Detecting Multi-file Vulnerabilities Using Code Property Graphs

Guilherme Gonçalves
INESC-ID
Lisbon, Portugal
guilherme.silva.goncalves@tecnico.ulisboa.pt

José Santos
INESC-ID
Lisbon, Portugal
jose.fragoso@tecnico.ulisboa.pt

Pedro Adão
INESC-ID
Lisbon, Portugal
pedro.adao@tecnico.ulisboa.pt

***Abstract*—The web is essential to daily life, with *JavaScript* being fundamental to web development. *Node.js* [1] extends its capabilities beyond browsers, enabling scalable applications. However, its dynamic nature and the large repository of potentially vulnerable packages in *NPM* [2] create significant security risks.**

This work focuses on improving the vulnerability detection capabilities of *Graph.js* by addressing its shortcomings in inter-procedural analysis and external module processing, which often lead to false positives. Our contributions include: (1) the development of an Extended Multi-version Dependency Graph (EMDG) that enhances inter-procedural analysis and enables multi-file reasoning by adding call and return nodes, argument and return edges, and merging EMDGs from the application's modules into a single graph; (2) the creation of three new detection algorithms—Top-Down, Bottom-Up with Pre-processing, and Bottom-Up Greedy—that effectively identify vulnerabilities within the graph; and (3) a new attacker-controlled object definition that uncovers previously missed vulnerabilities.

We evaluated our approach using a combined dataset from *VulcaN* [3] and *SecBench* [4], demonstrating that the Bottom-Up Greedy approach achieves 82% recall and 85% precision, resulting in a 15% reduction in false positives compared to *Graph.js*. Additionally, testing on a dataset of real-world *NPM* packages revealed 83% fewer reported vulnerabilities and an average speed improvement of 3 seconds. With the new definition, the number of reported vulnerabilities increase by 30%, yet the estimated precision remains the same.

***Index Terms*—Static Analysis, Graph Queries, Vulnerability Detection, Node.js**

I. Introduction

The web is integral to daily life, with JavaScript playing a key role in web development. Node.js [1] extends JavaScript's utility beyond browsers, enabling scalable web applications through an event-driven architecture and non-blocking I/O model. However, Node.js has security issues due to JavaScript's dynamic nature and NPM's [2] vast repository of potentially vulnerable packages.

Manual detection of vulnerabilities is challenging, leading to the adoption of static analysis tools like Graph.js and ODGen, which use graph-based approaches. Yet these tools have limitations. Particularly, Graph.js doesn't process a file's external modules and struggles with inter-procedural analysis, leading to false positives.

This work aims to enhance Graph.js by improving its detection capabilities. Specifically, the goals of this work are as follows:

1. ***Extended MDGs***: We enhanced the Graph Constructor Module to create an Extended Multi-version Dependency Graph (EMDG), improving inter-procedural analysis by introducing call and return nodes, as well as argument and return edges. The EMDG unifies graphs from multiple modules into interconnected sub-graphs.
2. ***New Detection Algorithms***: Three new algorithms were developed: Top-Down, Bottom-Up with Pre-processing, and Bottom-Up Greedy, each designed to find paths from sources to sensitive sinks with different graph navigation methods.
3. ***Attacker-Controlled Object Definition***: To further reduce false positives, we proposed a new attacker-controlled object definition, allowing the tool to identify previously missed vulnerabilities by considering overlooked taint introduction methods.
4. ***Evaluation on Combined Datasets***: Evaluating on VulcaN [3] and SecBench [4], the Bottom-Up approaches outperformed the Top-Down, with the Bottom-Up Greedy achieving 82% recall and 85% precision, reporting 15% fewer false positives than Graph.js and showing improvements over ODGen.
5. ***Evaluation on Real-World Dataset***: Using a dataset of real-world NPM packages, the Bottom-Up Greedy approach reported 83% fewer vulnerabilities and was 3 seconds faster on average compared to Graph.js, with estimated precision and recall of 83% and 81%, respectively. With the new definition, the number of reported vulnerabilities increase by 30%, yet the estimated precision remains the same.

This document is organized as follows: Section II covers the necessary background. Section III details the proposed changes to the graph generation algorithm and the new detection algorithms. Section IV evaluates these changes. Section V reviews related research. Finally, Section VI concludes with a summary and concluding remarks. We detail all the algorithms mentioned in this document in Appendix A.

II. Overview

This section provides the necessary background for understanding our work. First, in Section II-A, we overview the Node.js security model, including some of the most common vulnerabilities present in Node.js applications. Finally, we

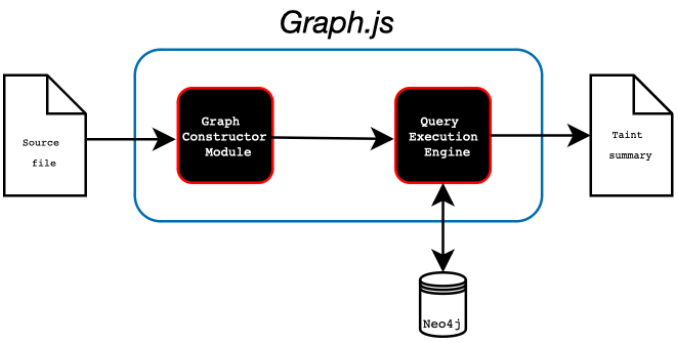

Fig. 1: *Graph.js Architecture Overview*

introduce Graph.js in Section II-B. Understanding Graph.js is crucial here, as our work builds upon it to enhance its functionality and address security concerns.

A. Node.js Security

The Node.js environment, while powerful and flexible, also introduces various vulnerabilities in applications. This section presents the Node.js security model, highlighting common vulnerabilities found in Node.js applications.

1) Taint-Style Vulnerabilities: Taint-style vulnerabilities are prevalent in Node.js applications, characterized by the flow of untrusted data (sources) to sensitive functions (sinks).

- ***Sources:*** Points where untrusted data enters the system, such as web forms, request bodies, and module parameters.
- ***Sinks:*** Functions that trigger security-sensitive behavior, like `eval`, `child_process.exec`, and `fs.readFile`.

Injection vulnerabilities are a subset of taint-style vulnerabilities, where attacker-controlled data is executed by the application. Examples include:

- ***Code Injection:*** Passing untrusted data to `eval` or `Function`, leading to arbitrary code execution.
- ***OS Command Injection:*** Influencing OS commands via `child_process.exec` or `child_process.spawn`.
- ***Path Traversal:*** Accessing restricted files via `fs.readFile` or `fs.createReadStream`.

2) Other Types of Vulnerabilities: Node.js applications can also exhibit other common vulnerabilities:

- ***Prototype Pollution:*** Altering built-in JavaScript properties, leading to severe consequences like remote code execution.
- ***Cross-Site Request Forgery (CSRF):*** Trick authenticated users into executing unwanted actions on a web application.
- ***Denial of Service (DoS):*** Disrupt a website or service, making it unavailable to users.

B. Graph.js

In this section, we present *Graph.js* [5], [6], whose architecture can be found in Figure 1. First, we present the running example in Section II-B1. Then, we discuss its limitations in Section II-B2.

```
const bar = require('./bar.js');
function f(x,y){
   if(x > 0){
       var a = bar.f("foo",y);
       eval(a);
   }
   else{
        var b = bar.g(x,0);
        eval(b);
   }
}
module.exports = f;
```

Listing 1: *Main module*

```
const foo = 4;
function f(a,b){
    eval(a);
    return b;
}
function g(a,b){
    eval(a);
    return 0;
}
module.exports = {f, g};
```

Listing 2: *Bar module*

1) Running Example: Listing 1 and Listing 2 depict the running example used in this section. The example involves two modules: `Main` and `Bar`. The `Main` module includes `Bar` and calls either `f` or `g` based on whether `x` is greater than `0`. For `f`, it passes `"foo"` and `y`; for `g`, it passes `x` and `0`. In both cases, `eval` is invoked with the function's return value. The `Bar` module defines `f` and `g`, both of which evaluate `a` using `eval`. `f` returns `b`, while `g` returns `0`.

2) Limitations: Although Graph.js can accurately detect vulnerabilities, it also has the following limitations:

- ***File-by-File Analysis***: Graph.js analyzes modules separately and assumes all function parameters are unsafe, leading to false positives. In the running example, when calling `Bar.f` with `"foo"` as the argument `a`, it results in `eval("foo")`, which is safe. However, if the module is analyzed on its own, this call to `eval` is incorrectly flagged as vulnerable because there is a path from the parameter `a` to the sink.
- ***Incorrect Inter-Procedural Analysis***: Graph.js marks a call's return value as tainted if any argument is tainted, which can cause false positives. In the running example, the return value of the call to `Bar.g` is marked as tainted, incorrectly flagging a vulnerability. This is a false positive because the tainted value is `0`, which poses no real security risk.

III. Multi-file Vulnerability Detection

In this section, we explain the proposed improvements to Graph.js's detection capabilities. First, we show how to build the extended MDGs (EMDGs) in Section III-A. Then,

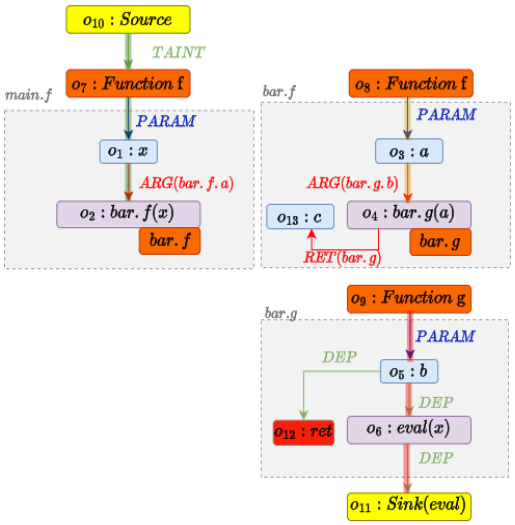

Fig. 2: *Extended MDG example*

in Section III-B, we present three methods for detecting vulnerabilities in the extended MDGs.

A. Multi-File Graph Construction

To address the limitations outlined in Section II-B2, we propose two main changes to the Graph Constructor module: adding new nodes and edges to improve inter-procedure analysis, and implementing a new graph generation algorithm to manage file dependencies. The following sections will detail each of these changes.

1) Extended Graph: Figure 2 displays an EMDG. It details its the nodes and edges and will serve as a reference throughout this section.

a) EMDG nodes: Starting with its nodes, an EMDG has the following nodes:

- ***Tainted Source Nodes*** (yellow): Represent unsafe data whose safety cannot be assured, often from user input (e.g., o_{10} in Figure 2).
- ***Unsafe Sink Nodes*** (yellow): Represent calls to risky functions/APIs (e.g., o_{11} in Figure 2).
- ***Value Nodes*** (blue): Represent objects and primitive values generated during execution (e.g., o_1, o_3, o_5).
- ***Call Nodes*** (purple): Represent function calls, including the called function's identifier (e.g., o_2, o_4, o_6).
- ***Function Nodes*** (orange): Represent declared functions (e.g., o_7, o_8, o_9).
- ***Return Value Nodes*** (red):Represent a function's return value (e.g., o_{12} for `Bar.g`).

b) EMDG edges: Now shifting the focus to the edges, an EMDG has the following edges:

- ***Property Edges***: Represent an object's structure. The edge $n_1 \xrightarrow{PROP(p)} n_2$ indicates that n_1 has a property p, with value n_2. For instance, the code snippet `n1 = "p": n2`, creates a node to represent the sub-object `p`, connecting it to `n1` through a property edge.
- ***New Version Edges***: Represent updates to objects. The edge $n_1 \xrightarrow{\texttt{NV(p)}} n_2$ shows that n_2 is a new version of n_1 after updating property p. For instance, the code snippet `a.x = 2`, creates a new version of a, a', connecting a and a' with a new version edge.
- ***Dependency Edges*** (green): Represent data relationships between nodes. The edge $n_1 \xrightarrow{\texttt{DEP}} n_2$ shows that n_2 depends on n_1. For instance, in Figure 2, the edge $o_6 \xrightarrow{\texttt{DEP}} o_7$.
- ***Parameter Edges***: Connect functions to their parameters. The edge $n_1 \xrightarrow{\texttt{PARAM}} n_2$ shows n_2 is a parameter of n_1. For instance, in Figure 2, the edge $o_{17} \xrightarrow{\texttt{PARAM}} o_1$.
- ***Taint Edges***: Link taint sources to functions. The edge `TAINT_SOURCE` $\xrightarrow{\texttt{TAINT}} n_2$ indicates that n_2 is potentially controlled by the attacker. For instance, in Figure 2, the edge $o_{21} \xrightarrow{TAINT} o_{17}$.
- ***Return Edges***: Connect function calls to their return values. The edge $n_1 \xrightarrow{\texttt{RET(f)}} n_2$ shows n_2 is the return value of `f` at callsite n_1. For instance, in Figure 2, the edge $o_{13} \xrightarrow{\texttt{RET(bar.g)}} o_4$.
- ***Argument Edges***: Connect function arguments to calls. The edge $n_1 \xrightarrow{\texttt{ARG(f.x)}} n_2$ indicates n_2 has a parameter x receiving the value from n_1. For instance, in Figure 2, the edge $o_2 \xrightarrow{\texttt{ARG(bar.f.b)}} o_{11}$.

2) Multi-file Algorithm: We enhanced inter-procedural analysis with new nodes and edges, but Graph.js still struggles with file dependencies. To address this, we introduce the *Graph Generation Algorithm* in Algorithm 1. This algorithm requires two inputs: a Directed Acyclic Graph (DAG) of module dependencies and a summary of the functions exported by each module.

We generate the DAG using the *Dependency Tree* NPM package, which analyzes all dependencies (regardless of being used or not) starting from a specified entry point, excluding Node.js built-in modules.

To summarize exported functions, we track assignments to `module.exports`, mapping each exported function to its original name. If an object is exported, we recursively map its function properties. This process is detailed in Algorithms 4 and 5.

The algorithm works by first generating the module dependency DAG, then processing it from sinks (modules without dependencies) to sources (modules without dependents). For each module, we build its EMDG and export function summary. When a module calls an external function, we integrate its graph using these summaries, resulting in a unified EMDG for the entire application.

Figure 2 shows a multi-file EMDG with two modules: `Bar` and `Main`. To create it, we first generate the dependency DAG: `Main` $\rightarrow$ `Bar`. We start by processing the `Bar` module since it has no dependencies, generating its EMDG and summarizing its exports. Then, we process the `Main` module. When a call to `Bar` is found in `Main`, we use the previously generated summary for `Bar` to include the calling function's graph.

Traversal	Description	Pattern
$\texttt{TaintedPath}^{\texttt{s}}_{\texttt{e}}$	Sequence of 0 or more edges connecting node `s` to node `e`. If `e` is not specified, return all distinct paths that start in `s`. If `s==e`, return `e`.	$\texttt{s} \xrightarrow{(DEP/NV/PROP/ARG)^+} \texttt{e}$
$\texttt{Call}^{\texttt{arg}}_{\texttt{f,p}}$	Match the argument edge connecting `arg` to a call node representing a call to function `f` on the parameter `p`, returning both `f` and `p`. If `f` and `p` are not specified, match all call nodes connected to `arg` and return all distinct pairs of functions and parameters where `n` is used as an argument.	$\texttt{arg} \xrightarrow{ARG(p)} \texttt{call(f)}$
$\texttt{Param}^{\texttt{p}}_{\texttt{f}}$	Matches the node representing function `f` and returns the node representing its parameter `p`.	-

TABLE I: *Base graph traversals*

Name	Path
Top-Down Taint Query	$(\texttt{TaintPath}^{\texttt{start}}_{\texttt{sink/ret}}) \cup (\texttt{TaintPath}^{\texttt{start}}_{\texttt{arg}} \circ \texttt{Call}^{\texttt{arg}})$
Bottom-Up Taint Query	$\texttt{Param}^{\texttt{p}}_{\texttt{f}} \circ \texttt{TaintPath}^{\texttt{p}}_{\texttt{sink}}$
Call Graph Query	$\texttt{Param}^{\texttt{p}}_{\texttt{f}} \circ \texttt{TaintPath}^{\texttt{p}}_{\texttt{arg}} \circ \texttt{Call}^{\texttt{arg}}_{\texttt{g,q}}$

TABLE II: *Algorithms' Queries*

B. Vulnerability Detection

With the changes described earlier, Graph.js can now accurately model vulnerabilities. To detect these vulnerabilities, we can use either a *Top-Down* or *Bottom-Up* approach to trace paths from the *Taint Source* node to a sensitive sink. Section III-B1 covers the Top-Down approach, while Section III-B2 details the Bottom-Up approach. Finally, Section III-B3 discusses the soundness of these methods.

1) Top-Down Vulnerability Detection: In this section, we explain how to detect vulnerabilities in EMDGs using a *Top-Down* approach. This approach traces call chains from callers to callees, starting at the *Taint Source* node and moving towards sensitive sinks. To implement this, we use the *Top-Down Taint Query* and Algorithm 8.

a) Top-Down Taint-Query: The Top-Down Taint Query, shown in Table II, identifies paths from a starting node (e.g., a Taint Source node) to a sensitive sink, a Return node, or a function call. It uses two basic traversals from Table I: *TaintPath* and *Call*. The *TaintPath* traversal connects the start node (*s*) to a sink or return node (*end*). The *Call* traversal finds calls to functions (*f*) on a parameter (*p*) and the corresponding argument (*arg*).

To find taint paths that reach a sensitive sink or a Return node, we simply use the *TaintPath* traversal. Additionally, we find taint paths that reach function calls by combining the *TaintPath* with the *Call* traversal (which finds function calls, its arguments and corresponding parameters). For example, in Figure 2, this query returns the green path, amongst others.

b) Top-Down Algorithm: The Top-Down Algorithm (Algorithm 8) connects call chains from callers to callees, behaving differently based on where the query stops. If it stops at a call node, it executes a Top-Down Taint query on the called function's sub-graph. If it stops at a return node or sensitive sink, it saves the path for reporting. The algorithm uses two lists: *results* (storing identified vulnerable paths) and *work_list* (holding incomplete paths). Initially, *results* is empty, and *work_list* starts with the Taint Source node. For example, to detect the vulnerability in Figure 2, we follow these steps:

1. We start by calling `Find_Taint_Paths` with `results` as an empty list and `work_list` initialized with $[[o_8]]$, where o_8 is the Taint Source node.
2. The algorithm pops the first path from `work_list`. Since the last node is `TAINT_SOURCE`, it runs the Top-Down Taint query, identifying the green path and updating `work_list` to $[[o_{10}, o_7, o_1, o_2]]$ (lines 30-34).
3. It pops the first path from `work_list`. The last node (o_2) is a call node, so `func` is `bar.f` and `param` is o_3. It then recursively calls `Find_Taint_Paths([],[[`o_3`]])` (lines 12-16):
   - 3.1. The only path in `work_list` ends at o_3, a parameter. Thus, it executes the Top-Down Taint query, identifying the yellow path and updating `work_list` to $[[o_3, o_4]]$ (lines 30-34).
   - 3.2. The path in `work_list` ends at a call node (o_4). Therefore, it recursively calls `Find_Taint_Paths([],[[`o_5`]])` (lines 12-16):
     - 3.2.1. The only path in `work_list` ends at o_5, a parameter. Consequently, it executes the Top-Down Taint query, identifying the red path and updating `work_list` to $[[o_5, o_6, o_{11}]]$ (lines 30-34).
     - 3.2.2. The path in `work_list` ends at a sensitive sink (o_{11}). Thus, it adds this path to `results` and returns `results`, as there are no paths left to analyze (lines 8-10).
   - 3.3. It append the current path with the returned path and adds it to `results`, updating it to $[[o_3, o_4, o_5, o_6, o_{11}]]$ (lines 17-28). It returns `results`, as there are no paths left to analyze (lines 1 and 2).
4. The algorithm now appends the paths (lines 17-28). Finally, it returns the `results` list with the vulnerable path $[[o_{10}, o_7, o_1, o_2, o_3, o_4, o_5, o_6, o_{11}]]$, as there are no paths left to analyze (lines 1 and 2).

2) Bottom-Up Vulnerability Detection: Unlike the Top-Down approach, the Bottom-Up approach traces call chains in reverse, starting from the sinks and working back to the

```
1  function g(x) {
2      let o = {}
3      o.foo = 33;
4      f(x, o);
5      eval(o.foo);
6  }
7  function f(y,z){
8     z.foo = y;
9  }
10 module.exports = g
```

Listing 3: *Overlooked Vulnerability Example*

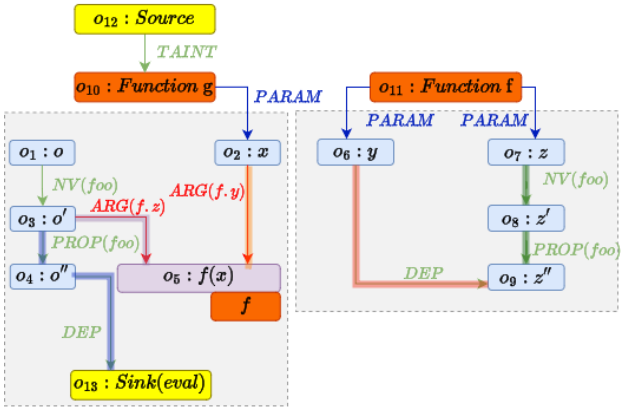

Fig. 3: *Corresponding MDG of Listing 3*

parameters of exported functions (i.e., functions connected to the Taint Source node). In this section, we present two algorithms that use this approach, along with the queries that support their execution.

a) Bottom-Up Queries: The Bottom-Up algorithms discussed in the following sections require the following queries:

- ***Bottom-Up Taint Query***: The Bottom-Up Taint query traces data flows from a sink to its origin. First, we use the *Param* traversal to identify a function's parameter. Then, we chain it with the *TaintPath* traversal, to trace taint flows from this parameter to a sensitive sink. For example, in Figure 2, this query returns the red path.
- ***Call Graph Query***: The Call Graph query connects callers to callees. This query first uses *Param* traversal to identify a function parameter, then chains the *TaintPath* and the *Call* traversals to follow this parameter to a function cal, identifying the called function's parameter (q). For example, in Figure 2, this query that o_3 is an argument of the parameter `b` of `Bar`'s function `g`

b) Bottom-Up Algorithm with Pre-Processing: The Bottom-Up algorithm with Pre-Processing first computes the transposed call graph using the Call Graph query. Next, it uses the Bottom-Up Taint query to identify potential vulnerabilities by tracing objects that reach a sink. Finally, it traverses the call chains in a bottom-up manner (from sinks to sources) to confirm these vulnerabilities using the transposed call graph and Algorithm 7. For instance, we detect the vulnerability in Figure 2 by following these steps:

1. We first construct the transpose call graph (`CGT`), using the Call Graph Query. `CGT` becomes $o_5 \rightarrow o_3 \rightarrow o_1$
2. We then run the Bottom-Up Taint query to identify the red path. Thus, we call `Confirm_Vuln(`o_5`, CGT)`, where o_5 represents the function `bar.g`'s parameter `b` and that parameter reaches a sensitive sink.
3. The algorithm initializes the `stack` list with `[`o_5`]` (line 1). It pops o_5 (parameter `b` of `bar.g`). Since `b` is not a parameter of an exported function, it adds the parameters that reach it from the transposed call graph. Consequently, it updates `stack` to `[`o_3`]`, where o_3 represents `bar.f`'s parameter `a` (lines 3-8).
4. It repeats for `a` (o_3). It updates `stack` to `[`o_1`]`, where o_1 represents `main.f`'s parameter `x` (lines 3-8).
5. Since `x` (o_1) is a parameter of the exported function `main.f`, the algorithm reports the vulnerability and ends its execution (line 5).

c) Bottom-Up Greedy Algorithm: The Bottom-Up Greedy algorithm confirms vulnerabilities by connecting only the necessary paths, caching them to avoid repetition. We start by identifying parameters that reach a sink using the Bottom-Up Taint query. Then, we use Algorithm 6 to traverse the call chains in a bottom-up manner (from sinks to sources) and confirm the vulnerability. The Call Graph query builds the transpose call graph as needed. To detect a vulnerability in the graph of Figure 2, follow these steps:

1. We start by running the Bottom-Up Taint query to find the red path in the graph. Then, we call `Confirm_Vuln(`o_9`,` o_5`)`, where o_9 represents `bar.g` and o_5 represents its parameter `b` (which reaches a sensitive sink).
2. Since o_9 (`bar.g`) is not exported, the algorithm uses the Call Graph query to identify parameters reaching o_5. This yields the yellow path. Consequently, it calls `Confirm_Vuln(`o_8`,` o_3`)`, where o_8 represents `bar.f` and o_3 represents its parameter `a` (lines 4-7).
3. It repeats for o_8 (`bar.f`) and o_3 (its parameter `a`), finding the green path. Therefore, it calls `Confirm_Vuln(`o_7`,` o_1`)`, where o_7 represents `main.f` and o_1 represents its parameter `x` (lines 4-7).
4. Finally, it reports the vulnerability since o_7 (`main.f`) is exported, and ends its execution (lines 1-2).

3) Soudness Issues: The algorithms discussed earlier detect vulnerabilities effectively with minimal false positives but are not sound, as they miss certain vulnerabilities. This section outlines how to make vulnerability detection as close to sound as possible.

a) Motivating Example: Listing 3 shows a scenario where our analysis fails to detect a vulnerability. Function `g`, which is exported, creates an object `o` with `foo` set to 33, and

Dataset	Total	Excluded				Manually Added	Considered
		Unavailable	Incorrect Annotations	Duplicated	Out-of-Scope		
VulcaN	236	10	7	0	0	59	278
SecBench	601	10	71	38	98	150	534
Total	837	20	78	38	98	209	812

TABLE III: *Summary of the vulnerabilities considered in each dataset*

calls function `f` with `x` and `o`. Inside `f`, `y` assigns its value to `z.foo`, thus modifying `o.foo` to `x`'s value. Function `g` then calls `eval(o.foo)`.

The vulnerability arises from `eval(o.foo)` in `g`. Since `f` sets `z.foo` (i.e., `o.foo`) to `y` (which equals `x`), an attacker can manipulate `x` to control `o.foo`, leading to code injection. The analysis fails to link the assignment `z.foo = y` in `f` with `eval(o.foo)` in `g`, missing the taint propagation from `x` to `o.foo` through `f`.

b) Unsound Attacker-Controlled Object Definition: As shown in the previous section, earlier algorithms fail to detect the vulnerability in the example because they rely on the definition of an attacker-controlled object in Algorithm 3. This definition states that an attacker controls the objects that are:

- Directly connected to parameters of exported functions, through new version, dependency or property edges (line 3).
- Reachable from parameters of exported functions through function calls (lines 7-8).

However, this definition fails in the example shown in Figure 3, where the parameter `x` of function `g` (o_2) does not have a direct or indirect path to `o.foo` (o_4), despite taint propagation through the call to `f(o,x)`.

c) New Attacker-Controlled Object Definition: To detect vulnerabilities in the motivating example from Listing 3, we propose replacing the function `Reaches` with `Reaches2.0` (Algorithm 2) in Algorithm 3. In `Reaches2.0`, `n1` reaches `n2` if they are directly or indirectly connected, or if taint propagation occurs within a function call.

In the example, the parameter `x` of function `g` (o_2) taints `o.foo` (o_4). In the call to `Reaches2.0`(o_2, o_4), line 1 of the algorithm is not satisfied (i.e, they are not directly connected), but line 2 holds with the following setup: o_2 as `n1` and `n1'`, o_4 as `n2`, o_3 as `n2'`, `y` as parameter `p` (node o_6), and `z` as parameter `q` (node o_7). Based on this, we verify the following conditions:

1. ***Reaches2.0(n2',n2)*** (line 3): We verify this condition because node o_3 (variable `o'`) connects to o_4 (`o.foo`) through a property edge (blue path in the graph).
2. ***n2'*** $\xrightarrow{ARG(q)}$ ***call*** (line 4): We verify this condition because node o_3 (`n2'`) connects to the call to `f` (o_5) (purple path in the graph).
3. ***Reaches2.0(n1,n1')*** (line 5): We verify this condition because o_2 corresponds to both `n1` and `n1'`.
4. ***n1'*** $\xrightarrow{ARG(p)}$ ***call*** (line 6): We verify this condition because node o_2 (`n1'`) connects to the same call node found in 2. (o_5) (orange path in the graph).
5. ***Reaches2.0(node(q),q')*** (line 7): We verify this condition because node o_7, representing the variable `z`, connects to o_9 through through new version and property edges (green path in the graph).
6. ***Reaches2.0(node(p),q')*** (line 8): We verify this condition because node o_6, representing the variable `y`, connects to o_9 (`q'`) through dependency edges (red path in the graph).
7. ***PropsTraversed(n2',n2) == PropsTraversed(node(q),q')*** (line 9): We verify this condition because the paths from `n2'` to `n2` (blue path) and from `node(q)` to `q'` (green path) follow the same sequence of property edges.

Consequently, this analysis flags o_4 as tainted by o_2 (the parameter `x`). Since `x` is a parameter of the exported function `g`, we report the vulnerability.

IV. Evaluation

In this section, we evaluate the effectiveness of our solution. Specifically, we aim to answer the following research questions:

- ***RQ1: Which of our three proposed algorithms is most effective for vulnerability detection?***
- ***RQ2: How much does our best algorithm improve detection over state-of-the-art tools?***
- ***RQ3: What is the impact of our new attacker-controlled object definition on the detection?***

A. Experimental Setup

To address our research questions, we need two datasets:

- ***Collected Dataset***: Consists of 32,137 popular real-world NPM packages retrieved from the NPM repository in September 2023. A package is considered popular if it had over 2,000 weekly downloads, according to Snyk's guidelines.
- ***Combined Dataset (VulcaN + SecBench)***: A ground truth dataset combining vulnerabilities from the VulcaN [3] and SecBench [4] datasets. We identified additional vulnerabilities and exploits, increasing the total to 812 vulnerabilities. The vulnerabilities picked from each dataset are summarise in Table III and the following:
  - ***VulcaN dataset***: Includes 957 npm package versions with vulnerabilities like Path Traversal (CWE-22),

CWE	Top-Down			Bottom-Up Pre-Processing			Bottom-Up Greedy		
	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1
CWE-22	0.95	0.84	0.89	0.97	0.87	0.92	0.95	0.86	0.90
CWE-78	0.94	0.95	0.94	0.93	0.97	0.95	0.94	0.95	0.94
CWE-94	0.77	0.80	0.78	0.87	0.84	0.85	0.77	0.82	0.79
CWE-1321	0.46	0.65	0.54	0.54	0.70	0.61	0.56	0.70	0.62
Total	0.80	0.84	0.82	0.82	0.84	0.83	0.82	0.85	0.84

TABLE IV: *Detection results in the Combined dataset*

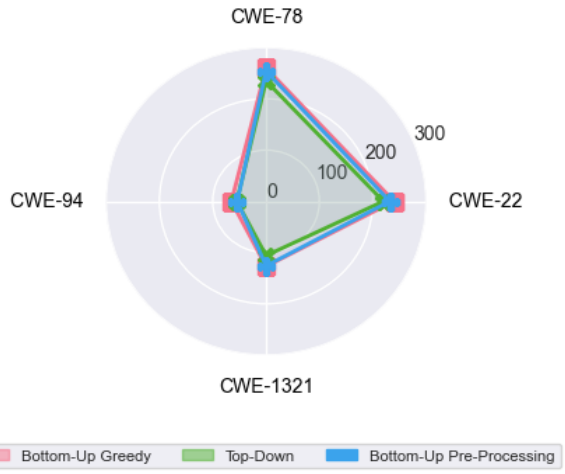

Fig. 4: *Vulnerable packages reported in the Collected dataset*

OS Command Injection (CWE-78), and Code Injection (CWE-94). We selected 174 packages containing Graph.js-targeted vulnerabilities, totaling 236 vulnerabilities. Excluded 17 due to incorrect annotations or external package issues.

– ***SecBench dataset***: Contains 600 vulnerable packages, including Regular Expression Denial of Service (CWE-1333). Excluded 217 vulnerabilities: 98 out-of-scope ReDoS, 71 incorrectly annotated, 38 already in VulcaN, and others unavailable or in TypeScript. Resulted in 384 vulnerabilities.

The evaluation also includes a comparison between the current version of Graph.js and ODGen, selected for its similar detection approach and favorable trade-off between effectiveness and precision as identified by Brito *et al.* [3]. The testbed consisted of a single 64-bit Ubuntu 22.04.3 server with 64GB of RAM and 2x Intel(R) Xeon(R) Gold 5320 2.2GHz CPUs, with a total analysis timeout set to five minutes.

B. RQ1: Which of Our Three Proposed Algorithms Is Most Effective for Vulnerability Detection?

To address this research question, we evaluated all algorithms against every package in the Combined dataset. A *False Positive* is any vulnerability report not annotated in the dataset, and a *True Positive* matches the annotations. Precision is computed as *TP/(TP + FP)*, recall as *TP/(TP + FN)*, and the F1-score as *(2 × Precision × Recall)/(Precision + Recall)*. The results are summarised in Table IV. Figure 4 shows that the number of vulnerabilities by each algorithm.

The Greedy Bottom-Up algorithm achieved the best balance with 82% recall and 85% precision. The Bottom-Up with pre-processing followed closely with 82% recall and 84% precision. The Top-Down algorithm had 80% recall and 84% precision, demonstrating its effectiveness despite slightly lower recall.

a) False Positive and Negative Analysis: Despite good precision and recall, all algorithms suffer from false positives and negatives. False positives occur due to incorrect labeling of the `require` function as a sink, lack of detailed insights into Node.js modules, and incorrect flags for recursive object assignments in prototype pollution. False negatives arise from incomplete support for JavaScript features like `arguments` and `this` keywords, and prototype pollution patterns involving third-party NPM packages and sources using the `arguments` keyword.

C. RQ2: How Much Does Our Best Algorithm Improve Detection Over State-of-the-Art Tools?

To address this research question, we evaluated Graph.js and ODGen on the Combined dataset, using the same methodology as before. Additionally, we ran the Bottom-Up Greedy algorithm on 5003 packages flagged as vulnerable by Graph.js, comparing the number of reported vulnerabilities and average analysis time. We did this, because we expect the number of reported vulnerabilities to decrease significantly in this dataset. On the Collected dataset, Graph.js analyzed packages file-by-file, while Bottom-Up Greedy analyzed them file-by-file and by entry points in the *main* attribute of *package.json* or

CWE	Total	ODGen					Bottom-Up Greedy				
		TP	FP	Recall	Precision	F1	TP	FP	Recall	Precision	F1
CWE-22	244	131	8	0.54	0.94	0.69	231	38	0.95	0.86	0.90
CWE-78	269	151	29	0.56	0.84	0.67	254	13	0.94	0.95	0.94
CWE-94	71	24	113	0.34	0.18	0.24	55	12	0.77	0.82	0.79
CWE-1321	228	37	21	0.16	0.64	0.26	127	57	0.56	0.70	0.62
Total	812	343	171	0.42	0.67	0.52	667	120	0.82	0.85	0.84

TABLE V: *Comparison of ODGen and Bottom-Up greedy approach*

CWE	Total	Graph.js					Bottom-Up Greedy				
		TP	FP	Recall	Precision	F1	TP	FP	Recall	Precision	F1
CWE-22	244	235	47	0.96	0.83	0.89	231	38	0.95	0.86	0.90
CWE-78	269	255	13	0.95	0.95	0.95	254	13	0.94	0.95	0.94
CWE-94	71	61	21	0.86	0.74	0.80	55	12	0.77	0.82	0.79
CWE-1321	228	132	60	0.58	0.63	0.63	127	57	0.56	0.70	0.62
Total	812	683	141	0.83	0.83	0.83	667	120	0.82	0.85	0.84

TABLE VI: *Comparison of Graph.js and Bottom-Up greedy approach*

Tool	Vulnerabilities	Vulnerable Packages	Avg Analysis Time
Graph.js	14186	5003	15.110s
Bottom-Up Greedy (file-by-file)	13894	4571	15.799s
Bottom-Up Greedy (multi-file)	2327	1255	12.323s

TABLE VII: *Results of the evaluation on the Collected dataset*

defaulting to *index.js*. We did not run ODGen on this dataset due to its long runtime.

a) Comparison with ODGen: Table V shows that the Bottom-Up Greedy algorithm improved recall by 40% and precision by 18% compared to ODGen, with 30% fewer false positives.

b) Comparison with Graph.js: Table VI shows that the Bottom-Up Greedy algorithm improves precision by 2% and reduces false positives by 15% compared to Graph.js, though it has a 1% decrease in recall. Additionally, as detailed in Table VII, the Bottom-Up Greedy algorithm reduced reported vulnerabilities in the Collected dataset by about 300 in file-by-file analysis and by 83% in multi-file analysis. It also performed 3 seconds faster on average, highlighting the benefits of combining inter-procedural queries with multi-file analysis.

c) Analysis of Collected Dataset Vulnerabilities: To ensure that the reduction in reported vulnerabilities wasn't due to missed true positives, we randomly sampled and manually reviewed 40 vulnerabilities detected uniquely by each method: Graph.js, file-by-file, and the multi-file approach. As depicted in Figure 5, the vulnerabilities identified by the multi-file approach are a subset of those found by the file-by-file method, which in turn are a subset of Graph.js results. Any true positives missed by the inner methods were considered false negatives. Based on this, we estimated the Recall, Precision, and F1-score for each method, as shown in Table IX. Since we are treating Graph.js as our ground truth, we assumed its recall to be 1 (the highest possible recall) in order compute its F1-score.

Our analysis revealed that the multi-file approach significantly improved precision from 34% in Graph.js to 83%, while maintaining a strong recall of 81%. Thus, the reduction of the reported vulnerabilities was not due to sacrificing recall.

D. RQ3: What is the impact of our new attacker-controlled object definition on the detection?

To address this research question, we applied the Bottom-Up Greedy algorithm with our new attacker-controlled object definition to all packages in the Collected dataset, identifying entry points as before. The results are summarise in Table X.

As Table X hints the number of reported vulnerabilities increased by about 30%, and the average package analysis time increased by approximately 1.8 seconds due to running additional queries.

a) Analysis of the Reported Vulnerabilities:: To confirm that the increase in reported vulnerabilities wasn't just due to false positives, we randomly selected and manually reviewed 120 newly reported vulnerabilities. Of these, 100 (83%) were true positives, and 20 (17%) were false positives. Based on this, we estimate that out of the 690 newly reported vulnerabilities, 575 are true positives and 115 are false positives. Using the previously estimated precision of 83%, the prior analysis had 1931 true positives and 396 false positives, which are a subset of the vulnerabilities detected using the new definition. This brings the total to 2506 true positives and 511 false positives, maintaining the same precision of 83%. Thus, we detected more vulnerabilities without sacrificing precision.

V. Related Work

Scientific research aims to enhance Node.js application security. This section introduces four static analysis tools for detecting vulnerabilities in Node.js applications: *ODGen*, *FAST*, *Nodest*, and *CodeQL*.

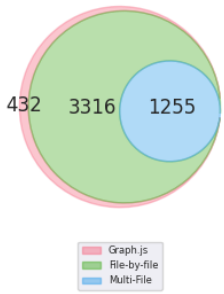

Fig. 5: *Vulnerable packages reported in the Collected dataset*

CWE	Graph.js		File-by-File		Multi-file		Total TP
	TP	FP	TP	FP	TP	FP	
CWE-22	1	9	0	10	9	1	10
CWE-78	1	9	1	9	9	1	11
CWE-94	0	10	1	9	8	2	9
CWE-1321	1	9	3	7	7	3	11
Total	3	37	5	35	33	7	41

TABLE VIII: *Sampling results in the collected dataset*

CWE	Graph.js			File-by-File			Multi-File		
	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1
CWE-22	1	0.33	0.50	0.90	0.45	0.6	0.90	0.90	0.90
CWE-78	1	0.37	0.53	0.91	0.50	0.65	0.82	0.90	0.86
CWE-94	1	0.30	0.46	1	0.45	0.62	0.89	0.80	0.84
CWE-1321	1	0.37	0.54	0.91	0.50	0.65	0.64	0.70	0.67
Total	1	0.34	0.51	0.93	0.48	0.63	0.81	0.83	0.82

TABLE IX: *Sample detection metrics in the collected dataset*

a) ODGen: ODGen, developed by S. Li *et al.*, uses a novel graph structure called the Object Dependency Graph (ODG) to capture object relationships in JavaScript, addressing shortcomings in previous methods. ODG integrates the Abstract Syntax Tree (AST) and employs a two-phased analysis for offline graph queries to detect vulnerabilities such as Prototype Pollution, OS Command Injection, and Path Traversal. ODGen outperforms JSJoern [7] and JSTap-vul [8] with fewer false positives (32%) and negatives, showcasing its enhanced accuracy.

b) FAST: M. Kang *et al.* [9] introduced FAST to tackle scalability issues in abstract interpretation tools like ODGen. FAST combines bottom-up and top-down approaches to construct a data-flow graph, analyzing only instructions directly dependent on the sink. It uses solvers like Z3 to generate exploits, improving accuracy and reducing false positives. FAST outperformed ODGen and CodeQL with lower false positive (11.8%) and negative (16.6%) rates, demonstrating superior scalability and vulnerability detection, including code and command injection, and path traversal.

c) Nodest: B. Nielsen *et al.* [10] developed Nodest to analyze essential modules within a package dynamically. It uses runtime feedback to decide which modules to analyze, maintaining two sets: *MSp* (modules to be analyzed) and *MSb* (modules not to be analyzed). Nodest identified 63 vulnerabilities, including two previously unknown ones, across 11 npm packages, highlighting its adaptability and effectiveness in detecting injection vulnerabilities.

d) CodeQL: Developed by GitHub, CodeQL [11] uses semantic code analysis, treating code as a database to query for patterns and potential issues. It can detect common Node.js vulnerabilities like command injection, path traversal, and prototype pollution. CodeQL allows for in-depth analysis of data flows and security vulnerabilities in large codebases.

These tools compete with Graph.js, which struggles with module detection. FAST and Nodest offer more scalable and efficient analysis than ODGen, CodeQL, and Graph.js. FAST uniquely generates exploits for detected vulnerabilities, reducing false positives. However, all tools still have false positives and negatives, and none detect all vulnerabilities that Graph.js can, such as prototype pollution, which FAST and Nodest miss.

VI. Conclusions

In this work, we addressed significant limitations in the vulnerability detection capabilities of Graph.js, particularly in inter-procedural analysis and handling of external modules. By introducing the Extended Multi-version Dependency

Tool	Vulnerabilities	Avg Analysis Time
Bottom-Up Greedy (unsound defintion)	2327	12.323s
Bottom-Up Greedy (new defintion)	3017	14.121s

TABLE X: *Results of the evaluation on the new definition on the Collected dataset*

Graph (EMDG) and developing three new detection algorithms (Top-Down, Bottom-Up with Pre-Processing and Bottom-Up Greedy), we enhanced the tool's ability to accurately identify vulnerabilities within Node.js applications. Additionally, our new attacker-controlled object helps the tool to detect vulnerabilities that it did not in the past.

Our evaluations on the Combined datasets (which includes the VulcaN and SecBench datasets) demonstrated that the Bottom-Up Greedy approach achieved a recall of 82% recall and 85% precision, reporting 15% fewer false positives compared to Graph.js. Furthermore, testing on a dataset of real-world NPM packages (the Collected dataset) revealed that the Bottom-Up Greedy approach reported 83% fewer vulnerabilities and improved analysis speed by an average of 3 seconds, with estimated precision and recall of 83% and 81%, respectively. Importantly, the introduction of the new attacker-controlled object definition resulted in an increase of the number of reported vulnerabilities, yet the estimated precision remains the same.

These results indicate that our modifications effectively reduce the number of reported false positives. Moreover, the new definition also allows the tool to detect vulnerabilities that it previously overlooked. Consequently, we have enhanced the tool's detection capabilities.

REFERENCES

[1] "Node.js," https://nodejs.org/en, 2023, accessed: 2023-10-03.

[2] "Node package manager," https://www.npmjs.com, 2023, accessed: 2023-10-03.

[3] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. F. Santos, and N. Santos, "Study of javascript static analysis tools for vulnerability detection in node.js packages," in *IEEE Transactions on Reliability*, 2023, pp. 1–16.

[4] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "Secbench.js: An executable security benchmark suite for server-side javascript," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1059–1070.

[5] M. Ferreira, M. Monteiro, T. Brito, M. E. Coimbra, N. Santos, L. Jia, and J. F. Santos, "Efficient static vulnerability analysis for javascript with multiversion dependency graphs," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, p. 25, June 2024. [Online]. Available: https://doi.org/10.1145/3656394

[6] M. Monteiro, "Explodeq.js: A library of queries to detect injection vulnerabilities in node.js applications," Master's thesis, Instituto Superior Técnico, 2023.

[7] "Jsjoern," https://github.com/malteskoruppa/phpjoern, 2024, accessed: 2023-10-29.

[8] A. Fass, M. Backes, and B. Stock, "Jstap: A static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 257–269. [Online]. Available: https://doi.org/10.1145/3359789.3359813

[9] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1059–1076.

[10] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: Feedback-driven static analysis of node.js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 455–465. [Online]. Available: https://doi.org/10.1145/3338906.3338933

[11] "Codeql," https://github.com/github/codeql, 2023, accessed: 2023-10-29.

APPENDIX A
DETAILED ALGORITHMS

Algorithm 1 `MF-MDG(dep,dag,summaries,graphs)`

```
mdg ← NULL
// Process dependencies first
for dep ∈ dag[main] do
    MF-MDG(dep,dag,summaries,graphs)
end for
// Process the file
mdg ← Generate_MDG(main)
for call ∈ mdg.external_calls do
    module ← call.module
    func ← summaries[module][call.func]
    graph ← graphs[module].Get_Func(func)
    mdg.Add_External_Func(graph)
end for
module_graphs[main] ← mdg
summaries[main] ← Get_Summary(mdg)
return mdg
```

Algorithm 2 `Reaches2.0(n1,n2):-`

```
Reaches(n1,n2) ∨
∃ n1',n2',q,p,call:
  Reaches2.0(n2',n2) & // n2 sub-object n2'
  n2' --ARG(q)--> call &
  Reaches2.0(n1,n1') & // n1' depends on n1
  n1' --ARG(p)--> call &
  Reaches2.0(node(q),q') & // q' sub-object q
  Reaches2.0(node(p),q') & // q' depends on p
  PropsTraversed(n2',n2) ==
PropsTraversed(node(q),q');  // the paths n2' → n2 and q → q' have the same sequence of property edges
```

Algorithm 3 `Controls(n)`

```
// get the function that contains n
f ← get_func(n)
params ← get_params(f)
if ∃ p ∈ params: Reaches(p,n) then
    if is_exported(f) then
        return True
    else
        callers ← get_calls(f,p)
        if ∃ call ∈ callers: Controls(call.arg) then
            return True
        end if
    end if
end if
return False
```

Algorithm 4 `Get_Summary(MDG)`

```
summary ← Map()
for export ∈ MDG.exports do
    init ← export.init
    prop ← export.property
    match init do
        case Function
            summary[prop] ← init.name
        case Object
            summary[prop] ← BuildObj(init)
end for
return summary
```

Algorithm 5 `BuildObj(obj)`

```
object ← Map()
for prop ∈ obj.properties do
    init ← prop.init
    name ← prop.name
    match init do
        case Function
            object[name] ← init.name
        case Object
            object[name] ← BuildObj(init)
end for
return obj
```

Algorithm 6 `ConfirmGreedy(func,param)`

```
if is_exported(func) then
    return True
else
    callers ← call_graph_query(param.name)
    for call ∈ callers do
        // caller is vuln ⟹ param is vuln
        if ConfirmGreedy(call.func,call.arg) then
            return True
        end if
    end for
end if
return False
```

Algorithm 7 `Confirm_Vuln(param,CGT)`

```
stack ← [param]
while stack.length != 0 do
    caller ← stack.pop()
    if caller.isExported() then
        return True
    end if
    // Params reaching caller
    stack.append(CGT[caller])
end while
return False
```

Algorithm 8 `Find_Taint_Paths(results,work_list)`

```
if work_list == [] then
    return results
else
    // Get current and remaining paths
    path,remaining_paths ← work_list.pop()
    node ← path.last()
    match node do
        case SINK|RETURN:
            results.append(path)
            return Find_Taint_Paths(results,remaining_paths)
        case CALL:
            // Get called function and param
            func ← node.func
            param_node ← get_param(func,node.previous())
            // Check taint propagation by the param
            param_paths ← Find_Taint_Paths([],[[param_node]])
            sink_paths,ret_path ← filter_paths(param_paths)
            sink_paths ← [path + cont | for cont in sink_paths]
            new_results ← sink_paths + results
            if ret_path ≠ NULL then
                // Param propagates taint, so continue the path
                full_path ← path + ret_path + node.ret
                new_worklist ← full_path + remaining_paths
                return Find_Taint_Paths(new_results, new_worklist)
            else
                // Param does not propagate taint, disregard the path
                return Find_Taint_Paths(new_results, remaining_paths)
            end if
        default:
            // Continue the path by running the taint query at node
            paths ← taint_query(node.Id)
            new_paths ← [path + cont | for cont in paths]
            new_worklist ← new_paths + remaining_paths
            return Find_Taint_Paths(results,new_worklist)
end if
```

Algorithm 9 `Filter_Paths(paths,start)`

```
// Get the paths that end in a sink
sink_paths ← paths.filter((path) => is_sink(path.last()))
// Get the paths that end in a return node
ret_paths ← paths.filter((path) => is_ret(path.last()))
if ret_paths.length > 0 then
    ret_path ← ret_paths.get(0)
else
    ret_path ← NULL
end if
return sink_paths,ret_path
```