete execution are reachable in the call graphs.

Naturally, any vulnerabilities that may exist in unreachable parts of the application code cannot affect the behavior of the applications. The applications altogether contain 69 320 functions, 8 026 modules, and 620 packages (including duplicates used by several applications). According to the computed call graphs, only 9 894 (14%) of the functions, 887 (11%) of the modules, and 367 (59%) of the packages are reachable, which gives an indication of the overall potential of call-graph-based security scanning.

7.3 Results for RQ2 (Call Graph Accuracy)

The results of the call graph precision and recall measurements are shown in Table 2. Jam finds that on average 84.35% of the call sites have a unique callee, compared to only 58.64% for *js-callgraph*. Also, 98.62% of the call edges observed in the concrete executions are detected by Jam, while the corresponding result for *js-callgraph* is only 48.16%.[22] Moreover, *js-callgraph* fails on 5 of the applications, either crashing with out-of-memory or producing a call graph with no nodes reachable from the entry (both indicated by '–'). The few missing edges in the Jam results are triggered by some rare cases where the soundness assumptions do not hold (see Section 5).

[22]We have excluded the package *esprima* from the recall measurements because it has been bundled using webpack. The low recall for *nodetree* and *ragan-module* with *js-callgraph* is due to limitations in its parser and lack of support for getters.

These results suggest that the call graphs produced by Jam are substantially more accurate than those produced by *js-callgraph*. Even though the recall for Jam is not perfect, a few false negatives is likely preferable to a large number of false positives or a significantly slower analysis.

7.4 Results for RQ3 (Analysis Time)

The "**Time (full)**" columns in Table 2 shows the time it takes Jam (and *js-callgraph*) to compute the call graphs for the applications including all dependencies. The analysis time for Jam varies from less than one second for the *toucht* application to around 23 seconds for *jwtnoneify*, and *js-callgraph* is orders of magnitude slower.

The relatively large time for *jwtnoneify* is explained by a heavy usage of, for example, the `forEach` function from the *lodash* library. The `forEach` function is a higher-order function that takes a collection (typically an array) and some iterator function that is called with each element in the collection as an argument. Because the rule *PARAM-CALL* from Figure 3 merges arguments from all call sites when a parameter is called, a massive amount of new entries are added to the α relation. This behavior is similar to what happens in a context-insensitive dataflow analysis. Perhaps surprisingly, despite the longer analysis time and the less precise call graph, the security scanner is still more precise than *npm audit* for this application. Nevertheless, investigating this outlier in more detail and improving its analysis time is an interesting challenge for future work.

The "**Time (modular)**" column in Table 1 shows the analysis time, when the call graphs for all direct dependencies of the application have been precomputed, which is a realistic situation in a scenario where many applications that share dependencies are being analyzed. The time includes aggregating the call graphs from the dependencies and computing the call graph for the entire application. The call graph construction is very efficient taking less than a second for all applications.

We conclude that the Jam full call graph analysis is highly efficient for most benchmarks. Furthermore, the modular approach ensures that all benchmarks are analyzed even faster, which is promising for, for example, IDE integration.

8 RELATED WORK

As discussed in the introduction, multiple studies show how JavaScript libraries are being used extensively, and how security vulnerabilities in such libraries cause serious problems for the applications [4, 5, 14, 17, 29, 33–36]. In particular, Zapata et al. [34] conclude that security scanning based on package dependencies considerably overestimates the implications of security vulnerabilities in libraries, and they suggest that many false positives may be avoided by performing analysis at the function level, however, they do not present such an analysis.

The dynamic call graph generators by Herczeg et al. [11] and Hejderup et al. [10] have been developed for JavaScript security scanning and function-level dependency management, but unlike our approach they require high-coverage test suites to avoid missing security issues.

Präzi [9] is an approach to reason about package dependencies that resembles Jam by relying on statically computed call graphs, but it is developed for Rust, not JavaScript. Eclipse Steady [27] is a similar approach for Java. It has recently been adapted to JavaScript [3], however, that work uses a simple program analysis that ignores most JavaScript language constructs. Mininode [15] is a tool for reducing the attack surface of Node.js applications by removing unused code. It includes a form of call graph construction, but it is unclear how it works for the dynamic features of JavaScript.

Multiple static analyzers already exist for JavaScript [6, 12, 13, 18, 20, 21]. Although they can in principle produce call graphs, none of these analyzers have been designed for the modular structure and heavy reuse of libraries in Node.js applications. Moreover, the light-weight static analyzers (e.g., [6, 20, 21]) are fast but tend to miss many call edges, whereas abstract-interpretation-based analyzers (e.g., TAJS [13] and SAFE [18]) do not yet scale to real-world Node.js applications. The study by Antal et al. [2] compares different static call graph construction tools for JavaScript, with very limited success for Node.js applications.

As explained in Section 5, a key component of Jam is the field-based analysis inspired by Feldthaus et al. [6], extended with access paths [22, 23] to enable modular reasoning. The experimental results with Jam and *js-callgraph* demonstrate the advantages of the modular analysis. The modular approach of Jam is inspired by componential analysis [7], which also as a first step computes summaries for modules (Scheme program components) and then combines the summaries to obtain the analysis result for the full program. As discussed in Section 5, Jam is designed to reach a useful compromise between precision, recall, and efficiency [19]. Although it is theoretically unsound, no security issues are missed in the experiments described in Section 7.

Another approach to security scanning is taint analysis, which not only considers the call graph but also the dataflow, and can thereby in principle safely dismiss some security warnings as harmless. The Nodest analyzer [25] extends TAJS with taint analysis and circumvents the scalability problem by attempting to avoid analyzing irrelevant modules, but still is orders of magnitude slower than Jam. Staicu et al. [30] also discuss the problem with package-dependency-level security scanning and propose a dynamic analysis to infer taint summaries for libraries. Such taint summaries can be used with, for example, the static analyzer LGTM,[23] which is designed to minimize the amount of false positives, at the cost of missing true positives, in contrast to Jam.

Change impact analysis is closely related to security scanning. Existing change impact analysis tools for JavaScript [1, 8] are designed for browser-based applications, not for reasoning about dependencies between modules in Node.js applications.

9 CONCLUSION

We have presented Jam, a modular call graph construction analysis that scales for Node.js applications, and we have shown how the produced call graphs can be used for security scanning. Due to Jam's modular design, call graphs can be computed for libraries and reused when computing call graphs for an application, and thereby scale for applications with complex dependencies.

Jam is designed to strike a balance between analysis precision (not producing an overwhelming amount of spurious call edges),

[23] https://lgtm.com/

recall (detecting almost all call edges that appear in concrete executions), and efficiency (analyzing real-world applications with thousands of functions within seconds).

We have shown experimentally on 12 Node.js applications that security scanning on the call graphs produced by Jam reports all true positive security warnings and reduces the number of false positives by 81% compared to the package-based security scanner *npm audit*. The analysis time for Jam using the modular approach is less than a second on average for our benchmarks, indicating that Jam is practically useful. Future work involves exploring more uses of the call graphs, for instance, change impact analysis as well as code navigation and refactoring in IDEs.

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).

REFERENCES

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2015. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 321–345. https://doi.org/10.4230/LIPIcs.ECOOP.2015.321

[2] Gabor Antal, Péter Hegedüs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static JavaScript Call Graphs: A Comparative Study. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 177–186. https://doi.org/10.1109/SCAM.2018.00028

[3] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2020. Code-based Vulnerability Detection in Node.js Applications: How far are we? *CoRR* abs/2008.04568 (2020). arXiv:2008.04568

[4] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 404–414. https://doi.org/10.1109/ICSME.2018.00050

[5] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 181–191. https://doi.org/10.1145/3196398.3196401

[6] 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*. IEEE Computer Society, 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[7] Cormac Flanagan and Matthias Felleisen. 1999. Componential Set-Based Analysis. *ACM Trans. Program. Lang. Syst.* 21, 2 (1999), 370–416. https://doi.org/10.1145/316686.316703

[8] Quinn Hanam, Ali Mesbah, and Reid Holmes. 2019. Aiding Code Change Understanding with Semantic Change Impact Analysis. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 202–212. https://doi.org/10.1109/ICSME.2019.00031

[9] Joseph Hejderup, Moritz Beller, and Georgios Gousios. 2018. *Präzi: From Package-based to Precise Call-based Dependency Network Analyses.* Working Paper. TU Delft.

[10] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 101–104. https://doi.org/10.1145/3183399.3183417

[11] Zoltán Herczeg, Gábor Lóki, and Ákos Kiss. 2019. Towards the Efficient Use of Dynamic Call Graph Generators of Node.js Applications. In *Evaluation of Novel Approaches to Software Engineering - 14th International Conference, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019, Revised Selected Papers (Communications in Computer and Information Science, Vol. 1172)*. Springer, 286–302. https://doi.org/10.1007/978-3-030-40223-5_14

[12] IBM Research. 2018. *T.J. Watson Libraries for Analysis (WALA).*

[13] 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 (Lecture Notes in Computer Science, Vol. 5673)*. Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

[14] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 102–112. https://doi.org/10.1109/MSR.2017.55

[15] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[16] 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*. IEEE / ACM, 83–93. https://doi.org/10.1109/ICSE.2019.00026

[17] 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.

[18] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proc. International Workshop on Foundations of Object Oriented Languages*.

[19] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[20] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. 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, 499–509. https://doi.org/10.1145/2491411.2491417

[21] Magnus Madsen, Frank Tip, and Ondrej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 505–519. https://doi.org/10.1145/2814270.2814272

[22] 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)*. 7:1–7:24.

[23] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 187:1–187:25. https://doi.org/10.1145/3428255

[24] 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.

[25] 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*. ACM, 455–465. https://doi.org/10.1145/3338906.3338933

[26] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 735–756. https://doi.org/10.4230/LIPIcs.ECOOP.2015.735

[27] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* (2020). https://doi.org/10.1007/s10664-020-09830-x

[28] Barbara G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Trans. Software Eng.* 5, 3 (1979), 216–226. https://doi.org/10.1109/TSE.1979.234183

[29] 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.

[30] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting Taint Specifications for JavaScript Libraries. In *Proc. 42nd International Conference on Software Engineering (ICSE)*.

[31] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static analysis with demand-driven value refinement. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 140:1–140:29. https://doi.org/10.1145/3360566

[32] 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*. ACM, 196–206. https://doi.org/10.1145/3178372.3179527

[33] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 351–361. https://doi.org/10.1145/2901739.2901743

[34] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 559–563. https://doi.org/10.1109/ICSME.2018.00067

[35] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. 2019. On the Impact of Outdated and Vulnerable JavaScript Packages in Docker Images. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 619–623. https://doi.org/10.1109/SANER.2019.8667984

[36] 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.