论文阅读报告

撰写人： 22S003100 孙骁　　**时间：** **6 月 5 号**

一、 标题

Mining Node.js Vulnerabilities via Object Dependence Graph and Query

二、 出处

2022 USENIX Security Symposium

三、 中英文摘要

Node.js 是一个流行的非浏览器 JavaScript (JS) 运行平台 (非浏览器指该运行环境支持后端开发) ，提供了很多有用的依赖包，人们也可以发布自行开发的程序包。然而有些时候不考虑安全性的编程容易受到攻击。当前有许多基于程序分析的方法来检测 Node.js 中的漏洞的研究工作，例如针对命令注入、原型污染等。但是它们大多是针对单一类型漏洞的，并不能够针对广泛的普适情况。另一方面，也有研究在 C/C++/PHP 上提出了基于图数据查询的方法，如代码属性图 (Code Property Graph, CPG) 来有效地挖掘漏洞，但是由于 JS 语言中频繁使用了动态语言特性，因此不直接适用。

在本文中，提出了具有混合分支敏感性和指向信息的流敏感和上下文敏感的静态分析方法，使用抽象解释生成一种新的图结构，称为对象依赖图 (Object Dependence Graph, ODG) 。该图将 JS 对象表示节点，将它们与抽象语法树 (Abstract Syntax Tree, AST) 的关系表示为边，并接受图形查询，尤其是对象查找和对象定义，以检测 Node.js 漏洞。

研究实现了一个名为 ODGEN 的开源原型系统，通过抽象解释为 Node.js 程序生成 ODG 并检查漏洞。根据最近的 Node.js 漏洞评估表明，ODG 与 AST 和控制流图 (Control Flow Graph, CFG) 搭配可以建模出 16 种漏洞类型中的 13 种，使用 ODGEN 的图数据查询功能检测了六种类型的漏洞，其正确报告了 180 个 zero-day 漏洞，到目前为止，共获取到了 70 个 CVE 标识符。

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 vulnerabilities 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 16vulnerability 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.

四、 研究的问题

Node.js 是一个流行的 JS 运行时环境，支持在 web 浏览器之外执行 JS 代码，例如后端代码。Node.js 生态系统繁杂，npm、yarn 等安装包管理工具支持安装数量为百万级，如此多的依赖包容易受到各种漏洞的攻击，例如命令注入、原型污染、路径遍历与内部属性篡改等。

过去的针对漏洞提出的程序分析方法并不通用，而基于其他语言提出的一大漏洞检测的最新进展旨在建立一个表示目标程序不同属性的图结构，并执行图数据查询挖掘漏洞。以代码属性图 CPG 为例，其结合了抽象语法树、控制流图以及程序依赖图，在 C、C++、PHP 的许多类型漏洞检测中被证明有效。然而 CPG 并不建模对象之间的关系，这正是 JS 语言广泛使用的，例如基于原型链的对象查找和 this 关键词的对象查找（尤其是使用 bind 的对象），即很多动态语言特性引发的对象相关性无法识别，因此 CPG 无法对基于对象的 JS 漏洞进行建模和检验，例如原型污染和内部属性篡改的识别都会存在困难。

同时，已有的静态 JS 分析通过抽象解释以及在线数据结构来建模对象及其关系。然而，这面临两大挑战：其一，它们的目标主要是特定类型的漏洞，因此在线数据结构通常仅包含一类或少量类别的漏洞类型特征，不适合离线检测普适的存在漏洞场景；其二，按照分支敏感性的角度，现有的 JS 分析顺序解释会影响准确性，并行解释会损害可扩展性。

五、 研究的目标

基于 CPG 无法基于语言的特点检测和处理 JS 程序中的很多经典漏洞这一前提，考虑在其基础上提出新的图结构，保留 CPG 支持图数据查询，可泛化检测漏洞的优良特性的同时针对 JS 语言特性进行改良，做到对于对象及其关联关系级别的常见漏洞可以进行常规检测。

本文提出了对象依赖图这种新型图结构，支持图数据查询，用于离线静态检测各种 Node.js 漏洞。其关键在于将 JS 对象表示为节点，并将对象之间和对象与 AST 节点之间关系表示为边。具体来说 ODG 包括对象之间的细粒度数据依赖关系，从而有助于污点式漏洞检测（如命令注入）。同时 ODG 还与 CPG 集成，以支持抽象解释中对于对象间关系的定义与查找。文章还构建 ODGEN 原型系统，在抽象解释过程中生成 ODG，从入口点开始遵循 AST 节点序列来定义和查找抽象作用域下 AST 节点的对象，然后将对象定义和查找记录为 ODG 的一部分，用于生成 CFG 和数据依赖关系。ODGEN 系统采用混合分支敏感，默认并行解释，若节点数量激增则会切换为顺序分支执行。

ODGEN 的有效性由实验给出，研究 CVE 数据库中所有最近的 Node.js 漏洞，并通过对 ODG 的图查询以及现有基于图形的代码进行建模，评估可检测出 16 个漏洞类别中的 13 种；同时在现实世界中的 Node.js 包上评估系统，可以检测到 43 个应用程序级 zero-day 漏洞，14 个为假阳性；137 个包级 zero-day 漏洞，84 个为假阳性；这些漏洞中的 70 个已经被 CVE 标识。

六、 研究的方法

6.1 引入与威胁模型

我们从一个例子出发，介绍检测漏洞时的威胁模型。

```
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.

如图 1 所示是一个仅有六行的简单示例代码。source1、source2 都由敌手控制，代表污点源；sink 函数是污点汇聚点，代表直接产生安全敏感的操作或直接泄露数据的位置。这段代码有两个漏洞：

1. 内部属性篡改。当敌手设置 source2 为__proto__时，第五行执行的指令就是修改__proto__属性指向的对象。在 JS 中，每个函数都有一个特殊的属性原型，prototype，指向一个原型对象。原型对象就是一个普通的 JS 对象，包含着一些属性和方法，可以被该函数创建的所有实例共享。当新建一个实例对象时，该对象会继承构造函数的原型对象中的属性和方法，如果需要访问的属性或方法在自身上找不到，那么对象会沿着原型链向上查找直至找到原型链的树根——Object 对象的原型对象。在 JS 中，每个对象都有一个__proto__属性，指向对象的原型对象。这里代码把 myFunc 实例的原型链切断了，myFunc 的__proto__属性原先指向 Func 的原型对象，现在被改为一个新的原型对象，配置为{‘ab’ source1}，寻找新的原型链可能找不到 x 属性，因此在第六行代码处调用该属性会报错，从而产生影响。又或者在 source1 中定义了 x 属性，而 x 属性被用于用户权限认证，在第六行代码调用中出现越权的后果。

2. 污点式漏洞，如命令注入。当敌手设置 source2 为 x 时发生。由原型链调用，第六行调用 x 属性时应该调用的是 Func 的原型对象的 x 属性获得‘ab’，但是通过第五行代码在 myFunc 实例中新建属性 x，同时赋值为 source1，可被敌手掌控，导致可能的注入。

示例代码的两个漏洞的关键是第 5 行的对象查找操作，ODG 正是特别侧重于建模对象定义、查找与赋值进行漏洞检测。

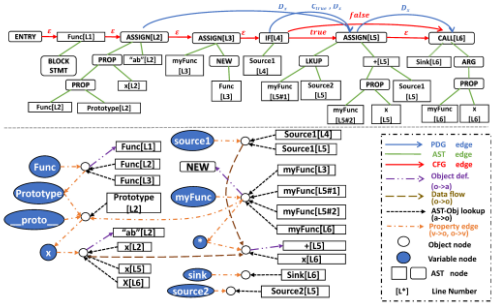

Figure 2: Object Dependence Graph (ODG, Bottom) Integrated with Code Property Graph (CPG, Top) of the Exemplary Code in Figure 1. For readers' convenience, we copied corresponding AST nodes from top to bottom and skipped several unimportant nodes and edges, such as `__proto__` of many objects, the global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

图 2 是基于图 1 示例代码生成的 ODG 图，顶部是结合 AST、CFG 和程序依赖图的代码属性图 CPG；底部则是 ODG。用于检测漏洞的过程如下图 3、4。

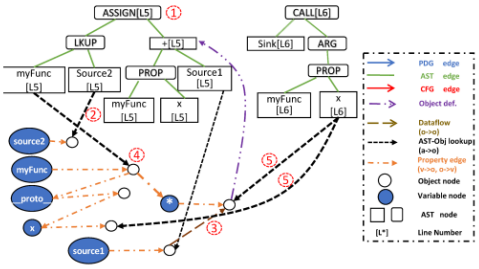

Figure 3: Nodes and Edges related to Graph Query for Internal Property Tampering Detection.

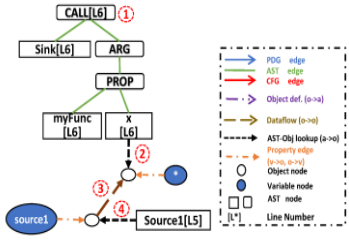

Figure 4: Nodes and Edges related to Graph Query for Taint-style Vulnerability Detection.

漏洞 1 如图 3：

1) 通过 AST 边查找具有属性调用的赋值语句找到 L5，AST 图例匹配到了 obj[prop]=value；
2) 沿着赋值目标进行查找发现变量 source2 由敌手控制；
3) 沿着赋值语句的等式右侧赋值内容进行查找发现变量 source1 也由敌手控制，自此判定赋值语句存在问题；
4) 对赋值对象的原型链进行查询，发现其原型中有属性 x；
5) 对后续属性调用语句进行查询，发现属性 x 在后续代码中被调用且直接受有问题的赋值语句影响，得到结论存在内部属性篡改漏洞。

漏洞 2 如图 4：

1) 通过 AST 查找污点汇聚点语句，定位到 L6；
2) 反向查询污点汇聚点语句的参数；
3) 参数被 source1 影响，而 source1 可被敌手指定；
4) 沿着 ODG 图查找，发现 source1 在 L5 代码中被用于赋值，定位到污点源，漏洞确认。

经过以上例子的描述，我们对于 ODG 图的工作流程有了一个简单的了解，下面给出研究针对的漏洞威胁模型。考虑 JS 级别的 Node.js 漏洞，而不考虑更低层次的漏洞，例如底层的引擎漏洞（V8 引擎）。JS 级别的漏洞分为两类，分别为应用程序级别和包级别。

应用程序级别漏洞，假设敌手对网络中的报文如 HTTP 请求或响应具有一定的控制权，即恶意客户使用应用程序。敌手的详细能力取决于应用程序内容：

1) 敌手控制的网络向有漏洞的服务器请求内容；例如 rollup-plugin-serve 插件用于在开发过程中充当本地 web 服务器，可以让开发者本地运行开发版本的应用程序，而无需每次更改代码时都重新构建并重新加载网页。使用 ODG 发现该插件存在路径遍历漏洞，有漏洞的代码使用 readFile 函数从客户提供的任意路径读取文件，不进行数据检测与清洗，

因此文件所在路径可能来自敌手。

2) 敌手控制的网络向有漏洞的客户端响应内容；以 github-growl 为例，是一个 github 的客户端通知系统，如果一个 issue 发布到了订阅的 github 仓库，那么客户端会发出 alert 弹窗。敌手可以发布带有操作系统指令的特制标题 issue，触发程序的命令注入漏洞。

依赖包级别漏洞，指依赖包或应用程序中导入的库存在的漏洞。假设敌手可以控制存在漏洞的库组件的输入。包级别漏洞不会独立出现而是必须与应用程序结合利用。一个针对包级别的漏洞会影响许多应用程序，因此必须正确对待组件的输入检测。还是以 github-growl 为例，其代码本身没有漏洞，而是引入的依赖包 growl 存在漏洞，这个漏洞同样也影响了 mqtt-growl，一种 mqtt 监视器。

6.2 对象依赖图

对象依赖图 ODG，使用图形表示法将抽象解释过程中生成的 JS 对象、变量值以及作用域表示为节点，关系表示为边。边包含 object 和 object 之间的属性、数据依赖等的对象关系以及对象定义和对象调用的 AST 关系。关于 ODG 图的组成内容详见表 1。

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 ($s \to s$)	A parent-child scope relation.
Variable lookup ($s \to v$)	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 ($o \to v$)	A property v of an object o.
Data dependency ($o \to o$)	Data dependency between two objects.
Control dependency ($a \to a$)	Control dependency between two AST nodes.
Procedures (P)	*All the ODG-related operations*
$Child^{EdgeType}_{parentNode}$	Getting the child node of *parentNode* with *EdgeType*
$AddEdge^{EdgeType}_{src \xrightarrow{p} dst}$	Adding an edge from *src* node to *dst* node with *EdgeType* and a property being either branch tags (*Br*) or a scope (*s*)
$GetEdge^{EdgeType}_{src}$	Getting all the edges start from *src* node with *EdgeType*
$AddNode^{NodeType}_{a}$	Adding a node from *a* with *NodeType*
$AddObj^{ObjType}_{a}$	Adding an object node from *a* with *ObjType* in *typeof* list and linking prototypical objects
$LkupVar^{s}_{Br}(n)$	Looking up a variable node under the scope (*s*) with branch tags (*Br*) and name *n*
$LkupObj^{s}_{Br}(n)$	Looking up object nodes under scope (*s*) with branch tags (*Br*) and name (*n*), i.e., $\{Child^{Br}_{v \xrightarrow{} o}\}$

关于 AST 类关系，主要指 Object def 和 AST-obj lookup，即对象定义和对象调用/查询赋值。前者对于每一个 AST 节点是唯一的，后者则可以有多条边。关于对象、变量与作用域之间的关系，可以重点关注$s \to s$、$s \to v$、$v \to o$、$o \to v$，作为一个根据给定变量查询其值的过程，系统首先使用$s \to v$查询当前作用域下是否有目标变量，没有则依照$s \to s$转换作用域，直至查询到对应变量，然后依照$v \to o$找到变量从属的对象，再根据$o \to v$找到对象下的对应属性值。

6.3 利用对象依赖图检测漏洞

在 ODG 图使用前，首先需要依照程序绘制 ODG。系统会依照 AST 采用抽象解释方法

解释每个 AST 节点，在解释的过程中为 ODG 生成节点与边，解释过程沿着控制流边流动。下面对于五类较为常见的代码语句进行简单的细节阐述：

1) 变量/对象属性的定义：ODGEN 试图从 ODG 中查找变量或属性，查找失败则创建新的变量和对象节点，连接需要连接的边；如果查找成功，ODGEN 重用节点，但是会创建新的边；
2) 分支语句（if/switch）：ODGEN 首先确定分支条件的值，并选择相应的分支。如果无法确定分支条件，则操作语义取决于分支敏感性。分支敏感模式下，为每个分支创建唯一的标记，并将分支标记与每个分支的抽象解释过程中创建的所有节点与边连接起来，当对语句的所有分支进行抽象解释后，ODGEN 会基于标记合并来自不同分支的节点和对象；分支不敏感的情况下顺序解释即可，如果有重复的对象与边将会覆盖。默认模式是分支敏感，如果对象数量爆炸（系统有判定函数，阈值设定为 1w），则 ODGEN 自动切换为分支不敏感；
3) 函数定义：若函数未在匿名闭包中定义，则添加变量节点，创建对象节点以及对象与变量之间的边，然后处理与原型相关的边；
4) 函数调用：调用前首先查找函数对象并创建相应的对象与控制流边；接着在调用时处理参数，更改当前作用域与点，沿着数据流边跳转 AST 节点处理直至 return 语句，处理返回对象并创建相应的数据流边。ODGEN 上下文敏感；
5) 循环/递归：广泛地抽象解释循环和递归调用，直到不再查找循环之外的新对象，设有最小最大限制。

针对已经绘制的 ODG 图的漏洞检测，被替代为图数据的查询过程，该查询过程具体由图遍历的方式实现，思路与 CPG 相近。将图遍历视作图中的点幂集映射到另一个点幂集的函数。文章中的设计将对象定义、对象调用、属性调用、原型链使用、数据流动、AST 节点匹配、控制流等基本操作给出相关的查询操作如表 2。这些基础图遍历函数支持函数结果组合、函数结果交集、函数结果并集操作，如表 3 所示将对应的漏洞类型对应为基础图遍历函数的操作组合形式，完成漏洞检测到图遍历操作的映射。

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

Traversal	Description
DEF_{obj}	Object Definition: $(a_1 = obj) \rightarrow o \rightarrow a_2$.
USE_{obj}	Object use: $(a_1 = obj) \rightarrow o \xrightarrow{reverse} a_2$.
PROP^{name}_{obj}	Property Lookup: $(a = obj) \rightarrow o_1 \rightarrow (v = name) \rightarrow o_2$.
$\text{PROTOTYPE}_{x[y]}$	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.
UNSANITIZED_{obj}	A Backward Unsanitized Dataflow traversal [15].
$\text{UNSANITIZEDSINK}_{sink}$	A Forward Unsanitized Dataflow traversal, i.e., a reverse version of UNSANITIZED_{obj}.
MATCH_{p}	This Match Traversal finds an AST node p [15].
$\text{VULASGMT}_{o1[o2]=o3}$	$\text{UNSANITIZED}_{o2} \bigcap \text{MATCH}_{o1[o2]=o3}$
$\text{VULASGMT}_{o1=o2[o3]}$	$\text{UNSANITIZED}_{o3} \bigcap \text{MATCH}_{o1=o2[o3]}$
ARG^{n}_{func}	A traversal matches a function $func$ and obtains its nth argument.
$\text{CTR}^{n}_{before/after}$	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	$\text{PROTOTYPELOOKUP}_{o1[o5]} \circ (\text{USE}_{o1} \bigcap \text{CTR}_{after}) \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
Direct	$\text{VULASGMT}_{o1=o4[o5]} \circ \text{DEF}_{o1} \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
Prototype Pollution	
__proto__	$\text{VULASGMT}_{o1=o4[o5]} \circ \text{DEF}_{o1} \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
constructor	$\text{VULASGMT}_{o4=o6[o7]} \circ \text{DEF}_{o4} \circ \text{VULASGMT}_{o1=o4[o5]} \circ \text{DEF}_{o1} \circ (\text{UNSANITIZED}_{o3} \bigcap \text{VULASGMT}_{o1[o2]=o3})$
Injection-related Vulnerabilities	
Command injection	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{Child\_process.exec}$
Arbitrary code exe.	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{eval}$
SQL injection	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{connection.query}$
Reflected XSS	$\text{UNSANITIZED} \circ \text{ARG}^{1}_{response.write}$
Stored XSS	$\text{UNSANITIZED} \circ (\text{ARG}^{1}_{connection.query} \bigcup (\text{ARG}^{1}_{connection.query} \circ \text{UNSANITIZED} \circ \text{ARG}^{1}_{response.write}))$
Improper File Access	
Path traversal	$(\text{UNSANITIZEDSINK}_{\text{PROP}^{write}_{\text{ARG}^{2}_{callback}}} \bigcap \text{CTR}_{after}) \circ (\text{UNSANITIZEDSINK}_{ReadFile} \bigcap \text{CTR}_{after}) \circ \text{PROP}^{*} \circ \text{ARG}^{1}_{callback} \circ \text{DEF AS } callback \circ (\text{ARG}^{1}_{CreateServer} \bigcup \text{ARG}^{2}_{CreateHttpServer})$
Arbitrary file write	$(\text{UNSANITIZEDSINK}_{\text{PROP}^{writeFile}_{fs}} \bigcap \text{CTR}_{after}) \circ \text{PROP}^{*} \circ \text{ARG}^{1}_{o1} \circ \text{DEF AS } o1 \circ (\text{ARG}^{1}_{CreateServer} \bigcup \text{ARG}^{2}_{CreateHttpServer})$

6.4 ODGEN 模块组成

ODGEN 是文章研究团队实现的一套原型开源系统，给出了链接地址：https://github.com/Song-Li/ODGen。这套系统支持对于给定的程序进行 ODG 绘制、存储、展

示与查询来检测程序漏洞。主要包含三个模块：

1) ODG 展示与查询：最重要的模块，ODG、AST 和 CFG 图被存储在内存中，依照 python 库 NetworkX 实现查询代码。同样也支持使用 pickle 存储于硬盘上以便后续查询。优势是使用 NetworkX 执行内存图处理，比图数据库的处理方式效率高很多；
2) JS 解析器：基于 Esprima，一个用于对 JS 代码词法语法分析的高性能解析器，生成 AST，再转换为标准 CPG 图的格式；
3) 抽象解释：使用 python 实现了定制的抽象解释器，建模了 JS 语言中流行的内置函数和功能。挑选的 AST 特征为在 Node.js 包中使用率大于 5%的。超时设置为 30s。

6.5 实验评估

文章实验评估是重点的内容，所占篇幅也较大，主要以解决以下五个问题的方式进行：

1. 最新的 Node.js 漏洞类别有哪些，ODG 是否可以建模这些漏洞？

通过了解 npm 上从 2019 年 1 月到 2020 年 9 月的 Node.js 包漏洞，包括从一些权威数据库 (MITRE 组织中央数据库和 synk.io 数据库) 中获取的信息，排除平台漏洞之后，总共检索到 330 个漏洞。经过手动检查以及对比数据库描述，总共分为 16 个类别：

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.

遵循 CPG 评估方法，手动分析描述这些漏洞时需要哪些代码表示，除了 CPG 之外，添加了 ODG 并试图了解其描述漏洞的能力。结论表明可以检测出 13 种类型，总计 302/330 种漏洞。无法检测的包含 DoS、Reg-DoS 以及不合理设计类别的漏洞。无法对于 Reg-DoS 建模是因为漏洞本质是正则表达式漏洞而非 JS 漏洞。而一些 DoS 是由于 event 循环引起，也不好建模，针对这些有专用的检测手段。输入的错误验证等不良设计不应该被统计入通用检测功能。

2. ODGEN 从大量的现实包中检测 zero-day 漏洞的能力如何？

2020 年 2 月 25 日，抓取 30 万个 npm 包，使用 ODGEN 检测相应的漏洞。目标漏洞是从上表 4 中排名靠前的内容中选择的。包含原型污染和内部属性篡改这些 JS 特有的漏洞。

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

由于时间限制，漏洞的数量实在太多，研究者手动检查并使用了周下载量超过 1000 次的应用程序与软件包。TP 表示在本地部署时可利用这个漏洞危害程序与包；FP 则表示无法生成漏洞或漏洞实则是预期功能，即假阳性。根据结果可以发现应用程序级漏洞明显少于依赖包漏洞，因为单个应用程序的使用本身少于调用依赖包。细分情况下，命令注入漏洞数量最多，因为 Node.js 经常被用作启动操作系统应用程序的客户端或服务器程序开发。原型污染作为一种新兴漏洞类型也广泛地被发现，XSS 漏洞少是因为 ODGEN 仅仅对 Node.js 框架提供的简单 web 服务器建模，而不对高级 web 框架建模，因此很多场景不适配。

3. ODGEN 的假阳率和假阴率如何？

实验设置 baseline，给出了对比的检测方法的类型、适用漏洞类型

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.

FP（假阳性）：比较两种通用静态检测器 JSJoern、JSTap-vul。在 30 万个依赖包上应用这两个工具，然后选择每周下载量前 20 的检测包进行手动验证。比较结果如下：

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%

可以看到两个 baseline 的假阳性率都很高，因为它们并不携带指向信息。而正因为缺乏这些信息，它们的检测模型会出现过拟合。由于动态和基于正则的检测方法假阳性率一般很低，因此不设置对照，一些可扩展性有限的检测器也没有实现对比。分析 ODGEN 中的假阳案例，主要原因有三：存在未建模的函数、存在无法解决的约束、预期功能。如 6.4 节所说，有些低使用率的函数和特性不支持，如果这些内容被用于数据清洗函数中，则可能会报告假阳性。执行 OS 指令的包可能会被检测为命令注入，即使这是它的正当用途。关于无法解决的约束的问题，下面给出一个例子表述：

```
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.

如上图，报告为原型污染，其在第七、八行发现两个可能污染原型链的漏洞。同时第八行的赋值可由敌手掌控，但是它恰好与被赋值方相同。

FN（假阴性）：使用 CVE 漏洞为数据。从 CVE 下载了直到 2020 年 2 月的五类漏洞历史包作为数据。可以看到对比结果中 ODGEN 真阳性最高，假阴性最低，这要归功于对 object 级数据依赖的建模。

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

假阴性主要有两个原因：存在未建模的函数、解释可能会超时。下面给出一个解释超时的例子，存在路径遍历漏洞，但是由于多次的递归调用，函数对应的节点数量超过阈值，采用混合分支策略到达不了存在漏洞的代码，而采用分支敏感策略执行则会出现节点爆炸的情况。

```
1   // curlrequest@1.0.1
2     exports.request = function(options,callback){
3       if (arguments.length === 1) {
4         exports.request.call(this, options, callback);
5       ... } // request calls itself.
6       if (options.retries) {
7         exports.request(options, function (err) {}
8       ... }  // request calls itself.
9         exports.copy(options);  // request calls copy.
10    }
11    exports.copy = function (obj) {
12      for (var i in obj) {
13        if (Array.isArray(obj[i])) {...}
14        else if (typeof obj[i] === 'object') {
15          copy[i] = obj[i] ? exports.copy(obj[i]) :
                null; // copy calls itself.
16          } else {...}
17        }
18      return copy;
19    };
```

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).

4. 系统的抽象解释模块代码覆盖率和性能开销如何？

代码覆盖范围有两个具体度量标准——语句覆盖率、函数覆盖率。前者定义执行的语句百分比；后者定义 ODGEN 分析的函数百分比，两个指标显示了 ODGEN 在分析漏洞方面的完整性。

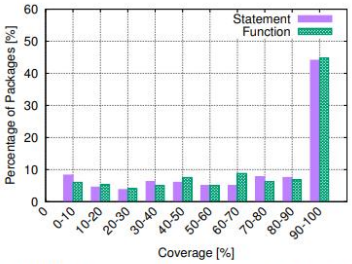

Figure 9: [RQ4-Coverage] Distribution of statement and function coverage (timeout: 30 seconds). One major reason of uncovered code is the runtime inclusion of JavaScript files depending on inputs.

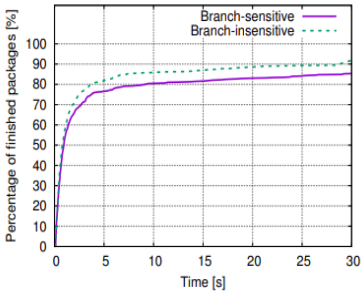

Figure 10: [RQ4-Performance] CDF graph of total execution time to finish analysis.

左图显示了在分析 500 个随机选择的 Node.js 包时语句和函数覆盖率的分布图，超时设置 30s。从 0-90 几乎均匀分布，而在 90-100%有一个飞跃的占比，其中大约 40%的包具有 100%的代码覆盖率。部分包覆盖率低的原因是：有一些代码直接从另一个包复制却没有使用，或者存在在线代码，在导出的函数中并没有调用。其次，有些包可能根据输入动态地包含一个文件，ODGEN 为静态分析方法，因此无法覆盖。其三，有些导出的函数以函数作为返回值，而调用该函数的程序包如果后续不调用这个函数，那么返回函数的代码是不执行的，也就不会被覆盖。

性能开销。如右图所示，任选 500 个包，对其运行 ODGEN，直到分析完成或超时。分支敏感状态下，ODGEN 在 30 秒内完成了 85%的包解析，分支不敏感时完成了 93%，评估结果为在大部分包上生成 ODG 可用可行。

5. 分支敏感对于 ODGEN 检测漏洞的影响如何？

ODGEN 采用的混合策略结合了准确性和扩展性两种优势，既可以对于不是非常复杂的循环和嵌套实现精确检测，又可以在对象节点数量迅速增长的情况下使用不敏感模式保证任务完成。

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

```
1 // limdu@0.9.4
2 exports.toSvmLight =
3   function(dataset, bias, binarize,
        firstFeatureNumber) {
4     var lines = "";
5     for (var i=0; i<dataset.length; ++i) {
6       var line = (i>0? "\n": "") +  // 2 objects
7       (binarize? (dataset[i].output>0? "1": "-1"):
            dataset[i].output) + // 2+1 objects
8       featureArrayToFeatureString(dataset[i].input,
            bias, firstFeatureNumber); // 54 objects
9       // 2*3*54 objects
10      lines += line;
11    }; // (2*3*54)^3=34,012,224 objects
12  ...
13  }
```

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).

6.6 实现细节

实验披露了 180 个 zero-day 漏洞。截至目前，已经修复了 12 个有漏洞的包。一些实现细节包括可以优化的点如下：

1) 支持的 JS 功能：原型实现参考了 AST 功能在 Node.js 包中使用的频率，超过 5%的包使

用的功能得到了实现。对于有未实现功能的包，ODGEN 仍可以分析，仅需要跳过不支持的部分；

2) 关于异步回调和事件：ODGEN 采用队列结构来存储注册期间的异步回调并逐个调用分析，这一执行逻辑并不能适用于全部的场景，这也是未来的研究方向之一；
3) 用于抽象解释中的循环和递归调用：ODGEN 执行 for 循环直至不再调用循环外的新对象；设置时间下限为 3，上限为 10，最大时间是为了避免死循环和过大的性能开销；最小时间是为了防止对于一些外部对象的建模不充分；
4) 动态服务文件：作为静态分析的经典限制，ODGEN 无法分析依赖于用户输入的动态服务文件。这个问题只能使用动态分析方法基于用户输入进行分析；
5) 数据检测消毒：ODGEN 在分析数据流时判定数据是否采用了清洗策略是依照人工给出的清洗函数列表进行比对，未来一个研究方向是实现清洗函数的自动化组合匹配；
6) ODGEN 是部分路径敏感的。如果是常量或可枚举，ODGEN 会计算布尔值、字符串值和整数值，对于 if 语句，如果值可以确定，ODGEN 仅抽象解释一个分支，否则并行解释两个分支。

七、 文章结论

文章利用抽象解释的手段给出了一种新型的图结构——对象依赖图 ODG。ODG 支持图数据查询，以此来检测和挖掘 Node.js 运行环境中的应用程序级与依赖包级别的漏洞，这其中尤以对象之间的依赖联系导致的漏洞为典型，如原型污染和内部属性篡改漏洞等。文章实现了一个开源系统 ODGEN，用于解析 JS 程序，使用抽象解释方法构建相应的 ODG 并存储用于图数据查询检测漏洞。ODGEN 系统经过了充分且复杂的实验得到了有利的实验结果。

八、 备注

通读整篇文章，我首先思考了这篇文章的贡献与可以入选顶会的原因。总结起来首要的条件就是文章所进行的工作是具有首创性的，在 JS 语言上没有合适的通配漏洞检测方法的情况下研究团队采用沿用并针对语言特点修改在 C、C++、PHP 等语言上表现出色的通配漏洞检测方法来首次实现了 Node.js 环境上的漏洞检测；其次这篇文章针对设计给出了开源的实现代码仓库，并对于设计的性能给出了极其细致且系统的实验结果来验证了设计和实现系统的可用性与实用性；最后难能可贵的一点是对于每一项实验中的结果，即使做到了与现有方案中对比的最优，研究团队依旧针对其出现的问题进行细致的原因分析，提供不足之处方便他人进行进一步的优化。

对于文章的内容优化，个人认为首先在文章中提到多次的问题例如对于非热点型的内置函数与功能的建模、对于 XSS 跨站脚本类漏洞的高级 web 框架的兼容性提升等都是研究的后续发展方向，事实上主要应对的就是在详尽的场景测试当中系统所暴露出的问题。另外文章提出的方案为纯粹的静态检测方法，是否可以将系统与 JS 的动态检测功能进行集成以实现一个能力更强的漏洞检测系统也是研究方向之一。