This is the submitted manuscript (pre-print version) of the article.
This is the version that represents the main opportunity for researchers to get input with regard to corrections and additions, before the peer-review process.

How to cite this publication
Please cite the final published version:


Publication metadata

Title: Modular call graph construction for security scanning of Node.js applications
Author(s): Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller
DOI/Link: https://doi.org/10.1145/3460319.3464836
Document version: Submitted manuscript (pre-print)
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.

KEYWORDS
static analysis, JavaScript, modularity

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 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

2https://dependabot.com/
3https://docs.npmjs.com/cli/audit
4https://snyk.io/
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 $G_{BC}$ 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 $G_{BC}$ 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, 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 scan 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 compared to modular call graphs. Moreover, compared to the state-of-the-art call graph construction tool js-callgraph, 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 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 vulnerability, and some appear through several dependency chains. For example, a prototype pollution vulnerability 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. 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.

```
write/node_modules/minimatch/minimatch.js:74:9
write/node_modules/minimatch/minimatch.js:114:0
write/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 1379 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 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:
1 require("./lib1.js");
2 console.log(lib1.filter([1, 2, 3]));
```
client1.js:

```javascript
11 const filter = require('./lib1.js').filter;
12 console.log(filter(x => x * x == 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. 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 static 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.

### Example 2

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

```javascript
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;
```

```javascript
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 Feldhaus 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, property names, variable names (including parameters), and program expressions, respectively. Given a single JavaScript file `f` as input, we compute a module summary `bf` consisting of three separate pieces of information:

- `bf.calls`: `Loc → P(Var)` 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.
- `bf.returns`: `Loc → P(Var)` is a function return summary, which for each function definition similarly describes its possible return values.
- `bf.props`: `Prop → P(Var)` 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 Mezetti 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')`.
- `Fun(f, l)` denotes a function definition in file `f` at line `l`.
- `Fun(f, l).Param[i]` denotes the `i`th parameter of the function definition in file `f` at line `l`.

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

11Source 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).

12We use the terms module and file interchangeably since a module is always stored in a single file in Node.js.
analyzing, for example, that \(x\) paths for the expressions aliased by the variable. A parameter is recursively computing the access paths for the sub-expression \(E.P\). For a property read

As an example, the access path \(\text{Fun(lib1,1)}.\text{Param[0]}\) describes the iteratee parameter of the \(\text{filter}\) 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

\[\text{Alias}_f : (\text{Var} \cup \text{Prop}) \rightarrow \mathcal{P}(\text{Exp})\]

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

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 \(\text{Alias}_f(X)\) or \(\text{Alias}_f(P)\), respectively. Transitive dataflow is taken into account later when the alias information is used.

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

\[\text{AccPaths}_f : \text{Exp} \rightarrow \mathcal{P}(\text{AccessPath})\]

The map is computed by \(\text{AccPaths}_f(E) = AP_{f}(E)\) for each expression \(E\) in \(f\), where \(AP\) is defined in Figure 2. The subscript \(V\) in \(AP\) and in the \(\text{lookup}\) auxiliary function ensures termination for recursions 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 \(\text{lookup}\) function to compute the access paths of the expressions that the alias analysis has determined to be aliased by \(P\).

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 \(\text{lookup}\) 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

### Summary Construction

The function call summary \(\beta_f.call\) is formed by grouping the access paths \(\text{AccPaths}_f(E)\) 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 \(\text{Fun}\) to refer to the function that wraps the analyzed file \(f\). Similarly, \(\langle f, \text{Main}\rangle\) denotes the location of that function.

The function return summary \(\beta_f.return\) is similarly computed by grouping the access paths assigned to the expressions of return statements in the function.

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

Example 3: Continuing Example 1, we obtain the module summaries \(\beta_{\text{client1}}\) and \(\beta_{\text{lib1}}\). Since the \(\text{filter}\) function is called in the outermost scope of the \text{client1.js} file, the call of \(\text{filter}\) is recorded as follows:

\[\beta_{\text{lib1}}.\text{calls}((\text{client1}, \text{Main})) = \{\langle \text{lib1}, \text{filter} \rangle, \ldots\} \]

Furthermore, the return summary of the \(\text{filter}\) function records the access path of the function returned:

\[\beta_{\text{lib1}}.\text{returns}(\langle \text{lib1}, 1 \rangle) = \{\langle \text{fun} \rangle, \text{lib1} \}\}

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

\[\beta_{\text{lib2}}.\text{props}(\text{sum}) = \{\langle \text{fun} \rangle, \text{lib2, 14} \}\]

\[\beta_{\text{lib2}}.\text{props}(\text{mul}) = \{\langle \text{fun} \rangle, \text{lib2, 15} \}\]

### 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, \(\beta\) calls (omitting the module name) denotes the combined call summary and is computed by \(\beta.\text{calls} = \bigcup_{E \in M} \beta_f.\text{calls}(E)\) for all \(E \in \text{Loc}\) 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 \(G = (V, E, \beta, \alpha)\) with nodes \(V \subseteq \text{Loc}\) corresponding to function definitions and edges \(E \subseteq \text{Loc} \times \text{Loc} \times \text{AccessPath}\) that represent the call edges. \(\beta\) is the combined module summary, and \(\alpha\) 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.
We explain in Examples 6 and 7 how these entries are produced.

Such a fixed point is guaranteed to exist since $E$ and $\alpha$ are the same as or depends directly or transitively on 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 $B$, 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., $ap = \langle m, () \rangle$, then the default exported function is extracted from the object property summary using the property name exports.

### Access Path Computation

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

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

- **Module Call** connects the access path $\langle m \rangle \ldots g$, 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 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 $B$, 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., $ap = \langle m, () \rangle$, then the default exported function is extracted from the object property summary using the property name exports.

### Example 6

The call to $\text{filter}$ on line 12 marked with red parentheses in Example 1 gives rise to the following entries in $\alpha$ and $E$:

```
\langle \text{lib1}, \text{filter} \rangle \rightsquigarrow \langle \text{lib1}, 1 \rangle
```

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

The next three constraint rules presented in Figure 3 model calls to functions returned by other functions ($\text{RETURN-CALL}$), calls to function parameters ($\text{PARAM-CALL}$), and calls to functions stored in object properties ($\text{PROP-CALL}$). These rules are applied iteratively since, for example, for the expression $E(O)$, 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., $\ldots \sim ap'\ldots$, then $ap'(\ldots)$ may obtain its function values from the return values of $ap$ functions. Those functions are retrieved using $ap \sim \text{Fun}(f,l)$. In the special case where $ap$ denotes a function definition $ap = \text{Fun}(f,l)$, we use the return values of that function directly. We retrieve the access paths $ap'$ representing return values using the function return summary $ap'' \in \beta_f(\text{return}((f,l)))$.

- **PARAM-CALL** ensures that if $ap''$ represents a call to a function value that comes from the $n$th parameter of a function $\text{Fun}(f,l)$, i.e., $\ldots ap'' \sim \text{Fun}(f,l), \text{Param}[n]$, then the function values of that

---

16It 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.
We use access path edge to together calls from different call sites to the same function. The access paths of the call sites are retrieved from the function call summary of some function and the possible function values of the argument of the call access path are in the object property summary for the argument of the call access path. We require the access path ends with an edge which finally by \( \text{EDGE} \) ensures that an edge is added in \( E \) between the caller and the callee:

\[
\begin{align*}
\langle \text{client1}, \text{Main} \rangle & \sim \langle \text{lib1}, \text{filter} \rangle \\
\langle \text{client1}, \text{Main} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle.
\end{align*}
\]

Furthermore, we saw in Example 3 that the return summary contains the fact that \( \text{filter} \) returns the function \( \text{Fun} \), and if the \( \text{MODULE-CALL} \) rule, we also have

\[
\begin{align*}
\langle \text{lib1}, \text{filter} \rangle & \sim \langle \text{lib1}, \text{filter} \rangle \\
\langle \text{lib1}, \text{filter} \rangle & \sim \langle \text{lib1}, \text{filter} \rangle.
\end{align*}
\]

Example 9 Consider the call to \( \text{iteratee} \) colored brown on line 5 in Example 1. The access path is \( \text{Fun} \langle \text{lib1}, 1 \rangle \). The call site is in the function call summary of some function and the possible function values \( \text{ap} \) include those found in the object property summary for \( \text{ap} \). We use \( \text{arg} \) for this argument of the call access path.

\[
\begin{align*}
\langle \text{client1}, \text{Main} \rangle & \sim \langle \text{lib1}, \text{filter} \rangle \\
\langle \text{client1}, \text{Main} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle.
\end{align*}
\]

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

\[
\begin{align*}
\langle \text{lib1}, 1 \rangle & \sim \langle \text{lib1}, 1 \rangle \\
\langle \text{lib1}, 1 \rangle & \sim \langle \text{lib1}, 1 \rangle.
\end{align*}
\]

which by \( \text{EDGE} \) results in the call edge:

\[
\begin{align*}
\langle \text{lib1}, 1 \rangle & \sim \langle \text{lib1}, 1 \rangle \\
\langle \text{lib1}, 1 \rangle & \sim \langle \text{lib1}, 1 \rangle.
\end{align*}
\]

Example 10 Consider the call to \( \text{sum} \) on line 20 in Example 2. From \( \beta_{\text{client2}} \cdot \text{calls} \) we have that the access path of \( \text{sum} \) is \( \langle \text{lib2}, \text{Arit} \rangle \). Again, the rule \( \text{OTHER-CALL} \) applies:

\[
\begin{align*}
\langle \text{lib2}, \text{Arit} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle \\
\langle \text{lib2}, \text{Arit} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle.
\end{align*}
\]

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

\[
\begin{align*}
\langle \text{lib2}, \text{Arit} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle \\
\langle \text{lib2}, \text{Arit} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle.
\end{align*}
\]

By \( \text{EDGE} \) this results in the following call edge:

\[
\begin{align*}
\langle \text{client2}, \text{Main} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle \\
\langle \text{client2}, \text{Main} \rangle & \sim \langle \text{lib2}, \text{Arit} \rangle.
\end{align*}
\]
Example 11  Let us consider an example program that requires multiple applications of \textit{return-call} and an application of \textit{transitive}, so that we can see how more complex calls are resolved.

\begin{verbatim}
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 }
\end{verbatim}

\begin{verbatim}
client3.js:
34 const lib = require('lib3');
35 const x = lib.h();
36 x();
\end{verbatim}

Let us consider an example program that requires multiple applications of \textit{return-call} and an application of \textit{transitive}, so that we can see how more complex calls are resolved.

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

Let us first consider the call to \texttt{h} ((\texttt{lib}.\texttt{h})). Since \texttt{Fun(lib3, 31) ∈ \beta_{\texttt{lib3}}-\texttt{props}(\texttt{h})}, we have by the \textit{module-call} rule:

\[ \langle \text{lib3}.h \rangle \sim \text{Fun(lib3, 31)} \]

By the \textit{edge} rule, we then have:

\[ \langle \text{client3}.Main \rangle \sim \langle \text{lib3}.h \rangle \]

We now consider how the analysis resolves \texttt{lib.h()} to \texttt{f}. By the \textit{other-call} rule we have:

\[ \langle \text{lib3}.h \rangle \sim \langle \text{lib3}.h() \rangle \]

From the resolution of the \texttt{lib.h()} call above, we have:

\[ \langle \text{lib3}.h \rangle \sim \text{Fun(lib3, 31)} \]

and during the module summary construction we have recorded access paths representing return values of \texttt{h}:

\[ \text{Fun(lib3, 27)}() ∈ \beta_{\text{returns}}(\langle \text{lib3}, 31 \rangle) \]

The \textit{return-call} rule then applies:

\[ \langle \text{lib3}.h() \rangle \sim \text{Fun(lib3, 27)}() \]

and by a second application of the \textit{return-call} rule:

\[ \text{Fun(lib3, 27)}() \sim \langle \text{lib3}.h() \rangle \text{Fun(lib3, 23)} \]

Finally, by the \textit{edge} rule, an edge is added from the main function of \texttt{client3} to \texttt{f} in \texttt{lib3}:

\[ \langle \text{client3}.Main \rangle \sim \langle \text{lib3}, 23 \rangle \]

The last call remaining is the call at line 36. For resolving that call we first apply the \textit{other-call} rule:

\[ \langle \text{lib3}.h() \rangle \sim \langle \text{lib3}.h() \rangle \sim \langle \text{lib3}.h() \rangle \]

Next, we apply the \textit{transitive} rule, based on the entries added to \texttt{a} by the two applications of the \textit{return-call} rule for the second call on line 35:

\[ \langle \text{lib3}.h() \rangle \sim \langle \text{lib3}.h() \rangle \sim \text{Fun(lib3, 23)} \]

From the function return summary we have

\[ \text{Fun(lib3, 21)} ∈ \beta_{\text{returns}}(\langle \text{lib3}, 23 \rangle) \]

and finally, by the \textit{return-call} rule:

\[ \langle \text{lib3}.h() \rangle \sim \text{Fun(lib3, 21)} \]

The \textit{edge} rule then adds an edge from the main function of \texttt{client3} to \texttt{e} in \texttt{lib3}:

\[ \langle \text{client3}.Main \rangle \sim \langle \text{lib3}.h() \rangle \sim \langle \text{lib3}, 21 \rangle \]

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 \(G_{BC}\) 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 \(a\), and \(E\) relations from \(G_{BC}\). The analysis then compiles 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 \(G_{BC}\) represents a partial result of \(G_{A_1, BC}\), we can compute \(G_{A_1, BC}\) faster with \(G_{BC}\) 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 \(ap" \sim ap\cdot q\) from Figure 3 also adds \(ap\cdot q ap"\) for any access path \(ap"\) that represents a callback to a built-in function \(q\). For example, if \(q\) is \texttt{map}, the analysis adds the above entry to \(a\) for each \(ap"\) in \(x[0]\) since \texttt{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 \(\beta_{\text{getters}}\) and \(\beta_{\text{setters}}\). For each property \(q\) that is read or written, we add \(\text{Fun(f, l)} \sim ap\) to \(a\) for each \(ap \in \beta_{\text{getters}}(q)\) for getters and \(ap \in \beta_{\text{setters}}(q)\) for setters.

We also added a special mechanism for handling the Node.js event system where events are registered using an \texttt{on} method and emitted using an \texttt{emit} method. Each module summary is augmented with an \texttt{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 \texttt{on} with two arguments where the first argument is a string, the map is extended with the access paths of the second argument. At \texttt{emit} calls, the corresponding access paths
We have not found many usages of dynamic module loads, and which nodes represent vulnerable functions. We describe known to use the call graph for security scanning, the analysis has to know

\( \text{VulnDesc} ::= (\text{AdvisoryID}, \text{PackageName}, \text{VersionRange}, \text{API-Pattern}) \)

\( \text{API-Pattern} ::= \{ \text{API-Pattern}, \ldots, \text{API-Pattern}\} \)

\( | < \text{ImportPath} > \)

\( | \text{API-Pattern} \cdot \text{Prop} \)

\( | \text{API-Pattern} \cdot O \)

Figure 4: API patterns.

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 \text{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 \( \beta.\text{props}(q) \) in \text{PROP-CALL} only considers the object property summary from the packages that are direct dependencies or direct dependents to the package with the callee. 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 \text{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 \text{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 \text{findNodes} (see Figure 5) to convert these patterns to source locations.\(^{18}\)

The grammar of the pattern language is shown in Figure 4. A vulnerability description (\text{VulnDesc}) consists of the advisory ID, the name of the package affected by the vulnerability, the range of affected versions, and an \text{API-Pattern} identifying the vulnerable parts of the library API. \text{AdvisoryID} is a number, and Module\text{leName} and \text{VersionRange} are strings. An \text{API-Pattern} can express disjunctions (\{\text{API-Pattern}, \ldots, \text{API-Pattern}\}), values obtained from loading modules (\(< \text{ImportPath} >\)), values read from properties (\text{API-Pattern} \cdot \text{Prop}), and return values of functions (\text{API-Pattern} \cdot O ). 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 \text{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, \( p = \{ p_1, \ldots, p_n \} \), as the union of the results of calling \text{findNodes} for each subpattern. For property read sequences on a module object without any calls, \( p = < m > \ldots q \), the function source locations are extracted similarly to the \text{MODULE-CALL} rule of Figure 3 (this rule also applies when the module object is read directly, i.e., \( p = < m > \), in which case the special \text{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 \( \beta.\text{props} \) and then the concrete source locations of these access paths from \( a \). For calls to returned values, \( p = p'()' \), the source locations represented by \( p' \) are extracted by calling \text{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 \( a \) (similar to the \text{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.

7 EVALUATION

We have implemented \text{JAM} (including the security scanner) in 3 000 lines of TypeScript code, using \text{acorn}\(^{20}\) and \text{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 \text{JAM}, compared to the \text{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 \text{JAM}, and how do they compare to call graphs computed by the existing tool \text{js-callgraph}?

\(^{17}\)\(https://www.npmjs.com/advisories\)

\(^{18}\)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.

\(^{20}\)\(https://www.npmjs.com/package/acorn\)

\(^{21}\)\(https://www.npmjs.com/package/acorn-walk\)
For these precision and recall measurements we dis-
tangle the module connectivity in 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 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).

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 `touch` application to around 23 seconds for `jwtoneify`, and js-callgraph is orders of magnitude slower.

The relatively large time for `jwtoneify` is explained by a heavy usage of, for example, the `forEach` function from the lodash library.

\(^{22}\)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.

---

**Table 1: Experimental results for security scanning.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Functions</th>
<th>Modules</th>
<th>Packages</th>
<th>JAM TP</th>
<th>JAM FP</th>
<th>JAM TP</th>
<th>JAM FP</th>
<th>JAM FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeappicon 1.2.2</td>
<td>6,165 (628)</td>
<td>1,393 (44)</td>
<td>13 (13)</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>touch 0.0.1</td>
<td>6,479 (61)</td>
<td>1,560 (4)</td>
<td>25 (1)</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>spotify-terminal 0.1.2</td>
<td>8,259 (61)</td>
<td>783 (3)</td>
<td>106 (1)</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ragan-module 1.3.0</td>
<td>839 (589)</td>
<td>85 (79)</td>
<td>61 (53)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>npm-git-snapshot 0.1.1</td>
<td>898 (357)</td>
<td>120 (55)</td>
<td>41 (35)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nodetree 0.3</td>
<td>1,557 (143)</td>
<td>15 (7)</td>
<td>4 (4)</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jwtoneify 1.0.1</td>
<td>27,703 (3,762)</td>
<td>1,869 (201)</td>
<td>93 (55)</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>foxx-framework 0.3.6</td>
<td>4,334 (1,124)</td>
<td>261 (124)</td>
<td>68 (51)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>npmgenerate 0.0.1</td>
<td>1,638 (530)</td>
<td>266 (31)</td>
<td>23 (19)</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>smrti 1.0.3</td>
<td>1,228 (732)</td>
<td>121 (116)</td>
<td>64 (56)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>writex 1.0.4</td>
<td>4,177 (1,237)</td>
<td>187 (90)</td>
<td>53 (42)</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>openbadges-issuer 0.4.0</td>
<td>6,043 (670)</td>
<td>1,366 (133)</td>
<td>69 (37)</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>69,320 (9,894)</td>
<td>8,026 (887)</td>
<td>620 (367)</td>
<td>8</td>
<td>26</td>
<td>8</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2: Experimental results for call graph construction.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Precision</th>
<th>Recall</th>
<th>Time (full)</th>
<th>Time (modular)</th>
<th>js-callgraph Precision</th>
<th>js-callgraph Recall</th>
<th>js-callgraph Time (full)</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeappicon 1.2.2</td>
<td>86.05%</td>
<td>100.00%</td>
<td>2.04s</td>
<td>0.07s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>touch 0.0.1</td>
<td>92.08%</td>
<td>100.00%</td>
<td>0.69s</td>
<td>0.01s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>spotify-terminal 0.1.2</td>
<td>92.79%</td>
<td>100.00%</td>
<td>0.73s</td>
<td>0.01s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ragan-module 1.3.0</td>
<td>87.42%</td>
<td>98.95%</td>
<td>1.39s</td>
<td>0.04s</td>
<td>80.00%</td>
<td>1.39%</td>
<td>1.84s</td>
</tr>
<tr>
<td>npm-git-snapshot 0.1.1</td>
<td>82.45%</td>
<td>94.78%</td>
<td>0.99s</td>
<td>0.04s</td>
<td>45.86%</td>
<td>85.32%</td>
<td>10.22s</td>
</tr>
<tr>
<td>nodetree 0.3</td>
<td>70.65%</td>
<td>100.00%</td>
<td>0.87s</td>
<td>0.03s</td>
<td>-</td>
<td>0.00%</td>
<td>12.93s</td>
</tr>
<tr>
<td>jwtoneify 1.0.1</td>
<td>71.43%</td>
<td>98.18%</td>
<td>23.01s</td>
<td>0.70s</td>
<td>68.44%</td>
<td>62.24%</td>
<td>61.30s</td>
</tr>
<tr>
<td>foxx-framework 0.3.6</td>
<td>89.14%</td>
<td>99.41%</td>
<td>2.37s</td>
<td>0.27s</td>
<td>59.81%</td>
<td>56.55%</td>
<td>614.37s</td>
</tr>
<tr>
<td>npmgenerate 0.0.1</td>
<td>97.42%</td>
<td>100.00%</td>
<td>1.80s</td>
<td>0.08s</td>
<td>66.20%</td>
<td>63.53%</td>
<td>3.77s</td>
</tr>
<tr>
<td>smrti 1.0.3</td>
<td>80.80%</td>
<td>96.51%</td>
<td>1.65s</td>
<td>0.09s</td>
<td>52.14%</td>
<td>64.20%</td>
<td>1,450.70s</td>
</tr>
<tr>
<td>writex 1.0.4</td>
<td>86.07%</td>
<td>100.00%</td>
<td>2.56s</td>
<td>0.18s</td>
<td>40.08%</td>
<td>52.04%</td>
<td>147.54s</td>
</tr>
<tr>
<td>openbadges-issuer 0.4.0</td>
<td>75.85%</td>
<td>95.59%</td>
<td>2.63s</td>
<td>0.11s</td>
<td>-</td>
<td>0.00%</td>
<td>12.93s</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>84.33%</td>
<td>98.62%</td>
<td>3.39s</td>
<td>0.14s</td>
<td>58.64%</td>
<td>48.16%</td>
<td>287.83s</td>
</tr>
</tbody>
</table>
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., 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 Nodeast 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,21 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), 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.

21https://lgtm.com/
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