International Journal of Information Security
https://doi.org/10.1007/s10207-020-00537-0

REGULAR CONTRIBUTION

DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules

Hee Yeon Kim[1] · Ji Hoon Kim[1] · Ho Kyun Oh[1] · Beom Jin Lee[2] · Si Woo Mun[3] · Jeong Hoon Shin[4] · Kyounggon Kim[5]

Abstract
The safe maintenance of Node.js modules is critical in the software security industry. Most server-side web applications are built on Node.js, an environment that is highly dependent on modules. However, there is clear lack of research on Node.js module security. This study focuses particularly on prototype pollution vulnerability, which is an emerging security vulnerability type that has also not been studied widely. To this point, the main goal of this paper is to propose patterns that can identify prototype pollution vulnerabilities. We developed an automatic static analysis tool called DAPP, which targets all the real-world modules registered in the Node Package Manager. DAPP can discover the proposed patterns in each Node.js module in a matter of a few seconds, and it mainly performs and integrates a static analysis based on abstract syntax tree and control flow graph. This study suggests an improved and efficient analysis methodology. We conducted multiple empirical tests to evaluate and compare our state-of-the-art methodology with previous analysis tools, and we found that our tool is exhaustive and works well with modern JavaScript syntax. To this end, our research demonstrates how DAPP found over 37 previously undiscovered prototype pollution vulnerabilities among 30,000 of the most downloaded Node.js modules. To evaluate DAPP, we expanded the experiment and ran our tool on 100,000 Node.js modules. The evaluation results show a high level of performance for DAPP along with the root causes for false positives and false negatives. Finally, we reported the 37 vulnerabilities, respectively, and obtained 24 CVE IDs mostly with 9.8 CVSS scores.

Keywords Node.js security · Automatic vulnerability extrapolation · Source code analysis · Prototype pollution

1 Introduction

Node.js is a run-time platform for JavaScript applications. JavaScript is one of the most popular programming languages, and it is mostly used within web applications. Node.js was initially developed in 2009 to provide asynchronous event-driven JavaScript runtime for scalable web application [21]. Considering that JavaScript was designed to be used in the domain of client-side scripting, many developers depend on third-party modules to bring in some server-side features, such as the I/O framework, and these modules are heavily dependent on each other. Most Node.js modules are managed by Node Package Manager (NPM), and the number of Node.js modules registered in NPM as of December 2020 is upward one million. A substantial number of developers are convinced that these third-party modules are secure and undoubtedly import them through NPM. However, if module "A" were vulnerable, other modules dependent on module "A" could also become vulnerable. For example, module "lodash," which features the highest module dependency, has 127,661 dependents as of December 10, 2020. Using the 127,661 other modules is potentially dangerous if "lodash" is revealed to be vulnerable. Nevertheless, research

✉ Kyounggon Kim
kkim@nauss.edu.sa

Hee Yeon Kim
sonysame@naver.com

Ji Hoon Kim
time4_ruin@naver.com

1 Department of Cyber Defense, Korea University, Seoul, Republic of Korea

2 Hayyim Security, Seoul, Republic of Korea

3 Alice&Mallory, Seoul, Republic of Korea

4 THEORI, Seoul, Republic of Korea

5 Department of Forensic Sciences, Naif Arab University for Security Sciences, Riyadh, Kingdom of Saudi Arabia

Published online: 13 February 2021

on module security remains insufficient in comparison to the importance of Node.js module security.

Most existing Node.js module vulnerability analysis tools focus on a statistical analysis based on the database of previously published vulnerabilities. For example, target module "A" is considered to be vulnerable, in that its version is known to have vulnerabilities [23,28,33,34], but this approach cannot explore previously undiscovered vulnerabilities. Other tools that try to detect security vulnerabilities in JavaScript programs without statistical analyses mostly focus on detecting bad coding, not on detecting vulnerabilities directly [30,36]. The main goal of this paper is to bridge this research gap by proposing empirical and a novel methodology that can automatically detect vulnerabilities, particularly prototype pollution vulnerabilities in Node.js modules. Prototype pollution vulnerability is an emerging, fatal vulnerability type, and there are few studies related to this issue. Staicu et al. [35] and Patnaik and Sahoo [25] studied methodologies on vulnerability detection in JavaScript source code, but the focus was on command injection and cross-site scripting (XSS) vulnerabilities. Issues surrounding injection and XSS vulnerabilities have been studied extensively for a long time [17,38]. These studies tested the vulnerability of the input value, which means they relied on run-time testing. However, run-time testing is dangerous if the module is malicious, so to prevent malicious modules from running and to shorten the testing time, we based our research on static analysis.

We observe that traditional static analysis has some limitations. Generally, there are mainly two approaches to analyzing JavaScript source code statically: the first approach involves extracting and analyzing the abstract syntax tree (AST) from JavaScript source code, and the second extracts and analyzes the control flow graph (CFG) from JavaScript source code. Since JavaScript adopts lexical scope, AST and CFG could be effectively used for static analysis. Previous tools and methods that extract and utilize AST and CFG are neither accurate nor suitable for the new features in ECMAScript6 (ES6) syntax because there are many specific cases that previous tools and methods cannot handle.

In light of this, this study proposes improved methodologies to extract and analyze AST and CFG that can overcome contemporary weaknesses. Our method identifies patterns of prototype pollution vulnerability that can occur in Node.js modules. We designated these patterns after analyzing modules known to have prototype pollution vulnerabilities. Then, we performed AST and CFG-based analyses and combined the results to test whether or not the target module contained vulnerable patterns. Furthermore, we conducted a data flow analysis (DFA) based on AST and CFG to improve the accuracy of the analysis. By applying these approaches, we consequently developed our proposed automatic static analysis tool, DAPP, to identify prototype pollution vulnerabilities in real-world modules. Additionally, through vulnerabilities detected by DAPP, we discovered a number of enlightening common features between modules containing prototype pollution vulnerabilities. This study also suggests prototype pollution vulnerability features.

The contributions of this paper are as follows.

- First, we introduce a novel automatic vulnerability analysis tool, DAPP, which can perform automatic analysis on all modules (approximately one million modules) registered in NPM in parallel.
- Second, we present the analysis on prototype pollution vulnerability types. This is the first study to suggest source code patterns that can occur in prototype pollution vulnerabilities. Our analysis also indicates how these vulnerability types may be mitigated.
- Third, we illustrate the limitations of previous analyses based on AST and CFG. We propose accurate and improved analysis methodologies based on AST, CFG, and DFA.
- Fourth, our implementation of DAPP is highly scalable with additional vulnerability patterns. DAPP can become a significant contribution to the software safety of using and developing with Node.js modules.
- Fifth, DAPP found 37 previously undiscovered vulnerabilities during our testing. The results of DAPP and proof of concept codes for the patched modules are publicly available at https://github.com/sonysame/DAPP.

The paper is organized as follows. Section 2 presents related works. Section 3 suggests prototype pollution vulnerability patterns in detail, and proposes two source code patterns that can identify prototype pollution vulnerabilities. Section 4 suggests improved analysis methodologies based on AST, CFG, and DFA. Section 4 also demonstrates how to patternize the control flow of the target program and prototype pollution vulnerability types. Section 5 presents the results of our automatic tool, DAPP, and the vulnerabilities found in real-world Node.js modules using this tool. Section 5 also suggests common features of modules containing prototype pollution vulnerabilities that we identified based on DAPP results. Section 6 gives out the evaluation results. The paper is concluded in Sect. 7.

2 Literature review

2.1 Node.js security

In 2012, Ojamaa and Düüna [22] emphasized the security weaknesses of the Node.js platform and described several security precautions to take when using Node.js platforms. The authors previously presented two previously undiscovered vulnerabilities that could be attributed to a denial of

service attacks and warned that Node.js was too underdeveloped to be used in security-critical applications. However, rapid development ensued, and Node.js became widely used and more stable. One of the well-known advantages of developing with Node.js is having the flexibility to combine libraries from third parties. In light of this, Patel [24] and Pfretzschner and ben Othmane [26] surveyed dependency-based vulnerabilities found in Node.js applications, and these studies concluded that the high dependency feature of Node.js modules can be critical for application security. De Groef et al. [6] also addressed security issues surrounding the flexibility of the platform in regard to loading third-party modules. In order to solve this problem, De Groef et al. [6] proposed a JavaScript security architecture called NodeSentry. NodeSentry adds web-hardening or access control policies when importing every third-party module. For example, if an imported module requests the URL of an incoming request, the NodeSentry framework works as a wrapper to intercept filtering and check input values dynamically. However, this policy is different for each module, and developers must manually define individual policies when using NodeSentry. Our study starts from the same position of examining the high-module dependency feature of Node.js. However, our research diverges to explore vulnerabilities directly in module code.

Staicu et al. [35] proposed SYNODE, an automated mitigation technique focused on command injection vulnerabilities in Node.js applications. SYNODE performs static analysis for input values that flow into injection APIs. If the values cannot be statically determined, it performs a run-time enforcement of security policies. This method involves constructing a partial abstract syntax tree (PAST) for each call site as the security policy that represents the expected structure of benign input values. During run-time, it checks whether or not the AST is following the pre-defined security policy. Our present work shares this idea of targeting Node.js modules and using AST to verify the structure of code. On the other hand, Gauthier et al. [9] proposed AFFOGATO, a taint analysis tool that can detect injection vulnerabilities in Node.js via grey-box taint analysis. AFFOGATO compares untrusted (i.e., tainted) strings in a program against string values in a security-sensitive function, or sink. If the two strings are similar, the tool reports the data flow between the source, where the tainted data had entered, and the sink, where the possible vulnerable point may be located. However, the aforementioned work focuses solely on command injection vulnerabilities, and the study is based on run-time testing that checks if input values are malicious. In contrast, our study identifies whether or not each Node.js module is exploitable in and of itself. Our static analysis is therefore advantageous, as it does not execute the modules to safeguard against potentially malicious target modules.

Along with this, Sun et al. [36] introduced NodeProf, a dynamic analysis tool for Node.js, that helps developers find bugs, performance bottlenecks, and bad coding practices. NodeProf performs AST-level instrumentation dynamically at run-time, wrapping AST nodes to enable additional operations to run before or after the original code is executed. AST-node wrapping makes it possible to bind state-to-code locations and can be used for analysis that requires event tracking at specific locations. Our tool and NodeProf are similar in that they both use AST for analysis. While NodeProf targets detecting bugs such as a concatenating "Undefined" variable to "String" variable, our tool focuses on detecting vulnerabilities within Node.js modules.

In 2015, Madsen et al. [19] proposed an "event-based call graph" to detect bad coding practices in Node.js applications. Similar to this study, our work also uses a flow graph to analyze Node.js applications. However, as the preceding work focuses on detecting bad coding practices that arise in event-driven programs, it differs significantly from our present research. Another work, Davis et al. [5], also proposed a comparative fuzzing testing tool for detecting concurrency bugs in event-driven Node.js applications called Node.fz.

2.2 Code analysis methodology in general

Gong et al. [12] presented DLint, a dynamic analysis approach that checks the source code quality rule, which is a pattern of code or execution behavior that should be avoided. The authors defined the run-time pattern of the JavaScript code corresponding to each code quality rule. DLint first intercepts all JavaScript code of the website before execution, and adds instructions to perform the analysis. After the browser loads the website, the analysis is executed and DLint examines if the specified run-time patterns exist. However, the run-time patterns defined in this study, unfortunately, were largely based on a single run-time event and were unable to handle multiple events occurring in a given sequence. Quinlan et al. [27] introduced techniques for specifying common bug patterns in C and C++. Although our work shares the idea of source code analysis with AST and CFG used in this study, the previous work is either limited to a single language or must be translated to Datalog, another language. Holland et al. [16] presents statically informed dynamic (SID) analysis that provides critical capabilities for detecting algorithmically complex vulnerabilities. SID analysis uses Loop Call Graph (LCG) to explore the nesting structure of loops and computes critical loop attributes. Source code analysis using AST is also used for plagiarism detection. Since AST directly describes the structure of a program, plagiarism detection is done simply by comparing AST structures [7]. To this end, CodeCompare [37], AST-CC [42] and CodEX [43] are tools that detect plagiarism using AST. CodEX, in contrast, supports both code-snippet search and plagiarism detection. This

study confirms that it is effective to compare the similarity of source code using AST.

2.3 Vulnerability detection and extrapolation methodology in general

Our study is more closely related to using vulnerability patterns rather than using code similarities. Using vulnerability patterns is an effective way to detect vulnerabilities, and as such, DAPP follows this approach. Yamaguchi et al. [39] firstly suggested the concept of vulnerability extrapolation, which is the detection of vulnerabilities using known unsafe patterns that are then passed on to manual auditing with potentially vulnerable codes flagged. It could be said that vulnerability extrapolation is an assisting methodology for exploring vulnerabilities. DAPP is a vulnerability extrapolation tool that uses patterns to find vulnerabilities, similar to the approach taken in [39]. However, Yamaguchi et al. focuses only on function call API usage patterns, whereas DAPP targets the whole source code.

The main disadvantage of vulnerability extrapolation is that it requires time-consuming manual auditing. Therefore, it is crucial to reduce false-positives. In fact, Yamaguchi et al. narrow the vulnerable candidate functions from 6,778 to 20 functions, finding one previously undiscovered vulnerability among 20 candidates. In contrast, DAPP narrows the vulnerable candidate modules from 30,000 to 75 modules, finding 37 previously undiscovered vulnerability among 75 candidates. One year after this study, Yamaguchi et al. [40] published another article regarding vulnerability extrapolation. The authors expanded on their previous study by extracting ASTs from vulnerable source code and target source code. Then, they analyzed and compared the ASTs using machine learning techniques. Even though our present study also extracts AST from each target source code, it differs from the research by Yamaguchi et al. in that we identify vulnerable patterns and are able to detect those patterns using AST without comparing AST similarities with itself. Grieco et al. [13] implemented VDISCOVER, a tool using machine learning to predict vulnerabilities, especially memory corruptions, in binary programs. Likewise, a number of other studies also based their methods on machine learning techniques for vulnerability detection [11,29,39].

3 Prototype pollution vulnerability patternization

In this study, we newly defined source code patterns to detect the prototype pollution vulnerability in Node.js modules. This section firstly describes the target vulnerability type of this study, "prototype pollution." Then, Sect. 3.2 shows two prototype pollution vulnerability source code patterns we created. How DAPP locates and identifies those two specific vulnerability patterns in the target source code will be illustrated in Sect. 4.

3.1 Prototype pollution

Due to the object structure of JavaScript, prototype pollution vulnerability occurs specifically in program operating on JavaScript engines, such as Node.js. In the JavaScript execution environment, it is possible to manipulate important values of objects to change execution flow. On the server side, like in the case of Node.js, arbitrary code execution is possible, which can cause serious security problems. All JavaScript data types excluding "null" and "undefined" have differing prototype properties. If the prototype data of a specific object, class, or function are modified, creating a new instance through the constructor of the modified object, class, or function will keep the properties modulated.

```
1  var obj2 = new Object()
2  obj2.__proto__.polluted = true
3  var obj = new Object
4  console.log(obj.polluted) //
       true
5  console.log(polluted) // true
```

Listing 1 Example of prototype pollution vulnerability

As shown in Listing 1, if the "__proto__" property of the object class instance is assigned or modulated, the properties of the new instance can be affected. In cases when prototype pollution vulnerability occurs, a double-reference (i.e., obj[a][b]=value) is necessary, and this is done by defining the first array value ("a" in obj[a][b]) as "__proto__." A triple-reference (i.e., obj[a][b][c]=value) also works by defining the first array value ("a" in obj[a][b][c]) as "constructor" and the second array value ("b" in obj[a][b][c]) as "prototype." Even if the target object class for the prototype pollution attack was not an Object, an attack can take place by referencing the "__proto__" property several times, as shown in Listing 2.

```
1  class A { }
2  var obj = new A()
3  obj.__proto__.__proto__.
       polluted = true
4  console.log(polluted) // true
```

Listing 2 Prototype pollution attack on non-Object class

If the target class inherits another class, the reference level should be deeper, as shown in Listing 3

```
1  class A { }
2  class B extends A { }
3  var obj = new B()
4  obj.__proto__.__proto__.
       __proto__.polluted = true
5  console.log(polluted) // true
```

Listing 3 Prototype pollution attack in inheritance

Prototype pollution can lead to critical vulnerabilities such as remote code execution, denial of service and property injection. Remote code execution can occur when the source code evaluates and executes the attribute of an object. For example, attacker can be able to execute arbitrary code if the attribute of an argument took by *eval* function can be polluted. Denial of service can occur when the attacker can pollute generic functions such as "toString," which is a method of Object class. Similarly, property injection can occur when the attacker can add new attributes by polluting the prototype of an object. For example, if the security property such as "Object.prototype.isAdmin" is falsified to "true," the attacker can achieve admin privileges [32].

3.2 Vulnerability pattern for prototype pollution vulnerability

Prototype pollution vulnerability generally occurs through performing a reference and assignment of object values recursively. It occurs within modules when users can freely approach and control the internal value of an object through property definition, object merging, or object copying. If an attacker can insert an argument, like "__proto__" property, as input without verification, prototype pollution vulnerability can also occur. We created the pattern for a prototype pollution vulnerability after analyzing the previous modules vulnerable to prototype pollution attacks. We were able to find common features among the previous modules based on the characteristics of prototype pollution attack. Our study proposes two source code patterns for prototype pollution vulnerabilities as follows: (i) Source code Pattern 1 is represented in Algorithm 1. Algorithm 1 is an example of a function setting given object's key with given value. The path of object's key is recursively assigned, and the given value is assigned at last. If the attacker can put keywords such as "__proto__" as an input of object's key, it can cause prototype pollution vulnerability; (ii) the same logic applies to source code Pattern 2, which is shown in Algorithm 2. However, the two patterns are separated because the control flow of the two patterns is different.

Algorithm 1 Prototype Pollution Vulnerability Source Code Pattern 1

```
1: Input path, value
2: i ← 0
3: while i < path.length do
4:     key ← path[i]
5:     if i = (path.length − 1) then
6:         object[key] ← value
7:     end if
8:     object ← object[key]
9:     i ← i + 1
10: end while
```

Algorithm 2 Prototype Pollution Vulnerability Source Code Pattern 2

```
1: Input path, value
2: i ← 0
3: while i < (path.length − 1) do
4:     key ← path[i]
5:     object ← object[key]
6:     i ← i + 1
7: end while
8: object[i] ← value
```

DAPP locates these two patterns through static analysis based on AST and CFG. The structure for Algorithms 1 and 2 can be shortened into AST nodes, and the relation between those nodes is analyzed by CFG, and this methodology will be described in the next section.

4 Methodology

4.1 Overview

Figure 1 is an overview of DAPP; we need vulnerability patterns as input. Section 3.2 discusses examples of vulnerability patterns focusing on prototype pollution vulnerability. DAPP has two phases: (i) analysis phase and (ii) pattern searching phase. These two phases run concurrently, and each has three parts. First, the target source code, as input, is given to the AST analyzer. The AST analyzer builds an AST from the given target source code and extracts variables and function information. Then, the pattern-searching phase implements a first filtering procedure using the output AST. DAPP traverses the AST and detects whether the AST nodes defined in the vulnerability pattern exist. Next, the AST from the AST analyzer is given to the CFG analyzer. The CFG analyzer builds a CFG from the given AST. Subsequently, the pattern-searching phase implements the second filtering procedure using the output CFG. The second filtering procedure identifies whether or not the output nodes from the first filtering procedure are located in the proper and targeted order. Finally, the variable and function information extracted from the AST analyzer and CFG from the CFG analyzer are given to the DFA analyzer. The DFA analyzer determines which data flow affects each variable value and how. The purpose of the DFA analyzer is to ascertain whether user input can reach the potentially vulnerable part. The third filtering procedure uses the results from the DFA. If a vulnerability is found, DAPP shows the vulnerability pattern type and specific location of the vulnerability in the target source code.

4.2 Basic concept

Abstract syntax tree (AST) is generally produced by the syntactic parser in a compiler, and is the result of each statement

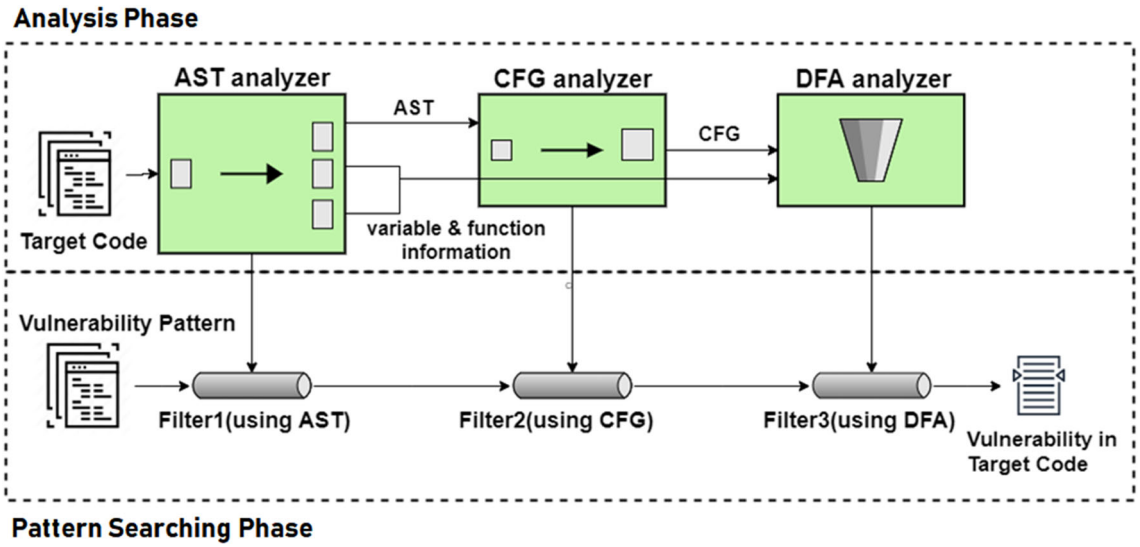

Fig. 1 Overview of DAPP

and expression in the source code being parsed to make it easy for computers to understand. Each string in the source code is split into a list of tokens by a scanner known as a lexical analyzer. For example, the "var a=1" string is split into a token list, var, a, and 1. Then, a syntactic parser transforms the token list into AST. AST is a tree format composed of multiple nodes, with many AST generators handling JavaScript program, such as esprima [15], babel-eslint [3], and acorn [1]. Our tool, DAPP, uses esprima to generate AST. Esprima is the most appropriate AST generator for DAPP because esgraph [8], the CFG generator, relies on esprima, which we are going to use. AST is advantageous in that it facilitates easy comparison and analysis of the source code by token unit.

The control flow graph (CFG) is a graphical representation of all possible execution flows of a program. It is commonly used in source code compilation and static code analysis. The CFG has been used in many areas of source code analysis, especially in security studies [41]. CFG is made up of AST nodes. In other words, CFG shows the order of AST nodes. Therefore, it must be preceded to construct AST nodes of the target source code in order to create CFG.

We can determine whether the control flow reaches the vulnerable statement via analysis of the CFG, while just using AST only points out the existence of a possibly vulnerable statement. Using CFG, it is also able to analyze the order of execution of the source code, which makes it possible for the analyzer to detect an ordered source code pattern made up of multiple sentences.

Each function of the source code, including the main program, is represented in the CFG. Each CFG starts from an entry node and ends at an exit node. Entry and exit nodes are created separately from the source code. The first CFG node of a function is connected to the entry node. When the function exits, such as a return statement or exception, it is connected to the exit node. A CFG node is a single expression, and the CFG edge has properties, such as true, false and exception. Edges have different properties depending on the connected node type. A program consists of elements like declaration statements, assignment statements, conditional statements, and loop statements. Each statement is expressed in the CFG, and the nodes are connected by edges with the appropriate properties. For example, statements such as declaration and assignment will have one incoming and one outgoing edge each in most cases. But for conditional statements, there are two outgoing edges with one "true" and one "false" property. We can analyze the data flow of a program based on the CFG we generated, which is explained in Sect. 4.7.

4.3 Searching prototype pollution vulnerability pattern based on AST nodes

Listing 4 is the AST node for the statement "object=object[key]" in the source code Pattern 1 and source code Pattern 2. Likewise, Listing 5 is the AST node for the statement "object[key]=value" in source code Pattern 1 and the statement "object[i]=value" in source code Pattern 2. To detect source code Pattern 1 in the target source code, first DAPP must detect whether both the AST nodes (Listing 4, Listing 5) exist. Then, if both AST in Listing 4 and 5 are detected during the first filtering procedure in Fig. 1, the second filtering procedure in Fig. 1 checks if the detected nodes for Listing 4 and 5 are in cycles, which will be discussed in Sect. 4.4.

In the case of detecting the source code Pattern 2, the first filtering procedure works in a similar way to detecting source

code Pattern 1. However, unlike source code Pattern 1, the second filtering procedure checks if the detected nodes for Listing 4 make a cycle by itself, and if the cycle is connected to the detected node for Listing 5. Therefore, the next step is to distinguish whether AST nodes make a cycle or a one-way connection. Patternizing the control flow which will be illustrated in Sect. 4.4 identifies this relationship.

```
{
    "type": "ExpressionStatement",
    "expression": {
        "type": "AssignmentExpression",
        "operator": "=",
        "left": {
            "type": "Identifier",
            "name": "object"// name can vary
        },
        "right": {
            "type": "MemberExpression",
            "computed": true,
            "object": {
                "type": "Identifier",
                "name": "object"// name can vary
            },
            "property": {
                "type": "Identifier",
                "name": "key"// name can vary
            },
        },
    },
}
```

Listing 4 Prototype Pollution Vulnerability AST Pattern 1

```
{
    "type": "ExpressionStatement",
    "expression": {
        "type": "AssignmentExpression",
        "operator": "=",
        "left": {
            "type": "MemberExpression",
            "computed": true,
            "object": {
                "type": "Identifier",
                "name": "object"// name can vary
            },
            "property": {
                "type": "Identifier",
                "name": "key"// name can vary
            },
        },
        "right": {
            "type": "Identifier",
            "name": "value"// name can vary
        },
    },
}
```

Listing 5 Prototype Pollution Vulnerability AST Pattern 2

4.4 Patternizing the control flow based on CFG

A control flow pattern is a subgraph of the control flow graph, which is defined based on the vulnerability pattern. The control flow pattern consists of one-way connections and cyclic relationships between control flow nodes. For example, statements in the "for" loop are in a cyclic relationship. In the implementation of pattern detection, the detection of one-way connections and loops is performed separately.

Identifying the one-way connection of nodes in CFG can be done by depth-first-search (DFS). For example, if the control flow pattern consists of two statements that are connected in one direction, DAPP performs DFS from the first statement node and searches for the second statement.

A cycle can be identified by using strongly connected components (SCC). When two different nodes u and v are selected in the directed graph, then these nodes are deemed strongly connected if both paths from u to v and v to u exist. An SCC can be generated by using the Kosaraju-Sharir algorithm [31]. It groups the nodes into SCCs, and the generated SCCs satisfy the following:

1. Two different nodes (u, v) from the same SCC always have the path from u to v and v to u.
2. The paths from u to v and v to u cannot exist at the same time if u and v are in different SCCs.

This means that if the two different nodes u and v are in a cycle, they are in the same SCC [31]. In the case of Fig. 2b, the generated SCC will be ((A), (B, C, D), (E)). Node B, C, and D are in the same SCC since they are connected in a loop, while node A and E belong to different SCCs. Thus, we can identify whether or not the CFG nodes are in a loop using SCC.

4.5 Intricacies of modern JavaScript languages

JavaScript uses a lexical scope. Declarations of variables and functions are determined at lexing time of compilation, so the scope of a function is determined when it is declared, not when the function is called.

There are three ways to declare a variable in JavaScript: var, let and const. Variables declared using var use a function-level scope [20]. On the other hand, variables declared using

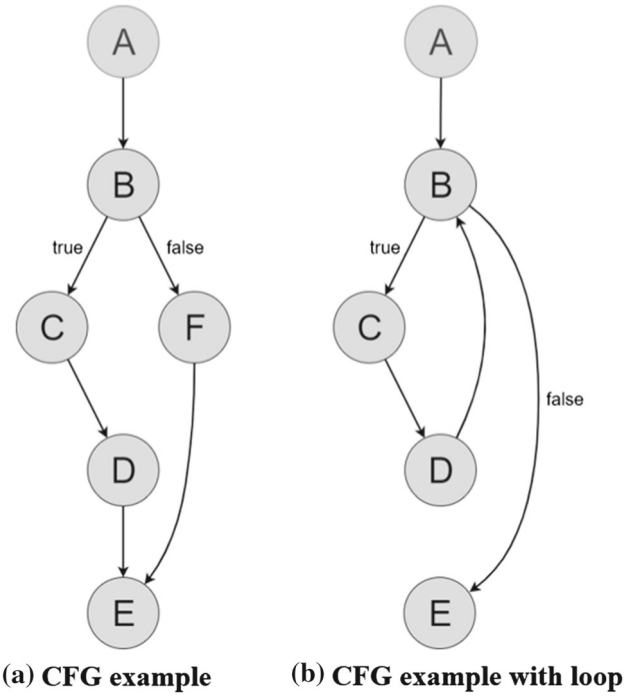

Fig. 2 CFG examples

let and const use a block-level scope, which means that they can be used within the brackets where they are declared. Apart from this, variables used without any declarations are declared as an implicit global variables. These variables are always declared as global even if they are used inside a function. Since the flow of data can vary depending on the scope of the variable, it is necessary to distinguish the scope of each variable.

In JavaScript, a variable or a function can be used before declaration, a process that is called hoisting. Hoisting is a feature of JavaScript that moves all declarations to the top of the current scope. Hoisting works differently depending on the type of variable. Declared var-type variables are moved to the top of the current scope via hoisting, and the variable can be used from the top of that scope. However, only declarations are hoisted, meaning that assignments that occur concurrently with a declaration are not. Unlike var-type variables, declarations of let and const-type variables are not hoisted to the top. Therefore, using let or const-type variables before declaration results in a reference error. Those variables are said to be in the "temporal dead zone" from the beginning of the scope until they are declared.

```
1    function foo(){ return 0; }
```

Listing 6 General function declaration

```
1    var foo = function (){ return
         0; }
```

Listing 7 Function declaration assigned to a variable

A function can be declared in various ways. Listing 6 and Listing 7 show two examples of function declaration in JavaScript. Both functions can be used by calling foo(), but the difference between the two functions lies in hoisting. A function declared in a general way without assignment to a variable, such as Listing 6, is hoisted to the top of the scope and can be used from the beginning of the scope. However, a function assigned to a variable, such as Listing 7, is not hoisted, since hoisting applies only to declarations, not assignments.

4.6 Extracting variable and function information

To analyze the control and data flow of a program, extraction of variables and functions is necessary. Our tool, DAPP, extracts variables and functions based on the AST of a source code considering the exact scope and hoisting mentioned in Sect. 4.5. Since DAPP generates the AST with esprima, the type of the AST nodes will be discussed based on the terminologies of esprima. For example, the term BlockStatement represent braces in source code. Extraction of variables and functions is completed through a total of three AST traversals. First, the AST analyzer traverses the AST and creates a new scope when it encounters an expression or statement that uses its own scope, such as BlockStatement or FunctionDeclaration. Scopes are organized in a tree, and the inclusive relationships of the scopes are represented by parent-child relationships between scope nodes. The Program node in the AST represents the whole source code, which forms the root node of the tree.

After scope creation is completed, the AST analyzer traverses the AST once more. Since it is permitted to assign and use variables without declaration in JavaScript, we separated variable extraction into two parts: extraction from declaration and extraction from assignment. The objective of the second traversal is to extract variables and functions from declaration nodes. Variable extraction and function extraction are executed in parallel. When the AST analyzer encounters a VariableDeclaration node, it first checks whether it is a duplicate of a declared variable in the same scope. If it is a duplicate, the analyzer continues to traverse. If not, the analyzer extracts the properties of the node, such as name and type. At the same time, the analyzer traverses the previously created scope tree to locate the appropriate scope of the variable. The scope that is found in this way depends on whether the variable type uses a function-level scope or block-level scope. Var-type variables use a function-level scope, while let and const-type variables use a block-level scope. When the AST analyzer encounters either a FunctionDeclaration or FunctionExpression node, the analyzer extracts the properties of the node, including its name, parameters, external scope, and internal scope. External scope is the scope of the function, which means where it can be accessed in the source

code. Internal scope is the scope of the function itself, similar to its location in the source code. Because every FunctionDeclaration node has a "child" FunctionExpression node, the analyzer checks the parent node during traversal to ensure that it has not been extracted as a duplicate.

After this function extraction, function parameter extraction is performed. The analyzer goes through the list of functions found earlier to extract function parameters. The scope of each parameter is defined from beginning to the end of the function. The type of the extracted function parameter is set differently from variables declared as var, let, or const.

Finally, the AST analyzer traverses the AST to extract implicit global variables. In this process, variables that are assigned a value without a declaration are extracted. When the AST analyzer encounters AssignmentExpression, it examines whether the identifier at the left of the assignment has already been extracted from declaration. If the variable has not been declared elsewhere, the analyzer extracts the variable as a global type.

4.7 DFA

Data flow analysis (DFA) is an analysis method that gathers data information on the control flow. This section describes data flow analysis methodology applied in DAPP. DAPP performs data flow analysis based on CFG nodes and variable/function information extracted from AST analysis. First, it is necessary to order each CFG node for data flow analysis. Sections 4.7.1 and 4.7.2 illustrate the algorithm we selected to order each CFG node for DFA. Section 4.7.3 describes the implementation of the DFA in detail.

4.7.1 Reverse post-order traversal

Post-order traversal is one of the DFS traversals. Using DFS traversals, all nodes in a tree or graph are visited once, and each node is assigned a unique number. In order to complete the DFA while visiting each node just once, it is necessary to sort each node into the proper topological order.

During post-order traversal, the node receives a number by recursively, after all outgoing edges have been visited. Reverse post-order traversal literally reverses the order of post-order traversal. If the nodes are numbered by reverse post-order, the nodes will be topologically sorted, such that each node receives its number before any of its successors [10,35]. For example, as shown in Table 1, the result of pre-ordering shown in Fig. 2a shows that node E is numbered before node F. However, since node E can be influenced by node F, node E must be numbered after node F for unhindered data flow analysis. The results of reverse post-ordering in both Fig. 2a, b in fact show that all nodes receive their number before their successors.

Table 1 Ordering CFG nodes

Order	Fig. 2a	Fig. 2b
Pre-order	(A, B, C, D, E, F)	(A, B, C, D, E)
Post-order	(E, D, C, F, B, A)	(D, C, E, B, A)
Reverse pre-order	(F, E, D, C, B, A)	(E, D, C, B, A)
Reverse post-order	(A, B, F, C, D, E)	(A, B, E, C, D)

4.7.2 Handling loop via dominator tree

Using only reverse post-order traversal, however, does present limitations. The normal DFS algorithm executes a search on a graph without prioritizing the edges if there are two or more outgoing edges attached to a node. It is critical to note that not determining prioritization can produce serious problems in data flow analysis if one of the outgoing edges is in a loop.

Figure 2b is an example of a CFG that includes a loop. When the analyzer reaches node B during DFS to generate reverse post-order, it must first select which branch to visit between node C and node E. If the analyzer visits node C first, the result of the reverse post-order is (A, B, E, C, D). If the data flow analyzer makes the traversal in this order, the analyzer visits node E before visiting nodes C and D. That is, although nodes C and D may be evaluated before evaluating node E in the real flow, the change of data flow that might occur at nodes C and D cannot be applied to node E. If the analyzer visits node E first, the result of reverse post-order becomes (A, B, C, D, E). Prioritizing the direction of reverse post-order properly in a loop situation is important for proper data flow analysis.

Identifying loops can be done by using a dominator tree [14]. A dominator tree is a tree of dominating relationships. A vertex v is dominated by vertex $u(\neq v)$ when every path from the root node to vertex v goes through vertex u [4]. We generated a dominator tree of a CFG using the Lengauer-Tarjan [18] algorithm. Algorithm 3 shows how to use the dominator tree to generate an appropriate reverse post-order with determining prioritization.

4.7.3 Implementation of data flow analysis

By using the above traversal order, we implemented a data flow analysis framework to first clarify values directly assigned, and second to gather tainted information on variables.

We also defined a context for each CFG node. A context contains variable information in its node. On each control flow node (hereinafter referred to as flownode), all context information evaluated in the previous flownodes are conjoined and copied to the current flownode. Based on this

Algorithm 3 Generate reverse post-order with determining prioritization

```
1: GetRPO(cfg){
2:   traverse ← function(Node){
3:     visited.add(Node)
4:     for prevNode of Node.prev do
5:       if !visited.has(prevNode) then
6:         for nextNode of Node.next do
7:           if nextNode dominates prevNode then
8:             latetraverse.add(nextNode)
9:           end if
10:        end for
11:      end if
12:    end for
13:    for nextNode of Node.next do
14:      if !visited.has(nextNode)
         and !latetraverse.has(nextNode) then
15:        traverse(Node)
16:      end if
17:    end for
18:    for late of latetraverse do
19:      traverse(late)
20:    end for
21:    POarray.push(Node);
22:  }
23: }
```

context, the current flownode is calculated and changes the data information that is mapped into variables.

JavaScript data types can be classified into two kinds: (i) primitive values and (ii) object references. Primitives are immutable—that is, they cannot be changed once created. Object references, in contrast, are mutable, and the values (attributes) can be changed after being created.

```
1   let a = 1;
2   a.a = 1;
3   console.log(a.a); // undefined
```

Listing 8 Example of primitive values

```
1   let a = {};
2   let b = a;
3   b.a = 1;
4   console.log(a.a); // 1
```

Listing 9 Example of object references

As shown in Listing 8 and Listing 9, the two data types work differently on the same operation. Therefore, these data types require different operational logic when gathering data information. DAPP overcomes this by making two information classes (DataInfo, ReferenceInfo) and a wrapper class (Candidate) that has two information class instances as attributes. Candidate instance is the final form of data information, which is mapped to appropriate variables and forms context.

Current context is calculated in two steps. First, the contexts of previous flownodes are joined to create the current context. This "union" method is called on each Candidate instance mapped in the same variable. Next, based on the AST nodes of the current flownode, the current context is reevaluated.

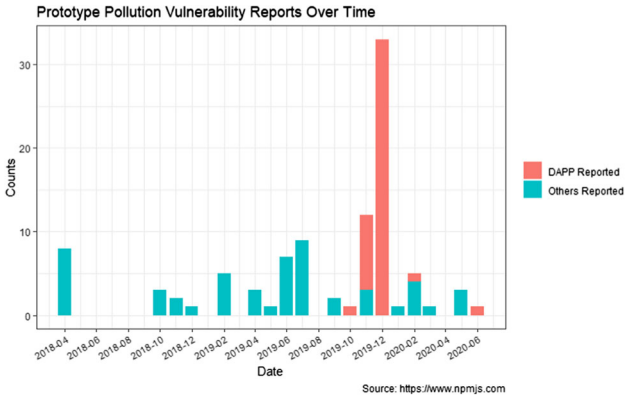

Fig. 3 Prototype pollution vulnerability reports in NPM advisory over time

5 Result

In this section, we present the results of running DAPP. Table 2 shows the modules in which we found vulnerabilities. The figures (Downloads and Dependents) in Table 2 are as of December 20, 2019. The detected modules can be classified into six types based on functionality, and this criterion is described in Sect. 5.1 in detail. We ran our tool, DAPP on 30,000 modules. We selected 30,000 modules on the basis of the highest downloads recorded in the NPM. Among 30,000 modules DAPP found 75 modules vulnerable. Verification on 75 modules done manually confirmed that 37 modules are actually vulnerable to prototype pollution attacks. We reported those 37 modules and proof of concept (PoC) for each vulnerable module through the NPM and Snyk. DAPP will not be public since lots of modules that have not yet been verified may be at risk and exploited by attackers using DAPP. Instead, DAPP results and proof of concept exploit codes for the patched modules are publicly available at https://github.com/sonysame/DAPP.

Among the 37 reported modules, 16 modules (A2, A3, A4, A7, B1, B2, B4, C1, C5, D1, D3, E1, E2, G1, G2 and G3) in Table 2 were patched after our reporting as of December 10, 2020. As we cannot disclose information regarding the unpatched modules, this section will provide detailed information about the patched module (B1 module in Table 2). Also, we will provide the proof of concept for the B1 module on behalf of all the patched modules, which can exploit the prototype pollution vulnerability.

B1 module is a dot notation-related module that provides methods for getting, setting, and deleting a property from

Table 2 List of vulnerabilities which DAPP found

Functionality	Module	Downloads	Dependents	Function	Identifier	CVSS score
Object-related modules	A1	161,838,339	43	Set	CVE-2020-7751	7.2
	A2	140,513,545	33			
	A3	70,762,220	240		Published in NPM	5.0
	A4	39,922,313	22	Set	CVE-2020-7707	9.8
	A5	1,179,963	9			
	A6	22,039	8	Set	CVE-2020-7721	9.8
	A7	6345	6		CVE-2020-7701	9.8
	A8	184	0		Published in NPM	
Dot notation-related modules	B1	551,155,879	522	Set	Published in NPM	7.3
	B2	5,694,387	375	Set		
	B3	546,832	9		CVE-2020-7717	9.8
	B4	372,582	44	Set	CVE-2020-7715	9.8
	B5	30,083	3	Set	CVE-2020-7716	9.8
	B6	1449	3	Set	Published in NPM	
Parsing-related modules	C1	5,276,429	82	Parse	CVE-2020-7719	9.8
	C2	2,454,505	39			
	C3	528,922	50	Parse	CVE-2020-7700	9.8
	C4	29,414	4	Parse	CVE-2020-7702	9.8
	C5	2099	3		CVE-2020-7704	9.8
Configuration-related modules	D1	17,995	6	Set		
	D2	7478	3	Set	CVE-2020-7714	9.8
	D3	6449	2	Set	CVE-2020-7706	9.8
	D4	4241	7	Set	CVE-2020-7724	9.8
JSON pointer-related modules	E1	466,048,725	95	Set		
	E2	7,950,332	211	Set	CVE-2020-7709	6.0
Wrapper-related modules	F1	37,698	2	Set	CVE-2020-7727	9.8
	F2	9285	4			
Other modules	G1	489,421,131	840	Set	CVE-2020-7720	9.8
	G2	69,224,455	1191		CVE-2020-7768	7.5
	G3	11,250,052	172			
	G4	95,924	9	Set	CVE-2020-7737	7.3
	G5	72,251	17	Set	CVE-2020-7718	9.8
	G6	16,278	3			
	G7	14,719	3	Set	CVE-2020-7725	9.8
	G8	8430	3	Set	CVE-2020-7722	9.8
	G9	5450	2	Set	CVE-2020-7703	9.8
	G10	4466	2		CVE-2020-7723	9.8

a nested object using a dot path. DAPP found the prototype pollution vulnerability in this B1 module. Among the two source code patterns we designated in Sect. 3.2 that can generate prototype pollution vulnerability, source code Pattern 1 was found in the B1 module by DAPP. Listing 10 is the actual vulnerable code in this B1 module, and Listing 11 is the Proof of Concept. The set function of Listing 10 takes an object, path and value to set the value of the given path in the object to the given value. Line 5 in Listing 11 polluted the "__proto__" property, "foo," to have the value, "bar." As a result, the value of "foo" property in all the other objects becomes "bar." It means that the attacker can write

an arbitrary value to the property of all the objects. This can be extended to remote code execution, denial of service, or property injection attack.

```
1  set(object, path, value){
2      if (!isObj(object) || typeof path !== "string"){
3          return object;
4      }
5
6      const root = object;
7      const pathArray = getPathSegments(path);
8
9      for (let i = 0; i < pathArray.length; i++){
10         const p = pathArray[i];
11         if (!isObj(object[p])){
12             object[p] = {};
13         }
14         if (i === pathArray.length - 1){
15             object[p] = value;
16         }
17
18         object = object[p];
19     }
20     return root;
21 }
```

Listing 10 Source code vulnerable to prototype pollution (B1)

The CFG analyzer uncovered that the nodes for line 15 and line 18 in Listing 10 are in cycle. Also, the DFA analyzer helped to gather variable/function information of the target code. We reported this vulnerability on Oct 14, 2019, and the patched source code was committed on Oct 23, 2019. The vendor added a function (Listing 12) that checks for disallowed keys while splitting object name with dot notation.

```
1  var B1 = require("B1");
2  var malicious_payload = "__proto__.
       foo";
3  var target = {};
4  console.log(target.foo); //
       undefined
5  B1.set({}, malicious_payload, "bar"
       );
6  console.log(target.foo); // bar
```

Listing 11 Proof of Concept (B1)

```
1      const disallowedKeys = ["
           __proto__", "prototype", "
           constructor"];
2      const isValidPath =
           pathSegments => !
           pathSegments.some(segment =>
            disallowedKeys.includes(
           segment));
3      if (!isValidPath(parts)) return
            [];
```

Listing 12 Filtering by keyname applied to module B1

5.1 Analysis based on result

We have analyzed all 37 modules that DAPP found vulnerable to prototype pollution attacks. Those modules are highly influential because of the high download count and number of dependents. The average number of downloads is 54,717,105, and the number of average dependents is 110. Moreover, we discovered that module functionality and the internal vulnerable function are important common characteristics.

The vulnerable module functionality can be classified into six types as follows:

(i) 22% of the 37 modules are related to objects. For example, if the target module is for getting or setting deep/nested objects, prototype pollution vulnerability can be generated during the setting process;
(ii) 16% of the 37 modules are related to dot notation. Those modules typically use dot notation string given via user input to set values on objects. Similar to the first type, prototype pollution vulnerability occurs during the setting process;
(iii) 14% of the 37 modules have a parsing functionality. For example, modules that assimilate language A to language B need to parse statements. Parsing statements includes dot notation followed by value storing, at which point prototype pollution vulnerability can occur;
(iv) 11% of the 37 modules are related to the configuration function that let users control configuration. However attackers can also use this functionality to store input values at a given path location, and prototype pollution vulnerability can be generated with malicious input;
(v) 5% of the 37 modules are related to JSON-pointer. Those modules work as JSON parsers, and prototype pollution attack can arise during nested JSON format string interpretation;
(vi) 5% of the 37 modules are related to the wrapper functionality. These modules generally parse user input to be compatible with the wrapped interface, and prototype pollution vulnerability can occur while the input is being parsed.

Furthermore, the internal function for each module where prototype pollution vulnerability occurred has common features. 62% of the 37 modules have vulnerability in setting-related functions. These functions work almost in same way and are intended to set values into target objects or paths. Likewise, the names of those functions are similar, such as "setPath," "SetPathValue," "setter," and "deepSet." Also 8% of the 37 modules have vulnerability in parsing-related functions, such as "parse_str." Therefore, it can be assumed that internal functions related to setting or parsing without a filtering process are presumably vulnerable to prototype pollution attacks.

6 Evaluation

In this section, we evaluate the experimental results of our tool DAPP with various real-world modules. To assess efficiency and effectiveness, we present the performance and coverage of DAPP in this section. Then, we highlight the root causes for false positives and false negatives.

Even though a substantial number of tools already exist that can find vulnerabilities in Node.js modules, most cannot detect the prototype pollution vulnerability type. Olivier's tool [2] is the only one that can detect a prototype pollution vulnerability. In our evaluation of DAPP, we compared our tool with Olivier's tool and demonstrated how DAPP improves on previously released static analysis tools.

6.1 Experiment setup

We performed the experiment on 100,000 modules to evaluate DAPP. We collected 100,000 module names from NPM using the open-source package "all-the-package-names." This package prints out the list of module names sorted by dependent count. Then, our tool DAPP tested each module. Initially, DAPP installs the target module from NPM. Then, DAPP combines the source code of the target module with the source code of required external modules using webpack. After gathering the entire source code of the target module, DAPP tests whether the target module contains the vulnerable pattern. After DAPP records the result, it removes the target module off of the computer in order to save memory. For our experiment, we used 25 computers equipped with an Intel i7-4790 CPU with 3.60 GHz and 16.0 GB of memory. Four thousand modules were tested on each computer.

6.2 Performance

The results of DAPP over time are shown in Fig. 4. Figure 4a shows the number of modules tested per minute, divided by those analyzed successfully and those with errors. DAPP successfully tested 74,342 out of 100,000 modules with an error rate of 25.68%. Furthermore, the graph shows that, after about 300 min, or 5 h, most of the modules were tested. It takes 6.1748 s (total time) for DAPP to test each module on average. Excluding time to install, remove, and bundle, it takes an average of 0.3554 s (analysis time) for DAPP to test each module. Furthermore, DAPP is able to analyze files with very large sizes. The file size of the target module ranges from 2 KB to 40 MB. Figure 4b shows analysis time and total time for each module based on seconds by file size. DAPP only takes about 100 s to cover a file as large as a 40 MB file.

6.3 Comparison

6.3.1 Comparison with previous prototype pollution detection tool

At NSEC18, Olivier Arteau [2] presented a tool that could detect a prototype pollution vulnerability in Node.js modules. It is a dynamic vulnerability analysis tool that targets only prototype pollution. In Olivier's tool, specific user input that can cause prototype pollution attacks are defined in advance. Specific user input here refers to the number of argu-

Fig. 4 DAPP results over time

Table 3 Types of modules vulnerable to prototype pollution attack which Olivier's tool could not detect

Type	A3	A4	A6	A7	A8	B2	B6	C1	C2	C3	C4	C5	D1	D2	D3	D4	E1	E2	F1	F2	G1	G2	G3	G5	G6	G10
Type1			O					O		O	O	O		O			O	O	O		O	O				O
Type2		O																								
Type3					O				O		O		O										O			
Type4		O		O		O																				
Other	O						O								O	O				O				O	O	

ments of the function and where malicious JSON is inserted. Olivier's tool performs fuzz testing on all the functions with pre-defined input, such as inserting the malicious JSON, "(JSON.parse('{"__proto__":{"test":123}}'), {})" into the arguments of all functions.

Listing 13 is an example of a function vulnerable to prototype pollution that both DAPP and Olivier's tool can detect. Since the vulnerability in this module is not published yet, we cannot disclose the name of the module.

```
1  function set(src, key, value) {
2      var keys, prop, obj;
3
4      keys = split(key, ".");
5      prop = keys.pop();
6      obj = src = JSON.parse(JSON.
           stringify(src));
7
8      while (obj && (key = keys.shift
           ())) {
9          obj = obj[key];
10     }
11     if (!keys.length) {
12         obj[prop] = value;
13         return src;
14     }
15     return undefined;
16 }
```

Listing 13 Detected module

The module in Listing 13 takes three arguments—"src," "key," and "value"—and sets the value of the property "key" of the "src" object to "value." Olivier's tool first executes the function with arguments containing malicious JSON and then checks if the prototype is polluted. However, DAPP detects vulnerabilities by statically checking whether the two AST patterns (Listing 4, Listing 5) are connected in the control flow according to the vulnerability pattern defined previously. The statement that matches AST Pattern 1 on line 9 of Listing 13 are in the loop, and the statement that matches AST Pattern 2 on line 12 of Listing 13 is connected to it.

Olivier's tool detects vulnerabilities while actually executing functions, so the behavior of the function can directly affect the analyst. Because the function executes as long as the parameter passing type matches regardless of the function's behavior, it can cause several problems. We analyzed the top 100,000 modules based on dependents using Olivier's tool, and in total, 58,882 out of 100,000 modules were tested with an error rate of 41.11%. We discovered problems such as infinite loop, file and folder creation, TCP connection attempt, and search by launching a browser. For example, if the function goes through an infinite loop, the function will continue to be executed until the analysis is stopped manually.

Among the 37 modules in which DAPP found vulnerabilities in this study, Olivier's tool could not detect any prototype pollution vulnerabilities in 26 of the modules. We analyzed and produced four main reasons why DAPP could detect the vulnerabilities while Olivier's tool could not. The four main reasons are:

- There are various ways to access the prototype data.
- A higher-order function was used.
- It is a vulnerability that cannot be triggered by simple function call.
- There are different ways to pass function parameters depending on the developer's preference.

We divided the 26 modules into 4 types for each reason. Table 3 shows the 26 modules in Table 2 classified into four types. A detailed explanation of the four reasons is as follows.

In order to exploit prototype pollution vulnerabilities, the attacker should modify the prototype data. As mentioned in Sect. 3.2, prototype pollution vulnerability occurs when attackers can control the internal value of an object through property definition, object merging or object copying. As each target module has a different way of parsing input values, there can be various ways to access the internal value of an object—that is, the prototype data—varying from module to module. Listing 14 shows different ways to access the "__proto__" property, "polluted," and assign the value "true." Thus, pre-defining the malicious user input is limited in that it can only cover a certain number of methods to approach the prototype data. Because of this reason, Olivier's tool could not detect prototype pollution vulnerabilities in 12 modules that turned out to be vulnerable by DAPP. This factor is named "Type1" in Table 3.

```
1 function("__proto__/polluted", true);
2 function({"__proto__.polluted":true});
3 function(["__proto__", "polluted"], true);
4 function("/__proto__/polluted", true);
5 function("__proto__.polluted=true");
6 function("__proto__[polluted]=true");
7 function("__proto__:polluted", true);
```

Listing 14 Various ways to access object properties

Furthermore, Olivier's tool cannot handle a higher-order function, which is a function that returns a function as a result. Olivier's tool firstly extracts the function name and then executes each function with malicious input. However, the result function of the higher-order function can also be vulnerable to prototype pollution vulnerabilities. DAPP can detect prototype pollution vulnerabilities in a higher-order function, because DAPP analyzes according to the source code structure and pattern, whereas Olivier's tool simply extracts function names from the target module, rendering it unable to consider the returned function of the higher-order function. In other words, if a vulnerable function is not loaded as the module is imported, Olivier's tool cannot detect the vulnerability. Listing 15 is an example of using a higher-order function to exploit a prototype pollution vulnerability. This factor is named "Type2" in Table 3, and it is applicable to the module "property-expr" in Listing 15. We discovered that this module is indeed vulnerable when using DAPP, but not when using Oliver's tool.

```
1 var expr = require("property-expr");
2 var f = expr.setter("__proto__.polluted");
3 f({}, true);
4 console.log(polluted); // true
```

Listing 15 Exploitation code for property-expr@2.0.2

Depending on the structure of the code, some exploits require prior work in order for the vulnerable function to work as intended. For example, we have found cases where the users need to additionally specify the behavior of target function or create a supplementary html file. Also, in the case of using a function that belongs to a certain class, the instance of the class must be created before function call. Listing 16 shows an example of a function using variables of the instance created by declaring a new object. The string "__proto__.polluted=true" is assigned with the object creation statement in line 2 in Listing 16, and it is exploited by calling the "parse" function of the object in line 3 in Listing 16. The "parse" function has no parameters in the function itself, so Olivier's tool cannot exploit that function. This factor is named "Type3" in Table 3, and five modules fall into this case.

```
1 const Templ8 = require("Templ8");
2 var tpl = new Templ8( "{{__proto__.polluted=true}}" );
3 tpl.parse();
4 console.log(polluted); //true
```

Listing 16 Exploitation code for Templ8@0.10.5

Lastly, the method of passing function parameters—that is, the number of parameters and the order of the parameters—can be changed simply to suit the developer's preference. Thus, pre-defining the malicious user input only covers a limited number of function formats, which leads to false negatives with Olivier's tool. However, the algorithm in which object deep copy occurs, which is how DAPP finds vulnerabilities, cannot simply be changed. This factor is named "Type4" in Table 3, and three modules are applicable in this case.

6.3.2 Improvement on previous static analysis tool

JSPrime [25] is a JavaScript static security analysis tool presented at BlackHat USA 2013. JSPrime also parses the AST of each target source code by using esprima, same as DAPP, to generate the AST. In order to compare the precision of analysis, we evaluated how correctly each tool can extract and analyze variable and function information from the AST.

When analyzing the JavaScript program, consideration for hoisting and the exact scope of each variable and function is important for accurate data flow analysis. However, even though JSPrime is one of the famous reliable analysis tools, it does not consider hoisting nor accurately extract scope of each variable and function. Considering intricacies of modern JavaScript languages in Sect. 4.5, we prepared variable cases, and Table 4 illustrates how our methodology can handle complicated and practical cases compared to JSPrime. If the scope of the variables are analyzed incorrectly, there can be confusion between variables using the same name within different scopes while analyzing the data flow. This can lead to incorrect analysis of the user input flow.

There are few existing control flow graph generators targeting JavaScript programs. We have extensively investigated and tested existing open-source CFG generators, and to the best of our knowledge, we believe that esgraph stands as the most well-functioning and representative CFG generator to date. Also, numerous other CFG generators depend on esgraph. Thus, our study selected esgraph to create the CFG from each target source code. Nevertheless, esgraph also faced great limitations. Esgraph creates the CFG from AST produced by esprima. Esprima handles 23 types of expres-

Table 4 Comparison with JSPrime: handling complicated and practical cases

Test case	DAPP	JSPrime	Details

/* hoisting variable */
function func(){
a = 1;
var a;
}
	O	X	JSPrime extracts hoisted variable as if it was two different variables

/* var, let scope */
{
var a;
let b;
}
	O	X	JSPrime does not support block-level scope of "let" variables

/* function-use scope */
function func(){
function func2(){
return 0;
}
return 0;
}
	O	X	JSPrime does not analyze the scope of a function

/* duplicate variable name */
var a = 1;
function func(){
var a = 2;
	O	O	Both can distinguish variables that have the same name in different scopes

/* duplicate var variable */
var a = 1;
var a = 2;
	O	X	JSPrime extracts the re-declared variable duplicately

/* implicit global variable */
function func(){
a = 1;
}
	O	X	JSPrime does not define the scope of an implicit global variable as global

/* function expression
assigned to variable */
var a = function(){
return 0;
}
	O	O	Both can handle variable declarations with an assigned function expression

/* function parameter */
function func(x, y) {
return x + y;
}
	O	O	Both are able to extract function parameters

/* class declaration, method */
class Person {
constructor(name) {
this._name = name;
}
	O	X	JSPrime cannot handle class and method declarations

Table 4 continued

Test case	DAPP	JSPrime	Details
get name() { return this._name; } set name(newName) { this._name = newName; } }			

sions and 20 types of statements, but esgraph cannot handle all of the expressions that esprima supports. Also, among the 20 types of statements, esgraph is unable to handle 8 of these statements. Our tool, DAPP, has forked esgraph. We developed and improved esgraph to make it capable of handling all 23 types of expressions and 20 types of statements. Furthermore, we adopted a short-circuit evaluation to improve the accuracy of esgraph.

As a result of this improved static analysis implementation, DAPP was able to produce meaningful results. Table 5 shows the number of detected AST Patterns 1 and 2 and the control flow pattern we defined. Since we conducted the experiment with 25 computers, Table 5 demonstrates the experimental results for each computer. The number of AST Pattern 1 detected is an average of 261 per one computer and 3936 for AST Pattern 2. However, as a consequence of using CFG and patternizing the connection relationship of each AST pattern, the number of pattern detects was significantly reduced. It efficiently narrows the number of candidate modules, which is potentially vulnerable to prototype pollution attacks.

6.4 False positives

We selected 30,000 modules on the basis of the highest downloads recorded in the NPM. Among the 30,000 modules, DAPP found 75 modules vulnerable. Then, we manually validated whether or not each module was vulnerable to prototype pollution attacks. We confirmed that 37 modules were indeed vulnerable among 75 candidates. We were able to define common factors among false positive modules that are definitively not vulnerable to prototype pollution attacks. As such, the root causes for false positives are:

- Filtering malicious user input such as "__proto__"
- Functions that are not exported

Filtering of user input is one of the ways to mitigate prototype pollution. Listing 17 is an example of a code that verifies key values using the "hasOwnProperty" method in the function that assigns a value to an object. The method "hasOwnProperty" returns a Boolean value indicating whether an object has a specific property. If "__proto__" and "constructor" are provided as input, "false" is returned, which can mitigate prototype pollution. In a similar way, prototype pollution can be prevented by using other functions that filter malicious input, such as "__proto__," "constructor," and "prototype" which caused false positives in our experiment.

```
1  function set (obj, keys, value) {
2    var tmp = obj || {}
3    ...
4    for (i = 0; i < keys.length; i++)
         {
5      ...
6      if (tmp.hasOwnProperty(key)) {
7        tmp = tmp[key]
8      }
9    }
10   ...
11   tmp[last] = value
12   ...
13 }
```

Listing 17 mergee@1.0.0

In Node.js, external modules can be loaded and used through the "require" function. Each module can be imported by using this function. This can be applied in the same way when importing JavaScript files. Listing 18 is an example of the code that exports the function so that the function can be used in another JavaScript file. Line 4 in Listing 18 exports a "set" function for use in other modules. If the function is not exported, the function cannot be directly called or used even if the JavaScript file containing the function is imported. Even if our control flow pattern exists in a function, a prototype pollution vulnerability attack cannot occur if the function is not exported, or the user cannot manipulate the parameters when the function is called from other code.

```
1  function set (obj, keys, value) {
2    ...
3  }
4  exports.set = set
```

Listing 18 Function exporting

Table 5 Analysis results of DAPP

List	Number of modules analyzed	Average module size (bytes)	AST Pattern 1	AST Pattern 2	CFG Pattern	Detection	Detection rate (%)	Error rate (%)
com1	3257	53,853	462	7512	65	34	1.04	18.58
com2	3107	76,752	515	3961	80	27	0.87	22.33
com3	3249	29,929	347	7064	48	30	0.92	18.78
com4	3253	38,759	293	3600	40	25	0.77	18.68
com5	3225	50,725	317	8749	44	27	0.84	19.38
com6	3241	59,706	306	3250	51	37	1.14	18.98
com7	3274	32,942	310	5066	34	22	0.67	18.15
com8	3141	21,293	235	2637	38	24	0.76	21.48
com9	3285	34,037	274	10,565	36	25	0.76	17.88
com10	3272	37,731	367	6256	22	19	0.58	18.20
com11	3009	29,473	194	6930	21	13	0.43	24.78
com12	3150	24,522	255	2826	21	19	0.60	21.25
com13	3113	22,419	184	2881	16	13	0.42	22.18
com14	2879	25,231	182	2111	18	16	0.56	28.03
com15	3067	25,999	186	2152	27	21	0.69	23.33
com16	3025	22,212	220	2995	38	16	0.53	24.38
com17	3117	36,630	252	2639	32	27	0.87	22.08
com18	3171	33,425	406	3595	21	10	0.32	20.73
com19	2954	30,721	246	1973	8	8	0.27	26.15
com20	2966	39,677	226	2193	16	17	0.57	25.85
com21	2865	21,599	200	3289	40	20	0.70	28.38
com22	2390	16,015	200	1960	29	14	0.59	40.25
com23	2318	11,448	108	1266	14	9	0.39	42.05
com24	2263	13,766	100	1190	114	12	0.53	43.43
com25	2273	15,811	141	1758	20	8	0.35	43.18
Average	2994	32,186	261	3936	35	19	0.65	25.14

6.5 False negatives

To evaluate false negatives, we analyzed existing prototype pollution vulnerabilities reported via NPM. A total of 65 modules were reported as vulnerable to prototype pollution vulnerabilities as of June 2020. We collected these 65 vulnerable modules and ran our tool, DAPP, against each module to determine whether DAPP could detect prototype pollution vulnerabilities. As a result of our experiment, 10 out of 65 modules were correctly detected by DAPP (5 for prototype pollution source code Pattern 1 and 5 for prototype pollution source code Pattern 2), and an error arose in only one module. Then, we examined 54 modules to analyze for false negatives, about which we discovered that the root causes for false negatives are:

- Use of recursive function
- Lack of analysis on linkage between functions
- Other trivial grammatical problems

Twenty false negatives were related to the use of recursive function in our experiment. A recursive function can be used instead of loop relation, the relation which is suggested in Algorithms 1 and 2 for the purpose of assigning an object's key recursively. For example, "merge-objects" module (Listing 19) is one case of a false negative. DAPP cannot detect these modules because our CFG analyzer does not interpret the statements in a recursive function as self-connected.

```
1  var mergeObjects;
2  mergeObjects = function(object1,
       object2) {
3  var key;
4  ...
5  for (key in object2) {
6      if (...) {
7          object1[key] = mergeObjects
               (object1[key], object2[
               key]);
8      }
9      else {
10         object1[key] = object2[key
               ];
11     }
12   }
13   return object1;
14 };
```

Listing 19 merge-objects@1.0.5

Five other false negatives have exactly the same structure as our suggested pattern. However, the lack of analysis on the use of the "forEach" statement and function-to-function relationship caused false negatives. For example, the "set-value" module (Listing 20) exhibits the prototype pollution vulnerability pattern 1. However, AST Pattern 2 (target[path]=value) is located in the "result" function, which is different from the "set" function where AST Pattern 1 (target=target[prop]) is located. In order for DAPP to interpret these modules as potentially vulnerable, it needs to analyze the function call relationship. Furthermore, other trivial grammatical problems can also cause false negatives, such as the use of an arrow function expression or ternary operator.

```
1  function set(target, path, value,
       options) {
2    ...
3    for (let i = 0; i < len; i++) {
4      ...
5      if (i === len - 1) {
6        result(target, prop, value,
             merge);
7        break;
8      }
9      target = target[prop];
10   }
11   ...
12 }
13
14 function result(target, path, value
       , merge) {
15     ...
16     target[path] = value;
17     ...
18 }
```

Listing 20 set-value@3.0.0

7 Conclusion

Since the number of developers using Node.js is increasing constantly, the security of Node.js is becoming increasingly important. However, in terms of Node.js module vulnerability, most existing tools are only equipped to verify modules via statistical analysis of previously published vulnerabilities using databases. Plus, other tools that can detect new vulnerabilities focus on the types of vulnerabilities that have already been widely studied such as cross-site scripting and injection vulnerabilities.

We introduced a novel methodology used to detect prototype pollution vulnerabilities in Node.js modules in this study. Our static analysis-based methodology is based on AST, CFG, and DFA. Vulnerability patterns, such as vulnerable structures of statements and the proper order of those statements, can be effectively detected using our proposed methodology. We implemented this methodology in DAPP, our proposed automatic vulnerability extrapolation tool for Node.js modules. In particular, we patternized prototype pollution vulnerabilities and performed an automatic analysis on real-world Node.js modules.

Previous analysis tools face problems handling intricate cases of modern JavaScript languages. DAPP overcomes such limitations, and even though many researchers mentioned that prototype pollution vulnerability is difficult to

detect because each target module must be first understood, DAPP is capable of automatically detecting prototype pollution vulnerabilities using static analysis for the first time.

DAPP is proven able to detect the existence of vulnerability patterns in each Node.js module in a matter of seconds. Our evaluation shows that DAPP performs well to the point that it can test a large number of target modules, including large files, in quick succession. Through the experimental validation process, we discovered and reported 37 vulnerabilities and obtained 24 CVE IDs mostly with 9.8 CVSS scores. Experimental results show that prototype pollution vulnerabilities are likely to occur in modules dealing with nested objects, particularly in the setting process or parsing.

Among 37 modules in which we found vulnerabilities in this study, 16 modules have been patched as of December 10, 2020. Based on the newly patched modules, we summarized how prototype pollution can be mitigated with current techniques in Appendix. If more modules are patched, analysis on mitigation for prototype pollution vulnerabilities and the limitations of current methods will be further expanded in our future research. Furthermore, DAPP is optimized for syntactic pattern matching of source code. DAPP is highly scalable, and it is easy to add new vulnerability patterns to the method. Adding and revising patterns while considering false positives and false negatives will increase the precision and detection rate of DAPP. Moreover, vulnerability types other than prototype pollution can be also patternized syntactically, and DAPP will be able to detect a more diverse array of vulnerabilities. Thus, DAPP will contribute significantly to the safe use and development of Node.js modules in forthcoming software.

Acknowledgements This work was supported as part of the Next Generation Security Leader Training Program (Best of the Best) funded by Korea Information Technology Research Institute (KITRI).

Author contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by H.Y.K., J.H.K., H.K.O., B.J.L. and S.W.M.. The first draft of the manuscript was written by H.Y.K. and J.H.K., and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding This work was supported as part of the Next Generation Security Leader Training Program (Best of the Best) funded by Korea Information Technology Research Institute (KITRI).

Availability of data and material The data and material that support the findings of this study are available from the corresponding author, Kyounggon Kim, upon reasonable request.
Code availability The data and material that support the findings of this study are available from the corresponding author, Kyounggon Kim, upon reasonable request.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent This article does not contain any studies with human participants.

Appendix A: Mitigations for prototype pollution vulnerability

After we reported those 37 vulnerabilities through NPM, some of the modules are newly patched to avoid prototype pollution attack. This process helped us to organize how to mitigate prototype pollution attack with today techniques. This section summarizes how to mitigate prototype pollution vulnerability.

Appendix A.1: Use Object.hasOwnProperty

The "Object.hasOwnProperty" method can be used to check the existence of certain properties of the target object. Thus, even if the value of a key such as "constructor" is specified, the reference can be prevented. In this way, unlike the "in" operator which works similar to the "Object.hasOwnProperty" method, the developer can prevent the prototype references. The difference between "Object.hasOwnProperty" and "in" operator is shown in Listing 21.

```
1  let obj = {foo: 1};
2
3  obj.hasOwnProperty("foo"); // true
4  obj.hasOwnProperty("constructor"); // false
5  obj.hasOwnProperty("__proto__"); // false
6
7  "foo" in obj; // true
8  "__proto__" in obj; // true
9  "constructor" in obj; // true
```

Listing 21 Difference between Object.hasOwnProperty and in Operator

Algorithm 4 shows how to mitigate prototype pollution by using "Object.hasOwnProperty" function. However, this method is not available for the new property settings that did not exist originally.

Algorithm 4 Example of using "Object.hasOwnProperty" to mitigate prototype pollution

```
1: function vulnerable_A(obj, key, value)
2: keylist ← key.split(".")
3: e ← keylist.shift()
4: if !obj.hasOwnProperty(e) then
5:     return obj
6: end if
7:     if keylist.length > 0 then
8:         if typeof(obj[e]) ≠ "object" then
9:             obj[e] ← {}
10:             vulnerable_A(obj[e], keylist.join("."), value)
11:         end if
12:     else
13:         obj[e] ← value
14:     end if
```

Appendix A.2: Make "__proto__" empty

Typically, to initialize object in JavaScript, it is usually initialized in the following ways.

```
1       let obj = {};
```

Listing 22 Initialize object

However, when initializing this way, the "__proto__" property of the "obj" in Listing 22 will refer to the prototype of the constructor Object. If the attacker contaminates the "__proto__" property of "obj," the "Object.prototype" is also contaminated. Since the global object also inherits the "Object.prototype," other contexts will refer to the contaminated property which can generate prototype pollution vulnerability.

The fundamental problem here is that "__proto__" of the object literal refers to the "Object.prototype," so contaminating the "__proto__" will also pollutes other objects. Therefore, by making "__proto__" empty, the developer can delete the reference of the "Object.prototype." Listing 23 and Listing 24 show how to initialize "__proto__" to null.

```
1       let obj = { __proto__ : null };
2       console.log(obj.__proto__); //
            undefined
```

Listing 23 How to initialize "__proto__" to null at the same time as initialization

```
1       let obj = Object.create(null);
2       console.log(obj.__proto__); //
            undefined
```

Listing 24 How to initialize "__proto__" to null using the "Object.create" method

Appendix A.3: Filtering by Keyname

This method prevents prototype pollution vulnerability by filtering the values of key with certain names such as "prototype," "__proto__," and "constructor."

Algorithm 5 Example for setting function that filters certain keywords

```
1: function vulnerable_A(obj, key, value)
2: keylist ← key.split(".")
3: e ← keylist.shift()
4: if keylist.length > 0 then
5:     if ["__proto__", "constructor", "prototype"].includes(e) then
6:         return false
7:     end if
8:     if typeof(obj[e]) ≠ "object" then
9:         obj[e] ← {}
10:         vulnerable_A(obj[e], keylist.join("."), value)
11:     end if
12: else
13:     obj[e] ← value
14: end if
```

Algorithm 5 shows how to mitigate prototype pollution by setting a keyword filter. Most of patched modules take this approach. Creating a mitigation with keyword filters can help prevent pollution without being limited to the execution environment.

Listing 25 is a code actually used to prevent prototype pollution of the "dot-prop" module. We have reported the vulnerability on Oct 14, 2019, and the patched source code was committed on Oct 23, 2019. The vendor added a function that checks disallowed keys while splitting object name with dot notation.

```
1       const disallowedKeys = ["
            __proto__", "prototype", "
            constructor"];
2       const isValidPath =
            pathSegments => !
            pathSegments.some(segment =>
             disallowedKeys.includes(
            segment));
3       if (!isValidPath(parts)) return
             [];
```

Listing 25 Filtering by keyname applied to "dot-prop" module

Appendix A.4: Prototype freezing

"Object.freeze" is a JavaScript native function that makes objects read-only. By applying this function to the object "Object.prototype," the properties of the "Object.prototype" will become unchangeable as shown in Listing 26.

```
1  Object.freeze(Object.prototype);
2  Object.prototype.foo = true;
3  console.log(Object.prototype.foo); // undefined
```

Listing 26 Example for prototype freezing

However, this method can cause serious errors when using libraries that require modification of the object prototypes. Therefore, this may not be applicable in all situations.

References

1. Acorn.: acorn (2019). https://www.npmjs.com/package/acorn. Online; Accessed 10 Dec 2019
2. Arteau, O.: Holyvier/prototype-pollution-nsec18 (2018). https://github.com/HoLyVieR/prototype-pollution-nsec18. Online; Accessed 10 Aug 2020
3. Babel-eslint.: babel-eslint (2019). https://www.npmjs.com/package/babel-eslint. Online; Accessed 10 Dec 2019
4. Christensen, H.K., Brodal, G.S.: Algorithms for finding dominators in directed graphs. PhD thesis, Aarhus Universitet, Datalogisk Institut (2016)
5. Davis, J., Thekumparampil, A., Lee, D.: Node. fz: fuzzing the server-side event-driven architecture. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 145–160. ACM (2017)
6. De Groef, W., Massacci, F., Piessens, F.: Nodesentry: least-privilege library integration for server-side Javascript. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 446–455. ACM (2014)
7. Duračík, M., Kršák, E., Hrkút, P.: Current trends in source code analysis, plagiarism detection and issues of analysis big datasets. Procedia Eng. **192**, 136–141 (2017)
8. Esgraph.: esgraph (2019). https://www.npmjs.com/package/esgraph. Online; Accessed 10 Dec 2019
9. Gauthier, F., Hassanshahi, B., Jordan, A.: A ffogato: runtime detection of injection attacks for node.js. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, pp. 94–99. ACM (2018)
10. Georgiadis, L., Tarjan, R.E., Werneck, R.F.: Finding dominators in practice. J. Graph Algorithms Appl. **10**(1), 69–94 (2006)
11. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv.: CSUR **50**(4), 56 (2017)
12. Gong, L., Pradel, M., Sridharan, M., Sen, K.: Dlint: dynamically checking bad coding practices in javascript. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 94–105. ACM (2015)
13. Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 85–96. ACM (2016)
14. Gupta, R.: Generalized dominators and post-dominators. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 246–257. ACM (1992)
15. Hidayat, A. esprima (2018). https://www.npmjs.com/package/esprima, https://esprima.org/. Online; Accessed 10 Dec 2019
16. Holland, B., Santhanam, G.R., Awadhutkar, P., Kothari, S.: Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 79–84. IEEE (2016)
17. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (S&P'06), pp. 6–pp. IEEE (2006)
18. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst.: TOPLAS **1**(1), 121–141 (1979)
19. Madsen, M., Tip, F., Lhoták, O.: Static analysis of event-driven node.js Javascript applications. In: ACM SIGPLAN Notices, vol. 50, pp. 505–519. ACM (2015)
20. Murthy, P.K.: Constructing a control flow graph for a software program, February 3 (2015). US Patent 8,949,811
21. nodejs.: nodejs (2019). https://nodejs.org/en/about/. Online; Accessed 10 Dec 2019
22. Ojamaa, A., Düüna, K.: Assessing the security of node.js platform. In: 2012 International Conference for Internet Technology and Secured Transactions, pp. 348–355. IEEE (2012)
23. OWASP: Owasp dependency check (2019). https://www.owasp.org/index.php/OWASP_Dependency_Check. Online; Accessed 10 Dec 2019
24. Patel, P.R.: Existence of Dependency-Based Attacks in NodeJS Environment. In: Creative Components. 91 (2018). https://lib.dr.iastate.edu/creativecomponents/91. Accessed 10 Dec 2019
25. Patnaik, N., Sahoo, S.: Javascript static security analysis made easy with jsprime. Blackhat USA (2013)
26. Pfretzschner, B., ben Othmane, L.: Identification of dependency-based attacks on node.js. In: Proceedings of the 12th International Conference on Availability, Reliability and Security, p. 68. ACM (2017)
27. Quinlan, D.J., Vuduc, R.W., Misherghi, G.: Techniques for specifying bug patterns. In: Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging, pp. 27–35. ACM (2007)
28. Retire.js.: Retire.js (2019). https://retirejs.github.io/retire.js/. Online; Accessed 10 Dec 2019
29. Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W.: Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. **40**(10), 993–1006 (2014)
30. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for Javascript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498. ACM (2013)
31. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Comput. Math. Appl. **7**(1), 67–72 (1981)
32. Snyk: Prototype pollution (2019a). https://snyk.io/vuln/SNYK-JS-JQUERY-174006. Online; Accessed 10 Dec 2019
33. Snyk: Snyk (2019b). https://github.com/snyk/snyk. Online; Accessed 10 Dec 2019
34. SourceClear: Sourceclear (2019). https://www.sourceclear.com/. Online; Accessed 10 Dec 2019
35. Staicu, C.-A., Pradel, M., Livshits, B.: Understanding and automatically preventing injection attacks on node.js. Technical report, Technical Report TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science (2016)
36. Sun, H., Bonetta, D., Humer, C., Binder, W.: Efficient dynamic analysis for node.js. In: Proceedings of the 27th International Conference on Compiler Construction, pp. 196–206. ACM (2018)
37. Tao, G., Guowei, D., Hu, Q., Baojiang, C.: Improved plagiarism detection algorithm based on abstract syntax tree. In: 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies, pp. 714–719. IEEE (2013)
38. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium, vol. 15, pp. 179–192 (2006)
39. Yamaguchi, F., Lindner, F., Rieck, K.: Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In:

Proceedings of the 5th USENIX Conference on Offensive Technologies, pp. 13. USENIX Association (2011)
40. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 359–368. ACM (2012)
41. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604. IEEE (2014)
42. Zhao, J., Xia, K., Fu, Y., Cui, B.: An ast-based code plagiarism detection algorithm. In: 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA), pp. 178–182. IEEE (2015)
43. Zheng, M., Pan, X., Lillis, D.: Codex: source code plagiarism detection based on abstract syntax tree. In: AICS, pp. 362–373 (2018)