NODEST: Feedback-Driven Static Analysis of Node.js Applications

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

Behnaz Hassanshahi
Oracle Labs
Australia
behnaz.hassanshahi@oracle.com

François Gauthier
Oracle Labs
Australia
francois.gauthier@oracle.com

ABSTRACT

Node.js provides the ability to write JavaScript programs for the server-side and has become a popular language for developing web applications. Node.js allows direct access to the underlying filesystem, operating system resources, and databases, but does not provide any security mechanism such as sandboxing of untrusted code, and injection vulnerabilities are now commonly reported in Node.js modules. Existing static dataflow analysis techniques do not scale to Node.js applications to find injection vulnerabilities because small Node.js web applications typically depend on many third-party modules. We present a new feedback-driven static analysis that scales well to detect injection vulnerabilities in Node.js applications. The key idea behind our new technique is that not all third-party modules need to be analyzed to detect an injection vulnerability. Results of running our analysis, NODEST, on real-world Node.js applications show that the technique scales to large applications and finds previously known as well as new vulnerabilities. In particular, NODEST finds 63 true positive taint flows in a set of our benchmarks, whereas a state-of-the-art static analysis reports 3 only. Moreover, our analysis scales to Express, the most popular Node.js web framework, and reports non-trivial injection vulnerabilities.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

KEYWORDS

Program analysis, Static analysis, Taint analysis, Security analysis, Node.js, JavaScript

ACM Reference Format:
Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. NODEST: Feedback-Driven Static Analysis of Node.js Applications. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338933

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338933

1 INTRODUCTION

Node.js is a platform to run JavaScript on the server-side. Node.js applications consist of modules and are managed by the NPM package manager. In contrast to client-side JavaScript applications that run in browsers, Node.js allows direct access to filesystem, operating system resources and databases, but does not provide any security mechanism such as sandboxing of untrusted code.

To detect injection vulnerabilities, tracking user-controlled inputs is needed. This type of analysis is known as taint analysis, which is a popular analysis to detect flows of data from untrusted sources to security-sensitive sinks. Due to the dynamic nature of JavaScript, existing dataflow analyses for statically-typed languages, such as Java, are not suitable in practice. For example, it is possible to construct an over-approximate callgraph in a pre-analysis step to refine and scale a more precise and expensive analysis for Java [16, 30]. However, constructing callgraphs for JavaScript programs requires handling dynamic dispatches, which requires type inference, which itself requires a precise callgraph. Abstract interpretation techniques that are designed to more closely model the program semantics compared to previous dataflow techniques have shown to be more precise and suitable for analyzing dynamic languages such as JavaScript [19, 22, 24]. A taint analysis can be specified as a client analysis in an abstract interpretation framework to detect vulnerable dataflows.

Abstract interpretation is a static analysis technique that computes a sound overapproximation of all possible program behaviors [12]. Statically computing all possible behaviors of a program using concrete language semantics is known to be undecidable. Therefore, abstract interpretation frameworks overapproximate concrete values and operations with abstract values and operations. Existing state-of-the-art static analysis techniques for JavaScript are conducted as a whole-program analysis, precisely analyzing all reachable code from the main entry point of the program [19, 22, 24]. These analysis techniques often fail to scale because of the highly dynamic nature of the JavaScript language. This problem is exacerbated in Node.js applications because they often consist of many NPM modules. For instance, many of the web-based Node.js applications are built upon libraries such as Express [3], which relies on many other NPM modules, making a single dependency transitively depend on about **30** modules; with an estimated **12,000** lines of JavaScript code. The standard whole-program analysis techniques often get stuck in a hard-to-analyze NPM module in the early stages of the analysis and are not able to produce any useful results.

In abstract interpretation, fine-tuning analysis precision and scalability manually is infeasible. The key idea in this paper is to automatically determine which modules in a Node.js application

can be approximated[1] with a wide abstraction to improve scalability while preserving precision. By precisely analyzing a set of modules while approximating the rest, our analysis is able to scale to Node.js applications and detect injection vulnerabilities.

Our analysis tool, NODEST, performs feedback-driven static analysis that is carried out through several iterations. It uses TAJS [19], an abstract interpretation framework for JavaScript, as the underlying analysis in each iteration. At the start of the process, the set of modules that need to be analyzed precisely includes all modules in the Node.js application and excludes all third-party modules. During each iteration, it applies heuristics to determine any new modules that need to be added to this set, or existing modules that need to be deleted from the set, and uses this information as feedback to the next iteration. Third-party modules that are not in this set are not analyzed (i.e., they are approximated) and their side effects are ignored.

We evaluated the effectiveness of our technique using benchmarks in [13, 21, 32], and additional real-world Node.js applications. We also compared our technique with the whole-program analysis in TAJS [8], to understand if NODEST helps scaling the analysis to find injection vulnerabilities in real-world applications that were not analyzable before. Our results show that NODEST scales to those applications and finds not only the previously known vulnerabilities, but also previously unknown zero-day vulnerabilities. Moreover, it achieves high precision and reports few false positives.

In summary, this paper makes the following contributions:

- We present a feedback-driven static analysis that scales to real-world Node.js applications (Section 4).
- We extend our feedback-driven analysis with a static taint analysis, which enables us to find non-trivial injection vulnerabilities in Node.js applications (Section 5).
- We evaluate our feedback-driven static taint analysis on real-world Node.js applications and report injection vulnerabilities that would otherwise be missed by a whole-program taint analysis. Moreover, we report new vulnerabilities that are not reported by existing dynamic analyses (Section 6).

2 MOTIVATING EXAMPLE

To better understand the challenges explained in Section 1, consider the code-snippet in Listing 1. This example shows a simplified Node.js application based on Express [3] that is vulnerable to a NoSQL injection attack. NoSQL, is a common term for nonrelational databases, in which queries and data are represented in JavaScript Object Notation (JSON) format. The purpose of the code is to provide information of a patient by specifying the phone number of the patient. In this example, the attacker can craft a JSON object (instead of a phone number) and send it as input through an HTTP request such that the query at line 9 satisfies all patients, instead of only a patient with a specific phone number. The attacker can thereby access the records of all patients.

The application starts by importing built-in and third-party modules between lines 1 and 4. Lines 4 and 5 instantiate the Express framework, a web framework that provides HTTP utility methods and middleware to build Node.js web applications.

[1]Throughout the paper, approximating a module refers to approximating the return value of `require("m")`, which imports a module named `"m"`, without analyzing `"m"`.

```
1  var http = require('http');
2  var yaml = require("js-yaml");
3  var mongo = require("mongodb");
4  var express = require("express");
5  var app = express();
6  ...
7  app.get("/patients", function (req, res) {
8      ...
9      q = {"Mobile": req.query.val};
10     ...
11     mongo.collection.find(q, {}, function(e, r){...});
12 });
```

Listing 1: Excerpt of Ankimedrec, a vulnerable Node.js application that whole program analysis fails to analyze.

When an HTTP request (`req`) comes in, it is passed to the application from Express at line 7. HTTP requests can be controlled by the attacker, hence `req` is a taint source[2]. The `query.val` property of `req` is used at line 11 through `q`, to query the MongoDB NoSQL database[3] using the `mongodb` database driver, which is imported at line 3. This application is vulnerable because an attacker-controllable (tainted) value is directly passed to `mongo.collection.find` as an argument, allowing an attacker to access sensitive data of patients with no restrictions.

To analyze this program and find the vulnerable taint flow, a whole-program abstract interpretation, such as TAJS [19], can be used, analyzing all the imported modules precisely based on the designed abstract domain and abstract operations until it reaches a fixpoint. TAJS times out while analyzing `js-yaml` at line 2. Note, however, that it is not necessary to analyze this module precisely to find the NoSQL injection at line 11. On the other hand, if `express` at line 4 is not analyzed precisely, the analysis will not be able to reason about the taint flow from an incoming HTTP request to the sink, hence failing to find the NoSQL injection vulnerability. In Section 5, we revisit this example and show how our feedback-driven analysis is able to scale and find the taint flow by approximating modules such as `js-yaml` while precisely analyzing modules such as `express`.

3 BACKGROUND: STATIC ANALYSIS IN TAJS

Our work is based on TAJS, an abstract interpretation framework for JavaScript. We extend several components of the original whole-program analysis in TAJS to adapt it to our proposed feedback-driven approach. The analysis in TAJS is based on the monotone framework [20] and uses a fixpoint solver that depends on a worklist algorithm. The program being analyzed is represented as a control flow graph. The abstract domain used by the analysis simulates the ECMAScript specification and, in high level, provides a callgraph and an abstract state for each context and flow graph node.

Fig. 1 shows the simplified definitions of the abstract domain in TAJS that are useful for understanding the new extensions proposed by our approach. *AnalysisLattice* maps node and context pairs to abstract states and the fixpoint solver needs to reach a fixpoint in this lattice. An abstract state maps *object addresses* to abstract

[2]The taint source location is the allocation site for a `req` object in the `http` module.
[3]To simplify the example, we have not included the database driver setup steps.

$$
\begin{aligned}
P &: \textit{property names} \\
L &: \textit{object addresses} \\
N &: \textit{nodes} \\
C &: \textit{contexts} \\
\textit{AnalysisLattice} &= N \times C \rightarrow \textit{State} \\
\textit{State} &= L \rightarrow \textit{Obj} \\
\textit{Obj} &= P \rightarrow \textit{Value} \\
\textit{Value} &= \textit{Undef} \times \textit{Null} \times \textit{Bool} \times \textit{Num} \times \textit{String} \times \mathcal{P}(L)
\end{aligned}
$$

Figure 1: Parts of the basic abstract domain in TAJS.

objects that are maps from *property names* to abstract values. Abstract values are modeled by the lattice *Value*. The details of each kind of value is discussed in detail in [19].

TAJS performs static analysis using the worklist algorithm in Algorithm 1. The worklist algorithm iterates over node and context pairs until a fixed abstract state is found for each node and context pair. The analysis starts by adding the initial node and context to the worklist. An element is removed from the worklist and analyzed until there are no more elements left. After analyzing the element, the current state propagates to its successors. If the state at the successor changes by this propagation (`propagate`$((n', c))$ is true), we add the successor to the worklist.

Algorithm 1 Worklist algorithm in TAJS

```
1:  add the initial node and context (n, c) to the worklist
2:  while worklist not empty do
3:      remove node and context (n, c) from worklist
4:      analyze (n, c)
5:      for all successors n' of n do
6:          if propagate((n', c)) then
7:              Add (n', c) to worklist
8:          end if
9:      end for
10: end while
```

Modelling the module loader in TAJS. The original module loader in TAJS mimics the require mechanism in Node.js [4]. Given a module name and the location from which `require` is called, it finds the first matching file following a precedence order specified by the require mechanism and runs the abstract interpretation analysis in TAJS to analyze it precisely.

4 FEEDBACK-DRIVEN ANALYSIS

In a standard whole-program analysis such as TAJS [19], the whole program is analyzed with the same level of precision across all modules. However, the analysis can be tuned for certain modules that are more critical. To automatically determine which modules can be approximated with a wide abstraction to improve scalability while preserving precision, we define MS_P as a set of modules that are analyzed precisely, and MS_B as a blacklist of modules that are not allowed to be added to MS_P (e.g., modules that are hard to analyze, which can be known a priori or during the feedback-driven analysis). Both MS_P and MS_B can initially be specified by the user and they should be disjoint sets. By default MS_P and MS_B are empty sets. If MS_P contains all modules used by the application, the feedback-driven analysis will be equivalent to running the normal whole-program static analysis, and if MS_P is empty, we do not analyze any third-party modules. Determining the right MS_P and MS_B sets manually can be difficult. Therefore, we design a feedback-driven analysis to automatically update these sets.

Algorithm 2 Feedback-driven Analysis

```
1: Inputs: MS_P and MS_B
2: Results = NULL
3: while hasChanged(MS_P) do
4:     Results ← ExtendedTAJS(MS_P)
5:     MS_P, MS_B ←processAnalysisResults(Results,MS_P,MS_B)
6: end while
7: report(Results.TaintFlows)
```

Algorithm 2 shows the main feedback loop of our analysis. This algorithm takes MS_P and MS_B as inputs and reports taint flows. The loop continues until MS_P reaches a fixpoint. At each iteration, we run an extended version of the standard analysis in TAJS[4]. We extend the module loader to load only modules that belong to MS_P. At the end of each iteration, we process the analysis results by running the algorithm shown in Fig. 3 for each third-party module in the application dependency tree (including transitive dependencies). This algorithm performs heuristics to update MS_P and MS_B sets at the end of each iteration. If these sets are modified, we run the ExtendedTAJS analysis again using the updated MS_P.[5] Fig. 3 is described in detail in Section 4.2.

4.1 Extending the Static Analysis in TAJS

In this section, we describe our extensions to the abstract domain in TAJS, how the module loader is modified to approximate modules that are not in MS_P, and changes to the underlying analysis in TAJS to handle values that originate from approximated modules.

Abstract domain extension. Fig. 2 shows the extensions added to the abstract domain in TAJS. We extend abstract values and abstract objects with a *TaggingWrapper* set. *TaggingWrapper* consists of *Tagging*, which provides additional information about the origin of an abstract value. *Tagging* can have three values: (1) *Module*(`M`) specifies that the abstract value originates from module `M`; (2) *SideEffect*(`M`) specifies that module `M` might have caused side-effects on the abstract value (which have been unsoundly ignored because we ignore side-effects for approximated modules); and (3) *ImpreciseWrite*(`M`) specifies that an abstract value (v) is written in an imprecise dynamic property write, i.e., $o[p] = v$ where p is approximated due to not analyzing `M`.

Joining and propagating *TaggingWrappers*. *TaggingWrappers* are joined by set-union and all built-in models are updated to propagate *TaggingWrappers*.

[4] Feedback-driven analysis is applicable for other abstract interpretation frameworks such as SAFE [24].

[5] Note that the ExtendedTAJS analysis does not reuse analysis results from the previous iterations.

$$
\begin{aligned}
M &: \textit{Module name} \\
SL &: \textit{Source-code location} \\
\mathcal{B} &= \textit{true} \mid \textit{false} \\
\textit{Value}' &= \textit{Value} \times \mathcal{P}(\textit{TaggingWrapper}) \\
\textit{Obj}' &= \textit{Obj} \times \mathcal{P}(\textit{TaggingWrapper}) \\
\textit{TaggingWrapper} &= \textit{Tagging} \times \mathcal{B} \times \mathcal{B} \\
\textit{Tagging} &= \textit{Module}(M) \\
&\mid \textit{SideEffect}(M) \\
&\mid \textit{ImpreciseWrite}(M) \\
&\mid \textit{Taint}(SL)
\end{aligned}
$$

Figure 2: Extensions to the abstract domain in TAJS. Highlighted parts are taint extensions introduced in Section 5.

Updated module loader. Before loading a third-party module `m` (line 1 in Listing 2), the updated module loader checks if `m` is in MS_P. If `m` is in MS_P, it is analyzed, otherwise it is approximated with the *tagging*: *Module*(`m`). Note that Node.js built-in modules are always analyzed.

```
1  var m = require('m');
2  var obj = {};
3  m.f(function(g) {
4    obj.a = g;
5  });
6  obj.a();
7  // rest of application
```

Listing 2: Approximating callbacks.

Calls to approximated functions. To understand how the analysis handles the values passed to the modules that are not analyzed, consider the code-snippet in Listing 2. In this example, because `m` is not in MS_P, it is not analyzed, i.e., the value of the variable `m` is approximated. If we do not analyze the call to the callback function passed as argument to `m.f` at line 3, we get a definite type error[6] at line 6 (because `obj.a` is `undefined`) and the dataflow to the rest of the application will be missed. Therefore, the analysis analyzes the call to the callback function with an approximated argument (`g`) labeled with the same tag used for `m.f`, which is the same tag used for `m`[7]. Next, the approximated value is written to `obj.a` at line 4, and the type of `obj.a` at line 6 is resolved to a value of any type including function. As a result, the analysis is able to continue analyzing the rest of the application.

Next, we show how the feedback-driven analysis uses the new extension to the abstract domain to identify the modules that should be analyzed, i.e., included in MS_P.

[6] All abstracted executions end in a `TypeError`.
[7] *Module*(`m`)

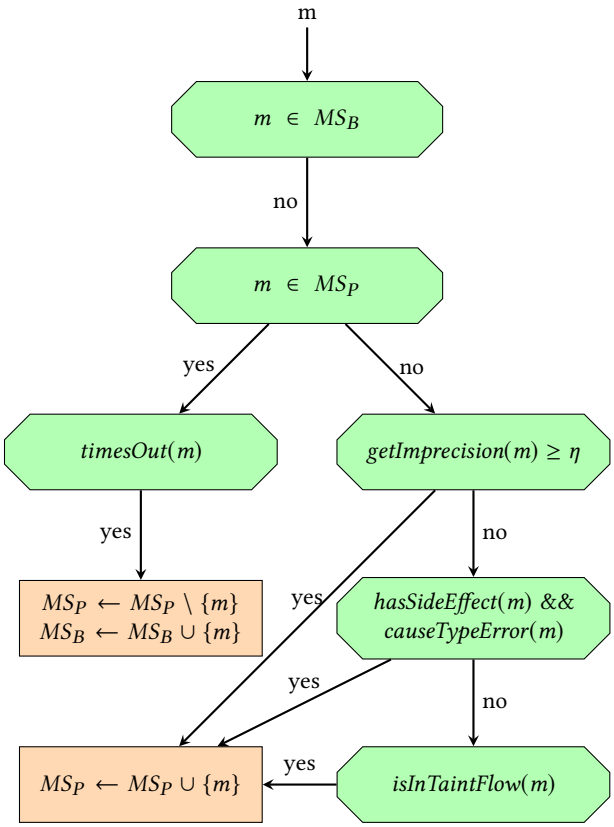

Figure 3: Flowchart for post-processing analysis results. Rectangular boxes indicate modifications to MS_B or MS_P, while the other boxes indicate predicates used by our heuristics. Post-processing terminates for an input module, *m*, when there is no transition to follow.

4.2 Post-processing Analysis Results

At the end of each iteration in Algorithm 2, we post-process the analysis results to determine whether the module sets MS_P and MS_B should be modified as depicted in Fig. 3. Recall that this is done for each third-party module in the application dependency tree (including transitive dependencies). Given a module `m`, if it is in MS_B, we end the post-processing step for this module because modules are never removed from MS_B. If `m` is not in MS_B, but in MS_P, we check if `m` has been too expensive to analyze. The predicate `timesOut(m)` is satisfied when `m` has taken a large fraction of the analysis time. In this case, `m` is moved from MS_P to MS_B.

If `m` is neither in MS_B nor in MS_P, we apply heuristics to determine whether `m` should be added to MS_P. The first heuristic is a precision heuristic which checks if not analyzing a module results in a large precision loss. In this heuristic, the function `getImprecision(m)` returns the number of source locations in which the *ImpreciseWrite*(`m`) tag (see Fig. 2) is read. If this number exceeds the preconfigured threshold η, `m` is added to MS_P. Otherwise, we apply our side-effect heuristic. The predicate `hasSideEffect(m) && causeTypeError(m)` checks whether a *SideEffect*(`m`) tag ends up in a

definite type error. If that is the case, we add `m` to MS_P. The last predicate (`isInTaintFlow(m)`) is related to the taint analysis, which is used as a client analysis in this paper, and is explained in Section 5.

To better understand the side-effect heuristic, consider the code-snippet in Listing 3, which uses the `setPrototypeOf` module to set the properties of the `router` object to the `proto` object. The program imports the module `setPrototypeOf`, defines a function and writes it to the `proto` variable from lines 2 to 8, and adds a property to `proto` at line 9. From lines 3 to 6, the function `router` is defined and the function `setPrototypeOf` is called with `router` and `proto` passed as arguments. This function call adds `proto` to the `router` object's prototype chain, which makes the properties of `proto` accessible through the `router` function object. In this example, the `router.handle` function called at line 4 is the function `proto.handle` defined at line 9. If the analysis fails to resolve `router.handle` correctly, it stops due to a definite type error at line 4.

Now we explain how our feedback-driven approach handles this example. Initially, the `setPrototypeOf` module is not included in MS_P. Because `setPrototypeof` is approximated, the analysis does not analyze it at line 6. Instead, it adds a side-effect tag[8], to the `router` and `proto` objects. The `router` function is returned at line 7 and because we do not overapproximate side-effects of `setPrototypeOf`, when the `router` function is called, the analysis cannot resolve `router.handle`, and the side-effect tag ends up in a definite type error. Therefore, `setPrototypeOf` is added to MS_P to be analyzed in the next iteration.

```
1  var setPrototypeOf = require('setPrototypeOf');
2  var proto = module.exports = function() {
3    function router(req, res, next) {
4      router.handle(req, res, next);
5    }
6    setPrototypeOf(router, proto);
7    return router;
8  };
9  proto.handle = function handle(req, res, next) {...}
```

Listing 3: Side-effect heuristic example.

Termination of feedback-driven analysis. Our feedback-driven analysis in Algorithm 2 is guaranteed to reach a fixpoint. Each iteration is guaranteed to terminate because the underlying analysis (ExtendedTAJS) is guaranteed to terminate and processAnalysisResults runs in linear time with respect to the modules used by the application. There is an upper bound for the number of iterations in the feedback loop in Algorithm 2 and because each iteration terminates, the entire algorithm is guaranteed to terminate. Note that MS_P is modified in each iteration, but a module is added to MS_P at most once during the entire feedback-driven analysis: when a module is moved from MS_P to MS_B, it remains in MS_B until the end of the analysis. Assuming that the number of third-party modules used by an application is n, at most n modules can be added to MS_P, resulting in at most n iterations. In iterations where no modules are added to MS_P (and MS_P has not reached a fixpoint), a module is moved to MS_B. Note that at most n modules can be added to MS_B, resulting in at most n extra iterations. Therefore, our feedback-driven analysis is guaranteed to terminate after at most $2n$ iterations.

[8] *SideEffect*(`setPrototypeOf`)

Algorithm 3 Optimized worklist algorithm in TAJS

```
1:  Input: θ
2:  add the initial node and context (n, c) to the worklist
3:  wlPhase = ORDINARY
4:  postponedWorklist = ∅
5:  visitationCounter = ∅
6:  while worklist not empty do
7:      remove node and context (n, c) from worklist
8:      if wlPhase == MAX-COV then
9:          if visitationCounter.get((n, c)) > θ then
10:             Add (n, c) to postponedWorklist
11:             continue
12:         end if
13:         visitationCounter.count((n, c))
14:     end if
15:     analyze (n, c)
16:     for all successors n′ of n do
17:         if propagate((n′, c)) then
18:             Add (n′, c) to worklist
19:         end if
20:     end for
21:     if worklist empty and wlPhase == MAX-COV then
22:         worklist = postponedWorklist
23:         Empty postponedWorklist
24:         wlPhase = ORDINARY
25:     else if wlPhase == ORDINARY and switchPhase() then
26:         visitationCounter.reset()
27:         wlPhase = MAX-COV
28:     end if
29: end while
```

4.3 Optimized Worklist Algorithm

To scale the static analysis to Node.js applications and increase code coverage, in addition to the feedback-driven analysis discussed in this section, we design an optimized worklist algorithm that aims to increase the coverage of worklist items before the analysis times out. This is done by introducing a new phase in the worklist algorithm, as shown in Algorithm 3. In this new phase, if a (n, c) pair is visited more than θ times, it is postponed to be processed in the ordinary worklist phase. When the worklist is empty, the ordinary worklist phase continues with the postponed worklist items.

The highlighted parts of Algorithm 3 show the new extensions to the standard worklist algorithm in Algorithm 1. The algorithm is performed in two phases: (1) `ORDINARY`, and (2) `MAX-COV`. The current phase is stored in `wlPhase`, which is initially set to `ORDINARY` (line 3). `postponedWorklist` (line 4) is a list containing worklist items that have been postponed during the `MAX-COV` phase. `visitationCounter` (line 5) is a map from (n, c) to an integer number, describing how many times (n, c) is visited in the current `MAX-COV` phase. During the `MAX-COV` phase, lines 9 to 13 make sure that a (n, c) pair is not analyzed more than θ times. Lines 21 to 28 switch between the

ORDINARY and MAX-COV phases: Lines 22 to 24 switch from MAX-COV to ORDINARY when the worklist is empty and postponedWorklist is assigned to the ordinary worklist to be processed later in the ORDINARY phase; Lines 26 to 27 switch from ORDINARY to MAX-COV based on the analysis execution time, making sure that multiple phases of MAX-COV are performed before the analysis times out.

5 STATIC TAINT ANALYSIS

In this section, we explain how our feedback-driven analysis can be extended to perform static taint analysis as a client analysis.

5.1 Incorporating Taint Analysis

To support taint analysis, we make slight modifications to the abstract domain (Section 4.1) and add one more heuristic to the post-processing algorithm (Fig. 3) in our feedback-driven analysis.

Abstract domain extensions. The highlighted parts of Fig. 2 show the modifications needed to support static taint analysis. *TaggingWrapper* is extended with two boolean flags: the first one indicates whether the value is tainted or not, and the second one indicates whether the value is a sink. To understand why we need these additional flags, consider the code: `var x = require("M").f(eval)`, where M is neither a taint source nor sink. The value returned from `require("M").f(eval)` might be a taint sink because `eval` is a sink. However, the tag present in the return value would still be *Module*(M). Therefore, by adding these flags we can precisely distinguish which abstract values originating from M might be taint sources or sinks. We also added the *Tagging* type, *Taint*(*SL*), specifying that the value is tainted by a taint source located at the source location *SL*.

Post-processing extension. To make sure we do not miss any taint flows because of not analyzing a module in the feedback-driven approach, we use the last heuristic (taint heuristic) in the post-processing step as shown in Fig. 3. The taint heuristic adds a module `m` to MS_P when `isInTaintFlow(m)` predicate holds. Intuitively, we add `m` to MS_P if it is either the source or the sink in a taint flow. Therefore, the predicate holds in two cases: (1) `m` is the *sink*: a tainted value flows to a sink with the tag *Module*(m), or (2) `m` is the *source*: a tainted value with the tag *Module*(m) flows to a sink and the sink is not approximated due to not analyzing a different module.

5.2 Taint Analysis Configuration

We perform a syntactic analysis that identifies sources and sinks in the application and third-party modules using method signatures. The source and sink definitions are provided as configurations by the user. For instance, if `eval` is marked as a sink, the syntactic analysis looks for occurrences of `eval` in the source-code files. It analyzes all files that are reachable through `require` function calls, where the argument is a constant string. If the syntactic analysis does not find any taint sources or sinks in a module `m` or its dependencies, loading module `m` will yield an approximated value tagged with (*Module*(m), *false*, *false*), which indicates that the approximated value is neither a taint source nor a sink.

5.3 Revisiting the Motivating Example

In this section, we show how our feedback-driven analysis extended with taint analysis finds the taint flow in Listing 1. In this example, our syntactic analysis marks the `http` request object allocated through the `express` module as source and `mongodb` as sink. Initially, MS_P is empty, so third-party modules added between lines 1 to 4 are approximated, and MS_B contains `mongodb`.

In the first iteration of the analysis, `express` is not analyzed, therefore `app` at line 5 is approximated. Because tainted data enters the application from the `express` module, it is labeled with a taint tag. At line 7, `app.get`, `req`, and `res` are all approximated and labeled with the same tag as `app`[9]. The `req` object flows to the variable `q`, ultimately reaching `mongo.collection.find`. This flow indicates that a tainted value has reached a sink. Because of the taint heuristic discussed earlier in this section, `express`, the approximated module from which the taint tag is originated, is added to MS_P.

In the next iteration of the analysis, MS_P contains `express`. To simplify this example, we skip the iterations where the analysis adjusts the MS_P and MS_B sets to analyze the `express` module. In the last iteration, MS_P contains the necessary modules to precisely identify that `req` is an `http` request object, so `req.query.val` is tainted. As `req.query.val` flows to `mongo.collection.find`, a taint flow is reported from an incoming http request to a NoSQL sink.

6 EVALUATION

We implemented our feedback-driven static taint analysis of Node.js applications in a tool called Nodest. To evaluate our technique we used a Windows 7 machine with an Intel Core i5-5300 CPU @ 2.3 GHz, and a JVM with 10GB memory. We use the benchmarks from [13, 21, 32], which are the existing program analysis frameworks for Node.js that detect injection vulnerabilities. We also analyze two more Express-based applications, `mongo-express` and `ankimedrec` which have not been analyzed by previous works. We have responsibly disclosed all the new vulnerabilities found by Nodest to the developers.

The benchmarks from [21, 32] have small test-drivers for npm modules/applications that exercise injection vulnerabilities. We use the test-drivers for some of these benchmarks in our evaluation. For applications that depend on networking libraries, such as the `http` module, we do not need the test-drivers because our analysis over-approximates all the incoming requests. Our evaluation answers the following three research questions:

- RQ1: Can static taint analysis detect taint flows in simple Node.js modules with high precision?
- RQ2: Is Nodest able to improve the scalability of a whole-program static taint analysis without missing any known taint flows?
- RQ3: How important is the optimized worklist algorithm for scaling static analysis of Node.js applications?

6.1 RQ1 - Precision

We will answer the first research question by comparing our static analysis with prior dynamic analysis works that detect taint flows. The static analysis in this experiment is the whole-program analysis in TAJS [8] extended with taint support. We use the benchmarks

[9] (*Module*(express), *true*, *false*)

Table 1: Results for whole-program analysis of module benchmarks. Analysis timeout is five minutes.

Module	No FN	No FP	Reached fixpoint
os-uptime	✓	✓	✓
chook-growl-reporter	✓	✓	✓
growl	✓	✓	✓
os-env	✓	✓	✓
fish	✓	✓	✓
mlog	✓	✓	✓
node-os-utils	✓	✓	✓
gm	✓	✓	✓
mongo-parse	✓	✓	✓
mongoosify	X	✓	X
printer	✓	✓	✓
kerb_request	✓	✓	✓
mixin-pro	✓	✓	✓
pidusage	✓	✓	✓
modulify	X	✓	X
system-locale	✓	✓	✓
mol-proto	X	✓	X
libnotify	✓	✓	✓
pomelo-monitor	✓	✓	✓
systeminformation	✓	✓	✓
node-wos	✓	✓	✓
git2json	✓	✓	✓
office-converter	✓	✓	✓
mongoosemask	✓	✓	✓
cocos-utils	✓	✓	✓

from [32] that are modules (not applications) for this experiment as well as the benchmarks used in [21]. All the benchmarks from [32] contain at least one known vulnerability. We exclude the modules in which exploiting the vulnerability requires interaction with the file-system. The [21] benchmarks contain both vulnerable and benign npms. The latter are used to test precision. Table 1 shows that NODEST finds the vulnerabilities in 22 out of 25 modules and does not reach a fixpoint in the remaining 3. The analysis does not report false positives in any of the 25 benchmarks. These results indicate that static analysis is suitable for detecting taint flows for Node.js modules, because the analysis reaches a fixpoint for almost all of these benchmarks and has high precision.

6.2 RQ2 - Scalability and Accuracy

To answer the second research question, we use those benchmarks from [32] that are applications, most of which depend on the Express [3] framework. We also test against `mongoosify` and `modulify`, which are two of the three modules that could not be analyzed by the whole-program analysis in Table 1. We do not include the third module because it has no dependencies. Furthermore, we evaluate our technique on one application from [13], `NodeGoat-v1.1`, and two new applications, `Ankimedrec` and `mongo-express`. All of these applications are based on Express [3] and use a MongoDB database [5].

For the Express-based applications, we have added a couple of source-code transformations to remove unsupported ES6 features [2]. Alternatively, we could have used an existing source-code transformation tool, such as Babel [1], to analyze such features. We chose not to invest time setting up Babel, since very few locations required transformations. Using Babel instead should not affect the analysis results. Moreover, we have annotated a couple of functions in Express to enable additional parameter-sensitivity [8]. Note that the underlying abstract interpretation analysis could be improved to avoid such transformations, however, the goal of this evaluation is to test the effectiveness of our feedback-driven analysis. Therefore, we use the underlying analysis as it is.

Feedback-driven analysis setup. The feedback-driven analysis presented in Section 4 relies on the following configurations: MS_P is initially empty except for `mongoosify` and `modulify`, for which MS_P is defined as {`mongoosify`} and {`modulify`} respectively, because they require test-drivers to exercise the injection vulnerabilities. MS_B is initialized with the following database modules (whose APIs are used as taint sinks): `mongodb`, `monk` and `sqlite3`. After the analysis runs for more than 50 seconds, the `timesOut(m)` predicate in Fig. 3 is triggered if a file in `m` spends more than 90% of the analysis time. For the precision predicate in Fig. 3 (`getImprecision(m) >=` η), we use $\eta = 20$.[10] We use $\theta = 2$ in Algorithm 3 and `switchPhase` triggers three times after 150, 800 and 1600 seconds. Note that these parameters have been chosen by one or two trial and errors, and they are not tuned to our benchmarks. We do not expect slight perturbations of these parameters to affect the analysis results significantly.

Feedback-driven analysis results. Table 2 shows the results of running NODEST with a 30-minute timeout for each analysis iteration. Note that the analysis might terminate before reaching the timeout if any of the post-processing heuristics are satisfied. "Identifying module sets" is the time spent to identify the module sets and "Analysis time" is the execution time for the last iteration, in which analysis is performed on the final version of module sets (module sets do not change in this iteration). "#Installed modules" is the number of third-party modules that are installed[11] during the installation of the application. $|MS_P|$ is the size of MS_P after reaching a fixpoint. "Eval", "Exec" and "NoSQL" indicate the number of true positive and false positive taint flows found for `eval`, `require("child_process").exec`, and NoSQL sinks provided by the database modules, respectively. We manually investigated the reported flows to determine whether they are true positives. However, the true positive taint flows might not be exploitable due to non-trivial sanitization.

NODEST is able to find taint flows in 10/11 benchmarks. We can also see that NODEST does not report any false positives for these benchmarks. Even though the analysis does not reach a fixpoint in the final iteration in 7 applications, it is still able to report true positive taint flows. As shown in this table, identifying module sets for `modulify` takes no time. The reason is that the taint flow in this application involves only its main module, which is included in MS_P initially, and the rest of the modules that are not analyzed are

[10] We have observed that lower thresholds can trigger at local precision losses (as compared to the precision loss spread throughout the application).
[11] We use `npm ls --prod` to count the number of unique modules.

Table 2: Results for feedback-driven analysis of Node.js modules/applications.

Application	Identifying module sets (hh:mm:ss)	Analysis time (mm:ss)	#Installed modules	$\|MS_P\|$	Eval TP/FP	Exec TP/FP	NoSQL TP/FP
NodeGoat	01:11:12	Times out	109	9	3/0	0/0	5/0
keepass-dmenu	00:09:49	00:54	13	1	0/0	1/0	0/0
ankimedrec	01:40:05	Times out	67	16	0/0	0/0	30/0
mongui	00:44:24	Times out	148	22	6/0	0/0	11/0
codem-transcode	00:01:00	Times out	31	3	0/0	1/0	0/0
Mock2easy	01:05:54	Times out	429	15	0/0	0/0	0/0
mongoosify	00:03:10	00:13	2	1	1/0	0/0	0/0
modulify	00:00:00	00:13	25	1	1/0	0/0	0/0
mongo_edit	00:20:50	Times out	25	9	1/0	0/0	0/0
mongo-express	01:04:36	Times out	103	19	0/0	0/0	2/0
mqtt-growl	00:05:01	21:22	74	5	0/0	1/0	0/0
Total	-	-	-	-	12/0	3/0	48/0

not affecting the taint flow. Therefore, the modules in MS_P do not need to change. Comparing the number of installed modules with the size of MS_P after reaching a fixpoint, we see that our feedback-driven analysis skips analyzing many modules, which makes our analysis more scalable. As an example, by installing `mongui`, 148 modules are installed, which are too many for a static analysis tool such as TAJS to scale. Nodest is able to identify 22 modules ($|MS_P|$) out of these 148 modules that are sufficient to detect the taint flows. Furthermore, there is no direct correlation between the time spent to identify module sets and the size of MS_P after reaching a fixpoint. The reason is that the duration of each iteration might vary a lot across different benchmarks.

Feedback-driven vs whole-program analysis. Next, we compare Nodest with the whole-program analysis in TAJS [19] on the same benchmarks to determine if the feedback-driven analysis improves the scalability without missing taint flows. Results for the whole-program analysis can be seen in Table 3. The timeout in this experiment is 30 minutes. This table shows that the whole-program analysis only reaches a fixpoint in one application (note that Nodest reaches a fixpoint in 4). We also observe that the whole-program analysis only finds taint flows in 4 applications, whereas Nodest finds taint flows in 10 applications. All the applications, except for `mqtt-growl`, in which the whole-program analysis finds taint flows has at most 31 installed modules, indicating that scalability of the whole-program analysis is correlated with the size of the application. The whole-program analysis finds a strict subset of the taint flows found by Nodest, which indicates that the feedback-driven analysis does not introduce any false negatives, even though it skips the analysis of some modules. Therefore, we conclude that our feedback-driven analysis is able to improve the scalability of the underlying static taint analysis without missing any known taint flows.

XSS injection vulnerabilities. Because some of our benchmarks are Express-based web applications, where reflected XSS[12] is common, we also conducted experiments to see if Nodest is able to detect XSS injection vulnerabilities. The taint source for XSS vulnerabilities is `http` request object and sinks are `http` response functions. In an Express application, sources are created in the framework and sinks are often used both in the framework and application. Therefore, Express applications are likely to have common taint flows. To distinguish such results, manual investigation is required. Below, we summarize our findings for this category of vulnerabilities.

[12]In a reflected XSS, attacker-controllable input that comes through an `HTTP` request is sent back to the client side through an `HTTP` response.

Table 3: Results for whole-program analysis.

Application	Analysis time (mm:ss)	Eval TP/FP	Exec TP/FP	NoSQL TP/FP
NodeGoat	Times out	0/0	0/0	0/0
keepass-dmenu	Times out	0/0	0/0	0/0
ankimedrec	Times out	0/0	0/0	0/0
mongui	Out of memory	0/0	0/0	0/0
codem-transcode	Times out	0/0	1/0	0/0
Mock2easy	Times out	0/0	0/0	0/0
mongoosify	Times out	0/0	0/0	0/0
modulify	Times out	1/0	0/0	0/0
mongo_edit	Times out	1/0	0/0	0/0
mongo-express	Times out	0/0	0/0	0/0
mqtt-growl	25:44	0/0	1/0	0/0
Total	-	2/0	2/0	0/0

Nodest reports XSS taint flows in six applications, which are all introduced by `express`. In these taint flows, the attacker-controllable input enters the application through `req.query.url` and is sent back to the client side through `res.end` if the request is invalid. However, the tainted value is sanitized, so it might not be exploitable. We also found other unique true positive XSS taint flows in four applications. Except for one application, the rest of the taint flows are not sanitized. Nodest reported false positive taint flows (infeasible dataflow) for only one application. The average time for inspecting and classifying the analysis results for XSS vulnerabilities was around 10 to 15 minutes.

New taint flow reports. Nodest is able to find new taint flows that are not reported by existing tools. It reports multiple `eval`

Table 4: Taint flows found using feedback-driven (FD) and whole-program (WP) analysis without optimized worklist.

	Eval		Exec		NoSQL	
Application	FD	WP	FD	WP	FD	WP
NodeGoat	3	0	0	0	4	0
keepass-dmenu	0	0	1	0	0	0
ankimedrec	0	0	0	0	0	0
mongui	0	0	0	0	0	0
codem-transcode	0	0	1	1	0	0
Mock2easy	0	0	0	0	0	0
mongoosify	1	0	0	0	0	0
modulify	1	0	0	0	0	0
mongo_edit	1	1	0	0	0	0
mongo-express	0	0	0	0	2	0
mqtt-growl	0	0	1	1	0	0
Total	6	1	3	2	6	0

vulnerabilities in `mongui`, which are not exercised and reported by existing dynamic analysis tools [13, 32]. In general, it is known that static analysis can achieve better coverage compared to dynamic analysis because the latter requires specific inputs at runtime that exercise these taint flows to be able to report them. Furthermore, all the NoSQL and XSS taint reports except for `NodeGoat`[13] are new and have not been reported before. Apart from Affogato [13], NODEST is the only tool that detects NoSQL and XSS taint flows.

6.3 RQ3 - Optimized Worklist Algorithm

To answer the third question, we performed the experiments in RQ2 without using the extended worklist algorithm (see Section 4.3). Table 4 shows the reported taint flows for the feedback-driven analysis (FD) and the whole-program analysis (WP), in both of which the optimized worklist algorithm is disabled. Note that the optimized worklist algorithm makes no difference in reaching a fixpoint in our benchmarks. As shown in Table 4, without using the optimized worklist, the feedback-driven analysis reports 15 taint flows (sum of FD columns) while the whole-program analysis reports 3 flows (sum of WP columns). On the other hand, using the optimized worklist helps the feedback-driven analysis to report 63 taint flows (see Table 2) while the whole-program analysis reports 4 flows in total (see Table 3). Therefore, we conclude that the optimized worklist algorithm improves the scalability of static taint analysis of Node.js applications by increasing coverage and reporting more true positive taint flows.

7 CASE STUDIES

In this section, we describe two case studies to show how our feedback-driven analysis is able to find non-trivial taint flows in complex Node.js applications. In the first case study, both the source and sink are in third-party modules, i.e., in the code not analyzed in the first iteration of the feedback-driven analysis (Section 7.1). The second case study shows an example of a complicated taint flow, which depends on multiple requests and a specific timing between these requests, which motivates the use of static instead of dynamic analysis (Section 7.2).

[13]The deliberately vulnerable application from OWASP [6].

```
1  var mqtt = require('mqtt')
2    , growl = require('growl')
3    , _ = require('underscore');
4  growl = _.throttle(growl);
5  mqtt.f(function (message) {
6    growl(message);
7  });
8
9  // underscore.js
10 _.throttle = function(func) {
11   return function() {
12     func.apply(this, arguments);
13   }
14 }
```

Listing 4: Simplified version of mqtt-growl.

7.1 Mqtt-growl

Listing 4 shows a simplified version of `mqtt-growl`, which has a remote code execution vulnerability (attacker controllable input reaches an `exec` sink). This code-snippet is simply importing three third-party modules from lines 1 to 3 (`mqtt`, `growl` and `underscore`) and the rest of the code does not have any sources or sinks, hence does not seem to be vulnerable in the first look. However, by combining these three modules, it enables an exploitable taint flow starting from a source in `mqtt`, passing through `underscore`, and finally reaching a sink in `growl`.

Our feedback-driven analysis goes through four iterations to identify all the modules that need to be analyzed and detect the vulnerable taint flow. Initially, all the three third-party modules from lines 1 to 3 are approximated (not analyzed). Note that even though `mqtt`, and hence `mqtt.f` from line 5 to 7 is approximated, the callback function passed as argument is called by the analysis and analyzed with approximated argument (`message`). Therefore, line 6 is reachable in all the iterations. The first iteration adds `underscore` to MS_P (the set of modules that need to be analyzed) because it has caused `growl(message)` at line 6 to be approximated while this function call is potentially part of a taint flow (see the taint heuristic in Section 5). The second iteration adds the `growl` module to MS_P because by analyzing `underscore`, the analysis reaches line 12 where `func` is approximated due to not analyzing `growl` and `arguments` is tainted. In the third iteration, the call to `func` at line 12 is resolved to the `growl` function object, and analyzed. Now, the tainted value passed to `growl` is approximated because of not analyzing `mqtt`, which includes the taint source (`message`). The tainted value flows to an `exec` sink and therefore, `mqtt` is identified to be potentially part of a taint flow and added to MS_P. Finally, in the fourth iteration, the analysis reports a taint flow from a source in `mqtt` to the `exec` function in `growl`.

7.2 Codem-transcode

`Codem-transcode` is a video transcoder, which receives requests through a simple HTTP API. The application has a XSS vulnerability, which depends on multiple requests and a specific timing

between these requests. Static analysis is suitable for finding this vulnerability because it overapproximates all possible orderings of incoming requests. NODEST is the first tool that reports this vulnerability. The application has a route for posting jobs (`POST /jobs`) and a route for getting all the jobs that are registered, but not completed (`GET /jobs`).

```
1  var slots = [];
2  // POST /jobs
3  postNewJob = function(request, response) {
4    var postData = "";
5    request.on('data', function(d) { postData += d; })
6    request.on('end', function() {
     ↪  processPostedJob(postData); } );
7  }
8  processPostedJob = function(postData) {
9    var job = Job.create(JSON.parse(postData));
10   slots.push(job);
11   ...
12 }
13 // GET /jobs
14 getJobs = function(request, response) {
15   var content = { jobs: slots };
16   response.end(JSON.stringify(content), 'utf8');
17 }
```

Listing 5: Simplified version of `Codem-transcode`.

A very simplified version of the implementation is shown in Listing 5. Lines 3 to 7 define the function that is called upon receiving a `POST` request to `/jobs` route. This function registers event listeners on the `request` object. It uses the `data` event to collect the data. Once all data is received (i.e., the `end` event is triggered), it calls `processPostedJob(postData)` to create a new job (`Job` object) for `postData`. Note that `postData` is tainted because the attacker can control it through HTTP requests. The code not shown in `processPostedJob` actually processes the job asynchronously and upon finishing the job, removes it from the `slots` array. Each `Job` object (that is processed but not finished yet) can be retrieved by a `GET` request to the `/jobs` route, which is handled through the function defined from line 14 to 17. As it can be seen, this function reflects back the tainted data to the client side, making the application vulnerable to an XSS attack. This example shows the advantage of using static analysis over dynamic analysis as the latter requires a test driver to trigger this behavior, while our analysis is able to find this taint flow by overapproximating the incoming requests and their orders.

8 RELATED WORK

Static analysis for JavaScript has a rich history, and different static analysis frameworks have been developed over the years. Because of its highly dynamic nature, however, the JavaScript language is very difficult to analyze both precisely *and* efficiently. In this work, we presented an approach to scale static analysis for JavaScript and enable precise taint analysis of Node.js applications.

Static analysis of JavaScript. In order to be practical, most, if not all static analysis frameworks for JavaScript [15, 19, 22, 24, 25] allow for precision and scalability trade-offs through context, field, heap, and loop sensitivity tuning [23, 27, 35]. When sensitivity tuning reaches its limit, however, auxiliary strategies have to be used to achieve acceptable precision and scalability. For example, the work by Madsen et al. [25] specifically addresses the issue of computing points-to analysis for JavaScript applications that depend on complex frameworks and libraries. To avoid the need for manually written *stubs*, their approach performs a use analysis that automatically infers points-to specifications. The work in [31] addresses the challenge of scaling points-to analysis for JavaScript through correlation tracking, a lightweight pre-analysis that identifies field reads and writes that must refer to the same property. Using correlation information, the subsequent points-to analysis avoids introducing spurious points-to edges, which leads to a sparser points-to graph.

In a constant quest to overcome the scalability challenges of JavaScript analysis, others have developed hybrid analyses, where the core static analysis uses dynamic information to restrict its search space, often at the cost of soundness. Wei and Ryder coined the term *blended analysis* to designate static analyses that use some dynamic analysis results as part of their computation. The work in [34] uses dynamic analysis to resolve calls to `eval` and to build a call graph that is then used by a static taint analysis [7]. Similarly, Tripp et al. [33] use concrete `location`, `referrer`, `URL`, `DOM` element values to perform partial evaluation that enables more precise string and taint analyses. While the aforementioned works all focused on JavaScript code embedded in web pages, the work in [26] specifically targets JavaScript web applications, and proposes the use of execution environment *snapshots*, as a lighter alternative to dynamic trace collection, to inform the subsequent static analysis. On the other hand, Andreasen et al. [9] used blended analysis to pinpoint root causes of imprecision and the work in [36] proposes to use dynamic information *after* the static analysis, to help pinpoint and fix root causes of imprecision, and to specialize the analysis to the code of interest. Compared to these works, while our approach is not sound, its unsoundness is more principled than relying on concrete executions, hence gaining better coverage.

Taint analysis for JavaScript. Previous work on static taint analysis for JavaScript relied either on points-to analysis [7, 14], or on information flow tracking [11, 17, 18, 28]. To the best of our knowledge, this is the first paper on abstract interpretation-based taint analysis for JavaScript. Other work has also explored dynamic taint analysis approaches [10, 13, 21, 29] to circumvent the precision and scalability challenges of static analysis at the cost of completeness.

9 CONCLUSION

We have presented a feedback-driven static analysis that improves the scalability of an existing state-of-the-art whole-program static analysis for Node.js applications. By automatically identifying the third-party modules of an application that need to be analyzed, we detect taint flows in critical modules. We evaluated our tool, NODEST, on existing benchmarks and additional real-world Node.js applications. The results show that our feedback-driven static analysis scales well and is able to find previously known and new zero-day injection vulnerabilities with high precision. NODEST can statically analyze real-world applications that no other static analysis tool has been able to analyze before. In particular, it is able to analyze the Express framework and report non-trivial taint flows.

REFERENCES

[1] 2019. Babel. https://babeljs.io/.
[2] 2019. ES6 Features. http://es6-features.org.
[3] 2019. Express web framework. https://www.npmjs.com/package/express.
[4] 2019. Module Loader in Node.js. https://github.com/nodejs/node/blob/master/lib/module.js.
[5] 2019. MongoDB database. https://www.mongodb.com/.
[6] 2019. OWASP vulnerable Node.js application. https://github.com/OWASP/NodeGoat.
[7] 2019. T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php.
[8] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 17–31.
[9] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP)*. ACM, 31–36.
[10] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail: A general-purpose platform for shadow execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 260–270.
[11] Andrey Chudnov and David A Naumann. 2010. Information flow monitor inlining. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*. IEEE, 200–214.
[12] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 238–252.
[13] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. 2018. Affogato: Runtime Detection of Injection Attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (SOAP)*. ACM, 94–99.
[14] Salvatore Guarnieri and V Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code.. In *USENIX Security Symposium*, Vol. 10. 78–85.
[15] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 177–187.
[16] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP)*. ACM, 13–18.
[17] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1663–1671.
[18] Daniel Hedin and Andrei Sabelfeld. 2012. Information-flow security for a core of JavaScript. In *Computer Security Foundations Symposium (CSF)*. IEEE, 3–18.
[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*. Springer-Verlag, 238–255.
[20] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7, 3 (1977), 305–317.
[21] Rezwana Karim, Frank Tip, Alena Sochurkova, and Koushik Sen. 2018. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* (2018).
[22] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 121–132.
[23] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects. In *Asian Symposium on Programming Languages and Systems*. Springer, 148–168.
[24] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. 2012. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *In Proceedings of the International Workshop on Foundations of Object Oriented Languages (FOOL)*.
[25] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 499–509.
[26] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 61–70.
[27] Jihyeok Park, Xavier Rival, and Sukyoung Ryu. 2017. Revisiting recency abstraction for JavaScript: towards an intuitive, compositional, and efficient heap abstraction. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP)*. ACM, 1–6.
[28] José Fragoso Santos and Tamara Rezk. 2014. An information flow monitor-inlining compiler for securing a core of javascript. In *IFIP International Information Security Conference*. Springer, 278–292.
[29] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 488–498.
[30] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 485–495.
[31] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 435–458.
[32] C-A. Staicu, M. Pradel, and B. Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on Node.js. In *25th Annual Network and Distributed System Security Symposium (NDSS)*.
[33] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 49–59.
[34] Shiyi Wei and Barbara G Ryder. 2013. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 336–346.
[35] Shiyi Wei and Barbara G Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[36] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript Static Analysis via Localization and Remediation of Root Causes of Imprecision. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 487–498.