# Mining Node.js Vulnerabilities via Object Dependence Graph and Query

Song Li, Mingqing Kang, Jianwei Hou[†],[*] and Yinzhi Cao
Johns Hopkins University, [†]Johns Hopkins University/Renmin University of China

## Abstract

Node.js is a popular non-browser JavaScript platform that provides useful but sometimes also vulnerable packages. On one hand, prior works have proposed many program analysis-based approaches to detect Node.js vulnerabilities, such as command injection and prototype pollution, but they are specific to individual vulnerability and do not generalize to a wide range of vulnerabilities on Node.js. On the other hand, prior works on C/C++ and PHP have proposed graph query-based approaches, such as Code Property Graph (CPG), to efficiently mine vulnerabilities, but they are not directly applicable to JavaScript due to the language's extensive use of dynamic features.

In the paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and accepts graph queries—especially on object lookups and definitions—for detecting Node.js vulnerabilities.

We implemented an open-source prototype system, called ODGen, to generate ODG for Node.js programs via abstract interpretation and detect vulnerabilities. Our evaluation of recent Node.js vulnerabilities shows that ODG together with AST and Control Flow Graph (CFG) is capable of modeling 13 out of 16 vulnerability types. We applied ODGen to detect six types of vulnerabilities using graph queries: ODGen correctly reported 180 zero-day vulnerabilities, among which we have received 70 Common Vulnerabilities and Exposures (CVE) identifiers so far.

## 1 Introduction

Node.js is a popular JavaScript runtime environment that executes JavaScript code outside web browsers such as being a web server to serve the client. Node.js ecosystem including millions of NPM packages is known to be vulnerable to a variety of vulnerabilities, such as command injection [1, 2], prototype pollution [3], path traversal [4], and internal property tampering [5–7]. In the past, researchers have proposed various program analysis-based approaches [1–3, 8–14] targeting individual vulnerability, such as command injection [1, 2] and prototype pollution [3]. However, despite their success, there is no general framework to detect all kinds of Node.js vulnerabilities.

One recent advance of vulnerability detection in languages other than JavaScript such as C/C++ and PHP is to build a graph structure representing different properties of a target program and perform graph queries to mine vulnerabilities. For example, researchers proposed a particular graph structure, called Code Property Graph (CPG), which combines Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). CPG is demonstrated to be effective in mining many types of vulnerabilities in C/C++ [15] and PHP [16]. However, CPG does not model object relations, such as object lookups based on prototype chain and `this` object lookup especially with a `bind` call. Therefore, it cannot model and detect popular object-based JavaScript vulnerabilities, such as prototype pollution [3] and internal property tampering [5–7].

At the same time, prior static JavaScript analysis works [1, 10–12, 17] model objects and their relations via abstract interpretation [18] together with an online data structure, such as a lattice. However, prior abstract interpretations face two major challenges. First, previous data structures are unsuitable for offline (i.e., post abstract interpretation) detections of a variety of vulnerabilities—in other words, their target is a specific type of vulnerability. The reason is that object information in these structures keeps changing during abstract interpretation. Thus, vulnerability-related object information is likely overwritten and lost in the final state. Second, existing JavaScript analysis—in terms of branch sensitivity—interprets all branches either in sequence, which compromises accuracy, or in parallel, which compromises scalability. Both cases lead to many false negatives: the former due to reduced detection capability and the latter due to excessive number of objects.

In this paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG accepts graph queries for the offline detection of a wide range

[*] The author contributes to the paper when she is visiting JHU.

of Node.js vulnerabilities. The key insight of ODG is to represent JavaScript objects as nodes and the relations among objects and between objects and AST nodes as edges. Specifically, ODG includes fine-grained data dependencies between objects, thus helping taint-style vulnerability detection such as command injection. At the same time, ODG is also integrated with CPG, or particularly Abstract Syntax Tree (AST) of CPG, to represent and preserve all object definitions and lookups (e.g., these via the prototype chain) in abstract interpretation for the offline detection of object-related vulnerabilities such as internal property tampering and prototype pollution.

We build a prototype system, called ODGen, to generate ODG during abstract interpretation. Specifically, ODGen starts from entry points and follows AST node sequence to define and lookup objects for each AST node under abstract scopes. Then, ODGen records object definitions and lookups as part of ODG, which are also used to generates CFG (if an object lookup is related to functions) and object-level data dependencies (if an object definition is derived from another object). ODGen is hybrid branch-sensitive because the default of ODGen is to abstractly interpret all branches in parallel, but ODGen switches back to sequential branch interpretation for a function if the number of object nodes explodes. ODGen has points-to information because different aliases of an objects point to the same object node in ODG.

To demonstrate the effectiveness of ODGen, we studied all recent Node.js vulnerabilities in the CVE database and modeled them with graph queries to ODG together with existing graph-based code representations. Our evaluation shows that 13 out of 16 vulnerability categories can be successfully modeled by graph queries to ODG+AST+CFG. We then evaluate ODGen on real-world Node.js packages. The results show that ODGen is able to detect 43 application-level zero-day vulnerabilities with 14 false positives and we also confirmed 137 package-level zero-day vulnerabilities with 84 false positive. We received 70 CVE identifiers for these vulnerabilities.

We make the following contributions in the paper.

- We design a novel graph structure, called Object Dependence Graph (ODG), to model JavaScript objects and their relations to AST node in terms of definition and use.
- We design offline graph queries that match object-related patterns for a variety of Node.js vulnerabilities, particularly internal property tampering and prototype pollution.
- We build a prototype, open-source system using abstract interpretation to generate ODG for Node.js packages.
- Our evaluation of ODGen on real-world NPM packages reveals 43 application-level and 137 package-level zero-day vulnerabilities (70 being assigned with CVE identifiers).

## 2 Overview

In this section, we start from a motivating example and then describe the threat model in detecting Node.js vulnerabilities.

```
1 function Func() {};
2 Func.prototype.x="ab";
3 myFunc = new Func;
4 if (source1)
5   myFunc[source2]=myFunc.x+source1; // internal
        property tampering
6 sink(myFunc.x); // taint-style vulnerability like
      command injection
```

Figure 1: An exemplary code.

### 2.1 A Motivating Example

Figure 1 shows a simple exemplary code with only six lines in motivating the use of ODG in vulnerability detection. Both `source1` and `source2` are controllable by an adversary and `sink` is a sink function, such as `exec` in command injection. The code has two vulnerabilities:

- Internal Property Tampering [5–7]. This vulnerability is triggered when `source2` is `"__proto__"`. Because the prototype chain of `myFunc` is overwritten at Line 5, the internal property `x` of `myFunc` is tampered. Specifically, when the code tries to access `myFunc.x` at Line 6, the object lookup in the property `x` fails as the prototype chain to `Func.prototype` is broken. This vulnerability may lead to a consequence like Denial of Service (e.g., the execution of Line 6 fails) or privilege escalation (e.g., if `myFunc.x` is used later as part of an authentication).
- Taint-style Vulnerability (e.g., command injection [1, 2]). This vulnerability is triggered when `source2` is `"x"`. The code will then create a new property `x` under `myFunc` directly with an adversary controllable value from `source1`. Next, when the code accesses `myFunc.x` at Line 6, the object lookup goes to `myFunc` directly instead of `Func.prototype`, leading to a possible injection.

What we learned from these two vulnerabilities is that the key is the object lookup `myFunc[source2]` at Line 5. Different lookups lead to different vulnerabilities—which motivates the design of ODG in modeling different object lookups in a graph for vulnerability detection. Another interesting observation worth noting is that the data dependencies are different for two vulnerability triggering conditions. In the case of internal property tampering at Line 5, we do not have a dataflow dependency between Lines 2 and 6 and the lack of such a dependency leads to the vulnerability. By contrast, in the case of a taint-style vulnerability, we have a dataflow dependency between Lines 5 and 6 (which does not exist before) and the existence of this dependency leads to the vulnerability.

Figure 2 shows the object dependence graph (ODG) integrated with code property graph (CPG) of the code in Figure 1. The top part of Figure 2 is CPG with AST, CFG and Program Dependence Graph (PDG) nodes and edges; the bottom part is ODG with object/name nodes, object lookup/definition edges to AST nodes (copied from top for clarity purpose), and property edges. Note that because ODG has object-level data dependencies, we do not need the statement-level data dependencies in PDG as part of CPG. We include these edges in

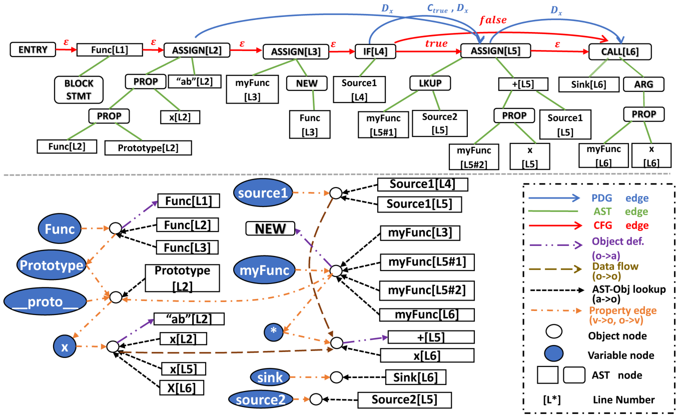


Figure 2: Object Dependence Graph (ODG, Bottom) Integrated with Code Property Graph (CPG, Top) of the Exemplary Code in Figure 1. For readers' convenience, we copied corresponding AST nodes from top to bottom and skipped several unimportant nodes and edges, such as __proto__ of many objects, the global object and many built-in objects.

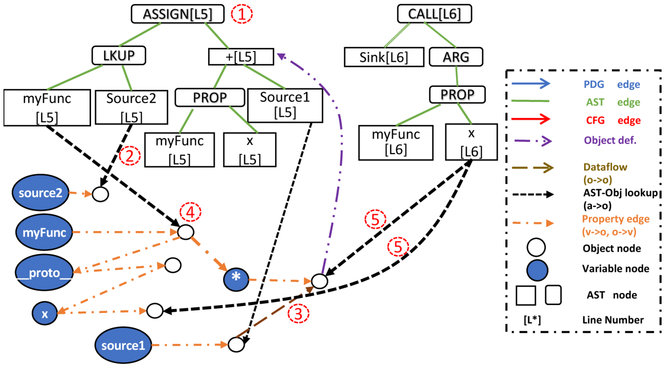


Figure 3: Nodes and Edges related to Graph Query for Internal Property Tampering Detection.

the figure for the purpose of a comparison. We now describe how to detect these two vulnerabilities via graph queries and more importantly how ODG edges contribute to the detection.

### 2.1.1 Query to Detect Internal Property Tampering

We summarize the detection of this internal property tampering vulnerability using ODG as follows. From a high-level perspective, ODGEN finds an object assignment statement via a property lookup, which is then followed by another property lookup statement. Both the lookup and the assigned values in the first statement are controllable by an adversary so that the prototype chain of the object can be tampered. Then, the property lookup in the second statement needs to have the tampered prototype chain involved. We extract related edges from Figure 2, show them in Figure 3 and describe below.

- ① AST pattern matching (obj[prop]=value). The query finds an assignment statement with a property lookup via AST edges, which is myFunc[source2]=myFunc.x+source1 at Line 5 of Figure 1.

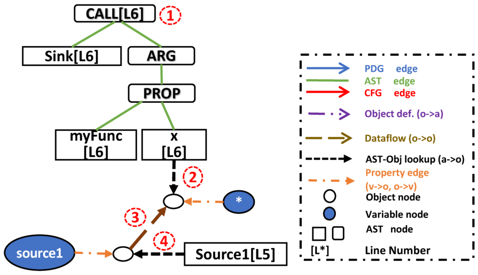


Figure 4: Nodes and Edges related to Graph Query for Taint-style Vulnerability Detection.

- ② Property in ① (prop) is controllable by an adversary. The query follows the object-level data dependencies to determine whether source2 is controllable by an adversary. Therefore, the value of source2 can be __proto__.
- ③ Assigned value in ① (value) is controllable by an adversary. The query follows the object-level data dependencies to determine whether myFunc.x+source1 can be controllable by an adversary.
- ④ Object in ① (obj) has a prototypical object and the prototypical object has a property. The query follows prototype chain of the object myFunc to find the prototype object myFunc.__proto__, which has a property x.
- ⑤ Property in ④ is used later in the control flow and has more than one possible lookup. The query follows the property x to find other uses of the object (myFunc.x at Line 6 of Figure 1) and ensures that it has a control dependency with the previous assignment.

### 2.1.2 Query to Detect Taint-style Vulnerability

The detection of a taint-style vulnerability using ODG can be summarized as finding a data dependency between the source object and the argument object in the sink function. We extracted related edges from Figure 2 and show them in Figure 4.

- ① AST Pattern matching for sink function (sink(arg)). The query finds a statement with a sink function invocation (i.e., sink(myFunc.x) at Line 6 of Figure 1).
- ② Object lookup for arg in ①. The query finds the object node in ODG.
- ③ Data dependency for the object in ②. The query follows object-level data dependency edges to determine whether the sink function argument can be influenced by a source.
- ④ AST Node for the source in ③. The query follows object lookup edges to find the AST node for the source.

Note that the handling of myFunc[source2] is implicit in the detection of this taint-style vulnerability. During ODG construction, ODGEN creates a so-called wildcard object with a property $*$ to represent myFunc[source2] for all kinds of possibilities. Then, myFunc.x can be resolved via two ways: one to Func.prototype.x and the other as myFunc.*. Therefore, our query can find an object-level data dependency between myFunc.* and source1.

### 2.2 Threat Model

In this subsection, we describe the threat model of vulnerabilities in scope of ODGen. ODGen considers all JavaScript-level Node.js vulnerabilities but excludes low-level ones, such as those related to the V8 engine. Specifically, such vulnerabilities can be categorized as two types: (i) application-level and (ii) package-level. We now describe these two in details.

#### 2.2.1 Application-level Vulnerabilities

An application-level vulnerability assumes that an adversary has some controls over contents in network connection, e.g., an HTTP request or a response, because the application is communicating with a malicious party. The detailed capability of the adversary also depends on the semantics of the application. We now describe two concrete senarios:

- Adversary-controlled network request to a vulnerable server. Say the application is a web server serving web contents to clients. An adversary can send HTTP requests with malicious contents to the server and trigger a vulnerability. Consider `rollup-plugin-serve`, which has a path traversal vulnerability (CVE-2020-7684) found by ODGen. The vulnerable code reads a file using `readFile` via an arbitrary path provided by the client without sanitization, i.e., the `filePath` value eventually comes from the `request` object controllable by a possible adversary.
- Adversary-controlled network response to a vulnerable client. Say the application is at client-side talking with servers. An adversary, i.e., a malicious server, can send HTTP responses with malicious contents to the application and trigger a vulnerability. Let us take a real-world, client-side github notification system, called `github-growl`, for example. `github-growl` gives an alert at the client side if a github issue is posted to a subscribed github repository. An adversary can post an issue with a crafted title with OS commands and trigger the command injection vulnerability in `github-growl`.

#### 2.2.2 Package-level Vulnerabilities

Packages in Node.js are libraries that are imported by other packages or applications. Package-level vulnerabilities assume that an adversary can control inputs to a vulnerable package (i.e., those accessible via `module.exports`), thus triggering the vulnerability. It is worth noting that package-level vulnerabilities are not stand-alone and have to be combined with applications for a possible exploitation.

The reason that Node.js community considers package-level vulnerabilities—which are demonstrated in both academic works [1, 2] and many prior CVEs [5, 19, 20]—are that one package-level vulnerability may affect many applications if the inputs to the package are not correctly sanitized. Take the previous `github-growl` for example. The application itself is not vulnerable, but the vulnerability lies in an imported package called `growl` (CVE-2017-16042). In fact, the vulnerable package also affects other applications, such as `mqtt-growl` a mqtt monitor based on growl, by making them vulnerable as well.

Table 1: Nodes, Edges, and Operations of ODG

| Name | Description |
|---|---|
| *Nodes (N)* | *A set of ODG nodes* |
| Object node ($o \in N_o$) | An object created in the abstract interpretation. |
| Scope node ($s \in N_s$) | An abstract interpretation scope. |
| Variable node ($v \in N_v$) | A variable under a scope or a property under an object. |
| AST node ($a \in N_a$) | An abstract syntax tree node. |
| *Edges (E)* | *A set of ODG edges* |
| Object def. ($o \xrightarrow{s} a$) | The AST node ($a$) defining the object $o$ under scope $s$. |
| AST-obj lookup ($a \xrightarrow{s} o$) | The object ($o$) used by the AST node ($a$) under $s$. |
| Scope hierarchy ($s \rightarrow s$) | A parent-child scope relation. |
| Variable lookup ($s \rightarrow v$) | A variable $v$ is defined under a scope $s$. |
| Var-obj lookup ($v \xrightarrow{Br} o$) | An object $o$ that $v$ points to with branch tags $Br$. |
| Property lookup ($o \rightarrow v$) | A property $v$ of an object $o$. |
| Data dependency ($o \rightarrow o$) | Data dependency between two objects. |
| Control dependency ($a \rightarrow a$) | Control dependency between two AST nodes. |
| *Procedures (P)* | *All the ODG-related operations* |
| $Child_{parentNode}^{EdgeType}$ | Getting the child node of *parentNode* with *EdgeType* |
| $AddEdge_{src \xrightarrow{p} dst}^{EdgeType}$ | Adding an edge from *src* node to *dst* node with *EdgeType* and a property being either branch tags ($Br$) or a scope ($s$) |
| $GetEdge_{src}^{EdgeType}$ | Getting all the edges start from *src* node with *EdgeType* |
| $AddNode_{a}^{NodeType}$ | Adding a node from $a$ with *NodeType* |
| $AddObj_{a}^{ObjType}$ | Adding an object node from $a$ with *ObjType* in *typeof* list and linking prototypical objects |
| $LkupVar_{Br}^{s}(n)$ | Looking up a variable node under the scope ($s$) with branch tags ($Br$) and name $n$ |
| $LkupObj_{Br}^{s}(n)$ | Looking up object nodes under scope ($s$) with branch tags ($Br$) and name ($n$), i.e., $\{Child_{LkupVar_{Br}^{s}(n)}^{v \xrightarrow{Br} o}\}$ |

Other than the aforementioned application- vs. package-level, we further classify Node.js vulnerabilities into two categories based on the vulnerability location, i.e., directly vulnerable where the package itself is vulnerable, and indirectly vulnerable where an imported package is vulnerable.

## 3 Object Dependence Graph

In this section, we describe the definition of Object Dependence Graph (ODG) and the operational semantics of the abstract interpretation and the procedure of constructing ODG.

### 3.1 Definitions

In this section, we define an Object Dependence Graph (ODG) as a representation, using graph notation, of all the JavaScript objects, variables and scopes generated during abstract interpretation as nodes and their relations as edges. These edges include object and AST relations (such as object definition and object lookup) and object relations (such as object property and object-level data dependency).

Table 1 summarizes different ODG nodes and edges. Objects, variables, scopes and AST are all represented as nodes and their relations as edges. We start from AST-related edges: object definition and AST-obj lookup. The former is used to locate the AST node where the object is defined when the object is used later. These types of edges are unique to one object node because an object is only defined once. The latter is used to reproduce object lookups in abstract interpretation. One AST node may have multiple AST-obj lookup edges because the AST node can be abstractly interpreted for multiple times in a for loop or a recursive call.

We then describe edges between objects, variables, and scopes. Note that we skipped branch tags (introduced later in Section 3.2) for a simple explanation. First, the combination of $s \rightarrow s$, $s \rightarrow v$, $v \rightarrow o$, and $o \rightarrow v$ edges can be used to resolve a statement like obj.prop during abstract interpretation. ODGEN first looks up obj under current scope using $s \rightarrow v$ and then follows the scope chain using $s \rightarrow v$ to find obj if the lookup under current scope fails. Once the variable is found, ODGEN follows $v \rightarrow o$ to find the object node and then $o \rightarrow v$ to find the prop. Then, $o \rightarrow o$ indicates the latter object has a data dependency on the former. For example, the object that myFunc[source2] points to at Line 5 of Figure 1 has an object-level data dependency on both objects that myFunc.x and source1 point to.

Next, we describe how ODG models points-to information via $v \rightarrow o$ edges. Say two variables a and b and an object property obj.v point to the same object. There is only one object node in ODG representing this object and three $v \rightarrow o$ edges from a, b and obj.v to the object node. Therefore, all three object lookups will resolve to the same object node during abstract interpretation.

### 3.2 Operational Semantics

In this subsection, we describe our abstract interpretation and the construction of ODG using operational semantics shown in Figure 5. From a high level, ODGEN abstractly interprets each AST node ($a$) based on the statement ($e$), generates nodes ($N$) and edges ($E$) for ODG, and then follows control-flow edges (which are generated during abstract interpretation) to the next AST node. During the abstract interpretation of each AST node, the state of ODGEN is represented as a tuple $\rho = (N, E, s, Br)$, where $N$ is all the ODG nodes, $E$ is all the ODG edges, $s$ is the current scope node, and $Br \subseteq S_{br}$ is a set of branch tags that represents the current conditional branch in the branch-sensitive mode. Each branch tag is a unique identifier representing the current conditional branch.

Now, we describe the operational semantics of the abstract interpretation of different statements in Figure 5. First, we start from the definition of either a variable or an object property in Figure 5. ODGEN attempts to look up the variable or the property from ODG. If the look-up fails, ODGEN creates new variable and object nodes and links corresponding nodes via edges; if the look-up succeeds, ODGEN reuses existing variable nodes but creates new edges for these nodes.

Second, we describe branching statements (i.e., IF and SWITCH in Figure 5). ODGEN first tries to determine the value of the branching condition and chooses corresponding branch(es). If the branching condition value cannot be determined, the operational semantics depends on branch sensitivity. (i) ODGEN creates a unique branching tag for each branch in the branch-sensitive mode and attaches the branching tag with all the nodes and edges created during the abstract interpretation of each branch. When all the branches of a statement are abstractly interpreted, ODGEN merges all the objects and nodes from different branches based on the tags for continued abstract interpretation. (ii) ODGEN sequentially performs abstract interpretation for all the branches in the branch-insensitive mode, i.e., the objects and edges created in later branches will overwrite those created in earlier branches. The default mode is branch sensitive, but ODGEN will switch to branch insensitive if the number of objects explodes, i.e., exceeding a certain number (e.g., 10k), for a given function.

Third, we describe function definition in Figure 5. ODGEN adds a variable node if the function is not defined in an anonymous closure, creates an object node and edges between the object and the variable nodes, and then handles edges related to prototypes.

Fourth, we describe function calls in Figure 5, which has two phase: pre-call and call. In the pre-call phase, ODGEN looks up the function object and creates corresponding object and control-flow edges. Then, in the call phase, ODGEN handles all the parameters, changes the current scope and this point, and then jumps to the AST node following a call edge. Finally, in the return statement, ODGEN handles return objects and creates corresponding dataflow edges. Because ODGEN handles function calls using the current scope and returns to the exact call site, ODGEN is considered as a context-sensitive approach.

Lastly, we describe loops in Figure 5. ODGEN abstractly interprets a loop (and a recursive call) extensively until no more new objects outside the loop (or recursive call) are being looked-up. ODGEN also sets up a minimum and a maximum limit for loops (and recursive calls).

## 4 ODG Queries for Node.js Vulnerabilities

In this section, we describe graph queries to ODG for all kinds of Node.js vulnerabilities. We first present how to model queries as several types of graph traversals in Section 4.1 and then describe how to represent all kinds of vulnerabilities via those graph traversals in Section 4.2.

### 4.1 Graph Traversals

A graph traversal, as defined in the CPG paper [15], is a function $T : P(V) \rightarrow P(V)$ that maps a set of nodes to another set of nodes on top of ODG, where $V$ is a set of ODG nodes and $P$ is the power set of $V$. There are multiple operations that can be performed on $T$:

- A function composition $\circ$. Two graph traversals $T_0$ and $T_1$ on $V$ can be chained together by $T_1 \circ T_0(V)$.
- A function intersection $\bigcap$. The results of two graph traversal $T_0$ and $T_1$ on $V$ can be intersected by $T_0 \bigcap T_1(V)$.
- A function union $\bigcup$. The results of two graph traversal $T_0$ and $T_1$ on $V$ can be unioned by $T_0 \bigcup T_1(V)$.

By those three simple operations, we can break a complex graph traversal into multiple basic traversal components shown in Table 2. These basic traversals

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(x,a,\rho) \Rightarrow if\ LkupVar^s_\varnothing(x), = \varnothing\ then\ (N,E,s,Br)\ else\ (N, E \cup \{AddEdge^{a\xrightarrow{s}o}_{x\xrightarrow{s}o'}\ where\ \forall o' \in LkupObj^s_{Br}(a)\}, s, Br)}\ (\text{VARIABLE})$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(let/var/const/\varnothing\ x,a,\rho) \Rightarrow (N\cup N_a := \{AddNode^{var}_{a.name}\}, E\cup\{AddEdge^{s\to v}_{s'\to n_a}, \forall n_a \in N_a\}, s, Br)}\ \text{where} \begin{cases} s' := s\ (BLOCK\_SCOPE) & let/const \\ s' := GLOBAL\_SCOPE & \varnothing \\ s' := upper\ FUNC/FILE\_SCOPE & var \end{cases}\ (\text{VARIABLE DEF})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (x,a.x,\rho)\Rightarrow (N_x,E_x,s_x,Br_x), (p,a.p,\rho)\Rightarrow(N_p,E_p,s_p,Br_p)}{(x[p]/x.\text{const},\rho) \Rightarrow \begin{cases} (N_x \cup \{p_{ov}(0), \forall p_{ov}\in P_{ov}\}, E_x \cup \{AddEdge^{o\to v}_{p_{ov}(0)\to p_{ov}(1)}, \forall p_{ov}\in P_{ov}\}, s, Br) & if\ on = \emptyset \\ (N_x, E_x \cup \{AddEdge^{a\xrightarrow{s}o}_{a\xrightarrow{s}n_o}, \forall n_o \in N_o\}, s, Br) & otherwise \end{cases}}$$

$$where \begin{cases} N_o := \{LkupObj^{o_x}_{Br_x}(o_p.name), \forall o_p \in Child^{a\xrightarrow{s}o}_{a.p}, \forall o_x \in Child^{a\xrightarrow{s}o}_{a.x}\} & P_{ov} := \{(AddNode^{var}_{p'}, o'), \forall o'\in Child^{a\xrightarrow{s}o}_{a.x}, \forall p' \in Child^{a\xrightarrow{s}o}_{a.p}\} & x[p] \\ N_o := \{LkupObj^{o_x}_{Br_x}(\text{const}), \forall o_x \in Child^{a\xrightarrow{s}o}_{a.x}\} & P_{ov} := \{(AddNode^{var}_{\text{const}}, o'), \forall o' \in Child^{a\xrightarrow{s}o}_{a.x}\} & x.\text{const}\end{cases}\ (\text{PROPERTY})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (x_1,a.x_1,\rho)\Rightarrow(N_{x_1},E_{x_1},s_{x_1},Br_{x_1}), (x_2,a.x_2,\rho)\Rightarrow(N_{x_2},E_{x_2},s_{x_2},Br_{x_2})}{(x_1\ op\ x_2, a, \rho) \Rightarrow (N_{x_1}\cup N_{x_2}\cup N_{new}, E_{x_1}\cup E_{x_2}\cup E_{dep}\cup E_{def}, s, Br)}\ \text{where} \begin{cases} N_{new} := \{AddObj^*_a, \forall o_1 \in Child^{a\xrightarrow{s}o}_{a.x_1}, \forall o_2 \in Child^{a\xrightarrow{s}o}_{a.x_2}\} \\ E_{dep} := \{AddEdge^{o\to o}_{u'\to o'}, \forall o' \in N_{new}, \forall u' \in \{Child^{a\xrightarrow{s}o}_{a.x_1} \cup Child^{a\xrightarrow{s}o}_{a.x_2}\}\} \\ E_{def} := \{AddEdge^{o\to a}_{o'\to a}, \forall o' \in N_{new}\}\end{cases}\ (\text{BINARY OP})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (k_1,a.k_1,\rho)\Rightarrow(N_{k_1},E_{k_1},s_{k_1},Br_{k_1}), (v_1,a.v_1,\rho)\Rightarrow(N_{v_1},E_{v_1},s_{v_1},Br_{v_1}),\ldots,(k_n,a.k_n,\rho)\Rightarrow(N_{k_n},E_{k_n},s_{k_n},Br_{k_n}),(v_n,a.v_n,\rho)\Rightarrow(N_{v_n},E_{v_n},s_{v_n},Br_{v_n})}{(\{k_1:v_1,\ldots,k_n:v_n\},a,\rho)\Rightarrow(O_a := \{AddObj^*_a\}\cup\{nv_i := AddNode^{var}_{a.k_i}, \forall i \in \{i,\ldots,n\}\}\cup\{\bigcup_{i=1}^n N_{k_i}\}\cup\{\bigcup_{i=1}^n N_{v_i}\}, \{\bigcup_{i=1}^n E_{k_i}\}\cup\{\bigcup_{i=1}^n E_{v_i}\}\cup E_{ov}\cup E_{vo}\cup\{AddEdge^{a\xrightarrow{br}o}_{a\xrightarrow{Br}oa}, \forall oa \in O_a\}, s, Br)}$$

$$,\text{where} \begin{cases} E_{vo} := \{AddEdge^{v\xrightarrow{br}o}_{nv_i\xrightarrow{Br}Child^{a\xrightarrow{s}o}_{a.v_i}}, \forall i\in\{1,\ldots,n\}\} \\ E_{ov} := \{AddEdge^{o\to v}_{ao\to v_i}, \forall i \in \{1,\ldots,n\}\}\end{cases}\ (\text{OBJECT LITERAL})\ \frac{\rho \Rightarrow (N,E,s,Br)}{(this,a,\rho)\Rightarrow(N,E\cup\{AddEdge^{a\xrightarrow{s}o}_{a\xrightarrow{s}o'}\ where\ \forall o' \in LkupObj^s_{Br}(\text{"}this\text{"})\}, s, Br)}\ (\text{THIS})$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(B_{pre},a,\rho)\Rightarrow(N\cup\{as := AddNode^{scope}_a\}, E\cup\{AddEdge^{s\to s}_{s\to as}\}, as, Br)}\ (\text{PRE BLOCK})\ \frac{(B_{pre},a,\rho)\Rightarrow\rho_{B_{pre}}, (S_1,\rho_{B_{pre}})\Rightarrow\rho_1,\ldots,(S_n,\rho_{n-1})\Rightarrow\rho_n}{(S_1,\ldots,S_n,\rho)\Rightarrow(N_{\rho_n}, E_{\rho_n}\cup\{AddEdge^{a\to a}_{a.S_i\to a.S_{i+1}}, \forall i \in \{1,\ldots,n-1\}\}, s_\rho, Br_{\rho_n})}\ (\text{BLOCK})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (let/var/const/\varnothing\ x, a.x, \rho)\Rightarrow(N_x,E_x,s_x,Br_x), (e,a.e,\rho)\Rightarrow(N_e,E_e,s_e,Br_e)}{(let/var/const/\varnothing\ x = e, \rho)\Rightarrow(N_x\cup N_e, E_x\cup E_e/\{GetEdge^{v\to o}_{LkupVar^s_\varnothing a.x}\}\cup\{AddEdge^{v\xrightarrow{br}o}_{LkupVar^s_\varnothing a.x\xrightarrow{Br}o'}\ where\ \forall o' \in Child^{a\xrightarrow{s}o}_{a.e}\}, s, Br)}\ (\text{ASSIGN})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (f,a.f,\rho)\Rightarrow(N_f,E_f,s_f,Br_f)}{(function\ f(p_1,\ldots,p_n),a,\rho)\Rightarrow(N_f\cup\{on := AddObj^{func}_{a.f}\}, E_f\cup\{AddEdge^{v\xrightarrow{br}o}_{LkupVar^s_\varnothing a.f.name\xrightarrow{Br_f}on}\}\cup\{AddEdge^{a\xrightarrow{s}o}_{a\xrightarrow{s}on}\}\cup\{AddEdge^{o\to a}_{on\to a}\}, s_f, Br_f)}\ (\text{FUNCTION DEF})$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(function\ (p_1,\ldots,p_n),a,\rho)\Rightarrow(N_f\cup\{on := AddObj^{func}_\varnothing\}, E_f\cup\{AddEdge^{a\xrightarrow{s}o}_{a\xrightarrow{s}on}\}\cup\{AddEdge^{o\to a}_{on\to a}\}, s, Br)}\ (\text{CLOSURE DEF})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (f,a.f,\rho)\Rightarrow(N_f,E_f,s_f,Br_f), (a_1,a.a_1,\rho)\Rightarrow(N_{a_1},E_{a_1},s_{a_1},Br_{a_1}),\ldots,(a_n,a.a_n,\rho)\Rightarrow(N_{a_n},E_{a_n},s_{a_n},Br_{a_n})}{(f(a_1,\ldots a_n),a,\rho)\Rightarrow(\bigcup_{i=1}^n N_{a_i}\cup S_c\cup\bigcup_{i=1}^n vn_{a_i}, \bigcup_{i=1}^n E_{a_i}\cup\{AddEdge^{s\xrightarrow{br}s}_{s\xrightarrow{br}s_c}, \forall s_c \in S_c\}\cup E_{call}\cup E_{vo}, S_c, Br)}$$

$$\text{where} \begin{cases} P_{sd} := \{(AddNode^{scope}_{a'_{def}}, a'_{def}), \forall a'_{def} \in a_{def}\},\ S_c := \{p_{sd}[0], \forall p_{sd} \in P_{sd}\}, a_{def} := \{Child^{o\to a}_{o'}, \forall o' \in Child^{a\xrightarrow{s}o}_{a.f}\}, E_{call} := \{AddEdge^{a\xrightarrow{s}a}_{a\xrightarrow{p_{sd}[0]}p_{sd}[1]}, \forall p_{sd} \in P_{sd}\} \\ P_{vo} := \{(s_c,\ AddNode^{var}_{a.a_i},\ Child^{a\xrightarrow{s_c}o}_{a.a_i}), \forall s_c \in S_c, \forall i \in \{1,\ldots,n\}\}, vn_{a_i} := \{p_{vo}[1], \forall p_{vo} \in P_{vo}\}, E_{vo} := \{AddEdge^{v\xrightarrow{br}o}_{p_{vo}[1]\xrightarrow{Br}p'_{vo}[2]}, \forall p_{vo} \in P_{vo}, \forall p'_{vo}[2] \in p_{vo}[2]\}\end{cases}\ (\text{PRE CALL})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (f(a_1,\ldots a_n),a_{pc},\rho)\Rightarrow\rho_{pc}, (B,a_B,\rho_{pc})\Rightarrow\rho_B}{(f(a_1,\ldots a_n),a,\rho)\Rightarrow\begin{cases}(N_{\rho_B}, E_{\rho_B}, s, Br) & \text{Call} \\ (N_{\rho_B}\cup\{n_{to} := AddObj^{obj}_a\}\cup\{n_{tv} := AddNode^{var}_{\text{"}this\text{"}}\}, E_{\rho_B}\cup E_{sv}\cup E_{vo}\cup E_{res}, s, Br) & \text{New}\end{cases}}, \text{where} \begin{cases} B := \{a'.B, \forall a' \in Child^{a\to a}_a\} \\ E_{sv}; := \{AddEdge^{s\to v}_{s_{\rho_{pc}}\to n_{tv}}\} \\ E_{vo} := \{AddEdge^{v\xrightarrow{br}o}_{n_{tv}\xrightarrow{Br}n_{to}}\} \\ E_{res} := \{AddEdge^{a\xrightarrow{s}o}_{a\xrightarrow{s}n_{to}}\}\end{cases}\ (\text{CALL, NEW})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (e,a.e,\rho)\Rightarrow(N_e,E_e,s_e,Br_e),\ \begin{matrix}\rho'_{if} := (N_e,E_e,s_e,Br_e\cup new\ br(a.if)) & \text{(branch sensitive)} \\ \rho'_{else} := (N_e,E_e,s_e,Br_e\cup new\ br(a.else)) & \text{(branch sensitive)} \\ \rho'_{else} := \rho'_{if} := (N_e,E_e,s_e,Br_e) & \text{(branch insensitive)}\end{matrix}\ , (B_{if},a.B_{if},\rho'_{if})\Rightarrow\rho_{if}, (B_{else},a.B_{else},\rho'_{else})\Rightarrow\rho_{else}}{(if(e)\{B_{if}\}else\{B_{else}\},a,\rho)\Rightarrow\ \begin{matrix}(N_{\rho_{if}}, E_{\rho_{if}}\cup\{AddEdge^{a\to a}_{a\to a.if}\}, s_{\rho_{if}}, Br_{\rho_{if}}) & C_{true} = True \\ (N_{\rho_{else}}, E_{\rho_{else}}\cup\{AddEdge^{a\to a}_{a\to a.else}\}, s_{\rho_{else}}, Br_{\rho_{else}}) & C_{false} = False\quad where\ C_{true} = \wedge\{Child^{a\xrightarrow{s}o}_{a_{\rho_e}}\}, C_{false} = \vee\{Child^{a\xrightarrow{s}o}_{a_{\rho_e}}\} \\ (N_{\rho_{if}}\cup N_{\rho_{else}}, E_{\rho_{if}}\cup E_{\rho_{else}}\cup\{AddEdge^{a\to a}_{a\to a.if}\}\cup\{AddEdge^{a\to a}_{a\to a.else}\}, s, Br) & else\end{matrix}}\ (\text{IF})$$

$$\frac{(x = x+1, a', \rho)\Rightarrow\rho_{x+1}}{(x++,a,\rho)\Rightarrow\rho_{x+1}}\ \frac{(x = x-1, a', \rho)\Rightarrow\rho_{x-1}}{(x--,a,\rho)\Rightarrow\rho_{x-1}}\ (\text{INC/DEC})\ \frac{(x_1 = x_1\ op\ x_2, a', \rho)\Rightarrow\rho_{x_1\ op\ x_2}}{(x_1\ aop\ x_2, a, \rho)\Rightarrow\rho_{x_1\ op\ x_2}}\ (\text{ASSIGN OP})\ \frac{\rho \Rightarrow (N,E,s,Br)}{(c,a,\rho)\Rightarrow(N\cup\{ao := AddObj^*_a\}, E\cup\{AddEdge^{a\xrightarrow{s}o}_{a\xrightarrow{s}ao}\}, s, Br)}\ (\text{CONST})$$

$$\frac{(e_1,a.e_1,\rho)\Rightarrow(N_{e_1},E_{e_1},s_{e_1},Br_{e_1}),\ldots,(e_n,a.e_n,\rho)\Rightarrow(N_{e_n},E_{E_n},s_{e_n},Br_{e_n}),}{(e_1,\ldots,e_n,a,\rho)\Rightarrow(\bigcup_{i=1}^n N_{e_i}, \bigcup_{i=1}^n E_{e_i}, s_{e_n}, Br_{e_n})}\ (\text{EXPRESSION LIST})\ \frac{(B_{try},a.B_{try},\rho)\Rightarrow(N_t,E_t,s_t,Br_t), (B_{catch},a.B_{catch},\rho_{B_{try}})\Rightarrow(N_c,E_c,s_c,Br_c)}{(try\{B_{try}\}catch\{B_{catch}\},a,\rho)\Rightarrow(N_t\cup N_c, E_t\cup E_c, s, br)}\ (\text{TRY-CATCH})$$

$$\frac{(e_1,a.e_1,\rho)\Rightarrow\rho_{e_1}, (B_1,a.B_1,\rho'_{e_1})\Rightarrow\rho_{B_1},\ldots,(e_n,a.e_n,\rho)\Rightarrow\rho_{e_n}, (B_n,a.B_n,\rho'_{e_n})\Rightarrow\rho_{B_n}\ where\ \rho'_{e_i} = \begin{cases}(N_{\rho_{e_i}}, E_{\rho_{e_i}}, s_{\rho_{e_i}}, new\ br(e_i)\cup Br_{\rho_{e_i}}) & \text{(branch-sensitive)} \\ (N_{\rho_{e_i}}, E_{\rho_{e_i}}, s_{\rho_{e_i}}, Br_{\rho_{e_i}}) & \text{(branch-insensitive)}\end{cases}}{(switch\ e_1\{B_1\}\ldots e_n\{B_n\},a,\rho)\Rightarrow(\bigcup_{i=1}^n\{if\ Child^{a\xrightarrow{s}o}_{a_{\rho_{e_i}}} = True\ then\ N_{\rho_{B_i}}\ else\ \varnothing\}, \bigcup_{i=1}^n\{if\ Child^{a\xrightarrow{s}o}_{a_{\rho_{e_i}}} = True\ then\ E_{\rho_{B_i}}\cup\{AddEdge^{a\to a}_{a\to a.B_i}\}\ else\ \varnothing\}, s, Br)}\ (\text{SWITCH})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (e,a.e,\rho)\Rightarrow(N_e,E_e,s_e,Br_e)}{(return\ e,a,\rho)\Rightarrow(N_e, E_e\cup\{AddEdge^{a\xrightarrow{s}o}_{a'\xrightarrow{s}o'}, where\ a' = AST_{caller}, o' = Child^{a\xrightarrow{s}o}_{a.e}\}, s, Br)}\ (\text{RETURN})\ \frac{(e,a.e,\rho)\Rightarrow\rho_e, (B_1,a.B_1,\rho_e)\Rightarrow\rho_{B_1}, (B_2,a.B_2,\rho_e)\Rightarrow\rho_{B_2}}{(e:\{B_1\}?\{B_2\},a,\rho)\Rightarrow if\ Child^{a\xrightarrow{s}o}_{a.\rho_e} = True\ then\ \rho_{B_1}\ else\ \rho_{B_2}}\ (\text{TERNARY})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (x_1,a.x_1,\rho)\Rightarrow(N_{x_1},E_{x_1},s_{x_1},Br_{x_1}),\ldots,(x_n,a.x_n,\rho)\Rightarrow(N_{x_n},E_{x_n},s_{x_n},Br_{x_n})}{([x_1,\ldots,x_n],a,\rho)\Rightarrow(\bigcup_{i=1}^n N_{x_i}\cup\{ao := AddObj^{array}_\varnothing\}\cup\{v_i = AddNode^{var}_i, \forall i \in \{1,\ldots,n\}\}, \bigcup_{i=1}^n E_{x_i}\cup\{AddEdge^{o\to v}_{ao\to v_i}, AddEdge^{v\xrightarrow{br}o}_{v_i\xrightarrow{Br}o_i}, where\ \forall o_i \in Child^{a\xrightarrow{s}o}_{a.x_i}, \forall i \in \{1,\ldots,n\}\}, s, Br)}\ (\text{ARRAY})$$

$$\frac{\rho \Rightarrow (N,E,s,Br), (e,a.e,\rho)\Rightarrow\rho_e, (B,a.B,\rho_e)\Rightarrow\rho_B}{(while\ (e)\{B\},a,\rho)\Rightarrow(N_{\rho_B}, E_{\rho_B}, s, Br)}\ (\text{WHILE})\ \frac{\rho \Rightarrow (N,E,s,Br), (e_1,a.e_1,\rho)\Rightarrow(a_{e_1},\rho_{e_1}), (e_2,a.e_2,\rho_{e_1})\Rightarrow\rho_{e_2}, (B,a.e_2,\rho_{e_2})\Rightarrow\rho_B, (e_3,\rho_B)\Rightarrow\rho_{e_3}}{(for(e_1;e_2;e_3)\{B\},a,\rho)\Rightarrow(N_{\rho_{e_3}}, E_{\rho_{e_3}}, s, Br)}\ (\text{FOR})$$

loop until $\rho_B$ or $\rho_{e_3}$ does not change or the number of looping reaches the threshold

Figure 5: Operational Semantics for ODG Construction.

include object definition and use from AST (DEF$_{obj}$ and USE$_{obj}$), property lookups (PROP$^{name}_{obj}$ and PROTOTYPE$_{x[y]}$), data-flows (UNSANITIZED$_{obj}$ and UNSANITIZEDSINK$_{sink}$), AST pattern matching (MATCH$_p$, VULASGMT$_{o1[o2]=o3}$, VULASGMT$_{o1=o2[o3]}$, and ARG$^n_{func}$) and control-flows (CTR$^n_{before/after}$).

Table 2: Basic Graph Traversals (edges are defined in Table 1)

| Traversal | Description |
|---|---|
| $\textsc{Def}_{obj}$ | Object Definition: $(a_1 = obj) \rightarrow o \rightarrow a_2$. |
| $\textsc{Use}_{obj}$ | Object use: $(a_1 = obj) \rightarrow o \xrightarrow{reverse} a_2$. |
| $\textsc{Prop}_{obj}^{name}$ | Property Lookup: $(a = obj) \rightarrow o_1 \rightarrow (v = name) \rightarrow o_2$. |
| $\textsc{Prototype}_{x[y]}$ | Prototype-related Property Lookup: $(a_0 = x) \rightarrow o_0 \rightarrow \{(v_k = \text{“\_\_}proto\text{\_\_”}) \xrightarrow{Br_k} o_k\}_{k>0, Br_{k+1} \subset Br_k} \rightarrow (v = y) \rightarrow o_{k+1}$, where $\{\}_k$ means repeating $k$ times. |
| $\textsc{Unsanitized}_{obj}$ | A Backward Unsanitized Dataflow traversal [15]. |
| $\textsc{UnsanitizedSink}_{sink}$ | A Forward Unsanitized Dataflow traversal, i.e., a reverse version of $\textsc{Unsanitized}_{obj}$. |
| $\textsc{Match}_p$ | This Match Traversal finds an AST node $p$ [15]. |
| $\textsc{VulAsgmt}_{o1[o2]=o3}$ | $\textsc{Unsanitized}_{o2} \bigcap \textsc{Match}_{o1[o2]=o3}$ |
| $\textsc{VulAsgmt}_{o1=o2[o3]}$ | $\textsc{Unsanitized}_{o3} \bigcap \textsc{Match}_{o1=o2[o3]}$ |
| $\textsc{Arg}_{func}^{n}$ | A traversal matches a function $func$ and obtains its $n$th argument. |
| $\textsc{Ctr}_{before/after}^{n}$ | A traversal follows control flow edges either forward ($after$) or backward ($before$). |

Table 3: Graph Traversals for Different Vulnerabilities

| Vulnerability | Graph Queries |
|---|---|
| Internal Property Tampering | |
| Prototypical | $\textsc{PrototypeLookup}_{o1[o5]} \circ (\textsc{Use}_{o1} \bigcap \textsc{Ctr}_{after}) \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$ |
| Direct | $\textsc{VulAsgmt}_{o1=o4[o5]} \circ \textsc{Def}_{o1} \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$ |
| Prototype Pollution | |
| __proto__ | $\textsc{VulAsgmt}_{o1=o4[o5]} \circ \textsc{Def}_{o1} \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$ |
| constructor | $\textsc{VulAsgmt}_{o4=o6[o7]} \circ \textsc{Def}_{o4} \circ \textsc{VulAsgmt}_{o1=o4[o5]} \circ \textsc{Def}_{o1} \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$ |
| Injection-related Vulnerabilities | |
| Command injection | $\textsc{Unsanitized} \circ \textsc{Arg}_{Child\_process.exec}^{1}$ |
| Arbitrary code exe. | $\textsc{Unsanitized} \circ \textsc{Arg}_{eval}^{1}$ |
| SQL injection | $\textsc{Unsanitized} \circ \textsc{Arg}_{connection.query}^{1}$ |
| Reflected XSS | $\textsc{Unsanitized} \circ \textsc{Arg}_{response.write}^{1}$ |
| Stored XSS | $\textsc{Unsanitized} \circ (\textsc{Arg}_{connection.query}^{1} \bigcup (\textsc{Arg}_{connection.query}^{1} \circ \textsc{Unsanitized} \circ \textsc{Arg}_{response.write}^{1}))$ |
| Improper File Access | |
| Path traversal | $(\textsc{UnsanitizedSink}_{\textsc{Prop}_{\textsc{Arg}_{callback}^{2}}^{write}} \bigcap \textsc{Ctr}_{after}) \circ (\textsc{UnsanitizedSink}_{ReadFile} \bigcap \textsc{Ctr}_{after}) \circ \textsc{Prop}^{*} \circ \textsc{Arg}_{callback}^{1} \circ \textsc{Def As}\ callback \circ (\textsc{Arg}_{CreateServer}^{1} \bigcup \textsc{Arg}_{CreateHttpServer}^{2})$ |
| Arbitrary file write | $(\textsc{UnsanitizedSink}_{\textsc{Prop}_{fs}^{writeFile}} \bigcap \textsc{Ctr}_{after}) \circ \textsc{Prop}^{*} \circ \textsc{Arg}_{o1}^{1} \circ \textsc{Def As}\ o1 \circ (\textsc{Arg}_{CreateServer}^{1} \bigcup \textsc{Arg}_{CreateHttpServer}^{2})$ |

### 4.2 Vulnerability Descriptions

In this subsection, we describe how to use graph traversals to represent four big categories of vulnerabilities in Table 3.

**Object-related Vulnerabilities** We describe graph traversals of two object-related vulnerability:

- Internal Property Tampering. Internal property tampering (IPT) [5–7] allows an adversary to alter an internal property, either under an object directly or a prototypical object, so that future property lookups are affected. IPT has two main conditions: (i) a vulnerable assignment statement controllable by an adversary, and (ii) a property lookup after (i). We list graph traversals of both prototypical and direct property tampering in Table 3 based on these two conditions.
- Prototype Pollution. Prototype pollution allows an adversary to alter a built-in function following the prototype chain. There are traditionally two prototype pollution patterns: one through `__proto__` (i.e., `obj.__proto__.toString`) and the other through `constructor` (i.e., `obj.constructor.prototype`). We describe graph traversals for both patterns in Table 3: The former has two vulnerable assignments before the target and the latter has three.

**Injection Vulnerabilities** Injection vulnerabilities allow adversaries to execute arbitrary code via injections into a sink function via user inputs. Such vulnerabilities are detected via finding a backward taint-flow from a sink to an adversary-controlled source and we model this taint-flow as $\textsc{Unsanitized} \circ \textsc{Arg}_{sink}^{*}$. The traversals for specific injection vulnerabilities are shown in Table 3.

**Improper File Access** Improper file access allows an adversary to either read or write files on the filesystem without a proper permission. We model two example types of vulnerabilities in Table 3.

- Path Traversal. Path (directory) traversal allows an adversary to navigate through directories via `../` to access local files. We model it from a web server creation, to the callback of HTTP(s) request, then to a file read (`ReadFile`), and finally to the HTTP(s) response in Table 3.
- Arbitrary File Write. Arbitrary file read allows an adversary to write to arbitrary files due to improper input validation. We model the vulnerability from a web server creation, to the callback, and then to the write to the file system in Table 3.

## 5 Implementation

We implemented an open-source prototype of ODGen at this repository (`https://github.com/Song-Li/odgen`). The implementation has three major parts:

- (i) ODG representation and query. The ODG together with AST and CFG is stored in memory and queried based on a Python library, NetworkX (`https://networkx.github.io/`). We also store ODG with AST and CFG using pickle, a Python object serialization method, to the harddisk for future queries. Note that we adopt NetworkX instead of a graph database like Neo4j, because we find that an in-memory graph management is more efficient than a graph database stored on the disk, especially during abstract interpretation.
- (ii) JavaScript parser. The JavaScript parser is based on Esprima (`https://esprima.org`) and we added implementations to convert AST from Esprima to the standard format of CPG, i.e., those accepted by joern [15] and phpjoern [21]. Note that we adopt the standard format so that we can compare ODG with CPG in the evaluation.
- (iii) Abstract interpretation. We implemented a customized abstract interpretation in Python and modeled popular built-in functions via JavaScript. Our implementation includes popular AST features that are used by >5% of

Table 4: [RQ1] Vulnerability coverage of different code representation for modeling vulnerability types in the CVE database between January 2019 and September 2020.

| **Vulnerability type** | **# of CVE** | **Code Representations** | | |
|---|---|---|---|---|
| | | CPG* | AST+ODG | AST+CFG+ODG |
| Prototype pollution | 71 | | (✓) | ✓ |
| Command injection | 67 | ✓ | ✓ | ✓ |
| Cross Site Scripting (XSS) | 60 | ✓ | ✓ | ✓ |
| Path (directory) traversal | 32 | (✓) | | ✓ |
| Arbitrary code execution | 18 | ✓ | ✓ | ✓ |
| Improper access control | 14 | ✓ | | ✓ |
| Internal property tampering | 11 | | (✓) | ✓ |
| Denial of Service (DoS) | 11 | | | |
| Regex DoS (ReDoS) | 9 | | | |
| Design errors | 8 | | | |
| Information exposure | 8 | ✓ | ✓ | ✓ |
| Arbitrary file write | 8 | (✓) | | ✓ |
| SQL injection | 5 | ✓ | ✓ | ✓ |
| SSRF | 4 | ✓ | | ✓ |
| CSRF | 2 | ✓ | | ✓ |
| Insecure HTTP | 2 | ✓ | ✓ | ✓ |
| Total | 330 | | | |

*: CPG = AST + CFG + PDG.
(✓): It can be detected but with reduced capability.

Node.js packages. Note that we set a timeout as 30 seconds in practice of analyzing Node.js packages.

# 6 Evaluation

In this section, we evaluate ODGEN by answering the following research questions.

- RQ1: What are the recent Node.js vulnerability types and is ODG capable of modeling them?
- RQ2: What is the capability of ODGEN in detecting zero-day vulnerabilities among a large number of real-world NPM packages?
- RQ3: What are the False Positives (FPs) and False Negatives (FNs) of ODGEN?
- RQ4: What is the code coverage and performance overhead of the abstract interpretation?
- RQ5: How will branch-sensitivity affect the vulnerability detection of ODGEN?

We performed our experiments on a server with 192 GB = 6*32GB RDIMM 2666MT/s Dual Rank memory, Intel$^{\circledR}$ Xeon$^{\circledR}$ E5-2690 v4 2.6GHz, 35M Cache, 9.60GT/s QPI, Turbo, HT, 14C/28T (135W) Max Mem 2400MHz, and 4 * 2TB 7.2K RPM SATA 6Gbps 3.5in Hot-plug Hard Drive.

## 6.1 RQ1: Historical Node.js vulnerability coverage

In this subsection, we answer the research question on the ODG's capability in modeling real-world Node.js package vulnerabilities. We start from querying the central database maintained by the MITRE organization together with information provided by the synk.io database for recent (i.e., January 2019–September 2020) vulnerabilities of Node.js packages on NPM. In total, we retrieved 330 vulnerabilities of Node.js packages after excluding vulnerabilities of Node.js platforms (e.g., those with underlying memory issues). We then manually go through the vulnerability by downloading the originally vulnerable package and analyze the code together with the descriptions on CVE and snyk.io to understand the vulnerability category. Table 4 shows all 16 vulnerability categories and corresponding # of CVEs in the database.

Next, we follow the evaluation methodology adopted in the CPG paper [15] to manually analyze what code representations are necessary in describing those vulnerability categories in Node.js. In addition to the code presentations in CPG, we add ODG and try to understand the capability of ODG in describing vulnerabilities. Note that the object-level data dependency is a more fine-grained version of statement-level data dependency in PDG, and thus we do not need to study PDG+ODG in the code representation.

Table 4 shows the analysis results: ODGEN is able to model 13 out of 16 vulnerability types, i.e., 302 out of 330 recent vulnerabilities. The rest vulnerability types are general Denial of Service, Regex Denial of Service (ReDoS), and bad designs. ODG cannot model ReDoS because it is caused by a vulnerable regex rather than JavaScript; ODG cannot model many other DoS because some of them are caused by the event loop. Fortunately, Staicu et al. [22] and Davis et al. [23] either detect or defend against DoS attacks. ODG cannot model vulnerabilities due to bad designs, e.g., incorrect validation of inputs—this is the same as the CPG paper, which leaves design errors out of scope as well.

## 6.2 RQ2: Zero-day Node.js vulnerabilities

In this research question, we evaluate the capability of ODGEN in detecting zero-day Node.js vulnerabilities both at the application-level and the package-level as described in Section 2.2. Specifically, we crawled 300K NPM packages on February 25, 2020 and applied ODGEN with graph queries to detect corresponding vulnerabilities. Our target vulnerability is selected from the top ones in Table 4; we also intentionally include those that are unique to JavaScript, such as prototype pollution and internal property tampering.

**Results.** Table 5 (the "# reported" column) shows a list of vulnerabilities found by ODGEN. Due to time limit and the extensive number of reported vulnerabilities, we manually checked and exploited all the vulnerable applications and these vulnerable packages with >1,000 weekly downloads. The "TP" column indicates that we can generate an exploit to compromise the package if deployed locally and the vulnerability is not an intended functionality of the package, and the "FP" column that we fail to generate a working exploit or the vulnerability is an intended functionality of the package, e.g., a package like `shell-utils` designed to execute arbitrary OS command. Lastly, the "# CVE" column is the total number of CVE identifiers that we obtained.

We first break down all the found vulnerabilities by application- vs. package-level in Table 5 . The number of application-level vulnerabilities is relatively small compared with the one of package-level. This is because the total number of Node.js standalone applications is also much smaller than the one of packages.

Table 5: [RQ2] A breakdown of zero-day vulnerabilities found by ODGen.

| | #Reported | #Checked | TP | FP | #CVE |
|---|---|---|---|---|---|
| Total | 2,964 | 264 | 180 | 84 | 70 |
| *App. vs. package breakdown* | | | | | |
| Application-level | 57 | 57 | 43 | 14 | 6 |
| Indirect Package-level | 34 | 34 | 15 | 19 | 0 |
| Direct Package-level | 2,873 | 173 | 122 | 51 | 64 |
| *Vulnerability type breakdown* | | | | | |
| Path traversal | 109 | 40 | 30 | 10 | 6 |
| Command injection | 1,253 | 108 | 80 | 28 | 52 |
| Arbitrary code execution | 183 | 17 | 14 | 3 | 8 |
| Internal property tampering | 910 | 46 | 24 | 22 | 0 |
| Prototype pollution | 492 | 36 | 19 | 17 | 4 |
| Cross Site Scripting (XSS) | 17 | 17 | 13 | 4 | 0 |

**(a) Vulnerable code:**

```
1 module.exports = function deparam( params ) {
2   var obj = {};
3   params.replace(/\+/g, ' ').split('&').forEach(
        function(v){
4     var param = v.split('='), key = decodeURIComponent(
          param[0]), cur = obj, i = 0;
5     ... // convert string "key" to array "keys",
          e.g., 'a[b][c]' -> ['a', 'b', 'c']
6     var keys_last = keys.length - 1;
7     if ( param.length === 2 ) {
8       val = decodeURIComponent( param[1] );
9       for ( ; i <= keys_last; i++ ) {
10        key = keys[i];
11        if (i < keys_last) {
12          cur = cur[key] || (keys[i+1] && isNaN( keys[i
              +1] ) ? {} : []);
13        } else {
14          cur = cur[key] = val;  // vulnerable location
15        }
16      }
17    }
18  });
19  return obj;
20 };
```

**(b) Exploit:**

```
1  var deparam = require("deparam");
2  var payload = "a[__proto__][toString]=123";
3  deparam(payload);
4  console.log({}.toString)
```

Figure 6: [RQ2] A package-level prototype pollution in deparam and the exploit code (It leads to an application-level vulnerability in PDX-Parks, a park search application).

We then break down these vulnerabilities by their types in Table 5. The number of command injection vulnerabilities is the most among all the vulnerability types as Node.js is commonly used as a client- or server-side utility application to start OS applications. We also find many prototype pollution vulnerabilities as this is a relatively new type. The number of XSS vulnerabilties is small because our prototype implementation only models the simple web server provided by the Node.js framework but not those advanced web frameworks.

**Case Study.** In this part, we describe a popular Node.js package, called `deparam`, which has two other variations on NPM, `node-jquery-deparam` and `jquery-deparam`. All three packages provide reverse functions for the famous jquery function `$.param()`, called `deparam`. The function `deparam` takes a parameterized query string and converts the string back into an object.

Table 6: Baseline Detectors (CI: Command Injection, ACE: Arbitrary Code Execution, PT: Path Traversal, PP: Prototype Pollution)

| Name | Type | In-scope vuln. | Original tool | Our impl.* (LoC) |
|---|---|---|---|---|
| JSJoern | static | CI, ACE, PT | phpjoern [21] | 260 (Java)+415 (Python) |
| NodeJsScan | regex | CI, ACE, PT | NodeJsScan [24] | N/A |
| JSTap-vul | static | CI, ACE, PT | JSTap [8] | 134 (Python) |
| Synode-det | static | CI, ACE, PT | Synode [2] | 74 (Java) |
| PPFuzzer | dynamic | PP | Arteau [3] | N/A |
| Nodest | static | CI, ACE | Nodest [1] | 288 (Java)+27 (Javascript) |
| Ensemble | The combination of the above six detectors. | | | |

*: Because some tools are not for vulnerability detection, target another language or are close-sourced, we have to retrofit them for evaluation of vulnerability detection. Note that we keep their static analysis part integral.

Table 7: [RQ3-FP] FP/(FP+TP) of general-purpose static detectors.

| JSJoern | JSTap-vul | ODGen |
|---|---|---|
| 15/(15+5) = 75% | 16/(16+4) = 80% | 84/(84+180) = 32% |

`deparam` is vulnerable to prototype pollution as shown in the simplified code of Figure 6 (a) and the exploit in Figure 6 (b). Specifically, when `deparam` constructs an object, it does not check whether a property lookup follows the prototype chain (Line 14 of Figure 6 (a)). Therefore, an adversary can pollute `Object.prototype.toString` using the code at Line 2 of Figure 6 (b): When the for-loop at Line 9 is executed for the second time, `toString` is polluted at Line 14.

Since one popular use of `deparam` is to parse the query string of an URL, it will lead to application-level vulnerabilities. We search the use of `deparam` on github and find a real-world vulnerable web application, called `PDX-Parks` (`https://github.com/meandavejustice/pdx-parks`), which allows a user to search for nearby parks with given latitude and longitude. `PDX-Parks` adopts `deparam` to decompose a query string into an object, thus being vulnerable. Specifically, we deployed the website locally and exploited the site via `http://localhost/parks?[__proto__][toString]=123`, which leads to a Denial-of-Service (DoS) for all legitimate requests. The reason is that `PDX-Parks` adopts `express`, which needs a correct `toString` function.

### 6.3 RQ3: FP and FN

In this subsection, we answer the research question of the false positives (FPs) and false negatives (FNs) of ODGen.

**Baseline Detectors.** We now introduce several baseline vulnerability detectors for the purpose of comparing with ODGen in Table 6 including the technique type (static vs. dynamic vs. regex) and their in-scope vulnerabilities. Because we modified several existing JavaScript static analysis tools, such as phpjoern, Synode, and JSTap, to detect Node.js vulnerabilities, we also make our modification open-source in the same URL as ODGen.

**False Positives.** In this part, we evaluate the false positives (FPs) of ODGen and compare it with two other general-

Table 8: [RQ3-FP] A breakdown of FPs of ODGEN.

| **Vulnerability** | Unmodeled function | Unsolvable constraints | Intended functionality |
|---|---|---|---|
| Command injection | 7 | 9 | 12 |
| Arbitrary code execution | 1 | 1 | 1 |
| Prototype pollution | 7 | 8 | 2 |
| Path traversal | 0 | 10 | 0 |
| Internal property tampering | 0 | 21 | 1 |

purpose, static detectors, i.e., JSJoern and JSTap-vul. We apply both tools on 300K Node.js packages and then select the detected packages with Top 20 weekly downloads for manual verification. Table 7 shows the comparison results. JSJoern and JSTap have very high FPs because they do not have points-to information. Due to the lack of points-to information, they have to make many over-approximations, which lead to wrong call edges. Note that we did not compare with either dynamic or regex based detectors on FPs, because they are using different techniques, which tend to have low FPs. We also did not compare with Synode-det or Nodest due to scalability issues: Nodest needs installations of all dependencies and Synode-det does not support many ES6 features.

We also manually inspect all the FPs for ODGEN and break down the FPs by vulnerability types and reasons in Table 8. There are three main reasons: (i) unmodeled built-in functions, (ii) unsolvable constraints, and (iii) intended functionalities. First, our prototype of ODGEN only models popular Node.js built-in functions, i.e., those used by more than 5% packages. If ODGEN does not model a unpopular function especially when it is used for sanitization, ODGEN may report a false positive. Second, ODGEN does not solve all the control- and data-flow constraints, but only calculates all possible constant values if they are available. Therefore, it is possible that ODGEN finds a path, but the constraints along the path cannot be satisfied. Third, some packages may be designed for a certain functionality, e.g., executing an OS command. ODGEN will detect them as command injection, but this is not a vulnerability.

Figure 7 shows an FP example of unsolvable constraints for prototype pollution. ODGEN reports it as prototype pollution because ODGEN finds two vulnerable assignments at Lines 7 (in the first loop run) and 8 (in the second loop run). Then, the assigned value at Line 8 is also controllable by the adversary. However, although the assigned value o at Line 8 is controllable by the adversary, it happens to be the same as the assignee cur[nameTokens[i]]. ODGEN needs to add additional constraints for the assigned value so that it can remove such an FP.

**False Negatives.** In this part, we evaluate the false negatives (FNs) of ODGEN by using a benchmark of legacy CVE vulnerabilities. Specifically, we downloaded historical packages (until February 2020) with five categories of vulnerabilities from CVE as a benchmark. It is worth noting that we exclude some vulnerabilities, such as XSS in this benchmark, because they involve many different web frameworks, many of which have not been modeled in our prototype implementation.

```
1      //pixi-gl-core@1.1.4
2      function getUniformGroup(nameTokens, uniform)
3      {
4         var cur = uniform;
5         for (var i = 0; i < nameTokens.length - 1; i++)
6         {
7            var o = cur[nameTokens[i]] || {data:{}};
8            cur[nameTokens[i]] = o;
9            cur = o;
10        }
11        return cur;
12     }
```

Figure 7: [RQ3-FP] A false positive example of prototype pollution reported by ODGEN.

```
// curlrequest@1.0.1
  exports.request = function(options,callback){
    if (arguments.length === 1) {
      exports.request.call(this, options, callback);
    ... } // request calls itself.
    if (options.retries) {
      exports.request(options, function (err) {}
    ... }  // request calls itself.
      exports.copy(options);  // request calls copy.
  }
  exports.copy = function (obj) {
    for (var i in obj) {
      if (Array.isArray(obj[i])) {...}
      else if (typeof obj[i] === 'object') {
        copy[i] = obj[i] ? exports.copy(obj[i]) :
             null; // copy calls itself.
        } else {...}
      }
    return copy;
  };
```

Figure 8: [RQ3-FN] A false negative example in detecting a legacy path traversal vulnerability (multiple recursive calls lead to object explosion and time-out).

Table 9 shows the false negatives of ODGEN and existing analysis tools in detecting CVE vulnerabilities. Clearly, ODGEN's true positives are the highest and false negatives are the lowest, i.e., outperforming all existing works in detecting legacy CVE vulnerabilities because of the modeling of object-level data dependencies. We breakdown all the FNs of ODGEN into two reasons in Table 10 and describe them below. First, we only modeled a limited number of built-in functions, i.e., those that are adopted by more than 5% of Node.js packages. Therefore, ODGEN may miss some data dependencies due to lack of modeling. Second, the abstract interpretation of ODGEN may time out and leave a partial ODG without finishing interpreting all Node.js functions.

We also show a specific FN example in Figure 8. This example has a path traversal vulnerability, but the abstract interpretation cannot reach the vulnerable code because of multiple recursive calls for both request() and copy() functions. The number of object nodes for each functions is over 15k and multiple recursive calls lead to an object explosion even with our hybrid branch sensitivity.

Table 9: [RQ3-FN] Comparison of ODGen with prior program analysis in detecting legacy CVE vulnerabilities.

| Detector | Total | | Command injection | | Prototype pollution | | Arbitrary code execution | | Path traversal | | Internal property tampering | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TP | FN | TP | FN | TP | FN | TP | FN | TP | FN | TP | FN |
| NodeJsScan | 5 | 251 | 2 | 73 | - | - | 2 | 29 | 1 | 86 | - | - |
| JSJoern | 39 | 217 | 22 | 53 | - | - | 5 | 26 | 12 | 75 | - | - |
| JSTap-vul | 52 | 204 | 27 | 48 | - | - | 5 | 26 | 12 | 75 | - | - |
| Synode-det | 7 | 249 | 6 | 69 | - | - | 1 | 30 | 0 | 87 | - | - |
| Nodest | 7 | 249 | 7 | 68 | - | - | 0 | 31 | - | - | - | - |
| PPFuzzer | 29 | 23 | - | - | 29 | 23 | - | - | - | - | - | - |
| Ensemble | 115 | 141 | 46 | 29 | 29 | 23 | 13 | 18 | 27 | 60 | 0 | 11 |
| **ODGen** | 189 | 67 | 67 | 8 | 40 | 12 | 20 | 11 | 55 | 32 | 7 | 4 |

Table 10: [RQ3-FN] A breakdown of reasons of FNs of ODGen.

| Vulnerability name | # Timeout | # Unmodeled function |
|---|---|---|
| Command injection | 4 | 4 |
| Prototype pollution | 9 | 3 |
| Arbitrary code execution | 5 | 6 |
| Path traversal | 22 | 10 |
| Internal property tampering | 2 | 2 |

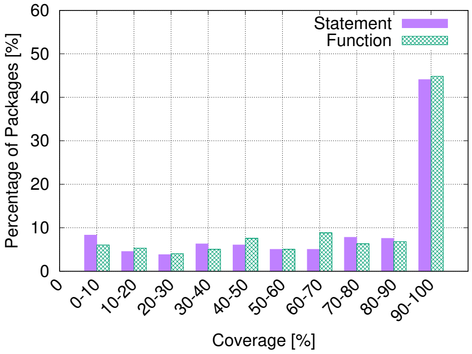


Figure 9: [RQ4-Coverage] Distribution of statement and function coverage (timeout: 30 seconds). One major reason of uncovered code is the runtime inclusion of JavaScript files depending on inputs.

### 6.4 RQ4: Abstract Interpretation Performance

We answer the research question on the code coverage and performance overhead of abstract interpretation implemented in ODGen.

**Code Coverage.** In this subsection, we answer the research question on the code coverage of ODGen's abstract interpretation in terms of two specific metrics: statement coverage and function coverage. Statement coverage defines the percentage of statements that are executed and function coverage the percentage of functions that are analyzed by ODGen. Both metrics show how complete ODGen is in analyzing Node.js packages. Figure 9 shows a distribution graph of statement and function coverages when analyzing 500 randomly-selected Node.js packages with a timeout as 30 seconds. The figure is almost an even distribution graph from 0 to 90% and then shows a sudden jump in 90–100%. Actually, about 40% of packages have 100% code coverage.

The reasons of a relatively low coverage of some packages are as follows. First, there are some dead code that are copied from another package or online that is not invoked from the exported function. Second, some packages may dynamically include a file depending on the inputs, which cannot be statically resolved. Third, some functions, particularly exported ones, will return another function as a return value—such returned functions will only be called if another package invokes them.

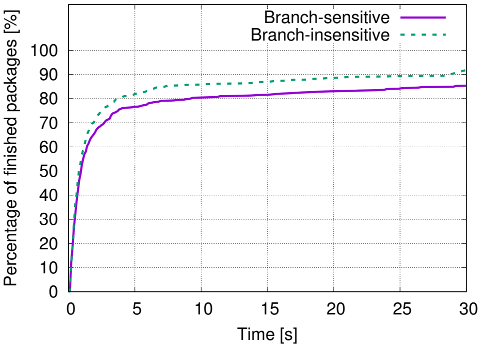


Figure 10: [RQ4-Performance] CDF graph of total execution time to finish analysis.

**Performance Overhead.** In this subsection, we answer the research question of the performance overhead of ODGen in generating ODG for real-world Node.js packages. Our methodology is as follows. We randomly select 500 Node.js packages and run ODGen against all the packages until the analysis finishes or time out. Figure 10 shows a CDF graph with 30 seconds as the time-out threshold: ODGen finishes analyzing 85% of packages within 30 seconds when being branch sensitive and 93% when being branch insensitive. This evaluation shows that ODGen is efficient in generating ODG for most of Node.js packages.

### 6.5 RQ5: Branch-sensitivity

In this subsection, we answer the research question on how branch-sensitivity affects the vulnerability detection of ODGen. Table 11 shows the number of detected vulnerabilities under different branch sensitivities. Clearly, the hybrid branch sensitivity adopted by ODGen detects the largest number of vulnerabilities: It combines both advantages, i.e., accuracy and scalability, with and without branch sensitivity.

Figure 11 shows why the hybrid branch sensitivity will help

Table 11: [RQ5] the number of detected legacy CVE vulnerabilities with branch sensitivity enabled and disabled.

| Vulnerability name | Hybrid | Branch-sensitive | Branch-insensitive |
|---|---|---|---|
| Command injection | 67 | 64 | 66 |
| Prototype pollution | 40 | 36 | 29 |
| Arbitrary code execution | 20 | 18 | 17 |
| Path traversal | 55 | 55 | 51 |
| Internal property tampering | 7 | 6 | 7 |
| Total | 189 | 179 | 170 |

```javascript
// limdu@0.9.4
  exports.toSvmLight =
    function(dataset, bias, binarize,
         firstFeatureNumber) {
      var lines = "";
      for (var i=0; i<dataset.length; ++i) {
        var line = (i>0? "\n": "") +  // 2 objects
        (binarize? (dataset[i].output>0? "1": "-1"):
             dataset[i].output) + // 2+1 objects
        featureArrayToFeatureString(dataset[i].input,
             bias, firstFeatureNumber); // 54 objects
        // 2*3*54 objects
        lines += line;
      }; // (2*3*54)^3=34,012,224 objects
    ...
    }
```

Figure 11: [RQ5] A false negative in detecting a legacy command injection vulnerability with branch-sensitive mode (The number of objects explodes and ODGEN times out).

the detection of more vulnerabilities. We annotate the source code with the number of object nodes in branch sensitivity enabled. Because the source code has multiple conditional expressions and a for loop, the number of object nodes quickly increases to over 34 million. ODGEN will reduce to branch insensitive mode in abstractly interpreting the code when object explosion is detected.

## 7 Discussion and Limitation

*Ethics: Responsible Disclosure.* We have disclosed all 180 zero-day vulnerabilities to their developers together with Proof of Vulnerability (PoV) under the help of snyk.io. All the details of these vulnerabilities can be found in the appendix. If we do not hear from the developer, we will publicly release the vulnerability after a 60-day disclosure window. So far, 12 vulnerable packages have already been fixed.

*Prototype Implementation and Limitation.* We now discuss several implementation choice and limitation.

- *Supported JavaScript Features.* Our prototype implementation follows the popularity of AST features among Node.js packages, i.e., we implemented those that are used by more than 5% of packages. Note that ODGEN can still analyze packages with unimplemented features but just skip the unimplemented part.
- *Asynchronous Callbacks and Events.* The prototype implementation of ODGEN adopts a queue structure to store asynchronous callbacks during registration and invokes them one by one. We acknowledge that this is just one of many possibilities that could happen in a real execution and leave the modeling of an event-based call graph like Madsen et al. [25] as a future work.
- *For-loop and Recursive Call in Abstract Interpretation.* As discussed in Section 3.2, ODGEN extensively executes a for-loop until no more new objects outside the loop are being looked-up. ODGEN also adopts a minimum time as three and a maximum as ten in abstractly interpreting for loops and recursive calls. The minimum value is designed in case some external objects are not modeled in depth; the maximum value is designed to avoid dead loop and reduce performance overhead.
- *Dynamically-included Files.* As a general limitation of static analysis, ODGEN cannot analyze any files that are dynamically included depending on user inputs. This can only be analyzed with user inputs and dynamic analysis.
- *Sanitization Functions.* The prototype implementation of ODGEN adopts a list of sanitization functions, e.g., parseInt, in analyzing dataflow. Currently, the list is generated manually and we leave it for the future work for automatic generation.

*Path-sensitivity.* ODGEN is partially path-sensitive, i.e., ODGEN will calculate boolean, string and integer values if they are either constant or enumerable. For an if statement, if the value can be determined, ODGEN will abstractly interpret only one branch; otherwise, ODGEN will abstractly interpret both branches in parallel.

## 8 Related Work

**Node.js Vulnerability Detection and Defense.** In the past, researchers have studied Node.js vulnerabilities and we discuss them based on their vulnerability types. Arteau [3] proposes a fuzzer to explore Node.js packages for prototype pollution. DAPP [13] uses AST and control-flow patterns to detect prototype pollution vulnerabilities with very high false positive and negative rates (50.6% and 84.6% respectively). ObjLupAnsys [12] detects prototype pollution by expanding object lookups and propagating taints during abstract interpretation. Nodest [1] proposed a closed-source detection framework to detect command injection vulnerabilities following the risks as mentioned by Ojamaa et al. [26]. Then, SYNODE [2] adopts a rewriting technique to enforce a template before executing a possible injection API like eval. Many prior works [22, 27, 28] propose to detect or defend against regular expression DoS (ReDoS); Davis et al. [23] propose to defend against Event Handler Poisoning (EHP) DoS attack.

Other than specific vulnerabilities, ConflictJS [29] studied and analyzed conflicts among different JavaScript libraries; Zimmermann et al. [30] studied the robustness of third-party Node.js packages and their influence on other packages' security. Researchers [31] have also proposed to study the binding layers of the Node.js for all kinds of vulnerabilities. Mininode [32] proposes to reduce the attack surface of Node.js and improve the overall security. As a comparison, ODGEN is the first general graph query-based framework of JavaScript for efficient detection of a variety types of Node.js vulnerabilities.

**Client-side JavaScript Security.** JavaScript is traditionally used at client-side as the scripting language and has been studied [33–36] long before the appearance of Node.js. Cross-site scripting (XSS) [37–41] and Cross-Site Script Inclusion attack (XSSI) [42] attacks are well studied on the client side. Malicious JavaScript has been studied by many prior works, such as HideNoSeek [43], JShield [44] and JSTap [8], for detection and defense. Researchers proposed to secure JavaScript via security policies, such as content security policy. Examples are like GateKeeper [45] and CSPAutoGen [46]. Program analysis [47, 48] have also been adopted at the client side for security analysis. Many prior works [49–52] have been proposed to restrict JavaScript, especially those from third-party, in a subset for security. We believe that ODG is able to analyze client-side JavaScript as well and leave those as our future work. In the evaluation, we compared ODGEN with JSTap, a client-side JavaScript analysis tool that can generate program dependency graph (PDG). The results show that ODGEN can detect more vulnerability than JSTap.

**Static Analysis of JavaScript.** TAJS [10] and JSAI [11] adopt abstract interpretation to analyze JavaScript programs for more accurate call graph generation and then detect type-related errors. Madsen et al. [25] propose event-based call graph to detect problems reported on StackOverflow. Brave's PageGraph [53] and its predecessor AdGraph [54] model the relations between different browser objects like scripts, DOM and AJAX during runtime with concrete inputs. JAW [55] models browser objects in a Hybrid Property Graph, which contains Event Registration, Dispatch and Dependency Graph, Inter-Procedural Call Graph, AST, PDG, and CFG. Guarnieri et al. [9] propose to adopt heap graph to model local object relations. SAFE [56] and follow-ups [17, 57] convert JavaScript to an IR form and adopt an internal structure for abstract interpretation. As a comparison, the lattice structure in TAJS and JSAI, the heap graph by Guarnieri et al., the Object Property Graph in the aforementioned ObjLupAnsys [12], and the data structure in SAFE change during abstract interpretation, which cannot be used offline for graph query, because many object-related information gets lost as the interpretation. PageGraph, AdsGraph and Hybrid Property Graph are offline structure, but they are designed to include browser objects rather than JavaScript objects. That is, none of these three can be used to detect JavaScript vulnerabilities in this paper.

**General Vulnerability Detection Framework.** Previous works, such as Program dependence graph (PDG) [58] and Combined C Graph (CCG) [59], have proved that it is effective to combine program analysis with graph representation to model data and control dependencies for operations in a program. Based on graph representation, many program analysis problems can be converted to graph-related problems, such as graph-reachability problem [60], graph query problem [15, 16, 61–63]. Specifically, Code Property Graph (CPG) is proposed by Yamaguchi et al. [15] as a general frame work combining CFG, DFG, and AST to detect C/C++ vulnerabilities. Later on, CPG is ported to PHP by Backes et al. [16] as an open-source tool called phpjoern [21]. As a comparison, ODGEN models object dependencies, such as object lookup/definition, which are unavailable in any of existing graph structures.

Other than graph-based frameworks, in the past, code analysis [64–68] has been also widely used to detect various vulnerabilities on different platforms. The concept of objects and relations between objects are also adopted in traditional program analysis and defenses [69, 70], such as Object Flow Integrity [70]. The concepts of objects in JavaScript are different from those on C/C++ due to the existence of prototype and runtime resolution, which makes traditional object analysis not applicable on JavaScript.

## 9 Conclusion

In this paper, we propose to generate a novel graph structure, called Object Dependence Graph (ODG), via abstract interpretation. ODG accepts graph queries to mine a variety of Node.js vulnerabilities, especially those related to objects such as prototype pollution and internal property tampering. We implement a prototype, open-source system, called ODGEN, to construct ODG via context- and flow-sensitive static analysis with hybrid branch sensitivity and points-to information. Our evaluation reveals 180 zero-day vulnerabilities and 70 of them have already been assigned with CVE identifiers.

## Acknowledgement

We would like to thank our shepherd, Alexandros Kapravelos, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA.

## References

[1] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: Feedback-driven static analysis of node.js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 455–465, Association for Computing Machinery, 2019.

[2] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node.js," 2018.

[3] O. Arteau, “Prototype pollution attack in nodejs application.” NorthSec, 2018.

[4] “Path traversal in npm package for node.js.” https://www.cybersecurity-help.cz/vdb/SB2019121218.

[5] “[CVE-2019-10790] internal property tampering affecting taffy package, all versions.” https://snyk.io/vuln/SNYK-JS-TAFFY-546521.

[6] “[CVE-2019-10805] internal property tampering affecting valib package, all versions.” https://snyk.io/vuln/SNYK-JS-VALIB-559015.

[7] “[CVE-2019-2391, cve-2020-7610] internal property tampering affecting bson package, versions >=1.0.0 <1.1.4.” https://snyk.io/vuln/SNYK-JS-BSON-561052.

[8] A. Fass, M. Backes, and B. Stock, “Jstap: A static pre-filter for malicious javascript detection,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, (New York, NY, USA), p. 257–269, Association for Computing Machinery, 2019.

[9] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, “Saving the world wide web from vulnerable javascript,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 177–187, 2011.

[10] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for javascript,” in *International Static Analysis Symposium*, pp. 238–255, Springer, 2009.

[11] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, “Jsai: a static analysis platform for javascript,” in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, pp. 121–132, 2014.

[12] S. Li, M. Kang, J. Hou, and Y. Cao, “Detecting node.js prototype pollution vulnerabilities via object lookup analysis,” in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[13] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, “DAPP: automatic detection and analysis of prototype pollution vulnerability in node. js modules,” *International Journal of Information Security*, pp. 1–23, 2021.

[14] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, “Abusing hidden properties to attack the node.js ecosystem,” in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[15] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, IEEE, 2014.

[16] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, “Efficient and flexible discovery of php application vulnerabilities,” in *2017 IEEE european symposium on security and privacy (EuroS&P)*, pp. 334–349, IEEE, 2017.

[17] J. Park, J. Park, D. Youn, and S. Ryu, “Accelerating javascript static analysis via dynamic shortcuts,” in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[18] P. Cousot and R. Cousot, “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*, pp. 238–252, 1977.

[19] “[CVE-2019-10768] prototype pollution affecting angular package, versions >=1.4.0-beta.6 <1.7.9.” https://snyk.io/vuln/SNYK-JS-ANGULAR-534884.

[20] “[CVE-2017-16042] arbitrary code injection affecting growl package, versions <1.10.0.” https://snyk.io/vuln/SNYK-JS-PM2-474345.

[21] “Parser utility to generate asts from php source code suitable to be processed by joern.” https://github.com/malteskoruppa/phpjoern.

[22] C.-A. Staicu and M. Pradel, “Freezing the web: A study of redos vulnerabilities in javascript-based web servers,” in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 361–376, 2018.

[23] J. C. Davis, E. R. Williamson, and D. Lee, “A sense of time for javascript and node.js: first-class timeouts as a cure for event handler poisoning,” in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 343–359, 2018.

[24] “Nodejsscan—nodejsscan is a static security code scanner for node.js applications.” https://ajinabraham.github.io/NodeJsScan/.

[25] M. Madsen, F. Tip, and O. Lhoták, “Static analysis of event-driven node.js javascript applications,” *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 505–519, 2015.

[26] A. Ojamaa and K. Düüna, “Assessing the security of node. js platform,” in *2012 International Conference for Internet Technology and Secured Transactions*, pp. 348–355, IEEE, 2012.

[27] J. Davis, F. Servant, and D. Lee, "Using selective memoization to defeat regular expression denial of service (redos)," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[28] Z. Bai, K. Wang, H. Zhu, Y. Cao, and X. Jin, "Runtime recovery of web applications under zero-day redos attacks," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[29] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: finding and understanding conflicts between javascript libraries," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 741–751, 2018.

[30] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 995–1010, 2019.

[31] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 559–578, IEEE, 2017.

[32] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, (San Sebastian), pp. 121–134, USENIX Association, Oct. 2020.

[33] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: A virtualized browser to sandbox third-party javascripts with enhanced security," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, (New York, NY, USA), p. 8–9, Association for Computing Machinery, 2012.

[34] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk, "Redefining web browser principals with a configurable origin policy," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, 2013.

[35] Z. Chen and Y. Cao, "Jskernel: Fortifying javascript against web concurrency attacks via a kernel-like structure," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 64–75, 2020.

[36] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic browser," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 163–178, 2017.

[37] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Precise browser-neutral prevention of cross-site scripting attacks," in *IEEE Symposium on Security and Privacy*, 2009.

[38] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense," in *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[39] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *Proceeding of the Network and Distributed System Security Symposium (NDSS.07)*, 2007.

[40] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise client-side protection against dom-based cross-site scripting," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 655–670, 2014.

[41] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1193–1204, 2013.

[42] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 723–735, 2015.

[43] A. Fass, M. Backes, and B. Stock, "Hidenoseek: Camouflaging malicious javascript in benign asts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1899–1913, 2019.

[44] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "Jshield: towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 466–475, ACM, 2014.

[45] S. Guarnieri and B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code," in *USENIX Security*, 2009.

[46] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 653–665, 2016.

[47] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 314–324, IEEE, 2015.

[48] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pp. 45–59, 2019.

[49] S. Maffeis, J. C. Mitchell, and A. Taly, "An operational semantics for javascript," in *Asian Symposium on Programming Languages and Systems*, pp. 307–325, Springer, 2008.

[50] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "Adsafety: type-based verification of javascript sandboxing," in *Proceedings of the 20th USENIX conference on Security*, pp. 12–12, USENIX Association, 2011.

[51] Google, "Google caja." http://code.google.com/p/google-caja/.

[52] "SES." https://github.com/tc39/proposal-ses.

[53] "Brave pagegraph," https://github.com/brave/brave-browser/wiki/PageGraph.

[54] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *IEEE Symposium on Security and Privacy*, May 2020.

[55] S. Khodayari and G. Pellegrino, "JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[56] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, vol. 10, Citeseer, 2012.

[57] S. Bae, H. Cho, I. Lim, and S. Ryu, "Safewapi: Web api misuse detector for web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 507–517, Association for Computing Machinery, 2014.

[58] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization.," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[59] D. A. Kinloch and M. Munro, "Understanding c programs using the combined c graph representation.," in *ICSM*, pp. 172–180, 1994.

[60] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[61] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61–S71, 2015.

[62] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 291–302, 2015.

[63] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*, pp. 797–812, IEEE, 2015.

[64] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis.," in *USENIX Security*, 2005.

[65] X. Zhang, A. Edwards, and T. Jaeger, "Using cqual for static analysis of authorization hook placement.," in *USENIX Security Symposium*, pp. 33–48, 2002.

[66] A. P. Sistla, V. Venkatakrishnan, M. Zhou, and H. Branske, "Cmv: Automatic verification of complete mediation for java virtual machines," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pp. 100–111, ACM, 2008.

[67] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, "A security policy oracle: detecting security holes using multiple api implementations," in *ACM SIGPLAN Notices*, vol. 46, pp. 343–354, ACM, 2011.

[68] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 6–pp, IEEE, 2006.

[69] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 648–659, IEEE.

[70] W. Wang, X. Xu, and K. W. Hamlen, "Object flow integrity," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1909–1924, 2017.

## Appendix

In this appendix, we list all the zero-day vulnerabilities found by ODGEN in Tables 12, 13, 14, 15, 16, and 17.

Table 12: A List of command injection zero-day vulnerabilities found by ODGen (80 in total).

| Package Name | Version | Status | CVE # | Package Name | Version | Status | CVE # |
|---|---|---|---|---|---|---|---|
| adb-driver | 0.1.8 | confirmed | CVE-2020-7636 | pomelo-monitor | 0.3.7 | confirmed | - |
| apiconnect-cli-plugins | 6.0.2 | confirmed | CVE-2020-7633 | promise-probe | 0.1.8 | fixed | CVE-2019-10791 |
| aws-lambda | 1.0.4 | fixed | CVE-2019-10777 | pulverizr | 0.7.0 | confirmed | CVE-2020-7603 |
| blamer | 0.1.13 | fixed | CVE-2019-10807 | push-dir | 0.4.1 | confirmed | CVE-2019-10803 |
| clamscan | 1.1.0 | confirmed | CVE-2020-7613 | pygmentize-bundled | 2.3.0 | confirmed | - |
| closure-compiler-stream | 0.1.15 | confirmed | CVE-2020-7604 | rpi | 0.0.3 | confirmed | CVE-2019-10796 |
| codecov | 3.6.1 | fixed | CVE-2020-7596/7597 | serial-number | 1.3.0 | confirmed | CVE-2019-10804 |
| compass-compile | 0.0.1 | confirmed | CVE-2020-7635 | strong-nginx-controller | 1.0.2 | confirmed | CVE-2020-7621 |
| compile-sass | 1.0.3 | fixed | CVE-2019-10799 | truffle-compile-vyper | 1.0.27 | submitted | - |
| curling | 0.3.0 | confirmed | CVE-2019-10789 | umount | 1.1.6 | confirmed | CVE-2020-7628 |
| devcert-sanscache | 0.4.6 | fixed | CVE-2019-10778 | vsce | 1.71.0 | confirmed | - |
| diskusage-ng | 0.2.4 | confirmed | CVE-2020-7631 | connection-tester | 0.2.0 | confirmed | CVE-2020-7781 |
| docker-compose-remote-api | 0.1.4 | confirmed | CVE-2020-7606 | buns | 1.1.6 | confirmed | CVE-2020-7794 |
| effect | 1.0.4 | confirmed | CVE-2020-7624 | monorepo-build | 0.1.9 | confirmed | CVE-2020-28423 |
| enpeem | 2.2.0 | confirmed | CVE-2019-10801 | s3-kilatstorage | 0.5.6 | confirmed | CVE-2020-28424 |
| fsa | 0.5.1 | confirmed | CVE-2020-7615 | geojson2kml | 0.1.1 | confirmed | CVE-2020-28429 |
| fsh | 0.0.2 | confirmed | - | image-tiler | 2.0.1 | confirmed | CVE-2020-28451 |
| get-git-data | 1.3.1 | confirmed | CVE-2020-7619 | curljs | 0.1.2 | confirmed | CVE-2020-28425 |
| git-add-remote | 1.0.0 | confirmed | CVE-2020-7630 | nuance-gulp-build-common | 0.0.1 | confirmed | CVE-2020-28430 |
| git-diff-apply | 0.19.7 | fixed | CVE-2019-10776 | ffmpeg-sdk | 0.0.5 | confirmed | CVE-2020-28435 |
| git-revision-webpack-plugin | 3.0.4 | confirmed | CVE-2020-7612 | lycwed-spritesheetjs | 1.2.2 | confirmed | - |
| git-tag | 0.2.0 | confirmed | - | wangzhe | 1.0.0 | confirmed | - |
| giting | 0.0.7 | fixed | CVE-2019-10802 | karma-ckb-reporter | 0.0.3 | confirmed | - |
| gulp-anybar | 1.0.1 | confirmed | - | surfboard | 0.1.0 | confirmed | - |
| gulp-scss-lint | 1.0.0 | confirmed | CVE-2020-7601 | ensure-module-latest | 1.0.9 | confirmed | - |
| gulp-styledocco | 0.0.3 | confirmed | CVE-2020-7607 | geojson2 | 0.1.8 | confirmed | - |
| gulp-tape | 1.0.0 | confirmed | CVE-2020-7605 | kill-process-occupying-port | 0.0.1 | confirmed | - |
| heroku-addonpool | 0.1.15 | confirmed | CVE-2020-7634 | shelljs.exec | 1.1.8 | confirmed | - |
| im-resize | 2.3.2 | fixed | CVE-2019-10787 | lycwed-spritesheetjs | 1.2.2 | confirmed | - |
| install-package | 0.4.0 | confirmed | CVE-2020-7629 | theme-core | 0.2.5 | confirmed | - |
| jscover | 1.0.0 | confirmed | CVE-2020-7623 | wc-cmd | 1.0.9 | confirmed | - |
| karma-mojo | 1.0.1 | confirmed | CVE-2020-7626 | gulp-tvm-tsc | 0.3.4 | confirmed | - |
| lsof | 0.1.0 | confirmed | CVE-2019-10783 | nuance-gulp-build-packers-dotnet | 0.0.0 | confirmed | - |
| mysql-dumper | 6.3.0 | confirmed | - | stream-jspm | 0.0.1 | confirmed | - |
| network-manager | 1.0.2 | confirmed | CVE-2019-10786 | hot-update-package | 1.0.6 | confirmed | - |
| node-key-sender | 1.0.11 | confirmed | CVE-2020-7627 | pstracker | 0.0.4 | confirmed | - |
| node-mpv | 1.4.3 | confirmed | CVE-2020-7632 | tile-web | 3.0.0 | confirmed | - |
| node-prompt-here | 1.0.1 | confirmed | CVE-2020-7602 | tvm | 0.8.14 | confirmed | - |
| npm-programmatic | 0.0.12 | confirmed | CVE-2020-7614 | nmcli-wrapper | 0.7.0 | confirmed | - |
| op-browser | 1.0.6 | confirmed | CVE-2020-7625 | gulp-shellexec | 0.4.4 | confirmed | - |

Table 13: A List of prototype pollution zero-day vulnerabilities found by ODGen (19 in total).

| Package Name | Version | Status | CVE # | Package Name | Version | Status | CVE # |
|---|---|---|---|---|---|---|---|
| asciitable.js | 1.0.2 | confirmed | CVE-2020-7771 | fun-map | 3.3.1 | confirmed | CVE-2020-7644 |
| bayrell-nodejs | 0.8.0 | submitted | - | grunt-util-property | 0.0.2 | confirmed | CVE-2020-7641 |
| blindfold | 1.0.1 | submitted | - | lodash._baseset | 4.3.0 | submitted | - |
| class-transformer | 0.2.3 | fixed | CVE-2020-7637 | jquery-deparam | 0.5.3 | submitted | - |
| debt | 0.0.4 | submitted | - | magico | 1.1.1 | submitted | - |
| dnspod-client | 0.1.3 | submitted | - | node-file-cache | 1.0.2 | submitted | - |
| draft | 0.2.3 | submitted | - | object-helpers | 0.0.4 | submitted | - |
| extend2 | 1.0.1 | submitted | - | parse-mockdb | 0.4.0 | submitted | - |
| fetch-wrap | 0.1.2 | submitted | - | propper | 1.3.0 | submitted | - |
| field | 1.0.1 | submitted | - | | | | |

Table 14: A List of Arbitrary Code Execution zero-day vulnerabilities found by ODGen (14 in total).

| Package Name | Version | Status | CVE # | Package Name | Version | Status | CVE # |
|---|---|---|---|---|---|---|---|
| @flammae/helpers | 0.0.3 | submitted | - | lisp-json-to-js | 0.4.1 | submitted | - |
| access-policy | 3.1.0 | comfirmed | CVE-2020-7674 | mosc | 1.0.0 | confirmed | CVE-2020-7672 |
| alt-class | 0.0.3 | comfirmed | - | node-extend | 0.2.0 | comfirmed | CVE-2020-7673 |
| cd-messenger | 2.7.26 | comfirmed | CVE-2020-7675 | node-import | 0.9.2 | confirmed | CVE-2020-7678 |
| couchdb-ddoc-test | 1.0.0 | comfirmed | - | node-rules | 4.0.2 | fixed | CVE-2020-7609 |
| inline-ng2-resources | 1.1.0 | submitted | - | pixl-class | 1.0.2 | fixed | CVE-2020-7640 |
| json-log-filter | 0.1.2 | submitted | - | thenify | 3.3.0 | comfirmed | CVE-2020-7677 |

Table 15: A List of Path Traversal zero-day vulnerabilities found by ODGen (30 in total).

| Package Name | Version | Status | CVE # | Package Name | Version | Status | CVE # |
|---|---|---|---|---|---|---|---|
| 11xiaoli | 1.1.0 | submitted | - | rollup-plugin-dev-server | 0.4.3 | submitted | - |
| 123qwe | 1.0.0 | submitted | - | rollup-plugin-serve-favicon | 0.4.7 | confirmed | CVE-2020-7684 |
| 1997server | 1.3.0 | submitted | - | rollup-plugin-serve | 1.0.1 | confirmed | CVE-2020-7683 |
| allserverming | 1.0.0 | submitted | - | rollup-plugin-server | 0.7.0 | confirmed | CVE-2020-7686 |
| entryhttp | 1.0.0 | submitted | - | static-server-g | 1.0.0 | submitted | - |
| fanwen | 1.0.0 | submitted | - | thy_server | 1.6.0 | submitted | - |
| fast-http | 0.1.3 | confirmed | CVE-2020-7687 | uekserver | 1.0.0 | submitted | - |
| jbbmyplay | 1.0.1 | submitted | - | w1703_server | 1.2.0 | submitted | - |
| lddll | 1.0.0 | submitted | - | waterfallhzw | 1.0.0 | submitted | - |
| lhm-ssi | 1.0.1 | submitted | - | wu456 | 1.0.0 | submitted | - |
| lserver | 1.0.9 | submitted | - | xuewarp | 1.0.0 | submitted | - |
| marked-tree | 0.8.1 | confirmed | CVE-2020-7682 | xxx-server-yyy | 1.0.1 | submitted | - |
| marscode | 1.0.1-0 | confirmed | CVE-2020-7681 | zlymain | 1.0.0 | submitted | - |
| musciplayer-szj | 2.0.0 | submitted | - | zzl-server | 1.0.5 | submitted | - |
| myserver123 | 1.0.0 | submitted | - | xhttpserver | 0.0.6 | submitted | - |

Table 16: A List of XSS zero-day vulnerabilities found by ODGen (13 in total).

| Package Name | Version | Status | CVE # | Package Name | Version | Status | CVE # |
|---|---|---|---|---|---|---|---|
| buildseverlzz | 1.0.0 | submitted | - | sheepy | 0.1.1 | submitted | - |
| hxsstatic | 1.0.8 | submitted | - | simple_server | 0.1.0 | submitted | - |
| lserver | 1.0.9 | submitted | - | simplewebserver | 1.2.0 | submitted | - |
| lymph-server | 1.2.0 | submitted | - | xxx-server-yyy | 1.0.1 | submitted | - |
| lyss | 0.0.1 | submitted | - | zzl-selver | 1.0.3 | submitted | - |
| min-http | 1.0.6 | submitted | - | zzl-server | 1.0.5 | submitted | - |
| node-servers | 1.0.3 | submitted | - | | | | |

Table 17: A List of internal property tampering zero-day vulnerabilities found by ODGen (24 in total).

| Package Name | Version | Status | CVE # | Package Name | Version | Status | CVE # |
|---|---|---|---|---|---|---|---|
| anyargs | 1.0.5 | submitted | - | leFunc | 1.2.5 | submitted | - |
| citeproc-js-node | 0.0.3 | submitted | - | lethexa-adt | 0.0.13 | submitted | - |
| diso.router | 6.0.3 | submitted | - | optometrist | 1.0.1 | submitted | - |
| domlib | 1.0.7 | submitted | - | resorting-key | 1.0.0 | submitted | - |
| hyperdrive-ui | 4.0.2 | submitted | - | solar | 0.1.6 | submitted | - |
| x-validator | 0.1.0 | submitted | - | immutable-record-class | 3.8.1 | submitted | - |
| ini | 2.0.0 | submitted | - | lazy-cache | 2.0.2 | submitted | - |
| acos-kelmu | 0.1.1 | submitted | - | bare | 0.0.2 | submitted | - |
| charity-base-form | 1.9.0 | submitted | - | common-codegen-tests | 2.2.3 | submitted | - |
| cookiemonster | 1.1.0 | submitted | - | deherd-scraper-engine | 1.2.11 | submitted | - |
| ikagaka.nar.js | 3.0.3 | submitted | - | jquery-register | 1.1.1 | submitted | - |
| ndx-modified | 0.1.2 | submitted | - | ng-pipe | 1.4.10 | submitted | - |

# A Artifact Appendix

## A.1 Abstract

In the paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and accepts graph queries—especially on object lookups and definitions—for detecting Node.js vulnerabilities.

We implemented an open-source prototype system, called ODGEN, to generate ODG for Node.js programs via abstract interpretation and detect vulnerabilities. Our evaluation of recent Node.js vulnerabilities shows that ODG together with AST and Control Flow Graph (CFG) is capable of modeling 13 out of 16 vulnerability types. We applied ODGEN to detect six types of vulnerabilities using graph queries: ODGEN correctly reported 180 zero-day vulnerabilities, among which we have received 70 Common Vulnerabilities and Exposures (CVE) identifiers so far.

In this artifact evaluation, we claim that OPGEN is capable of detecting all six types of vulnerabilities and found all the zero-day vulnerabilities.

## A.2 Artifact check-list (meta-information)

- **Algorithm:** Mining Node.js Vulnerabilities via Object Dependence Graph and Query
- **Data set:** We use the self-generated dataset and it is included in the docker image
- **Run-time environment:** Ubuntu 20.04 is recommanded and tested. The main software dependencies are Python 3.7+, pip, npm, and Node.js 12+
- **Run-time state:** No
- **Metrics:** Number of detected vulnerable packages
- **Output:** The testing results are located in the "logs" folder of the running directory. All the detected vulnerable packages will be output to the "succ.log" file; All the un-detected packages will be output to the "results.log" file. You can get the number of the successfully detected packages by running "cat ./logs/succ.log | wc -l", during or after the running process.
- **Experiments:** You can download and load the docker, or set up the environment from the source code. Then run the pre-written scripts and see the results.
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 10 to 30 mins
- **How much time is needed to complete experiments (approximately)?:** 200 mins
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPL v3.0
- **Data licenses (if publicly available)?:** GPL v3.0
- **Archived (provide DOI)?:**

## A.3 Description

### A.3.1 How to access

We provide two methods for testing, loading the docker image is highly recommanded:

- A docker image

  We uploaded our docker to Docker Hub. You can pull it by running

  **docker pull iamthesong/odgen:latest**

  Then you can attach to this docker by running

  **docker run -it iamthesong/odgen bash**

  After loading it, you should be able to see the environment

- A repository for the source code

  If you are not able to access the virtual machine and can not load the docker image, you can also try to clone our source code from the GitHub repository https://github.com/Song-Li/ODGen/tree/24d68fa810cae8c028cf36f269461e178c198c98 (commit hash: 24d68fa810cae8c028cf36f269461e178c198c98) and follow the instructions in the README.md to set up the environment.

### A.3.2 Hardware dependencies

Recommended

- CPU: 16 cores
- Memory: 16GB

Minimum

- CPU: 4 cores
- Memory: 4GB

### A.3.3 Software dependencies

If you want to start with the source code, Ubuntu 20.04 is recommended. This artifact requires Python 3.7+, pip, npm, and Node.js 12+.

## A.4 Installation

### A.4.1 Docker image

We prepared a docker image on Docker Hub. You can follow the commands mentioned in section A.3.1 to download the load the docker.

### A.4.2 Source code

**Setup the Environment** If you want to start with the source code, we recommend you to use Ubuntu 20.04., you can simply cd into the source code folder and install the software dependencies by running:

**./ubuntu_setup.sh**

After the packages are successfully installed, you can setup the environment by running:

**./install.sh**

The script **install.sh** will install a list of required Python and Nodejs dependencies. Once finished, the environment is setup and we are ready to go.

**Verify the Installation** You can run the script **odgen_test.py** to verify the installation. The command is:

**python3 ./odgen_test.py**

If the environment is successfully set up, you should be able to see the tests are finished without errors. The end of the outputs should be like:

```
Ran 3 tests in XXXs
OK
```

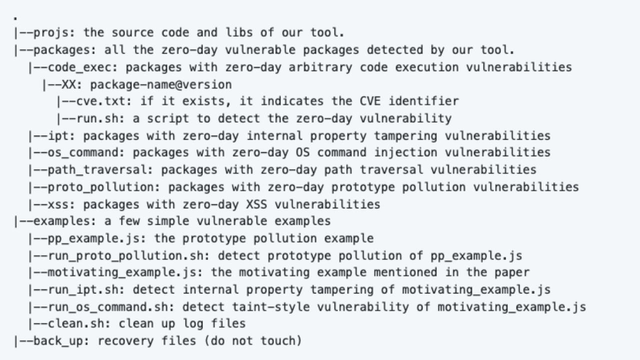
## A.5 Experiment workflow

As we claimed in the Abstract section and the Contributions part of section 1 in our paper, our main claim that needed to be evaluated is we found 43 application-level and 137 package-level zero-day vulnerabilities. Our tool can successfully found the vulnerabilities of those packages. We prepared the dataset and the related scripts to run our tool on top of the packages.

Besides the main claim, other evaluation results, including the performance, the code coverage, and the false-negative rate of our paper are also reproducible and reproduced by the reviewers. I will also include the steps and datasets to reproduce the related evaluation results in the next section.

### A.5.1 File organization of our Docker Image

Once you log into the docker, all the files and folders are organized as follows:



```
.
|--projs: the source code and libs of our tool.
|--packages: all the zero-day vulnerable packages detected by our tool.
  |--code_exec: packages with zero-day arbitrary code execution vulnerabilities
    |--XX: package-name@version
      |--cve.txt: if it exists, it indicates the CVE identifier
      |--run.sh: a script to detect the zero-day vulnerability
  |--ipt: packages with zero-day internal property tampering vulnerabilities
  |--os_command: packages with zero-day OS command injection vulnerabilities
  |--path_traversal: packages with zero-day path traversal vulnerabilities
  |--proto_pollution: packages with zero-day prototype pollution vulnerabilities
  |--xss: packages with zero-day XSS vulnerabilities
|--examples: a few simple vulnerable examples
  |--pp_example.js: the prototype pollution example
  |--run_proto_pollution.sh: detect prototype pollution of pp_example.js
  |--motivating_example.js: the motivating example mentioned in the paper
  |--run_ipt.sh: detect internal property tampering of motivating_example.js
  |--run_os_command.sh: detect taint-style vulnerability of motivating_example.js
  |--clean.sh: clean up log files
|--back_up: recovery files (do not touch)
```

### A.5.2 Dataset

**Dataset 1: Zero-day vulnerable packages**

- **dataset:** The 174 zero-day vulnerable packages that found by our tool. (Note that after our reporting, there are eight packages that are unpublished from NPM. Currently, we only have source code for 173 packages + one package, which is unpublished but cached on our server.)
- **location:** ∼/packages
- the CVEs they got: ∼/packages/xx/package-name@version/cve.txt (if exists)
- a script that runs the analysis on each of these folders/projects and detects the vulnerabilities: ∼/packages/xx/package-name@version/run.sh

  where xx = code_exec, ipt, os_command, path_trasversal, proto_pollution, and xss.

Note that considering the large size of the dataset, we are not able to upload the dataset to the GitHub repository. We uploaded the zipped dataset to Google Drive and if you are testing it by the source code, please download it, unzip it, and put it in the root directory of your machine.

**Dataset 2: Legacy vulnerable packages**

- **dataset:**The legacy vulnerable packages dataset mentioned in Section 6.3 of the paper, including 75 command injection vulnerable packages, 31 code execution vulnerable packages, 52 prototype pollution vulnerable packages, 87 path traversal vulnerable packages, and 11 internal property tampering (IPT) vulnerable packages.
- **location:** We uploaded it as a zip file to the GitHub Repo (https://github.com/Song-Li/legacy_benchmark)

**Dataset 3: Randomly selected packages**

- **dataset:** The 500 randomly selected packages from the NPM database.
- **location:** We uploaded it as a zip file to the GitHub Repo (https://github.com/Song-Li/random_500_npm.git)

### A.5.3 Play with the examples

In the ∼**/examples** folder of the Docker image, we have a few simple vulnerable examples for you to get familiar with our tool. You can try the **run_ipt.sh, run_os_command.sh** or **run_proto_pollution.sh** to run our tool on top of the pp_example.js(a prototype pollution) example and the motivating_example.js(the motivating example introduced in our paper). You can also write your modules, use a similar command and test them out.

## A.6 Evaluation and expected results

### A.6.1 Evaluation

**Zero-day vulnerable packages detection (Dataset 1)** Totally we have six different types of vulnerabilities, they are command injection, code execution, prototype pollution, path traversal, cross-site scripting, and internal property tampering. Each of them can be tested by running a command in the root directory of the source code:

- Command injection: ./scripts/os_command.sh
- Code execution: ./scripts/code_exec.sh
- Prototype pollution: ./scripts/prototype_pollution.sh
- Path traversal: ./scripts/path_traversal.sh
- Cross-site scripting: ./scripts/xss.sh
- Internal property tampering: ./scripts/ipt.sh

To reproduce the results, you can pick a vulnerability type and run the corresponding script.

Note that the scripts will try to run our tool parallelly, so you will not see the progress. Once you run a script, you should be able to see a message that says "new instance". You can check how many processes are still running by the command: *screen -ls*. You can also attach to a specific process by running: *screen -r XXX*(XXX means the name of the screen). Once all the processes are finished, you can check the result and run another script.

The testing results are located in the *logs* folder of the running directory. All the detected vulnerable packages will be output to the *succ.log* file; All the un-detected packages will be output to the *results.log* file. You can get the number of the successfully detected packages by running *cat ./logs/succ.log | wc -l*, during or after the running process.

If you finished checking one vulnerability type, please run *./clean.sh* to remove the logs and temporary files before checking another one.

Note that since the order of the testing functions is randomized, you may encounter some un-detected packages. For the un-detected packages, you may run them independently follow the instructions in README.md, or, go to *∼/packages/vulneralbility-type/package-name@version/* and run the *run.sh*

**False negative rate (Dataset 2)** The false-negative rate is introduced in Table 9 of the paper, which is measured on top of the legacy vulnerable packages. The steps to reproduce it is:

- Login to our Docker by the command *docker run -it iamthesong/odgen bash*
- Make sure you are in the root directory of the docker, and download the dataset by *git clone https://github.com/Song-Li/legacy_benchmark.git*
- Go into the downloaded dataset by *cd legacy_benchmark/* and unzip the dataset by *unzip legacy_benchmark.zip*
- Go into the source code directory by *cd /projs/ODGen/*. Test a type of vulnerability by *./odgen.py -t VUL_TYPE --list /root/legacy_benchmark/VUL_TYPE.list -aq --nodejs --timeout 120 --parallel 16*

Note that

- There are two locations in the last command that use *VUL_TYPE*. *VUL_TYPE* should be replaced by *os_command, ipt, proto_pollution, path_traversal* or *code_exec*
- The *--parallel* argument is used to run ODGen parallelly. In my case, I use 16 to indicate that I want to run 16 processes together. You can adjust the argument based on the number of CPU cores of your device
- The *--timeout* argument is used to set the timeout of a single test. We recommend 300 to make sure it works. In most cases, 120 should be enough.

If the number is less than expected, we need to run multiple times on those packages. You can simply run the same command multiple times (without cleaning the logs), and the results will be logged to the */root/projs/ODGen/logs/succ.log* file, cumulatively. To remove the duplicates, you can go into the */root/projs/ODGen/logs/* folder and run *awk '!x[$0]++' succ.log > outfile.succ*. The generated file *outfile.succ* will be the detected list.

**Code coverage (Dataset 3)** The code coverage rate is introduced in Figure 9 of the paper, which is measured on top of the 500 randomly selected Node.js packages. We prepared the statement-level code coverage API and the randomly selected 500 packages for testing.

Steps to reproduce:

Table 1: Expected Detection Results for Zero-day Vulnerable Packages

| | Command Injection | Code Execution | Prototype Pollution | Path Traversal | Cross-site Scripting | IPT |
|---|---|---|---|---|---|---|
| #Packages | 80 | 14 | 19 | 30 | 13 | 24 |
| #Unpublished | 2 | 4 | 0 | 0 | 0 | 0 |
| #Expected | 76∼78 | 9∼10 | 17∼19 | 30 | 12∼13 | 23∼24 |

Table 2: Expected True Positive Packages on Legacy Vulnerable Packages

| | Command Injection | Code Execution | Prototype Pollution | Path Traversal | IPT | Total |
|---|---|---|---|---|---|---|
| #Packages | 75 | 31 | 52 | 87 | 11 | 256 |
| #Claimed True Positive | 67 | 20 | 40 | 55 | 7 | 189 |
| #Expected True Positive | 67 | 20∼21 | 39∼40 | 55∼ 56 | 7∼ 10 | 189∼194 |

Table 3: Reproduced Code Coverage Rate

| Code Coverage | Percentage of Packages |
|---|---|
| 0% to 10% | 5.52% |
| 10% to 20% | 5.25% |
| 20% to 30% | 8.01% |
| 30% to 40% | 2.76% |
| 40% to 50% | 2.76% |
| 50% to 60% | 6.63% |
| 60% to 70% | 2.76% |
| 70% to 80% | 6.35% |
| 80% to 90% | 11.60% |
| 90% to 100% | 48.34% |

- Login to our Docker by the command *docker run -it iamthesong/odgen bash*

- Make sure you are in the root directory of the docker, and download the dataset by *git clone https://github.com/Song-Li/random_500_npm.git*

- Go into the downloaded dataset by *cd random_500_npm/* and unzip the dataset by *unzip ./random_500.zip*

- Go into the source code directory by *cd  /projs/ODGen/*.

- Update the source code to the latest version by *git pull*

- Start the testing by running *./odgen.py -t os_command -ma –list  /random_500_npm/random_500.list --timeout 30 --parallel 20*

- During the running process, you can go to the tools folder by *cd /root/projs/ODGen/tools* and check the results on the fly by running *python get_code_coverage_dis.py*. This script will output the results directly. You can run this command multiple times to see how the code coverage changes during the evaluation.

You can check how many processes are running by *screen -ls*. If all processes are finished, you can check the final result. Note that the code coverage raw data is logged in *ODGen/logs/stat.log*. You can also take a look if you want!

Note that not all of the packages will report code coverage, There are two reasons for that:

- Since the packages are randomly selected, there are many packages that do not meet the requirement of the NPM standard. For example, some of them do not have an entrance file, some of them do not include a package.json file, and some of them are demo packages without any meaningful content. For those packages, ODGen will not report the code coverage;

- It is possibly happening for packages running into a timeout. ODGen will not output the code coverage results for timeout packages since those results can not reflect the real code coverage of ODGen.

### A.6.2 Expected results

**Zero-day vulnerable packages detection** The number of all the packages, unpublished packages, expected detected packages and the estimated running time are listed in Table 1

**False negative rate** The number of all the packages, claimed true positive packages, expected detected packages are listed in Table 2

**Code coverage** The distribution of the code coverage should be comparable to Figure 9 of the paper. The results that reproduced by the reviewers are listed in Table 3

## A.7 Troubleshooting

**Zero-day vulnerable packages detection** If you can not get the expected results, you can try to restart the docker and see if it can run smoothly without the influence of the cache.

**False negative rate** If you can not get the expected results, you can try to:

- When you run the tool multiple times, try to change the number of *--parallel* each time. For example, we can use *--parallel 17* for the first time, and *--parallel 19* for the second time. In that way, each process may start from different packages and it may be faster to generate the results.

- Since the number of packages with code execution vulnerability is not very large. If your device has enough computing resources, for example, more than 20 CPU cores. You can try to set the *--parallel* argument to *--parallel 31* to make sure every vulnerable package can use an independent process. After doing this, you can check how many packages are still running by using *screen -ls*.

## A.8 Experiment customization

You are very welcome to test our tool on top of your customized packages. To do so, please go to the ∼/example folder and write your package follow the NPM package standard, or write a module like the ∼/example/pp_example.js and the ∼/example/motivating_example.js.

Once you prepared the module, you can check out the README.md file in the source code repository and follow the instructions to run the corresponding commands.