Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs

MAFALDA FERREIRA, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
MIGUEL MONTEIRO, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
TIAGO BRITO, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
MIGUEL E. COIMBRA, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
NUNO SANTOS, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
LIMIN JIA, Carnegie Mellon University, USA
JOSÉ FRAGOSO SANTOS, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal

While static analysis tools that rely on Code Property Graphs (CPGs) to detect security vulnerabilities have proven effective, deciding how much information to include in the graphs remains a challenge. Including less information can lead to a more scalable analysis but at the cost of reduced effectiveness in identifying vulnerability patterns, potentially resulting in classification errors. Conversely, more information in the graph allows for a more effective analysis but may affect scalability. For example, scalability issues have been recently highlighted in ODGen, the state-of-the-art CPG-based tool for detecting Node.js vulnerabilities.

This paper examines a new point in the design space of CPGs for JavaScript vulnerability detection. We introduce the Multiversion Dependency Graph (MDG), a novel graph-based data structure that captures the state evolution of objects and their properties during program execution. Compared to the graphs used by ODGen, MDGs are significantly simpler without losing key information needed for vulnerability detection. We implemented Graph.js, a new MDG-based static vulnerability scanner specialized in analyzing *npm* packages and detecting taint-style and prototype pollution vulnerabilities. Our evaluation shows that Graph.js outperforms ODGen by significantly reducing both the false negatives and the analysis time. Additionally, we have identified 49 previously undiscovered vulnerabilities in *npm* packages.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; **Automated static analysis**; • **Security and privacy** → **Software and application security**; • **Theory of computation** → **Program analysis**.

Additional Key Words and Phrases: Static Analysis, JavaScript, Vulnerability Detection

ACM Reference Format:
Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. *Proc. ACM Program. Lang.* 8, PLDI, Article 164 (June 2024), 25 pages. https://doi.org/10.1145/3656394

Authors' addresses: Mafalda Ferreira, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal, mafalda.baptista@tecnico.ulisboa.pt; Miguel Monteiro, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal, miguel.figueiredo.monteiro@tecnico.ulisboa.pt; Tiago Brito, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal, tiago.de.oliveira.brito@tecnico.ulisboa.pt; Miguel E. Coimbra, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal, miguel.e.coimbra@tecnico.ulisboa.pt; Nuno Santos, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal, nuno.m.santos@tecnico.ulisboa.pt; Limin Jia, Carnegie Mellon University, Pittsburgh, USA, liminjia@andrew.cmu.edu; José Fragoso Santos, INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal, jose.fragoso@tecnico.ulisboa.pt.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/6-ART164
https://doi.org/10.1145/3656394

1 INTRODUCTION

Static analysis tools based on Code Property Graphs (CPGs) [61] have become increasingly popular in recent years. CPGs, as originally introduced for analyzing C/C++ functions, are a graph-based representation that combines Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Program Dependence graphs (PDG). This rich data structure captures patterns for common vulnerabilities, such as buffer overflows and format strings. Tools that use CPGs start by generating the program's CPG and then iterating through the CPG in search of patterns that indicate potential vulnerabilities. Because search patterns are customizable through graph traversal queries, CPGs offer high flexibility in detecting a broad spectrum of vulnerability types.

CPGs have been adopted for many languages [2, 7, 10, 29, 36] and other analytical scopes, including detection of GDPR compliance violations [4, 19, 48] and data privacy breaches [32]. The original CPG data structure has been adapted to fit the specificities of programming languages and analyses. For example, Li et al. [36] developed ODGen, a tool that extends CPGs with an Object Dependence Graph (ODG) to detect vulnerabilities in Node.js applications. ODGen performs the analysis on a combined CPG-ODG data structure, where ODG's nodes represent objects, variables, and scopes, while edges capture relations between them, enabling the detection of taint-style and prototype pollution vulnerabilities. Currently, ODGen is the static vulnerability scanner for *npm* packages with the most favorable trade-off between effectiveness and precision [6].

Even with these advances, a fundamental challenge in using CPGs for vulnerability detection is deciding what information to include in the graphs so that the definition and identification of vulnerability patterns are both straightforward and precise. On the one hand, including less information limits the number of vulnerability patterns that can be precisely defined on the graph; analysis can either over-approximate, resulting in high false positives, or under-approximate, leading to high false negatives. For instance, using the original CPG alone, without the ODG component, is insufficient for detecting vulnerabilities in JavaScript programs, leading to false negatives. On the other hand, incorporating more program properties into the graph decreases the performance of both graph construction and query execution. Prolonged query execution may result in timeouts, causing the analysis to miss vulnerabilities. This has been observed for ODGen, for which scalability issues have been recently highlighted [27, 63]. Furthermore, complex graphs that intertwine multiple types of program representation into the same structure are challenging to reason about, resulting in complex vulnerability queries whose correctness guarantees are difficult to ascertain. Particularly, the construction of such graphs follows intricate semantic rules that make it hard to understand the formal properties of the graphs, both in relation to the underlying concrete program semantics and in relation to the true negatives and false positives allowed by the analysis. For example, how can we be sure that if the program has a code injection vulnerability, then the code injection pattern must exist in the constructed graphs?

This paper examines a different point in the design space of CPGs for JavaScript vulnerability detection. In particular, we observe that the AST and CFG contain a lot of extraneous information not needed for vulnerability detection. Furthermore, we note that with the current ODGs [27, 36, 63], vulnerability queries must jump back and forth between the CPG and the ODG of the given program in search of complex graph patterns that are not only difficult to specify and reason about, but also degrade the performance of the analysis. We ask the question: *can we design one unified graph to capture all the essential information for common JavaScript vulnerability detection?*

To this end, we introduce Multiversion Dependency Graphs (MDGs), an efficient graph-based data structure for statically detecting common vulnerabilities in JavaScript programs, which encompasses two types of analysis: (1) classical shape analysis [25, 51], where each object is mapped to the set of properties it may have during execution; and (2) dependency analysis [33, 47], associating the

objects and values created during execution with the values on which they depend. One key insight of our work is that the graph and query complexity can be reduced by relying on a *single graph that models the evolution of objects and properties* of the given program over time. Specifically, we create new versions of objects and properties each time an object is updated. Version information allows us to keep track of both data flows and the execution order in a single graph (more in §2), greatly reducing graph and query complexity. A second insight is that using a summary fixed-pointed representation for loops and recursive function calls reduces graph and query complexity without increasing false positives in almost all vulnerability detection tasks in our datasets. To understand the formal properties of the constructed MDGs, we formalize our MDG construction algorithm and prove that it is sound, i.e., the generated MDGs are an over-approximation of the concrete execution traces. This implies that if a program has a vulnerability that can be detected through the analysis of the program trace, then the corresponding vulnerability pattern occurs in the generated MDG.

To evaluate the effectiveness of our approach, we implemented Graph.js[1], a static vulnerability scanner for JavaScript code. Graph.js focuses on analyzing *npm* packages from the Node.js ecosystem, known to have numerous vulnerabilities [1, 6, 53, 64]. To analyze a package, Graph.js generates its MDG, which it then stores in a Neo4j graph database. Graph.js proceeds to detect vulnerabilities by executing specific queries written in Cypher. Currently, Graph.js can detect three different kinds of taint-style vulnerabilities as well as prototype pollution vulnerabilities.

Our evaluation of Graph.js on two curated datasets [5, 6] shows that our tool significantly outperforms ODGen, the state-of-the-art tool, with lower false negatives and shorter analysis time. In particular, Graph.js detects 82% of the reported vulnerabilities in the ground truth datasets, surpassing ODGen by 1.63×, with 1.23× the precision. On average, Graph.js completes its analysis of 603 packages within 4.61 seconds and can analyze 95% in under 10 seconds. In 99% of the cases, Graph.js's MDG are smaller than the ODGs generated by ODGen, with only 0.14× the nodes and 0.42× the edges. Moreover, with Graph.js, we have identified 49 previously undiscovered vulnerabilities in *npm* packages, which we have responsibly disclosed to the package developers.

2 MOTIVATION AND OVERVIEW

In this section, after briefly reviewing CPGs, we introduce a motivating example of a vulnerable JavaScript code (§2.1) and provide an overview of our proposed approach (§2.2).

In general, CPG-based vulnerability detection approaches have important advantages. *(1) Generality and modularity:* The graph serves as a universal structure for detecting a spectrum of vulnerabilities; variations in vulnerabilities are addressed in the query phase, allowing graph reuse and eliminating overhead in graph reconstruction. *(2) Compositionality:* Code changes only require partial reconstructions of the CPG and rerunning pertinent queries instead of a full-scale analysis.

To facilitate tracking dependencies across objects and properties in JavaScript, ODGen augments a program's CPG with another data structure called the Object Dependency Graph (ODG) [36]. ODG's nodes can represent variables or objects. Between the CPG and ODG, a total of seven types of edges are used, including *object definition edges* for linking objects to the AST node where the object was declared; *data flow edges* for connecting one object to another; *property edges* for associating properties with objects; and *AST-OBJ lookup edges* for linking nodes between CPG and ODG. ODGen has been shown to be effective in detecting many Node.js vulnerabilities [27, 36].

2.1 Motivating Example

Figure 1a presents an exemplary vulnerable JavaScript code sample that offers insight into how code property graphs can be used to detect vulnerabilities. The `git_reset` function (lines 3-9) initiates a

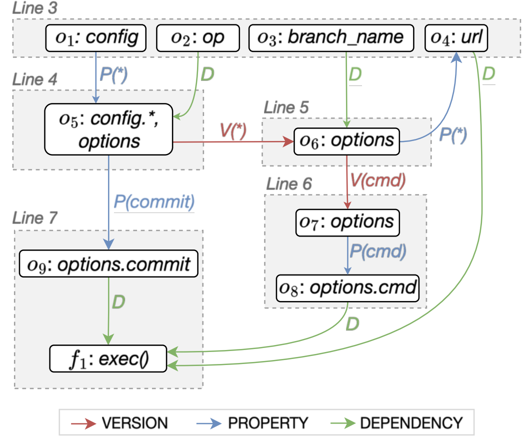
[1] https://github.com/formalsec/graphjs

```
1 const exec = require('child_process').exec;
2
3 function git_reset(config, op, branch_name, url) {
4     const options = config[op];
5     options[branch_name] = url;
6     options.cmd = 'git reset';
7     exec(`${options.cmd} HEAD~${options.commit}`);
8 }
9 module.exports = git_reset;
```

(a) Vulnerable code.

```
1 const reset = require('git_reset');
2 const config = { reset: { commit: 1, branch: 'main'} }
3 // Resets HEAD to HEAD~1
4 reset(config, 'reset', 'main', 'origin/main');
```

(b) Example of a benign use of the package.



(c) MDG of the program on the left.

```
1 const reset = require('git_reset');
2 const config = { reset: { commit: '| rm -rf /'} }
3 // Removes all files
4 reset(config, 'reset', 'xx', 'xx');
```

(d) Exploit for command injection vulnerability.

```
1 const reset = require('git_reset');
2 const malicious_fn = function() { for (;;) {}}
3 reset({}, '__proto__', 'toString', malicious_fn);
4 // Executes non-ending loop
5 {}.toString();
```

(e) Exploit for prototype pollution vulnerability.

Fig. 1. A motivating example, with a command injection and prototype pollution vulnerability.

shell process that executes a git reset command, designed to revert a git repository to a previous commit. As shown in line 9, the function `git_reset` is exported, and thus can be called by an attacker with crafted inputs. The function accepts four input values. When called using the benign inputs illustrated in Figure 1b, the `git_reset` function internally assigns the value `'origin/main'` to `config['reset']['main']` (line 5) and runs the shell command `'git reset HEAD~1'` (line 7). The embedded commit number `'1'` in this string is sourced from the `config['reset']['commit']` property, initialized in line 2 of Figure 1b before the `git_reset` function invocation. This function hides two exploitable *taint-style* and *prototype pollution* vulnerabilities.

Taint-style vulnerability: Occurs when untrusted input from a given source reaches a vulnerable sink without undergoing proper validation or sanitization, potentially leading to malicious exploitation or unintended behavior. Specifically, the program in Figure 1a contains an exploitable command injection vulnerability, i.e., by using the payload shown in Figure 1d, an attacker can prompt the `exec` function to run the command `'git reset HEAD~1 | rm -rf /'`, which deletes all local files. In general, to detect such vulnerabilities, the analysis needs to perform the following steps: (*1*) identify potential unsafe sources (e.g., function inputs) and unsafe sinks (e.g., `exec`) and (*2*) determine whether tainted inputs from sources can reach the sinks. At a high level, CPGs enable (*1*) searching through AST for known sink functions, and (*2*) tracking data dependency information between the untrusted sources (tainted inputs) and arguments used by the sink function.

Prototype pollution vulnerability: It arises when an attacker manages to manipulate the prototype of an object, leading to side effects such as Denial-of-Service (DoS) or arbitrary code execution. Figure 1e illustrates that a prototype pollution vulnerability within the `git_reset` function can be

exploited to induce a DoS by substituting JavaScript's built-in `toString` function with the function referenced by the variable `malicious_fn`. The next invocation of `toString` will enter the infinite loop of `malicious_fn`, causing the program to hang. The prototype is polluted on lines 4 and 5 of Figure 1a. At a high level, to detect prototype pollution, CPG-based approaches need to collect the following information: (*1*) the location where the prototype pollution happens, i.e., an object lookup followed by an object assignment over the initial lookup (e.g, lines 4 and 5); and (*2*) whether tainted inputs from sources can reach both the properties and the value assigned (`op`, `branch_name` and `url` respectively). The former can be obtained from the AST and the latter needs dependency analysis, similar to the previous example.

2.2 Overview of MDGs

To design a new and simpler CPG data structure for JavaScript, we first narrow down the following vulnerabilities: *prototype pollution* (CWE-1321 [56]), and *taint-style vulnerabilities*, which include OS command injection (CWE-78 [58]), arbitrary code execution (CWE-94 [59]), and path traversal (CWE-22 [57]). These vulnerabilities include representative information that CPGs should capture. We do not foresee technical difficulties in handling other vulnerabilities outlined in prior work [36], as they use similar information. The information needed for identifying the above-mentioned vulnerabilities includes: data dependencies between variables and objects, sequences of read/write operations to objects and their properties, and a mapping between variables used in the source code and heap objects/values that exist at runtime. We are able to use one graph, Multiversion Dependency Graph (MDG), to capture all of these. Next, we present our MDGs and associated queries using the example in Figure 1a.

Multiversion Dependency Graph (MDG), a simpler graph: The MDG for the function `git_reset` is shown in Figure 1c. The graph is generated by abstractly executing the program line by line.

The MDG consists of two types of nodes: *objects* and *function calls*. Object nodes represent objects or primitive values computed during the execution of the program. Each node has a label $o_x{:}v_y$, where o_x identifies a specific object version, and v_y denotes the name of the variable (or variables) in a given line of the source code pointing to the node's object or value. Nodes in each gray box are typically objects whose properties are accessed while the line of code indicated by the line number above the gray box is abstractly executed. For instance, node o_5, created during the analysis of line 4, refers to an object pointed to by: (*i*) the `options` variable and (*ii*) a property (`op`) of the object referenced by the `config` variable. We will explain the wildcard `*` notation later. Function call nodes, denoted as $f_x{:}v_y()$, represent the invocations of a function $v_y()$ in a specific line and are identified by the label f_x. Figure 1c includes node f_1 corresponding to the `exec()` function call on line 7.

MDGs have three types of directed edges: *property edges*, *version edges*, and *dependency edges*, and grows as the program is abstractly executed. Next, we detail how the graph in Figure 1c is generated. Our analysis is applied to one statement at a time in a forward manner, with the MDG being updated as the analysis proceeds.

- *Line 3:* The analysis creates four object nodes, one for each parameter: o_1, o_2, o_3, and o_4.
- *Line 4:* First, the analysis identifies a property lookup, `config[op]`. As the object that represents `config` (o_1) has no known property, the analysis lazily initializes a new property in o_1, creating a new node o_5 for the accessed property `op`, and then adding a property edge $o_1 \xrightarrow{P(*)} o_5$, meaning that o_5 is a property of o_1. Since the property name is not known at static time, the property edge is labeled with '*'. Additionally, the analysis creates a dependency edge $o_2 \xrightarrow{D} o_5$, indicating that this dynamic property's name depends on the value of the variable `op` (o_2). Finally, the analysis updates the value of the program variable `options` to the newly created object o_5.

- *Line 5:* This statement involves a dynamic property update in `options[branch_name]`. In this case, the analysis creates a new version of the updated object. In the example, a new object version for `options` (o_6) is created from its previous version o_5, linked via a version edge $o_5 \xrightarrow{V(*)} o_6$. Then, the analysis adds a property edge $o_6 \xrightarrow{P(*)} o_4$, indicating that o_6 has a dynamic property whose value is represented by o_4. In addition, since the value of `branch_name` is not available at static time, the analysis adds a dependency edge $o_3 \xrightarrow{D} o_6$, meaning that o_3 represents the name of the dynamic property. Importantly, when a new version is created, all program variables that referred to the old version are updated to the new one; hence, after line 5, the variable `options` is updated to o_6.
- *Line 6:* Here, the analysis identifies a property update of the static property `cmd`, which follows the same steps as the dynamic one, except that no dependency edges are created as the name of the property is known at static time. So, the analysis creates a new version of `options` (o_7) linked from its former version with a version edge $o_6 \xrightarrow{V(cmd)} o_7$ and creates another node o_8 for the written property `options.cmd` and connects both nodes with a property edge $o_7 \xrightarrow{P(cmd)} o_8$.
- *Line 7:* The last line executes a call to the function `exec()`. First, the analysis must evaluate the lookup expressions `options.cmd` and `options.commit`. The first expression trivially evaluates to o_8, since o_7 defines the property 'cmd'. For the second lookup, the analysis first constructs node o_9 and connects it to the initial version of `options`, o_5, with a property edge $o_5 \xrightarrow{P(commit)} o_9$, because although the algorithm is only reading the property now, it existed from the beginning. Then, similarly to *Line 4*, the analysis returns the object nodes that contain property `commit`. However, in this case, it will find two versions: o_9, because o_5 is the latest version that contains property `commit`; and o_4, because a more recent version of the object (o_6) has a dynamic property, which may have overwritten property `'commit'`. Having evaluated both expressions, the analysis creates three dependency edges from the resulting nodes to the node representing the function call, $o_8 \xrightarrow{D} f_1$, $o_9 \xrightarrow{D} f_1$ and $o_4 \xrightarrow{D} f_1$.

As shown above, the MDG captures how objects evolve over time as their properties are updated; these updates are captured by new version edges. For instance, object o_6, referenced by the variable `options`, is a new version of object o_5, resulting from the update of the object's dynamic property in line 5. Object updates can be identified by these version edges. Property edges $P(p)$, where p is the property name, capture the internal structure of objects during the program's execution. They denote that the object pointed to by the edge is a sub-object of the object version from which the edge originates. For example, the property edge $o_1 \xrightarrow{P(*)} o_5$ indicates that o_5 is a sub-object of object version o_1 derived from the dynamic property `op` (line 4). The dependency edges D denote data dependencies between values/objects in two cases: (i) when a sub-object is looked up (read) by a property name, as seen in dependencies $o_2 \xrightarrow{D} o_5$, $o_8 \xrightarrow{D} f_1$, $o_9 \xrightarrow{D} f_1$, or $o_4 \xrightarrow{D} f_1$, and (ii) when the name of the property being altered depends on a dynamic value, as in the dependency $o_3 \xrightarrow{D} o_6$.

MDG queries: MDG allows for simpler and more efficient query specifications for vulnerability detection. Firstly, *the order between operations delineated by writes can be easily determined due to the tracking of multiple versions of objects*. Specifically, given a version edge $o_x \xrightarrow{V(p)} o_y$, we know that the instruction on line $line_y$ that led to the creation of object version o_y is a write operation and that it was executed after the instruction on line $line_x$ that resulted in the prior object version o_x. This characteristic facilitates the identification of prototype pollution vulnerabilities. For example, a typical instance of prototype pollution, such as the one present in the vulnerable function shown in Figure 1c, occurs when there is an object lookup in a property p_1, followed by an assignment of a value v to a property p_2 of the obtained sub-object, where an attacker controls p_1, p_2, and v. The MDG in Figure 1c allows for the identification of this pattern, where one can easily identify a

tainted lookup dependent on o_2, followed by a tainted property update dependent on o_3 and o_4. The sequential ordering is captured by the version edge $o_5 \xrightarrow{V(*)} o_6$.

Secondly, *the MDG is self-contained, avoiding the need for costly AST and CFG visits*. Specifically, the MDG's edge types encapsulate essential information, allowing data flows to be tracked through simple graph traversals across these edge types. This characteristic is particularly beneficial for detecting taint-style vulnerabilities. For instance, in Figure 1c, once we establish that the sensitive sources are the `git_reset`'s input parameters (i.e., o_1, o_2, o_3, and o_4), and the sink is the `exec` function call node (i.e., f_1), it becomes apparent that the graph includes paths from all tainted inputs reaching the sink. For example, we can trace the sequence of dependencies from o_1 using $o_1 \xrightarrow{P(*)} o_5 \xrightarrow{V(*)} o_6 \xrightarrow{P(*)} o_4 \xrightarrow{D} f_1$.

3 SOUND MDGS FOR JAVASCRIPT

In this section, we formally define Multiversion Dependency Graphs (MDGs) (§3.1) and an abstract interpretation-based analysis that computes them for a core of JavaScript (§3.2). We then present a soundness theorem that establishes the guarantees of the proposed analysis (§3.3). A complete account of the analysis, including the proof of its soundness, can be found in [21].

3.1 Syntax of Multiversion Dependency Graphs

A Multiversion Dependency Graph, denoted $\hat{g} = (\hat{V}, \hat{E}) \in \hat{\mathcal{G}}$, tracks the structure and evolution of objects during the program execution and the data dependencies between the values that the program manipulates. The graph nodes, $\hat{V}$, are taken from the set of *abstract locations*, $\hat{l} \in \hat{\mathcal{L}}$, and represent objects and primitive values computed during program execution. In the MDG of the running example in Figure 1c, these abstract locations are represented with the label o_x. The graph edges $\hat{E}$, taken from the set of labeled edges, connect pairs of abstract locations. Each edge is annotated with its type, a label τ, given by the grammar: $\tau ::= \mathsf{D} \mid \mathsf{P}(p) \mid \mathsf{P}(*) \mid \mathsf{V}(p) \mid \mathsf{V}(*)$.

Edges labeled with D are *Dependency Edges*. An edge $\hat{l}_1 \mapsto_{\mathsf{D}} \hat{l}_2$ means that the value/object represented by $\hat{l}_2$ is computed using (depends on) the value/object represented by $\hat{l}_1$. For instance, $\hat{l}_2 \mapsto_{\mathsf{D}} \hat{l}_5$ in Figure 1c resulted from the property lookup on line 4, where the property name depends on the value of variable op. Edges labeled with $\mathsf{P}(p)/\mathsf{P}(*)$ are *Property Edges*. A known-property edge $\hat{l}_1 \mapsto_{\mathsf{P}(p)} \hat{l}_2$ means that the object represented by $\hat{l}_1$ has a property named p mapped to a value represented by $\hat{l}_2$. An unknown-property edge $\hat{l}_1 \mapsto_{\mathsf{P}(*)} \hat{l}_2$ has the same meaning as the known one, except that the property name cannot be determined statically. For instance, $\hat{l}_1 \mapsto_{\mathsf{P}(*)} \hat{l}_5$ in Figure 1c comes from the property lookup on line 4, where the property is represented by the variable op. Edges labeled with $\mathsf{V}(p)/\mathsf{V}(*)$ are *Version Edges*. A known-property version edge $\hat{l}_1 \mapsto_{\mathsf{V}(p)} \hat{l}_2$ means that the object represented by $\hat{l}_2$ is a new version of the object represented by $\hat{l}_1$, resulted from an update of its property p. An unknown-property version edge $\hat{l}_1 \mapsto_{\mathsf{V}(*)} \hat{l}_2$ has the same meaning as the known one, except that the name of the updated property is not known at static time. The version edge $\hat{l}_5 \mapsto_{\mathsf{V}(*)} \hat{l}_6$ in Figure 1c results from an update of a dynamic property on line 5.

MDGs form a lattice under standard subset inclusion; formally, given two MDGs $\hat{g}_1 = (\hat{V}_1, \hat{E}_1)$ and $\hat{g}_2 = (\hat{V}_2, \hat{E}_2)$, $\hat{g}_1$ is said to be lower than or equal to $\hat{g}_2$, written $\hat{g}_1 \sqsubseteq \hat{g}_2$, if and only if $\hat{E}_1 \subseteq \hat{E}_2$.

In the following, we write $\hat{g}[\hat{l}, p]$ to denote the set of abstract locations associated with the object represented by $\hat{l}$ via property p. These locations may be directly connected to $\hat{l}$ via a property edge labeled with $\mathsf{P}(p)$ or connected to a previous version of $\hat{l}$, as we do not duplicate properties that are not updated when creating a new version of an object. For instance, Figure 1c, $\hat{g}[\hat{l}_7, cmd] = \{\hat{l}_8\}$. The abstract graph over-approximates the program's concrete heap state, so $\hat{g}[\hat{l}, p]$ may contain more than one abstract location, for instance, branches of an if statement updating an object differently,

would result in different object versions at the join of the branches. For instance, in Figure 1c, $\hat{g}[\hat{l}_7, commit] = \{\hat{l}_4, \hat{l}_9\}$.

3.2 Computing Multiversion Dependency Graphs

We formalize our analysis for computing MDGs for a core of JavaScript.

Core JavaScript syntax: Our core of JavaScript includes expressions and statements, shown below. Expressions $e \in \mathcal{E}xp$ include values and program variables. Statements $s \in \mathcal{S}tm$ include: assignments, binary operations, property lookups, property assignments, new object creation, if and while statements, sequencing, and function calls. Each statement that computes new values or objects has a unique index i, which we explain later with analysis rules.

$$\begin{array}{lll} e \in \mathcal{E}xp & ::= & v \in \mathcal{V} \mid x \in \mathcal{X} \\ s \in \mathcal{S}tm & ::= & x := e \mid x :=_i e_1 \oplus e_2 \mid x :=_i e.p \mid x :=_i e_1[e_2] \mid e_1.p :=_i e_2 \mid e_1[e_2] :=_i e_3 \\ & & x := \{\ \}^i \mid \mathsf{if}(e)\{s_1\}\,\mathsf{else}\,\{s_2\} \mid \mathsf{while}(e)\{s\} \mid s_1; s_2 \mid x := f(e_1, ..., e_n) \end{array}$$

Abstract variable store: To connect program variables to the objects that they represent, we use abstract variable stores $\hat{\rho} \in \widehat{Sto} : \mathcal{X} \rightharpoonup \wp(\hat{\mathcal{L}})$ that map program variables to sets of abstract locations, where $\hat{\rho}(x)$ denotes the set of abstract locations that x represents. Given an expression e and an abstract store $\hat{\rho}$, the evaluation of e under $\hat{\rho}$, written $[\![e]\!]_{\hat{\rho}}$, denotes the set of abstract locations that e represents. For instance, $[\![options]\!]_{\hat{\rho}} = \{\hat{l}_8\}$ in the MDG of Figure 1c. Observe that $\hat{\rho}(x)$ only contains the newest versions of the objects associated with x. Abstract stores form a lattice under the standard pointwise subset inclusion; formally: a store $\hat{\rho}_1$ is said to be lower than or equal to a store $\hat{\rho}_2$, written $\hat{\rho}_1 \sqsubseteq \hat{\rho}_2$, if and only if $\mathsf{dom}(\hat{\rho}_1) \subseteq \mathsf{dom}(\hat{\rho}_2)$ and $\forall x \in \mathsf{dom}(\hat{\rho}_1).\ \hat{\rho}_1(x) \subseteq \hat{\rho}_2(x)$. Focusing only on the dependency aspect of the graph, we can say a graph/store is lower than another if it contains fewer dependency edges. If the sets of abstract locations and program variables are finite, then so are the lattices of MDGs and abstract stores.

Auxiliary graph functions: Next, we explain two auxiliary functions used in graph construction.

The function $\mathsf{NV}_i(\hat{g}, \hat{\rho}, L_1, p_1)$ is used to create a new version of objects represented by locations in L_1, due to an assignment to property p_1. Here i is the index of the statement where this function is called. It returns a tuple $(\hat{g}', \hat{\rho}', L_1')$, where $\hat{g}'$ is the updated graph, L_1' is the set of abstract locations representing newly created objects, and $\hat{\rho}'$ is the updated store with the occurrences of older version locations replaced by their corresponding newer versions.

The function $\mathsf{AP}_i(\hat{g}, L, p_1)$ extends a set of objects represented by locations in L with a property edge $\mathsf{P}(p_1)$ and returns the new graph $\hat{g}'$. The index i has the same meaning as above. If a location $\hat{l}_1$ in L already has a property edge labeled as $P(p_1)$, then no action is taken; otherwise, a new location $\hat{l}_2$ is allocated and $\hat{l}_1 \mapsto_{\mathsf{P}(p_1)} \hat{l}_2$ is added to the graph.

The above functions have an alternate version for properties whose values are computed dynamically. For example, on line 4 of the program in Figure 1a, the property name (op) is non-static. In this case, $\mathsf{AP}^*_i(\hat{g}, L_1, L_p)$ extends each object in L_1 with an unknown-property edge pointing to an abstract location that depends on all locations in L_p, where L_p is the set of abstract locations that represent the non-static property p. If a location $\hat{l}_1$ in L_1 does not have a property edge labeled as $P(*)$, then a new location $\hat{l}_2$ is allocated and $\hat{l}_1 \mapsto_{\mathsf{P}(*)} \hat{l}_2$ is added to the graph; otherwise, the dependencies in L_p are added to the existing property. Analogously, $\mathsf{NV}^*_i(\hat{g}, \hat{\rho}, L_1, L_p)$ creates a new version of all objects corresponding to locations in L_1, making each new object depend on all locations in L_p. These dynamic versions of the rules ensure that locations denoting dynamically

$$\textsc{Assign-Op}\quad \frac{[\![e_j]\!]_{\hat{\rho}} = L_j \ |_{j=1,2} \qquad \hat{l}_i = \mathsf{alloc}(\hat{g}, i) \qquad \hat{g}' = \hat{g} \uplus \left\{\hat{l} \mapsto_{\mathrm{D}} \hat{l}_i \mid \hat{l} \in L_1 \cup L_2\right\}}{\mathcal{A}(x :=_i e_1 \oplus e_2, \hat{g}, \hat{\rho}) \triangleq (\hat{g}', \hat{\rho}[x \mapsto \{\hat{l}_i\}])}$$

$$\textsc{New Object}\quad \frac{\hat{l}_i = \mathsf{alloc}(\hat{g}, i) \qquad \hat{\rho}' = \hat{\rho}[x \mapsto \{\hat{l}_i\}] \qquad \hat{g}' = \mathsf{AddNode}(\hat{g}, \hat{l}_i)}{\mathcal{A}(x := \{\ \}^i, \hat{g}, \hat{\rho}) \triangleq (\hat{g}', \hat{\rho}')}$$

$$\textsc{Static Property Lookup}\quad \frac{[\![e]\!]_{\hat{\rho}} = L \qquad \hat{g}' = \mathsf{AP}_i(\hat{g}, L, p) \qquad L' = \{\hat{l}' \mid \hat{l} \in L \ \wedge \ \hat{l}' \in \hat{g}'[\hat{l}, p]\}}{\mathcal{A}(x :=_i e.p, \hat{g}, \hat{\rho}) \triangleq (\hat{g}', \hat{\rho}[x \mapsto L'])}$$

$$\textsc{Dynamic Property Update}\quad \frac{[\![e_i]\!]_{\hat{\rho}} = L_j \ |_{j=1}^{3} \quad (\hat{g}'', \hat{\rho}', L_1') = \mathsf{NV}_i^*(\hat{g}, \hat{\rho}, L_1, L_2) \qquad \hat{g}' = \hat{g}'' \uplus \{\hat{l}_1 \mapsto_{\mathrm{P}(*)} \hat{l}_3 \mid \hat{l}_1 \in L_1' \ \wedge \ \hat{l}_3 \in L_3\}}{\mathcal{A}(e_1[e_2] :=_i e_3, \hat{g}, \hat{\rho}) \triangleq (\hat{g}', \hat{\rho}')}$$

$$\textsc{If Statement}\quad \frac{\mathcal{A}(s_j, \hat{g}, \hat{\rho}) = (\hat{g}_j, \hat{\rho}_j) \ |_{j=1}^{2}}{\mathcal{A}(\mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}, \hat{g}, \hat{\rho}) \triangleq (\hat{g}_1 \sqcup \hat{g}_2, \hat{\rho}_1 \sqcup \hat{\rho}_2)}$$

$$\textsc{While}\quad \frac{\mathbf{lfp}(\mathcal{A}(s))(\hat{g}, \hat{\rho}) = (\hat{g}', \hat{\rho}')}{\mathcal{A}(\mathsf{while}(e)\{s\}, \hat{g}, \hat{\rho}) \triangleq \hat{g}', \hat{\rho}'}$$

Fig. 2. Selected Graph Construction Analysis: $\mathcal{A}(\hat{g}, \hat{\rho}, s) \triangleq \hat{g}', \hat{\rho}'$.

computed properties are connected to the updated/looked-up objects via dependency edges; these are essential for effective taint propagation.

Analysis rules for graph construction: We formalize our graph construction using a declarative function $\mathcal{A}$. We write $\mathcal{A}(s, \hat{g}, \hat{\rho}) = (\hat{g}', \hat{\rho}')$ to mean that the analysis of statement s starting from the initial abstract state ($\hat{g}$, $\hat{\rho}$) results in the final abstract state ($\hat{g}'$, $\hat{\rho}'$). Selected rules for $\mathcal{A}$ are shown in Figure 2. The rule [Assign-Op] evaluates both expressions, e_1 and e_2, and creates a new location, $\hat{l}_i$, representing the result of the binary operation; this new location is then set to depend on all the locations to which e_1 and e_2 evaluate and the variable x is set to the singleton set containing $\{\hat{l}_i\}$ in the abstract store. The [New Object] rule generates an abstract location for the created object, calling the function alloc with the unique identifier i and the current graph $\hat{g}$. An abstract allocation does not necessarily generate a fresh abstract location; we choose to always generate the same abstract location for the same literal object. This means that objects created within a loop are represented by the same abstract location, avoiding object explosion.

The [Static Property Lookup] rule evaluates the expression e denoting the object being inspected, obtaining a set of locations L; it then uses the function AP to extend the objects in L with the property p in case they do not define it; finally, it obtains the set of locations L' representing the values of property p in the objects in L and sets the variable x to L' in the abstract store. The [Dynamic Property Lookup] Rule (omitted) is analogous, except that AP^* is used with an additional argument corresponding to the set of locations representing the dynamic value of the property.

Figure 3 illustrates the graphs and stores after applying the analysis rules for lines 4 and 5 of the running example in Figure 1a. The graph on the left is a subgraph of Figure 1c. The abstract stores $\hat{\rho}_k$ represent the content of the abstract store $\hat{\rho}$ after analyzing line k. The edges in blue are generated as a result of analyzing line 4 and those in red are generated when analyzing line 5.

The rule [Dynamic Property Lookup] is used when analyzing line 4 of the example. Here s = `options` $:=_i$ `config[op]`. First, `config` and `op` evaluate to $\{\hat{l}_1\}$ and $\{\hat{l}_2\}$, respectively. Then, the rule calls $\mathsf{AP}_i^*(\hat{g}, \{\hat{l}_1\}, \{\hat{l}_2\})$, which (1) extends $\hat{l}_1$ with the dynamic property $*$, represented by the abstract location $\hat{l}_5$, via edge $\hat{l}_1 \mapsto_{\mathrm{P}(*)} \hat{l}_5$, and (2) adds a dependency edge $\hat{l}_2 \mapsto_{\mathrm{D}} \hat{l}_5$, as the property being looked up depends on the value of $\hat{l}_2$. Finally, it sets the variable `options` to $\hat{l}_5$ in the store.

The [Dynamic Property Update] rule evaluates the expressions e_1, e_2, and e_3, respectively denoting the object being updated, the property to be updated, and the assigned value, obtaining three sets of locations L_1, L_2, L_3; then, the rule uses the function NV^* to create a new version of all the objects

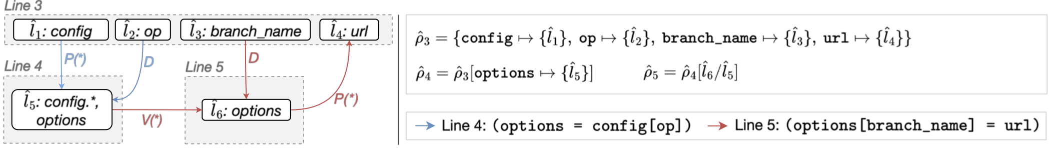


Fig. 3. Sub-MDG of the motivating example in Figure 1a.

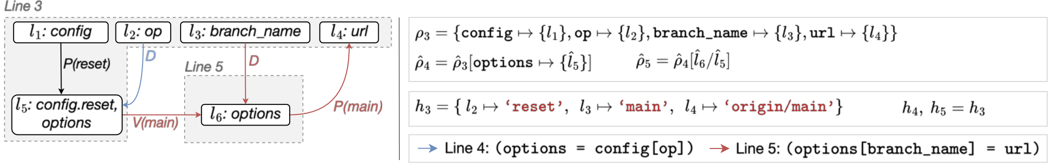


Fig. 4. Concrete sub-MDG of the motivating example in Figure 1a.

in L_1, updating the abstract store and graph accordingly; finally, the rule adds a property edge corresponding to the property being assigned to the new version objects contained in L_1'. The rule [Static Property Update] is similar except that NV is used with p being the last argument.

Back to our example, rule [Dynamic Property Update] is applied when analyzing line 5, where $s =$ `options[branch_name]` $:=_i$ `url`. First, expressions `options`, `branch_name` and `url`, are evaluated to $\{\hat{l}_5\}$, $\{\hat{l}_3\}$ and $\{\hat{l}_4\}$, respectively. Then, the rule calls $\mathsf{NV}_i^*(\hat{g}, \hat{\rho}, \{\hat{l}_5\}, \{\hat{l}_3\})$, which (1) creates a new version of object $\hat{l}_5$, represented by the abstract location $\hat{l}_6$, via edge $\hat{l}_5 \mapsto_{\mathsf{V}*} \hat{l}_6$, and (2) updates the variable `options` to the new abstract location $\hat{l}_6$ in the abstract store. Finally, the rule extends $\hat{l}_6$ with the dynamic property $*$, via edge $\hat{l}_6 \mapsto_{\mathsf{P}(*)} \hat{l}_4$, as `url` is represented by the abstract location $\hat{l}_4$. The [If] rule evaluates both branches of the if statement and combines the results using the least upper bound operator. Finally, the [While] rule computes the least fixed point of the analysis on the body of the loop. When analyzing a loop $\mathsf{while}(e)\{s\}$ on a state $(\hat{g}, \hat{\rho})$, we must find the smallest state $(\hat{g}', \hat{\rho}')$ such that: $(\hat{g}, \hat{\rho}) \sqsubseteq (\hat{g}', \hat{\rho}')$ and $\mathcal{A}(s, \hat{g}', \hat{\rho}') = (\hat{g}', \hat{\rho}')$. Such fixed point is guaranteed to exist because the set of abstract states forms a finite lattice and the analysis is monotone: for any $\hat{g}_1$ and $\hat{g}_2$, $\hat{\rho}_1$ and $\hat{\rho}_2$, and statement s, it holds that: $(\hat{g}_1, \hat{\rho}_1) \sqsubseteq (\hat{g}_2, \hat{\rho}_2) \implies \mathcal{A}(s, \hat{g}_1, \hat{\rho}_1) \sqsubseteq \mathcal{A}(s, \hat{g}_2, \hat{\rho}_2)$.

3.3 Soundness

To better understand the formal properties of our MDG, we first define a concrete semantics for the core language and then show that the MDGs generated by our analysis overapproximate the concrete object layout and structure in the concrete semantics. At a high level, the property established here guarantees that our graph generation algorithm consistently handles dynamic properties such that the abstract graph does not miss any edges that the concrete one generates.

Instrumented concrete semantics: Similar to the abstract store used in the analysis semantics, we define concrete stores, $\rho \in \mathcal{S}to : \mathcal{X} \rightharpoonup \mathcal{L}$, to map program variables to locations. To track the values computed during execution, we use heaps $h \in \mathcal{H} : \mathcal{L} \rightharpoonup \mathcal{V}$ that map locations to values. Compared to directly using values and variables, using the store simplifies dependency tracking. Object structure and dependencies are modeled through the concrete multiversion dependency graphs, $g = (V, E) \in \mathcal{G}$, which are analogous to their abstract counterparts in all respects except that nodes are taken from the set of concrete locations, $V \subseteq \mathcal{L}$, all edge types are known, and any given location l can only be connected to at most one other location via a given property edge $\mathsf{P}(p)$.

Figure 5 shows selected rules for the instrumented big-step semantics of Core JavaScript. The semantics rules take the form $\langle g, h, \rho, s \rangle \Downarrow_c \langle g', h', \rho' \rangle$, meaning that the evaluation of statement s in the initial concrete MDG g, heap h, and store ρ, yields the final graph g', heap h', and store ρ'.

$$\textsc{Dynamic Property Lookup}\quad \frac{[\![e_i]\!]_\rho = l_i \,|_{i=1}^{2} \qquad p = h(l_2) \qquad g[l_1, p] = l' \qquad \rho' = \rho[x \mapsto l'] \qquad g' = g \uplus \{l_2 \mapsto_{\mathsf{D}} l'\}}{\langle g, h, \rho, x :=_i e_1[e_2]\rangle \Downarrow_{\mathsf{c}} \langle g', h, \rho'\rangle}$$

$$\textsc{Dynamic Property Update}\quad \frac{[\![e_i]\!]_\rho = l_i \,|_{i=1}^{3} \qquad p = h(l_2) \qquad (g'', \rho', l') = \mathsf{NV}_{\mathsf{c}}(g, \rho, l_1, p, l_2) \qquad g' = g'' \uplus \{l' \mapsto_{\mathsf{P}(p)} l_3\}}{\langle g, h, \rho, e_1[e_2] := e_3\rangle \Downarrow_{\mathsf{c}} \langle g', h, \rho'\rangle}$$

Fig. 5. Fragment of Instrumented Concrete Semantics: $\langle g, h, \rho, s\rangle \Downarrow_{\mathsf{c}} \langle g', h', \rho'\rangle$

Analogously to the abstract semantics, the concrete semantics also keeps track of the entire history of the program execution by creating a new version of each object whenever it is updated.

The [Dynamic Property Lookup] Rule evaluates the expressions e_1, e_2, respectively denoting the object and property being looked up, and obtains the locations l_1 and l_2. Then, it obtains the location l' denoting the property value and updates the store accordingly. Finally, it adds a dependency edge $l_2 \mapsto_{\mathsf{D}} l'$, as the property being looked up depends on the value of l_2. The [Dynamic Property Update] Rule evaluates the expressions e_1, e_2, and e_3, respectively denoting the object being updated, the property to be updated, and the value assigned to that object, obtaining three locations l_1, l_2, l_3; then the rule extends the graph with a new version of the object represented by l_1 and the edge $l_1 \mapsto_{\mathsf{D}} l'$. Analogously to the abstract counterpart, it adds a property edge corresponding to the property being assigned (l_3) to the new version object (l'). The NV_c function is similar to NV_i function, except that the property name is resolved to a static value, instead of the wildcard '*'.

Similarly to Figure 3, Figure 4 illustrates the concrete sub-MDG, resulting from evaluating the first five lines of the example of Figure 1c, where `config` is the object `{ reset: {}}`, and `op`, `branch_name` and `url` are strings *'reset'*, *'main'*, and *'origin/main'*, respectively. The concrete stores ρ_k represent the content of the concrete store ρ after analyzing line k, and the heaps h_k represent the content of the heap h after analyzing line k. In contrast to the abstract sub-graph of Figure 3, since we know the object's structure and values in the initial state, line 4 only adds a dependency edge $l_2 \mapsto_{\mathsf{D}} l_5$, also mapping the variable `options` to l_5. Line 5 contains a dynamic property update, and, similarly to the abstract sub-graph, we create a new version of the object represented by l_5, and extend the newly created version (l_6) with a property edge pointing to l_4. However, as we know the value of the property being updated, instead of using the wildcard '*', we use the property name 'main'.

Analysis guarantees: We establish that the abstract MDG overapproximates concrete MDG. We define abstraction functions, $\alpha : \mathcal{L} \rightharpoonup \hat{\mathcal{L}}$, that map concrete locations to abstract locations; i.e., $\alpha(l) = \hat{l}$ means that the concrete location l is represented by $\hat{l}$ in the abstract domain. Intuitively, an abstract store $\hat{\rho}$ over-approximates a concrete store ρ according to an abstraction function α if all the variables in the domain of ρ are over-approximated by $\hat{\rho}$. Analogously, an abstract graph $\hat{g}$ over-approximates a concrete graph g if all the edges of g have corresponding edges in $\hat{g}$. Definitions 3.1 formalizes the relation between abstract graphs and concrete graphs.

Definition 3.1 (MDGs Over-Approximation). An abstract MDG $\hat{g}$ is said to over-approximate a concrete MDG g via abstraction function α, written $\hat{g} \sim_\alpha g$, if and only if the following hold:

- $\forall l_1, l_2 \,.\, l_1 \mapsto_{\mathsf{D}} l_2 \in g \implies \alpha(l_1 \mapsto_{\mathsf{D}} l_2) \in \hat{g}$
- $\forall l_1, l_2, p \,.\, l_1 \mapsto_{\mathsf{P}(p)} l_2 \in g \implies \alpha(l_1 \mapsto_{\mathsf{P}(p)} l_2) \in \hat{g} \,\vee\, \alpha(l_1 \mapsto_{\mathsf{P}(*)} l_2) \in \hat{g}$
- $\forall l_1, l_2, p \,.\, l_1 \mapsto_{\mathsf{V}(p)} l_2 \in g \implies \alpha(l_1 \mapsto_{\mathsf{V}(p)} l_2) \in \hat{g} \,\vee\, \alpha(l_1 \mapsto_{\mathsf{V}(*)} l_2) \in \hat{g}$

Where $\alpha(l_1 \mapsto_\tau l_2)$ is used to mean $\alpha(l_1) \mapsto_\tau \alpha(l_2)$.

In the following, we write $\hat{g}, \hat{\rho} \sim_\alpha g, \rho$ to mean that $\hat{g}$ over-approximates g and $\hat{\rho}$ over-approximates ρ according to α. Theorem 3.2 states that if the initial abstract state over-approximates the initial concrete state, then the final abstract state also over-approximates the final concrete state.

Table 1. Base graph traversals. Notation as defined in Section 3.1.

Traversal	Description	Pattern
BasicPath_n^s	Finds sequence of distinct k nodes connecting o_s to o_n, via $k+1$ distinct edges. If n is not specified, return all distinct paths that start in o_s. If $s = n$, return o_n.	$o_s \{\xrightarrow{\tau_k} o_k\}^k \xrightarrow{\tau_n} o_n$, where $k \geq 0$
UntaintedPath_n^s	Finds paths that include an object assignment followed by an object lookup on the same property p. Returns the path.	$(o_1 \xrightarrow{V(p)} o_2 \{\xrightarrow{\tau_k} o_k\}^k \xrightarrow{P(p)} o_3)$, where $k \geq 0$
TaintPath_n^s	Finds a dependency path between node o_s and node o_n. If n is not specified, return all distinct paths that start in o_s.	$\text{BasicPath}_n^s \setminus \text{UntaintedPaths}_n^s$
Arg_f^i	Matches a function f and returns its i-th argument [61].	-
ObjLookup_i^*	Searches for an object lookup via dynamic property. Returns o_k.	$o_i \xrightarrow{P(*)} o_k$
$\text{ObjAssignment}_{p,v}^{*,i}$	Searches for an object o_i assignment via dynamic property. Returns o_v and o_p.	$o_i \xrightarrow{V(*)} o_v \xrightarrow{P(*)} o_p$

Table 2. Graph Traversals for detecting taint-style and prototype pollution vulnerabilities.

Vulnerability	Graph Queries
Command Execution	$\text{TaintPath}^s \circ \text{ARG}_f^n$, where $\{ (f, n) \in \text{Sinks}_{command_execution}, o_s \in \text{Sources} \}$
Code Injection	$\text{TaintPath}^s \circ \text{ARG}_f^n$, where $\{ (f, n) \in \text{Sinks}_{code_injection}, o_s \in \text{Sources} \}$
Path Traversal	$\text{TaintPath}^s \circ \text{ARG}_f^n$, where $\{ (f, n) \in \text{Sinks}_{path_traversal}, o_s \in \text{Sources} \}$
Prototype Pollution	$(\text{ObjLookup}_i^* \circ \text{ObjAssignment}_{p,v}^{*,i}) \circ (\text{TaintPath}_i^s \cap \text{TaintPath}_v^s \cap \text{TaintPath}_p^s)$, where $o_s \in \text{Sources}$

Theorem 3.2 (Soundness with Full Knowledge). *For all graphs $\hat{g}, \hat{g}', g, g'$, abstract stores $\hat{\rho}, \hat{\rho}'$, concrete stores ρ, ρ', heaps h and h', statement s, and abstraction function α, it holds that:*

$$\mathcal{A}(s, \hat{g}, \hat{\rho}) = (\hat{g}', \hat{\rho}') \wedge \hat{g}, \hat{\rho} \sim_\alpha g, \rho \wedge \langle g, \rho, h, s \rangle \Downarrow_c \langle g', h', \rho' \rangle \implies \exists \alpha'. \, \alpha' \geq \alpha \wedge \hat{g}', \hat{\rho}' \sim_{\alpha'} g', \rho'$$

4 GRAPH.JS

To validate our approach, we implemented Graph.js, a novel static vulnerability scanner for JavaScript code, based on MDG graphs. Here, we describe the graph queries performed by Graph.js to detect taint-style and prototype pollution vulnerabilities and the implementation of Graph.js.

Basic graph traversals: A graph traversal, as proposed by Yamaguchi et al. [61], is a function $\mathcal{T} : \mathcal{P}(\mathcal{V}) \rightarrow \mathcal{P}(\mathcal{V})$, that maps a set of nodes to another set of nodes, where $\mathcal{V}$ is a set of nodes and $\mathcal{P}(\mathcal{V})$ is the power set of $\mathcal{V}$. This definition allows for chaining multiple traversals together, e.g., $\mathcal{T}_0 \circ \mathcal{T}_1$ represent two graph traversals $\mathcal{T}_0$ and $\mathcal{T}_1$ chained together, using a function composition $\circ$. It also allows for filtering traversals, e.g., $\mathcal{T}_0 \backslash \mathcal{T}_1$ represent a graph traversal $\mathcal{T}_0$, excluding the paths included in $\mathcal{T}_1$.

Table 1 defines elementary traversals, which are the building blocks for more complex traversals for finding vulnerabilities (summarized in Table 2). We first define BasicPath_n^i, which finds a path between o_i and o_n. For instance, the MDG of Figure 1c contains a basic path between o_1 and o_8, presented as $o_1 \xrightarrow{P(*)} o_5 \xrightarrow{V(*)} o_6 \xrightarrow{V(cmd)} o_7 \xrightarrow{P(cmd)} o_8$. The other traversals are built upon this notion, adding more restrictions to the path. For instance, TaintedPath_n^i returns all paths returned by BasicPath_n^i, except those that are also included in UntaintedPath_n^i (explained later in this section).

In Table 2, Sources represent untrusted input, e.g., user input, and Sinks_t represent functions that are classified as unsafe sinks for vulnerability of type t. The list of Sinks considered by Graph.js can be set dynamically via a configuration file, where each sink is defined by a JavaScript native function or a function imported from an external package f, and the sensitive argument(s) n.

Taint-style vulnerability queries: The detection of the three taint-style injection vulnerabilities, i.e., code injection, command injection, and path traversal, share the same graph traversal pattern and differ from each other only in their unsafe sink functions. Code injection vulnerabilities, which consist of injecting code that is later executed by the server, can involve sinks such as `eval` and `Function()`. The sinks of command injection vulnerabilities, which entail executing arbitrary commands in the server's operating system, include `exec`, `child_process.spawn`, and `child_process.execFile`. Path traversal vulnerabilities, which allow for accessing restricted files on the server by injecting malicious input, use sinks such as `fs.readFile` and `fs.createReadStream`.

Graph.js detects such scenarios by searching for paths connecting the tainted source to an unsafe sink in the MDG of the program being analyzed. The traversals for each taint-style vulnerability are shown in Table 2. First, Graph.js performs a graph traversal $\textsc{TaintPath}^s$ in the MDG, which returns all paths that start in o_s but excludes untainted paths. Untainted paths contain a new version edge `V(prop)` followed by an object lookup of the same property `P(prop)`; this pattern indicates that that (tainted) property has been overwritten and is no longer tainted through that path. This traversal is chained with $\textsc{Arg}^n_f$ to return tainted paths that end in argument n of an unsafe function call f.

Prototype pollution vulnerability queries: A prototype pollution allows an attacker to manipulate the prototype of an object. In this work, we focus specifically on `Object.prototype`, which is the topmost object on every prototype chain. Graph.js's prototype pollution queries search for an object lookup where the attacker controls the property, followed by an object assignment over the result of the initial lookup where the attacker controls both the property and the value assigned. Table 2 shows the traversal for prototype pollution. In a first step, Graph.js performs a graph traversal $\textsc{ObjLookup}^*_i$ chained with $\textsc{ObjAssignment}^{*,i}_{p,v}$, which together search for an object lookup ($o_i \xrightarrow{P(*)} o_k$), followed by a sub-object assignment ($o_k \xrightarrow{V(*)} o_v \xrightarrow{P(*)} o_p$). This traversal is then chained with three $\textsc{TaintPath}^s_n$, that, similarly to taint-style vulnerabilities, check if the attacker controls o_i, o_v and o_p, sequentially. We can identify this pattern in Figure 1c, where there is an object lookup, followed by an object assignment over the result of the initial lookup, by identifying the pattern $o_1 \xrightarrow{P(*)} o_5 \xrightarrow{V(*)} o_6 \xrightarrow{P(*)} o_4$. The property of the first lookup o_5, the property of the sub-object assignment o_6 and the value of the assignment o_4 are tainted through paths $o_2 \xrightarrow{D} o_5$, $o_3 \xrightarrow{D} o_6$, and o_4, respectively.

Implementation: Graph.js takes *npm* packages as input and reports potential vulnerabilities. It is composed of two processing pipelines: *MDG generator* and *graph engine*. The MDG generator is implemented with 6K lines of TypeScript code and is responsible for parsing and transpiling JavaScript programs to the core JavaScript and then producing the corresponding MDG. Graph.js uses Esprima v4.0.1 [14] for parsing before generating the program's AST and CFG in line with the original CPGs introduced by Yamaguchi et al. [62]. Then, the MDG builder creates the MDG. The query engine consists of 500 lines of Python code and is responsible for importing the MDG into a graph database and executing a set of queries on the MDG. We used Neo4j v4.2.1 [40] as the graph database engine and wrote two Cypher [39] queries with 80 lines of code, one for the taint-style vulnerabilities and the other for the prototype pollution.

5 EVALUATION

In this section, we evaluate the effectiveness and performance of Graph.js against *npm* packages. Specifically, our evaluation aims to answer the following three central research questions:

- **RQ1:** How effective is Graph.js in detecting vulnerabilities and how does it compare to ODGen?
- **RQ2:** Can Graph.js find zero-day security vulnerabilities in real-world *npm* packages?
- **RQ3:** What is Graph.js's performance and how does it compare to ODGen?

Table 3. Summary of the reference datasets per vulnerability type: *"Raw Total"* show the total number of packages in the dataset; *"Total"* show the number of packages excluding incorrect annotations and duplicates.

Vulnerability Type	**CWE**	*VulcaN*		*SecBench*		**Total**	**Distribution (%)**
		Raw Total	Total	Raw Total	Total		
Path Traversal	CWE-22	5	5	170	161	166	27.5%
Command Injection	CWE-78	92	87	101	82	169	28.03%
Code Injection	CWE-94	41	33	40	21	54	8.96%
Prototype Pollution	CWE-1321	98	94	192	120	214	35.49%
Total		236	219	503	384	603	100%

5.1 Experimental Setup

To answer our research questions, we leverage three datasets. Two of these, are complementary vulnerability datasets from prior work which we use as ground truth: VulcaN [6] and SecBench [5]. They have different vulnerability distributions over vulnerability types, summarized in Table 3. Combined, these two reference datasets provide a comprehensive set of vulnerabilities for our evaluation. The third dataset, which we call Collected, is a set of popular packages (>2K weekly downloads) that we downloaded from the *npm* repository. Next, we describe these three datasets.

- *Dataset 1 (VulcaN):* VulcaN [6] is a vulnerability benchmark with 957 *npm* package versions that contain confirmed Node.js vulnerabilities, reported in the GitHub Advisory Database [22]. Each package contains one or more vulnerabilities, each of which is annotated with the sink and source line number. Out of the 957 packages, we selected all 174 that contain vulnerabilities that Graph.js targets: code injection, command injection, path traversal, and prototype pollution. These selected packages contain a total of 236 vulnerabilities. Out of the 236 vulnerabilities, we excluded 17 that either have incorrect annotations (e.g., the annotated vulnerability type is different from the correct type and the correct type is outside our scope) or are located in an external imported package, whose source is unavailable for analysis.
- *Dataset 2 (SecBench):* SecBench [5] comprises 601 vulnerable packages, reported in the GitHub Advisory Database [22], Snyk [49], and Huntr.dev [24]. Each package only includes a single vulnerability and is annotated with the sink line number. Out of the 601 vulnerabilities, we selected a total of 384. We excluded 217 vulnerabilities in total: 98 refer to out-of-scope ReDoS vulnerabilities, 71 are incorrectly annotated (e.g., non-existent or wrong sink line, or missing files), and 38 were already included in VulcaN. The rest of the excluded cases either are unavailable for download or their file type was TypeScript. While our methodology applies to TypeScript, Graph.js uses a JavaScript parser that cannot handle TypeScript.
- *Dataset 3 (Collected):* Contains 32,137 (~32K) popular real-world *npm* packages, that we crawled from the *npm* repository in September, 2023. Following Snyk's guidelines, we consider a package popular if it had more than 2,000 weekly downloads at the time of collection.

To compare Graph.js with prior work, we set up the open-source implementation of ODGen [36] and run it on our ground truth datasets. We chose ODGen because it is the state-of-the-art CPG-based vulnerability detection tool for *npm* packages. We evaluated Graph.js on the same set of vulnerability types for which ODGen was evaluated. Similarly to ODGen, our evaluation does not include XSS and SQL injection vulnerabilities because the identification of these types of vulnerabilities relies on application-specific sinks. Furthermore, Brito et al. [6], the authors of VulcaN, present an empirical study for evaluating JavaScript vulnerability detection tools on *npm* packages, and elects ODGen as offering the most favorable trade-off between effectiveness and precision, ranking highest in precision and fourth in overall effectiveness among the assessed tools.

Table 4. Effectiveness and precision of Graph.js and ODGen for the VulcaN and SecBench datasets combined.

CWE	Total	Graph.js						ODGen					
		TP	FP	TFP	Recall	Precision	F1	TP	FP	TFP	Recall	Precision	F1
CWE-22	166	161	56	30	0.97	0.84	0.9	131	6	0	0.79	1	0.88
CWE-78	169	160	151	9	0.95	0.95	0.95	111	110	77	0.66	0.59	0.62
CWE-94	54	47	20	13	0.87	0.78	0.82	23	86	84	0.43	0.21	0.29
CWE-1321	214	126	112	85	0.59	0.60	0.59	39	18	13	0.18	0.75	0.29
Total	603	494	339	137	0.82	0.78	0.80	304	220	174	0.50	0.64	0.56

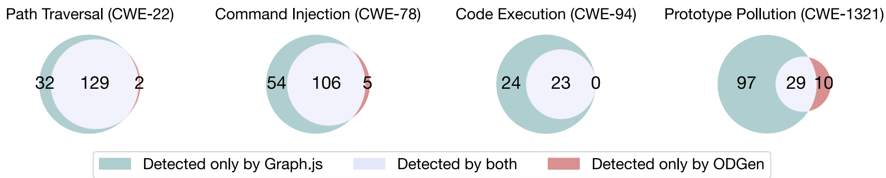


Fig. 6. Venn diagram of the vulnerabilities detected by Graph.js and ODGen.

Our testbed consisted of 6 64-bit Ubuntu 22.04.3 servers with 64GB of RAM and 2x Intel(R) Xeon(R) Gold 5320 2.2GHz CPUs. We conducted the experiments involving the reference datasets on a single server. To analyze the Collected dataset, we used six servers to distribute the load and speed up the analysis. We set the total analysis timeout to five minutes.

5.2 RQ1: Effectiveness in Vulnerability Detection

We assess Graph.js's effectiveness in detecting vulnerabilities and compare it to ODGen by running both tools on our ground truth datasets: VulcaN and SecBench. Results are summarized in Table 4.

True positives (TP): We consider a reported vulnerability a true positive if the vulnerability type and sink line number reported by the tools match the dataset annotations. For ODGen, a report is also considered a true positive if it only correctly detects the vulnerability type but does not pinpoint the sink code line[2]. We include all vulnerabilities reported by ODGen until it times out.

The columns titled "TP" in Table 4 show that Graph.js can detect 1.63× more vulnerabilities than ODGen, identifying 494 versus 304 vulnerabilities. In particular, Graph.js finds twice as many code injection vulnerabilities (CWE-94) and three times as many prototype pollution vulnerabilities (CWE-1321) as ODGen. The improvement is largely attributable to Graph.js building simpler graphs (as detailed in §5.4), enabling it to complete analysis more quickly than ODGen. In 95% of the cases, ODGen timed out without detecting any vulnerability, struggling particularly to recognize prototype pollution patterns. A contributing factor is that ODGen's abstract interpretation often fails to complete analysis of prototype pollutions involving recursion and loops. In §5.5, we present a case study of a prototype pollution vulnerability where Graph.js's version edges and summary fixed-pointed representation for loops enable a speedy detection, whereas ODGen times out.

Regarding the specific vulnerabilities each tool can identify, Figure 6 reveals that the set of vulnerabilities detected by Graph.js largely subsumes those found by ODGen. Apart from 17 vulnerabilities detected exclusively by ODGen, Graph.js identifies all other vulnerabilities that

[2]We saw many instances of such reports from ODGen. Since the lack of information about the sink may be an issue related to the implementation, not the approach itself, such a report is credited as a true positive. Thus, our reported TP for ODGen is a conservative upper bound.

ODGen detects, i.e., 94%. The reasons for Graph.js missing said vulnerabilities are similar to those for false negatives in general. The main reason for false negatives in Graph.js is the set of unimplemented JavaScript functionalities not represented by the MDG, resulting in missing dependency edges in the graph. Currently, MDGs do not provide full support for the `arguments` and the `this` keywords, some array operations, and `Function.prototype.call()`. When compared to taint-style detection, Graph.js exhibits a lower TP rate in prototype pollution detection. This lower rate is partly because prototype pollution patterns often involve third-party *npm* packages, such as *for-own* and *for-in*, which are instrumental in leading to vulnerabilities. However, since the code of these external packages is not represented in the MDG, the prototype pollution query fails to recognize the associated vulnerability pattern. Additionally, prototype pollution sources frequently use the `arguments` keyword, which, as noted above, is not fully supported by MDGs.

False positives (FP) and true false positives (TFP): We consider a false positive (FP) when a vulnerability is reported by a tool but was not annotated as such in the original dataset. However, it is important to note that both tools often discovered additional confirmed vulnerabilities not annotated in the datasets. This occurrence is not unusual, given that the datasets are not complete. For instance, SecBench reports only one vulnerability per package, yet it is common for vulnerable packages to contain multiple exploitable unsafe sinks; e.g., CVE-2019-10783 describes three exploitable sinks for *lsof_v0.1.0* while SecBench only reports one. Therefore, in our classification, we specifically designate a result as a true false positive (TFP) only when it does not correspond to an actual, exploitable vulnerability for which we have been able to generate a successful exploit.

Table 4 presents both of these metrics under the columns "FP" and "TFP". Although Graph.js reports a higher number of false positives than ODGen (339 versus 220, respectively), the datasets are incomplete. Consequently, a false positive identified by a tool could be a real, unannotated vulnerability in the dataset, making it a true positive. To account for potential inaccuracies in false positive reporting due to incomplete dataset annotations, we focus on the TFP metric. Analyzing this metric reveals that Graph.js outperforms ODGen by reporting 37 fewer true false positives.

In Graph.js, the main causes for TFPs are as follows. For taint-style TFPs, a tainted value reaches an unsafe sink, but it only occurs under highly specific circumstances, which prevent a successful exploitation of the vulnerability. In the case of prototype pollution TFPs, this issue arises because our graph traversals do not evaluate *if* conditions, which leads to the reporting of recursive object assignments as sinks, even if cases where the *if* condition is not executed. While these assignments may contribute to the vulnerability's existence, they do not directly pollute `Object.prototype`.

In comparison, ODGen has no TFPs in path traversal (CWE-22). This is because ODGen uses very specific queries that only search for the unsafe sinks in the context of a web server, i.e., the tainted path must pass in functions `CreateServer` or `CreateHttpServer`. For prototype pollution vulnerabilities, ODGen has only 13 TFPs, though it also has a low TP rate, as discussed above.

Precision, recall, and F1-score: Table 4 also presents the precision, recall, and F1-score of both Graph.js and ODGen. Precision is computed as $TP/(TP+TFP)$, where TP only includes the annotated vulnerabilities. Recall is calculated as $TP/(TP+FN)$. F1-score is determined using the harmonic mean of precision and recall: $(2\times \mathit{Precision} \times \mathit{Recall})/(\mathit{Precision} + \mathit{Recall})$. Globally, Graph.js achieves a precision of 78%, which is an increase of 14 points over ODGen's precision. The most significant improvement of Graph.js over ODGen is observed on the recall, which rises from 50% to 82%, respectively, playing a decisive role in boosting Graph.js's F1-score by 1.42× that of ODGen.

Takeaway 1: Graph.js detects 82% of the reported vulnerabilities in the ground truth datasets, outperforming ODGen by 1.63×. It achieves a total precision of 78%, which is 1.23× higher than ODGen's precision, and an F1-score of 80%, surpassing ODGen by 1.42×.

Table 5. Vulnerabilities found by Graph.js in the Collected dataset

Vulnerability	Reported	Checked	Exploitable	Unreported	FP
Path Traversal	1,223	26	4	3	21
Command Injection	384	159	71	26	91
Code Injection	701	201	10	4	191
Prototype Pollution	361	33	16	16	15
Total	2,669	419	101	49	318

5.3 RQ2: Vulnerability Detection in the Wild

We assess Graph.js' capability to detect zero-day vulnerabilities, by applying Graph.js to analyze the 32K *npm* packages from the Collected dataset. We define a vulnerability as zero-day if (1) a human expert confirms the vulnerability with a generated exploit, (2) we cannot find any information about the vulnerability online, and (3) it is not an intended functionality of the package.

Table 5 summarizes our results. Initially, Graph.js reported 2,669 vulnerabilities (see the column "Reported"). From those, we randomly sampled 396 packages to manually analyze, corresponding to 419 vulnerabilities (column "Checked"). To limit the manual effort, we prioritized analyzing packages with less than 10 files and code injection and command injection vulnerabilities, as they typically are simpler, making it easier to create the exploits to confirm the vulnerabilities, per our prior experience. From the 419 vulnerabilities we manually checked, we successfully created an exploit for 101 of them (column "Exploitable"); 49 of them were not previously reported and were not an intended functionality of the package.

We detected 318 non-exploitable vulnerabilities (i.e., false positives), mainly due to the presence of sanitization functions between tainted sources and unsafe sinks, or tainted data reaching unsafe sinks only under highly specific circumstances, making the creation of an exploit exceptionally challenging or even impossible, especially in code injection vulnerabilities. In particular, the high false positive rate in detecting code injection vulnerabilities is primarily caused by us considering the Node.js function `require` as an unsafe sink. This assumption is not always true. Although an attacker is able to control the imported package name, most times it is not able to also execute an exported function of that package or control the arguments. This is not manifested as an issue in the ground truth datasets, because there were few `require` functions with dynamic package names.

Ethical disclosure: We performed responsible disclosure for the confirmed vulnerabilities. We contacted all package maintainers, either directly or via Snyk, to explain the discovered vulnerabilities, along with proof of vulnerability. We were unable to find contact information for 3 of them. We provided a 30-day response deadline, and if we do not receive any communication from package maintainers within 30 days we report the vulnerability to CVE (MITRE) [12]. At the time of submission, we obtained 3 CVEs: CVE-2023-26156, CVE-2023-49210, and CVE-2023-40582. We received 6 replies, of which 4 developers confirmed the vulnerability: #1 deprecated the package and #2 produced a fix, as a result of our reporting; #3 asked us to create a pull request with a warning, and #4 said that the package was not being maintained anymore, and won't investigate a possible fix. #5 asked for a timeline extension, and #6 did not agree with our assessment.

Takeaway 2: Graph.js found 101 exploitable vulnerabilities in the *Collected* dataset, where 49 of them were not previously reported and were not an intended functionality of the package.

5.4 RQ3: Performance Evaluation

Execution time: We measure the time taken by Graph.js and ODGen to detect vulnerabilities in each *npm* package from both our reference datasets, consisting of 160 packages from VulcaN and

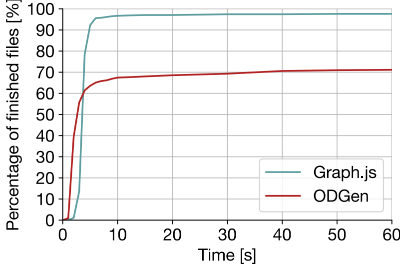


Fig. 7. CDF of total time to finish analysis.

Table 6. Average time, taken by each analysis phase.

CWE	#	Graph.js			ODGen		
		Graph	Traversals	Total	Graph	Traversals	Total
CWE-22	166	1.51s	2.2s	3.71s	1.19s	0.5s	1.69s
CWE-78	169	1.65s	2.11s	3.82s	3.75s	0.56s	4.32s
CWE-94	54	4.52s	2.15s	6.81s	1.4s	0.44s	1.84s
CWE-1321	214	2.4s	2.97s	5.47s	3.42s	12.04s	15.45s
Total	603	2.1s	2.44s	4.61s	2.68s	2.73s	5.41s

384 from SecBench. Figure 7 plots the cumulative distribution function (CDF) of the percentage of packages that each tool managed to analyze according to the analysis time of each package. We depict the first 60 seconds, although each package can take as much as 5 minutes to be processed as per our pre-defined analysis timeout. We can see that Graph.js completes the analysis for almost all the 544 packages from both datasets. Within just 10 seconds, Graph.js had already finished analyzing more than 95% of the packages. The remaining long tail represents only 10 packages that Graph.js was unable to analyze within the 5-minute limit, accounting for 1.8% of the total. In stark contrast, ODGen showcases limited scalability, successfully analyzing only 71.5% of the packages.

Interestingly, in the initial five seconds, ODGen outperforms Graph.js by analyzing packages considerably faster. For instance, by the 2-second mark, ODGen had already analyzed 39.5% of the packages, while Graph.js had completed only 1.1% of the total. To understand the reasons behind this difference, we calculated the average package analysis times of Graph.js and ODGen for packages that did not time out, grouped by vulnerability type. We further break down the time into two phases: graph construction and graph traversals. Table 6 summarizes these findings. In the table, the columns “Graph” and “Traversals” indicate the average time each tool takes to build its graph and execute the corresponding query. “Total” represents the total time.

Considering all vulnerability types, Graph.js analyzes a package on average 0.8 seconds faster than ODGen. However, a more detailed analysis by vulnerability type reveals some interesting insights. On the one hand, ODGen’s graph traversal is considerably more efficient for most taint-style vulnerabilities (CWE-22, CWE-78, and CWE-94), with Graph.js potentially taking up to 4.8 times longer to process a package. This efficiency in ODGen can be attributed to its queries being natively implemented in Python as part of the tool, whereas Graph.js relies on Neo4j’s query engine, which is slower. Consequently, taint-style detections tend to complete more quickly in ODGen than in Graph.js, explaining the performance discrepancy highlighted in the CDF. Notably, for prototype pollution (CWE-1321), the situation is reversed: ODGen takes significantly longer to analyze prototype pollution vulnerabilities. This delay is primarily due to the considerable expansion in the size of its ODG caused by patterns associated with prototype pollution vulnerabilities [36].

Takeaway 3: Graph.js’s average package analysis time is 4.61 seconds, and it successfully analyzes 98.2% of all packages from our reference datasets, demonstrating greater scalability than ODGen.

Graph complexity: We assess the complexity of MDG, by measuring the number of nodes and edges for all analyzed packages, and compare it with ODGen. Table 7 presents the graph size of Graph.js and ODGen for the reference datasets combined, grouped by the number of lines (LoC) of the analyzed package. To ensure a fair comparison with ODGen, we included the AST and CFG nodes used to generate the final MDG, even though they are not used in the queries. The “#” column represents the total number of packages and the “# Graphs” column represents the number

Table 7. Graph complexity of Graph.js and ODGen for the VulcaN and SecBench datasets combined.

LoC	#	Graph.js					ODGen				
		# Graphs	Nodes		Edges		# Graphs	Nodes		Edges	
			Avg.	Min/Max	Avg.	Min/Max		Avg.	Min/Max	Avg.	Min/Max
[0, 50)	220	220	185	38/687	252	46/976	199	5,594	1,851/103,655	2716	1,241/32,405
[50, 100)	116	116	569	200/1,092	797	264/1,504	85	7,227	1,820/61,487	3,582	1,227/16,386
[100, 200)	98	98	1,180	576/2,474	1,640	826/3,292	67	25,989	1,820/477,121	10,444	1,227/73,891
[200, 1k)	87	85	3,095	701/10,416	4,356	963/14,058	37	23,064	3,978/97,070	11,734	3,101/40,540
[1k, 2k)	11	10	9,474	6,284/18,006	13,585	9,170/25,674	2	14,016	12,041/15,992	12,014	9,253/14,776
[2k, 30,605]	12	5	47,841	15,121/62,158	72,237	20,875/89,677	0	-	-	-	-
Total	544	534	1539	38/62,158	2,209	46/89,677	390	11,187	1,820/477,121	5,146	1,227/73,891

of graphs that each tool was able to generate before timing out. Note that the total number of packages is different than the number of vulnerabilities presented in previous sections, because one package may contain more than one vulnerability (see §5.1). Similarly to §5.4, we measure the number of edges and nodes of the graph generated by ODGen for each vulnerability type alone. The graphs built by Graph.js are significantly simpler, having on average 7.2× fewer nodes and 2.3× fewer edges than ODGen. MDGs grow linearly with the number of lines of code given that our fixed-point computation algorithm only generates a single node per allocation site, re-using the same node in every iteration. Conversely, ODGen allocates a new node every time an object initializer command is analyzed, leading to the object explosion problem noted by its authors.

Takeaway 4: In 99% of the cases, Graph.js generates graphs significantly smaller than ODGen.

5.5 Case Study

We highlight a case study, sourced from the reference datasets, that showcases a prototype pollution vulnerability in the context of a loop. Figure 8 presents a code snippet adapted from the *npm* package *set-value v3.0.0*, which is used to set nested properties on an object using dot notation, and is susceptible to a prototype pollution vulnerability (CVE-2021-23440). The MDG of the prototype pollution vulnerability presented in Figure 8 is illustrated in Figure 9. The edges are numbered according to their creation timestamp.

The initial graph contains o_1, o_2, and o_3, as these are the function parameters. In line 2, object o_4 *(path)* is created with a dynamic property, which depends on o_1 (edges ① and ②). In line 3, o_4 is extended with property *length* (edge ③). When we execute the first iteration of the loop, variable `obj` is mapped to o_1; so, in line 6, we update a dynamic property on o_1 with a dynamic value, similarly to the running example (edges ④,⑤, ⑥ and ⑦). In line 8, we have two versions of `obj`: o_1, if the *if* branch was not executed, and o_4 otherwise; so, we only extend o_1 with a dynamic property edge (⑧,⑨), as o_7 already has one, and map `obj` to $\{o_8, o_9\}$. Now, we execute the second (and last) iteration of the loop. In line 6, we update a dynamic property on $\{o_8, o_9\}$ with a dynamic value, as before. Due to our cyclic representation, we generate the same abstract location for the same literal object. So, we add edges ⑤,⑥, ⑦ and ⑩ when updating o_8, and edges ⑤,⑥, ⑦ and ⑪ when updating o_9. Finally, similarly to the first iteration, extend o_8 and o_9 with a dynamic property edge (⑫ and ⑬). We can easily recognize the prototype pollution pattern by identifying the pattern $o_1 \xrightarrow{P(*)} o_9 \xrightarrow{V(*)} o_7 \xrightarrow{P(*)} o_6$. The property of the lookup o_9, the property of the sub-object assignment o_7 and the property of the first lookup o_8 are tainted through paths $o_6 \xrightarrow{D} o_9$, $o_6 \xrightarrow{D} o_7$, $o_7 \xrightarrow{D} o_8$, respectively.

6 DISCUSSION

Currently, Graph.js suffers from two types of limitations. One pertains to the inherent limitations of static analysis, as it cannot precisely analyze programs that rely on dynamic features of the language. Graph.js only analyzes dynamic function calls that can be resolved statically (e.g., we know statically to which function a given variable/property points to) and does not support dynamic

```
1  function setValue(obj, dotPath, value) {
2      const path = dotPath.split(".")
3      for (let i = 0; i < path.length; i++) {
4          const key = path[i]
5          if (i === path.length - 1) {
6              obj[key] = value
7          }
8          obj = obj[key]
9      }
10 }
```

Fig. 8. SetValue function (CVE-2021-23440)

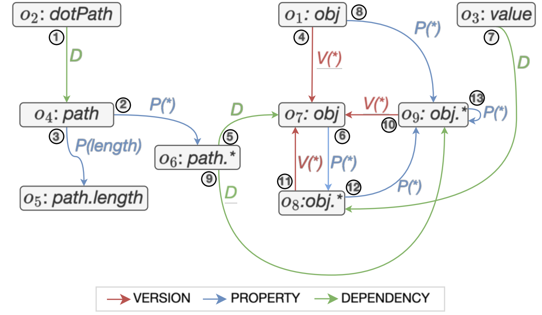


Fig. 9. MDG of the program given in Figure 8

code evaluation with eval and the Function constructor. It handles arrays similarly to objects, evaluating indexes as property names, but it may introduce ambiguities, e.g., when the number of elements cannot be determined statically. Nonetheless, Graph.js is able to analyze packages with unimplemented features but can miss vulnerabilities.

Graph.js outperforms ODGen due to our design decision of not to keep the full AST and CFG information and not to unfold loops and recursive calls. One implication is that the order of two property reads `x.a` and `x.b` that are not separated by updates to `x`, cannot be distinguished in MDG. However, such reads could be reordered while preserving the program semantics. The evaluation results indicate that the precision we gave up allowed Graph.js to be more performant in finding vulnerabilities that we target.

The precision of Graph.js is bounded by the queries presented in §4, which may not encode all the patterns that match a specific vulnerability type, or detect if proper sanitization was employed. Graph.js's queries can be expanded to identify other taint-style vulnerabilities, such as SQL injection, without modifying the underlying MDG. For instance, to detect SQL injections, one can supply common sinks like `mysql.connection.query`. The query can also be extended to not report program-specific sanitization functions, reducing false positives.

The soundness proofs established that the abstract graph over-approximates the concrete graph. We could go one step further and define concrete semantics that uses a regular object graph, as opposed to the multiversion one. Collapsing the multiversion graph to include only the latest version would yield the regular object graph. Using this concrete semantics, we can define concrete attack traces. The soundness proof could then be used to show that if there is a real attack trace on the concrete semantics, then there is a corresponding pattern in the concrete MDG, which in turn, means that the pattern exists in the abstract graph, i.e., our MDG does not miss any vulnerabilities.

7 RELATED WORK

Program analysis using CPGs: CPGs [61] have been used for detecting security vulnerabilities in web application code for various languages, including PHP [2, 46], JavaScript [10, 29, 36], and WebAssembly [7]. Khodayari and Pellegrino [29] explored CPGs for detecting cross-site request forgery vulnerabilities in client-side JavaScript. Li et al. [35] introduced a CPG version for identifying prototype pollution vulnerabilities in Node.js applications using an Object Property Graph (OPG). OPG enhances the original CPG by representing JavaScript objects, including variable names and properties. ODGen [36] evolved from this work, introducing the Object Dependence Graph (ODG) to detect Node.js vulnerabilities using graph queries. However, ODGen's combined CPG-ODG structure is complex and lacks soundness proofs. Our work addresses these limitations by tracking

object versions using an MDG to improve the effectiveness and trustworthiness of the analysis. Although MDGs were designed to address JavaScript-specific challenges, their underlying ideas can also be applied to reasoning about mutation in other dynamic languages, such as PHP and Python.

Vulnerability detection tools for Node.js applications: Various studies [1, 6, 53, 64] have reported a profusion of security vulnerabilities in *npm* packages and Node.js applications. To detect them, many existing tools employ dynamic code analysis [9, 23, 52–54, 60], sometimes in combination with symbolic execution Xiao et al. [60]. Some approaches focus on specific vulnerabilities, such as prototype pollution [30, 35] or code and command injection vulnerabilities [41]. Others, also generate exploits [9, 27, 43]. Brito et al. [6] studied several static vulnerability scanners, including ODGen and CodeQL [10], and found that ODGen offers the best trade-off between effectiveness and precision among the evaluated tools. Graph.js, while sharing the goal of detecting vulnerabilities in *npm* packages with the aforementioned work, explores a distinct analysis technique.

Vulnerability detection for client-side JavaScript code on the browser. Researchers have utilized a variety of code analysis techniques to detect malicious JavaScript code [8, 15, 16], vulnerabilities in JavaScript code in web applications [29, 34, 37, 38, 46, 55], and browser extensions [17, 50, 63]. DoubleX [17], leverages an Extension Dependence Graph for enhanced automated detection of vulnerable extensions. Although our focus is predominantly on server-side *npm* packages, Graph.js can in principle be adapted to the client side. Such an extension would require handling browser-specific APIs and analysis of event handlers. We leave this exploration for future research.

Static analysis of JavaScript code: Abstract interpretation [11] analyzes a program in an abstract domain rather than the concrete domain in which it operates. Some tools leverage abstract interpretation for analyzing errors in JavaScript code [3, 13, 26, 28, 31, 42, 44, 45]. Although CPG (and MDG) construction is not abstract interpretation in the classical sense defined by Cousot and Cousot [11], abstractly executing the program to construct the CPG shares similarities with computations in an abstract domain, as both aim to capture higher-level properties of programs. Points-to-analysis for JavaScript [18, 25, 51] aims to identify potential memory locations that a pointer or reference variable might target. While Graph.js integrates points-to information within its MDG, it includes additional information on the program's behavior to enhance the vulnerability analysis. In particular, instead of computing an abstract representation of the final concrete states, it models the entire program execution with new version edges.

8 CONCLUSIONS

We introduce Multiversion Dependency Graph (MDG), an efficient graph-based data structure for statically detecting common vulnerabilities in JavaScript programs. MDG relies on a single graph that models the state evolution of objects and properties, greatly reducing graph and query complexity. We implemented Graph.js, a static vulnerability scanner for *npm* packages based on MDG graphs, that detects taint-style and prototype pollution vulnerabilities. Our evaluation shows that Graph.js detects 82% of the reported vulnerabilities in the ground truth dataset, and is able to analyze 95% of the packages in under 10 seconds. Additionally, we have identified 49 previously undiscovered vulnerabilities in *npm* packages.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and insightful feedback. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the 2021.06134.BD and SFRH/BD/146698/2019 grants, and projects UIDB/50021/2020, Future Enterprise

Security Initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab), Carnegie Mellon CyLab, SmartRetail (ref. C6632206063-00466847), financed by IAPMEI, and RIGA (2022.03537.PTDC).

DATA AVAILABILITY STATEMENT

The Graph.js's artifact is available at Ferreira et al. [20]. The open-source repository contains the most up-to-date version and is available at https://github.com/formalsec/graphjs.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. Association for Computing Machinery, New York, NY, USA, 385–395. https://doi.org/10.1145/3106237.3106267

[2] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P '17)*. IEEE Computer Society, Los Alamitos, CA, USA, 334–349. https://doi.org/10.1109/EuroSP.2017.14

[3] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API Misuse Detector for Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. Association for Computing Machinery, New York, NY, USA, 507–517. https://doi.org/10.1145/2635868.2635916

[4] Christian Banse, Immanuel Kunz, Angelika Schneider, and Konrad Weiss. 2021. Cloud Property Graph: Connecting Cloud Security Assessments with Static Code Analysis. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD '21)*. IEEE Computer Society, Los Alamitos, CA, USA, 13–19. https://doi.org/10.1109/CLOUD53861.2021.00014

[5] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 1059–1070. https://doi.org/10.1109/ICSE48619.2023.00096

[6] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. 2023. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages. *IEEE Transactions on Reliability* 72, 4 (2023), 1324–1339. https://doi.org/10.1109/TR.2023.3286301

[7] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* 118 (2022), 102745. https://doi.org/10.1016/j.cose.2022.102745

[8] Yinzhi Cao, Xiang Pan, Yan Chen, and Jianwei Zhuge. 2014. JShield: Towards Real-Time and Vulnerability-Based Detection of Polluted Drive-by Download Attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. Association for Computing Machinery, New York, NY, USA, 466–475. https://doi.org/10.1145/2664243.2664256

[9] Darion Cassel, Wai Tuck Wong, and Limin Jia. 2023. NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P '23)*. IEEE Computer Society, Los Alamitos, CA, USA, 1101–1127. https://doi.org/10.1109/EuroSP57164.2023.00068

[10] CodeQL. 2024. https://codeql.github.com/.

[11] 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 '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[12] CVE - Mitre. 2024. Mitre corporation homepage. https:/cve.mitre.org/.

[13] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A parallel abstract interpreter for JavaScript. In *Proceedings of 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, USA, 34–45. https://doi.org/10.1109/CGO.2015.7054185

[14] Esprima. 2021. ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/index.html.

[15] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1899–1913. https://doi.org/10.1145/3319535.3345656

[16] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC) (ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 257–269. https://doi.org/10.1145/3359789.3359813

[17] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and*

Communications Security (CCS '21). Association for Computing Machinery, New York, NY, USA, 1789–1804. https://doi.org/10.1145/3460120.3484745

[18] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[19] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2023. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. In *44th IEEE Symposium on Security and Privacy (S&P '23)*. IEEE Computer Society, Los Alamitos, CA, USA, 2817–2834. https://doi.org/10.1109/SP46215.2023.10179395

[20] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. *Graph.js PLDI24 Artifact Evaluation*. https://doi.org/10.5281/zenodo.10936488

[21] Mafalda Ferreira, Miguel Monteiro, Tiago Brito, Miguel E. Coimbra, Nuno Santos, Limin Jia, and José Fragoso Santos. 2024. Technical Report: Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. https://doi.org/10.5281/zenodo.10933020

[22] GitHub Advisory Database. 2023. https://github.com/advisories.

[23] Liang Gong. 2018. *Dynamic Analysis for JavaScript Code*. Ph. D. Dissertation. University of California, Berkeley.

[24] Huntr.dev. 2023. https://huntr.dev/.

[25] Dongseok Jang and Kwang-Moo Choe. 2009. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. Association for Computing Machinery, New York, NY, USA, 1930–1937. https://doi.org/10.1145/1529282.1529711

[26] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of International Static Analysis Symposium (SAS '09)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

[27] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *44th IEEE Symposium on Security and Privacy (S&P '23)*. IEEE Computer Society, Los Alamitos, CA, USA, 1059–1076. https://doi.org/10.1109/SP46215.2023.10179352

[28] 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 '14)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

[29] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *30th USENIX Security Symposium (SEC '21)*. USENIX Association, USA, 2525–2542. https://www.usenix.org/conference/usenixsecurity21/presentation/khodayari

[30] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. 2022. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules. *International Journal of Information Security* 21, 1 (2022), 1–23. https://doi.org/10.1007/s10207-020-00537-0

[31] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Press, 541–551. https://doi.org/10.1109/ASE.2015.28

[32] Immanuel Kunz, Konrad Weiss, Angelika Schneider, and Christian Banse. 2023. Privacy Property Graph: Towards Automated Privacy Threat Modeling via Static Graph-based Analysis. *Proceedings of Privacy Enhancing Technologies (PETS)* 2023, 2 (2023), 171–187. https://doi.org/10.56553/popets-2023-0046

[33] Aditya Kurniawan, Bahtiar Saleh Abbas, Agung Trisetyarso, and Sani Muhammad Isa. 2018. Static Taint Analysis Traversal with Object Oriented Component for Web File Injection Vulnerability Pattern Detection. *Procedia Computer Science* 135 (2018), 596–605. https://doi.org/10.1016/j.procs.2018.08.227

[34] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The Unexpected Dangers of Dynamic JavaScript. In *Proceedings of the 24th USENIX Security Symposium (SEC '15)*. USENIX Association, USA, 723–735. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[35] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/3468264.3468542

[36] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *Proceedings of the 31st USENIX Security Symposium (SEC '22)*. USENIX Association, USA, 143–160. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

[37] Mike Ter Louw and V.N. Venkatakrishnan. 2009. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *30th IEEE Symposium on Security and Privacy (S&P '09)*. IEEE Computer Society, Los Alamitos,

CA, USA, 331–346. https://doi.org/10.1109/SP.2009.33

[38] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '09)*. The Internet Society. https://www.ndss-symposium.org/ndss2009/document-structure-integrity-a-robust-basis-for-cross-site-scripting-defense/

[39] Neo4j. 2023. Cypher Query Language. https://neo4j.com/developer/cypher/.

[40] Neo4j. 2023. Graph Database and Analytics. https://neo4j.com.

[41] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. Association for Computing Machinery, New York, NY, USA, 455–465. https://doi.org/10.1145/3338906.3338933

[42] Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP '20)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.16

[43] Christoforos Ntantogian, Panagiotis Bountakas, Dimitris Antonaropoulos, Constantinos Patsakis, and Christos Xenakis. 2021. NodeXP: NOde. js server-side JavaScript injection vulnerability DEtection and eXPloitation. *JISA* 58 (2021), 102752. https://doi.org/10.1016/j.jisa.2021.102752

[44] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript Static Analysis via Dynamic Shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. Association for Computing Machinery, New York, NY, USA, 1129–1140. https://doi.org/10.1145/3468264.3468556

[45] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript Web Applications Using SAFE 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, 59–62. https://doi.org/10.1109/ICSE-C.2017.4

[46] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1757–1771. https://doi.org/10.1145/3133956.3133959

[47] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31th IEEE Symposium on Security and Privacy (S&P '10)*. IEEE Computer Society, Los Alamitos, CA, USA, 317–331. https://doi.org/10.1109/SP.2010.26

[48] Faysal Hossain Shezan, Zihao Su, Mingqing Kang, Nicholas Phair, Patrick William Thomas, Michelangelo van Dam, Yinzhi Cao, and Yuan Tian. 2023. CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-language Code Property Graph. In *Proceedings of 30th Annual Network and Distributed System Security Symposium (NDSS '23)*. The Internet Society. https://doi.org/10.14722/ndss.2023.24610

[49] Snyk. 2023. https://snyk.io/.

[50] Dolière Francis Somé. 2019. EmPoWeb: Empowering Web Applications with Browser Extensions. In *40th IEEE Symposium on Security and Privacy (S&P '19)*. IEEE Computer Society, Los Alamitos, CA, USA, 227–245. https://doi.org/10.1109/SP.2019.00058

[51] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP '12)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 435–458. https://doi.org/10.1007/978-3-642-31057-7_20

[52] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (SEC '18)*. USENIX Association, Baltimore, MD, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[53] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '18)*. The Internet Society. https://doi.org/10.14722/ndss.2018.23076

[54] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting Taint Specifications for JavaScript Libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 198–209. https://doi.org/10.1145/3377811.3380390

[55] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *Proceedings of the 23rd USENIX Security Symposium (SEC '14')*. USENIX Association, San Diego, CA, 655–670. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/stock

[56] The MITRE Corporation. 2023. CWE-1321: Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution'). https://cwe.mitre.org/data/definitions/1321.html.

[57] The MITRE Corporation. 2023. CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'). https://cwe.mitre.org/data/definitions/22.html.

[58] The MITRE Corporation. 2023. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html.

[59] The MITRE Corporation. 2023. CWE-94: Improper Control of Generation of Code ('Code Injection'). https://cwe.mitre.org/data/definitions/94.html.

[60] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing hidden properties to attack the node. js ecosystem. In *20th USENIX Security Symposium (SEC '21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/xiao

[61] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *35th IEEE Symposium on Security and Privacy (S&P '14)*. IEEE Computer Society, Los Alamitos, CA, USA, 590–604. https://doi.org/10.1109/SP.2014.44

[62] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *36th IEEE Symposium on Security and Privacy (S&P '15)*. IEEE Computer Society, Los Alamitos, CA, USA, 797–812. https://doi.org/10.1109/SP.2015.54

[63] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. 2023. CoCo: Efficient Browser Extension Vulnerability Detection via Coverage-guided, Concurrent Abstract Interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2441–2455. https://doi.org/10.1145/3576915.3616584

[64] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (SEC '19)*. USENIX Association, Santa Clara, CA, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

Received 2023-11-16; accepted 2024-03-31