Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability

Mingqing Kang, Yichao Xu, Song Li[†#], Rigel Gjomemo[‡],
Jianwei Hou[*#], V.N. Venkatakrishnan[‡], and Yinzhi Cao

Johns Hopkins University, [†]*Zhejiang University*,
[‡]*University of Illinois Chicago*, [*]*Renmin University of China*

{mkang31, yxu166, yinzhi.cao}@jhu.edu, songl@zju.edu.cn,
{rgjome1, venkat}@uic.edu, and houjianwei@ruc.edu.cn

***Abstract*—Taint-style vulnerabilities, such as OS command injection and path traversal, are common and severe software weaknesses. There exists an inherent trade-off between analysis scalability and accuracy in detecting such vulnerabilities. On one hand, existing syntax-directed approaches often make compromises in the analysis accuracy on dynamic features like bracket syntax. On the other hand, existing abstract interpretation often faces the issue of state explosion in the abstract domain, thus leading to a scalability problem.**

In this paper, we present a novel approach, called FAST, to scale the vulnerability discovery of JavaScript packages via a novel abstract interpretation approach that relies on two new techniques, called *bottom-up* and *top-down* abstract interpretation. The former abstractly interprets functions based on scopes instead of call sequences to construct dynamic call edges. Then, the latter follows specific control-flow paths and prunes the program to skip statements unrelated to the sink. If an end-to-end data-flow path is found, FAST queries the satisfiability of constraints along the path and verifies the exploitability to reduce human efforts.

We implement a prototype of FAST and evaluate it against real-world Node.js packages. We show that FAST is able to find 242 zero-day vulnerabilities in NPM with 21 CVE identifiers being assigned. Our evaluation also shows that FAST can scale to real-world applications such as NodeBB and popular frameworks such as total.js and strapi in finding legacy vulnerabilities that no prior works can.

1. Introduction

Taint-style vulnerability [1]–[3] is a common type of software weakness where an adversary-controlled source input reaches a sensitive sink function without being sanitized, e.g., injection of third-party code from a source into a sink. Examples of such vulnerabilities are OS command injection (where an adversary injects OS commands into the sink), path traversal (where an adversary injects path fragments to access unauthorized resources), and arbitrary code execution (where an adversary injects and executes JavaScript). Taint-style vulnerabilities often lead to severe consequences like server hijacking and information leaks.

The detection of taint-style vulnerabilities requires discovering data flows from attacker-controlled sources to sensitive sinks. The classic syntax-directed static approach is to first construct call and control-flow graphs and then generate and track data flows following control-flow paths. While scalable for some languages, this approach is challenging especially for JavaScript—a prototype-based language with many dynamic features—due to the inherent tradeoff between analysis scalability and accuracy. One of the major issues of existing approaches (with several variations manifested in prior works [4]–[7]) is that the dynamic features of JavaScript often introduce call edges that cannot be resolved without contexts. Examples of such dynamic features include but are not limited to function calls related to bracket syntax with string concatenation and function pointer lookup based on variables defined in a closure. As a result, these approaches may often miss a large number of call edges that are not explicitly visible statically. That is, syntax-directed approaches achieve scalability with compromised analysis accuracy on call edges.

To deal with this problem, one popular research direction [8]–[12] is to use abstract interpretation, which abstractly simulates execution for dynamic call edges. Specifically, abstract interpretation stores call contexts, including dynamic ones in the abstract domain, e.g., a lattice or a graph, so that they can be fetched for call edge resolution. However, while abstract interpretation accurately resolves dynamic call edges with call contexts in the abstract domain, one major challenge is *scalability*: the corresponding code (e.g., those containing vulnerability) may not even be reached within a reasonable amount of time. For example, according to our experiments, ODGen [8] fails to finish analyzing more than 50% of Node.js packages of more than 2K Lines of Code (LOC) and that number jumps to 90% of Node.js packages of more than 60K (LoC) even given enough time (24 hours). Fundamentally, existing JavaScript abstract interpretations [8]–[12] explore all program state-

[#]. The two authors contribute to the paper when they are either studying or exchanging at Johns Hopkins University.

ments, e.g., all conditional branches, thus being prone to state explosion in the abstract domain. That is, the number of objects in the abstract domain may be exponential when several conditional statements are embedded.

Ideally, the best solution for the state explosion problem would be to abstractly interpret only the subset of statements having control- or data-dependency with the taint-style sink. In this ideal case, abstract interpretation would follow control-flow paths from sources to sinks, skipping unrelated conditional branches or it would follow data-flow paths to skip statements unrelated to the sink. However, while intuitively simple, the challenge of this solution is that it requires accurate control- or data-flow graphs, which can only be built by abstract interpretation itself. Therefore, for Javascript and similar languages with dynamic features, there exists a 'chicken-and-egg' problem: first, the construction of an accurate control-flow graph, let alone a data-flow graph, needs abstract interpretation due to dynamic call edges. However, a scalable abstract interpretation approach that skips branches unrelated to the taint-style sinks, needs a control-flow graph with dynamic call edges and a data-flow graph.

Putting aside the accuracy-scalability tradeoff, another major challenge facing existing JavaScript static analysis is asynchronous function calls—especially those involving `Promise` [13], a relatively new yet popular feature, which was introduced in ES6 (2015) and used by 23% of randomly selected 10K NPM packages. The main reason is that a `then` function depends on where the corresponding Promise object is resolved. If the resolution is in a synchronous function, the `then` function is invoked immediately after definition; by contrast, if the resolution is in an asynchronous function like `setTimeout` callback, the `then` function is invoked after the callback function. Currently, none of the existing approaches are able to deal with this new feature.

In this paper, we describe a novel system, called FAST (Fast Abstract Interpretation for Scalability), to detect and exploit JavaScript taint-style vulnerabilities. FAST tackles the scalability-accuracy tradeoff by scaling existing abstract interpretation via two new techniques—*bottom-up abstract interpretation* and *top-down abstract interpretation*. Specifically, the *bottom-up abstract interpretation* constructs a control-flow and call graph including asynchronous edges introduced via `Promise`, and an intra-procedural data-flow graph. FAST's novelty in this step is to *follow function scopes instead of call sequences* (as prior work does) for abstract interpretation. This enables FAST to efficiently analyze a function from the beginning to the end only once, rather than repeating the analysis once per function call. Additionally, to capture JavaScript's complexity of function call resolution, FAST constructs a novel *functional dependency graph (FDG)* that describes how functions create, resolve, or trigger the execution of other functions. FDG enables FAST to accurately and efficiently resolve function calls until all the needed information (e.g., function pointers) is available and annotated.

Top-down abstract interpretation constructs an inter-procedural data-flow graph following specific control-flow and data-flow paths. The insight is that FAST only analyzes a subset of statements that are related to the next function in the control-flow graph, called an intermediate sink, along the control-flow path. That is, the top-down abstract interpretation prunes the program and only analyzes statements with control- and data dependencies on the possible taint-style sink, making it scalable compared with traditional abstract interpretation.

After discovering vulnerable paths to taint-style sinks, FAST verifies whether the vulnerability is exploitable via symbolic constraint solving. Specifically, FAST annotates each object in the abstract domain with a symbol, converts the annotated structure together with object relations to constraints, and asks a solver to determine whether such constraints can be satisfied. If satisfiable, FAST generates an exploit for further human verification; otherwise, FAST tries another control-flow path and repeats the top-down abstract interpretation until all paths are exhausted.

We implemented a prototype of FAST as a flow-, context-, and path-sensitive abstract interpretation tool in detecting taint-style vulnerabilities. Our evaluation shows that FAST detects 242 zero-day, exploitable vulnerabilities on Node.js packages that cannot be detected by state-of-the-art detectors. We responsibly disclosed all the zero-day vulnerabilities to the developers and have obtained 21 CVE identifiers. At the same time, we compare FAST with ODGen and CodeQL [7], [8], two state-of-the-art Javascript vulnerability detectors, and show that FAST is scalable in detecting 10 out of 13 vulnerabilities in large Node.js packages or applications (e.g., Content Management Systems) with more than 10K Lines of Code while ODGen detects none. FAST is also able to automatically generate exploits for about half of the detected vulnerabilities (true and false positives combined), which significantly reduces human efforts in vulnerability confirmation.

We make the following contributions in the paper:

- We propose a two-phase abstract interpretation approach, which generates a control-flow graph in the first phase to guide the second phase for an efficient analysis.
- We implement a prototype, open-source static tool, called FAST, to detect taint-style vulnerabilities.
- Our evaluation shows that FAST significantly outperforms state-of-the-art vulnerability detection tools in reducing false negatives.

2. Motivation and Challenges

In this section, we describe the challenges in analyzing realistic JavaScript packages and motivate FAST's design.

2.1. A Motivating Example

Figure 1 contains a simplified version of an utility Node.js application that we will use to describe the problem and then illustrate our approach. The code compresses files under a given path using a selected algorithm.

Specifically, the `compress` function, called in Line 51, receives in input several options, including the name of

```javascript
1  // util.js
2  const childProcess = require("child_process");
3  const logger = require("./logger");
4  function promisify(fn) {
5   return function (arg) {
6    return new Promise(function executor(resolve,reject){
7      fn(arg, function cb(err, res) {
8          if (err != null) return reject(err);
9          resolve(res);
10     });
11   });
12  };
13 }
14 function execProcess(method){
15   return promisify(childProcess[method]);
16 }
17 async function deflate(options) {
18  const flush_pending = (strm) => {
19   const s = strm.state;
20   // let f(n) = 2*n^2,
      ↪ after k iterations, there are f^k(n) objects
21   let len = s.pending;     // n objs
22   if (len > strm.avail_out)
23    len = strm.avail_out;   // 2*n objs
24    strm.avail_out -= len;  // 2*n^2 objs
25   };
26   for (;;){ // ...       k*k iterations
27    while (...) { //...   k iterations
28      flush_pending(strm); // ...
29    }
30   }
31 }
32 async function compress(options) {
33  switch (options.alg) {
34   case 'zip':
35     return await deflate(options);
36   case 'xz':
37     var command = ["xz", "--stdout", "-k"];
38     if (!options.path)
39       command.push("data");
40     else
41       command.push(options.path);
42     command = command.join("")
43     logger.log(`xz, ${command}`);
44     return await execProcess("exec")(command);
45  }
46 }
47 module.exports = function Util() { };
48 module.exports.prototype.compress = compress;
49 // exploit code, under attacker control
50 const Util = require('util.js');
51 (new Util()).compress({ 'alg': 'xz', 'path': '; touch
    ↪ exploit #' });
```

Figure 1: A motivating example with a command injection vulnerability (the function pointer at Line 7 is the sink).

the compression algorithm (`options.alg`) and the path of the file to compress (`options.path`). Based on the value of (`options.alg`), the function executes lines 34–35 or 37–44. In the latter path, it builds a command from the options and dispatches that command to be executed in Line 44. This path is vulnerable to Operating System (OS) command injection, allowing an adversary to execute arbitrary OS commands.

The code utilizes a popular `promisify` function (Lines 4–13), which converts an asynchronous function (e.g., `childProcess.exec`) to return a `Promise` object. The vulnerable data flow starts from `options.path` (stored as part of Line 32 as the source) to the `command` object at Line 44, and then ends up as the function parameter `arg` of the sink function at Line 7. The exploit code of this vulnerability (generated by FAST and verified manually) is shown at

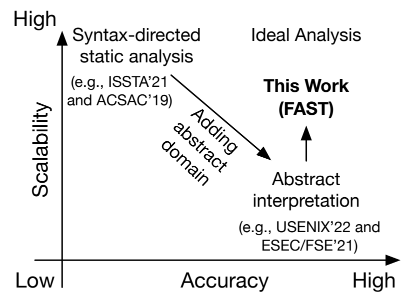

Figure 2: A visualization of accuracy-scalability trade-off in static JavaScript vulnerability detection.

Lines 49–51, which are under attacker control and where an adversary may inject an OS command string instead of a legitimate path as the `options.path` property.

2.2. Vulnerability Detection Challenges

We describe two major challenges in detecting and confirming this taint-style vulnerability. They are i) accuracy-scalability trade-off, and (ii) vulnerability validation.

2.2.1. Challenge I: Accuracy-Scalability Trade-Off

An ideal, static JavaScript vulnerability detection method should be both scalable and accurate. Nevertheless, in practice, real-world JavaScript vulnerability detection tools have to balance the trade-off between analysis accuracy and scalability. This trade-off is depicted in Figure 2. Current approaches are located either on the left top corner (scalable but less accurate) or the right bottom corner (accurate but less scalable) in Figure 2.

On one hand, the accuracy of existing approaches is hindered by JavaScript's large number of dynamic features that strongly depend on runtime values and are challenging to determine statically without call contexts [5], [14]. These include function calls related to `Promise` resolution and rejection, heavy use of function pointers to call functions, and callbacks that depend on function pointers. In our example (Figure 1), such features are manifested in three locations: (i) the function pointer `fn` at Line 7, (ii) the object lookup at Line 15, and (iii) the asynchronous execution of the callback function at Line 7. First, it is challenging to resolve `fn` statically because `fn` is defined as the function parameter in the closure of `promisify`. Second, the resolving of `childProcess[method]` depends on the function parameter `method` at Line 14, which is passed to the function at Line 44 as a string. Lastly, although the callback function `cb` is registered at Line 7, the asynchronous function is only executed at Line 44 when `await` is waiting for all promises to be settled. In fact, classic static analysis [4] cannot resolve either `fn` (Line 7) or `childProcess[method]` (Line 15), leading to missing call edges in the control-flow graph and thus false negatives in the detection.

On the other hand, several approaches use abstract interpretation, which mimics execution of the code in an abstract domain [8], [10] to deal with the dynamic features of JavaScript. However, improved analysis accuracy naturally comes with degraded scalability. More specifically,

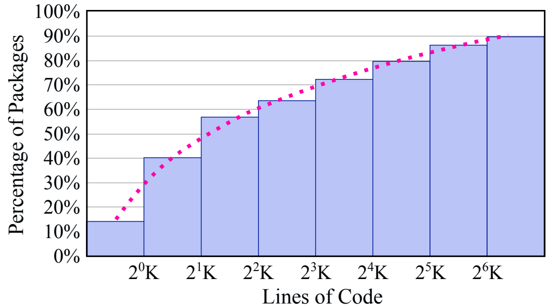

Figure 3: The percentage of packages that ODGen cannot scale to analyze vs. Lines of Code (LoC). When the LoC exceeds 64,000 (i.e., 2^6 K), over 90% of packages have the scalability issues under the analysis of ODGen. Note that we consider ODGen fails to scale the analysis for a given package if the code coverage stays stable for over ten minutes and the analysis does not finish.

abstract interpretation often suffers from the issue of object explosion. That is, the number of involved objects may increase exponentially, leading to a large amount of space to store objects and excessive amount of time to determine each object afterwards. Let us use the `deflate` function (Lines 17–31) in Figure 1 as an example to describe the scalability issue. The listed code is refactored from C/C++ code, which flushes pending outputs as much as possible. Let us assume that each iteration of the embedded loop (Lines 26–30) has n objects. The number of objects becomes $2n^2$ after the `flush_pending` function call because the abstract interpretation stores all the possibilities of conditional statements (Line 22). Then, $2n^2$ becomes the new n in another iteration, leading to an exponential increase of objects.

Scalability Challenge of Abstract Interpretation. We perform two experiments to better understand this scalability problem. First, we analyze Node.js packages with a state-of-the-art abstract interpretation tool, namely ODGen [8], and show the percentage of Node.js packages with the scalability issue as the Line of Code (LoC) increases. Specifically, we consider that an analysis of a given Node.js package has a scalability issue if the code coverage stays stable for over ten minutes and the analysis does not finish. Note that we believe that this is a reasonable estimation of the scalability issue as the ODGen paper adopts 30 seconds as the timeout value threshold.

Figure 3 shows the percentage of Node.js packages having a scalabilty issue vs. LoC. The percentage clearly increases from around 10% with under 1,000 LoC to over 90% with more than 64K LoC. The results show that while ODGen—the state-of-the-art abstract interpretation tool on JavaScript—is capable of analyzing many NPM packages especially those with less than 1K LoC, it cannot scale to big packages when LoC is large.

Second, we identify several code patterns that are difficult to analyze using abstract interpretation, based on manual, empirical analysis of Node.js packages that ODGen fails to scale. In other words, the existence of such patterns

TABLE 1: Percentage of Node.js Packages with Certain Hard-to-analyze Patterns for Abstract Interpretation

Pattern	% Packages
Recursive calls (including indirect ones)	17.62%
Embedded loops	10.13%
Loops + binary operation	29.40%
Loops + conditional statement	27.39%
Loops + conditional expression	8.82%
Loops + boolean OR operation	11.68%
Conditional statement/expression + binary operation	53.74%

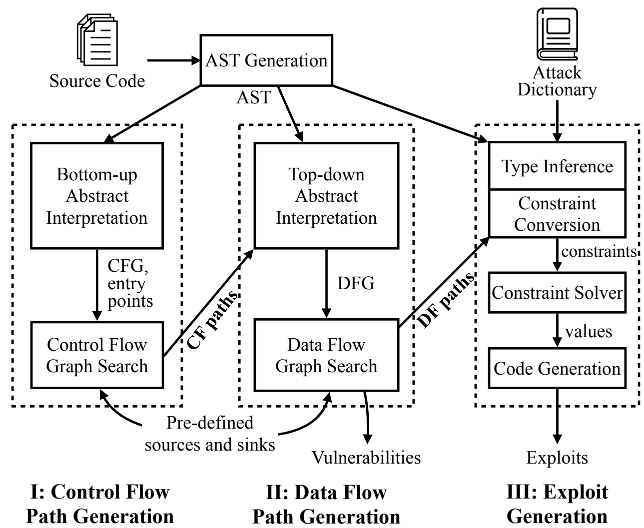

Figure 4: System architecture diagram

will significantly increase the total number of objects. Then, we follow an approach (which is similar to prior work [15]) and measure the percentage of Node.js packages that has the corresponding pattern. Table 1 shows the percentage of packages with such patterns in randomly-selected 10K packages. Many code patterns, such as the combination of loops and binary operations, are very popular, which further motivates the design of FAST.

2.2.2. Challenge II: Vulnerability Validation

The second challenge is how to validate a detected vulnerability as a true positive. Specifically, current approaches, e.g., those adopted by ODGen [8] and Nodest [10], report a vulnerability if there exists a data flow between a source and a sink, and then rely on human efforts to filter false positives, e.g., those with eventual explicit or implicit sanitization. For example, the ODGen authors can only inspect a small portion (i.e., less than 10%) of their reported vulnerabilities due to the total amount of manual work that is needed.

It is challenging to automatically validate vulnerability with exploit generation. Let us take a look at our motivating example in Figure 1. Such validation requires precise modeling of control-flows, e.g., the `switch` case at Line 36 and the `if` statement at Line 38, and data-flows, e.g., `arg` at Line 7, which is `command` at Line 44 and composed at Line 42, as constraints. Then, the validation needs to ensure that all the control-flow constraints can be satisfied and the data-flow allows the injection of third-party code, particularly OS commands in this example.

2.3. Threat Model

Our threat model considers all taint-style vulnerabilities [1]–[3] are in scope, i.e., those that can be modeled as one taint flow from a source (e.g., an object related to user inputs) and a sink (e.g., a sensitive built-in function). This threat model is the same as some prior works, such as Synode [4] and Nodest [10]. Specifically, we consider the following vulnerability types in the evaluation: (1) OS Command Injection, (2) Path Traversal, and (3) Arbitrary Code Execution. Note that some vulnerabilities, such as prototype pollution and internal property tampering, are out of scope of the paper, because they cannot be modeled by one taint flow.

3. Solution Overview

We show an overview of FAST's architecture in Figure 4. FAST has three stages: (i) control-flow path generation that uses bottom-up abstract interpretation to construct the control-flow graph and find a path between entry points and sink function(s), (ii) data-flow path generation that uses top-down abstract interpretation to generate accurate and informative data-flow paths following a control flow path from Stage (i), and (iii) exploit generation to convert data-flow paths into constraints and solve the constraints for exploit generation.

Now, we explain how FAST tackles the aforementioned two challenges in Section 2.2 using our motivating example. **[Scalability] Bottom-up and and top-down abstract interpretation.** First, the bottom-up abstract interpretation performs an intraprocedural analysis of each function scope without following interprocedural paths. This strategy avoids heavy-weight analysis following inter-procedural call edges. Second, the top-down abstract interpretation prunes statements based on control- and data-dependencies, thus skipping statements leading to state explosion. Intuitively, since the sink of our motivating example is at Line 7, which depends on Line 44, our two-phased abstract interpretation avoids the second phase from analyzing the function `deflate` by skipping the `case` branch at Line 34–35, thus scaling the analysis.

Let us explain these two phases in detail using our motivating example (Figure 1). Figure 5 illustrates the first phase of the analysis of the example, bottom-up abstract interpretation. FAST pushes all functions in the current scope into a stack in Step (1) while abstractly interpreting statements in the current scope and interacting with the abstract domain (i.e., Object Dependence Graph [8]). FAST creates call graph nodes for each function defined in the scope, e.g., `promisify` and the anonymous function (Line 5) in Step (2), and links functions together based on call relations, e.g., the anonymous function and the `Promise` constructor in Step (3). Calls that cannot be resolved are dealt with by delaying such resolution until all the information is available. In particular, FAST uses *functional dependency graphs (FDG)*, a novel representation of dependencies between function calls, to capture resolution information. For instance, FAST creates an unresolved lookup path, e.g., LP1 in Step (4) and LP2 in Step (5), waiting for a variable like a function parameter to be instantiated in the abstract domain. Finally, when the variable `method` in `execProcess` is instantiated in Step (6) as a string "exec", FAST uses this information to resolve LP2 as a call to `childProcess.exec`. We describe FDGs in more detail in the next section. The result of the bottom-up abstract interpretation is a control-flow, data-flow, and call graph.

Figure 6 illustrates the second phase of our approach, top-down abstract interpretation of the `compress` function, which operates on the control-flow graph built by the first phase and interacts with an empty abstract domain separated from the first phase. In particular, this phase first extracts source-sink paths and then it builds a data-flow graph by abstractly interpreting only the instructions that have dependencies with the sink. In our example, FAST skips lines 34, 35, 39, and 43, which have no dependencies with the sink. Avoiding such unnecessary abstract interpretation is a key improvement of FAST over prior work.
[Vulnerability Validation] Constraint Solving. FAST generates exploits for a detected vulnerability from two-phased abstract interpretation. Specifically, FAST first annotates all object relations in the abstract domain and then extracts control- and data-flow constraints for a constraint solver. Lastly, FAST generates code as the exploit for the vulnerability validation.

Now, let us use our motivating example to explain the process. Figure 7 shows the annotated object graph with objects as nodes and relations as edges. Then, FAST can directly extract control- and data-flow constraints from the graph. Let us explain the details. Node i is the source and Node j the sink. FAST extracts two types of constraints: data- and control-flow. First, the data-flow constraint (shown as "from the data flow path" in the graph) extraction is a backward traversal of the graph from the sink j to i with the string concatenation operation annotated on Node j until the traversal reaches all the constants. Second, the control-flow constraints (shown as "From condition A and !B in the graph) are annotated on the edge of `command`. Similarly, FAST traverses backward from Nodes A and B to generate both constraints. After constraint extraction, FAST combines all the constraints, asks a solver to provide a solution, and generates exploits (Lines 47–51 in Figure 1).

4. Design

In this Section, we present the design details of FAST's three stages: (I) bottom-up abstract interpretation, (II) top-down abstract interpretation, (III) exploit generation.

4.1. Stage I: Control-Flow Path Generation

The goals of this stage are the creation of a control-flow graph (CFG) of the code and finding a control-flow path between sources and sinks. The novelties of this stage are as follows. First, FAST follows scopes to abstractly interpret each function without following outgoing call edges. Second, it annotates function call dependencies using a novel

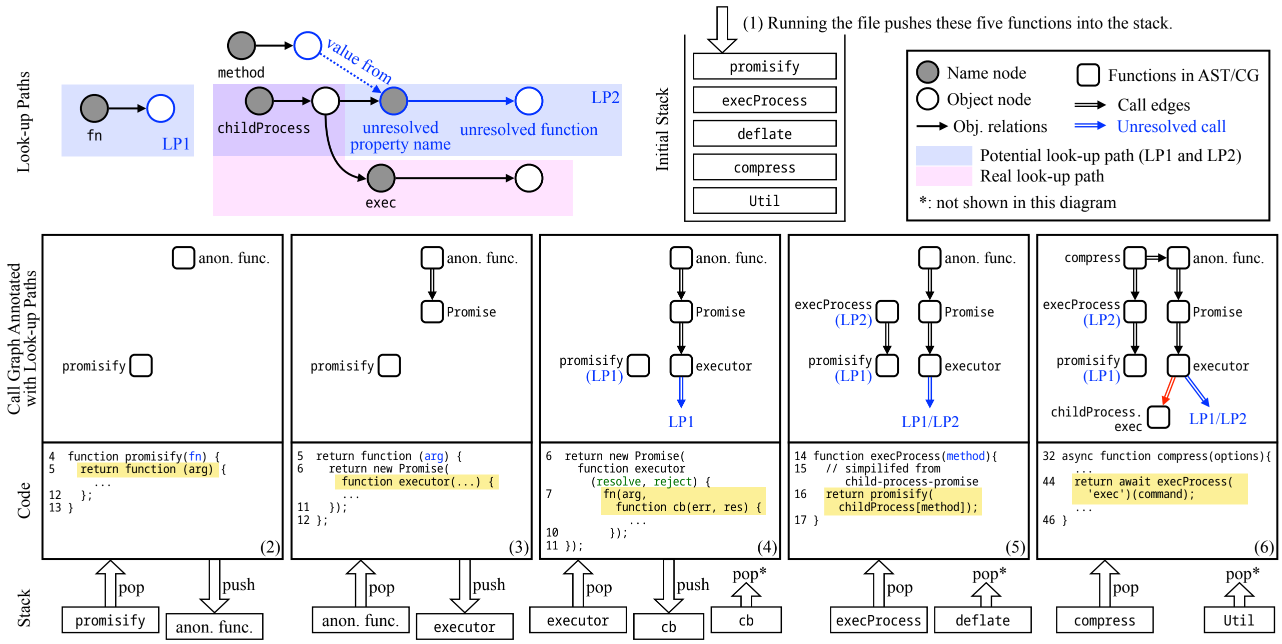

Figure 5: An illustration of bottom-up abstract interpretation using Figure 1 as an example (LP1 and LP2 are two lookup paths of the function pointer `fn` at Line 7. LP2 is resolved at Step (6), leading to a function call to `childProcess.exec`. Note that "Initial Stack" contains five functions that are in the file scope and pushed by FAST during initial scanning.)

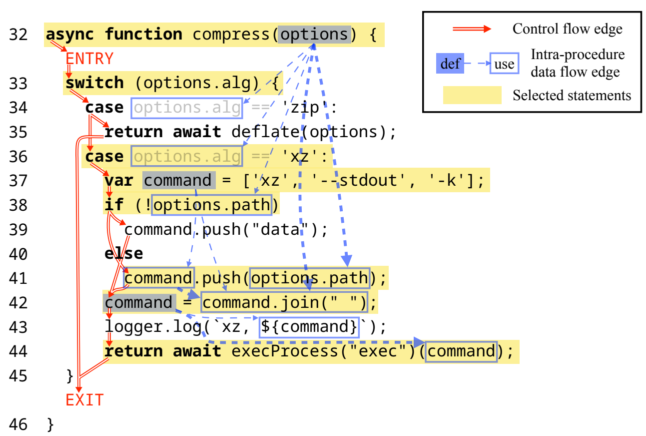

Figure 6: An illustration of top-down abstract interpretation of the `compress` of Figure 1 following control- and intra-procedural data-flow path.

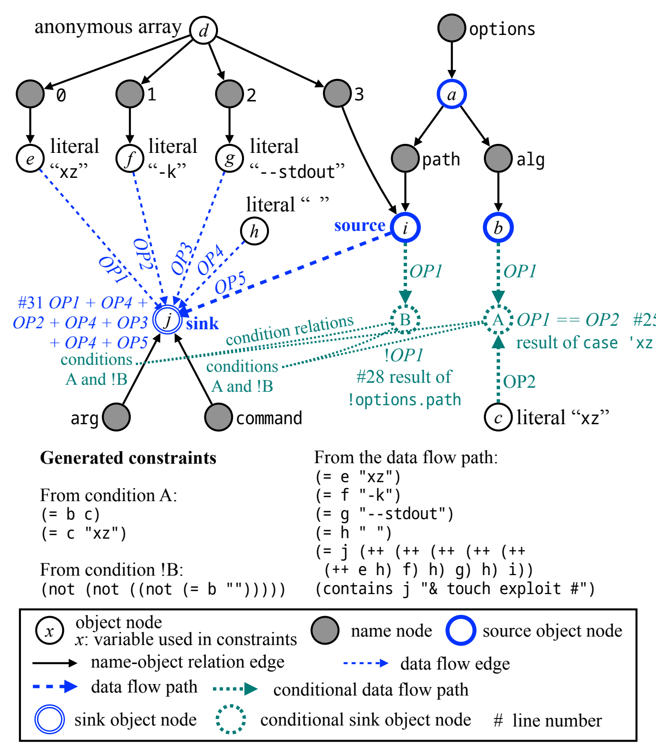

Figure 7: Inter-procedural data-flow graph with control- and data-flow constraints annotated of the motivating example.

functional dependency graph (FDG), and generates accuracy call graph based on FDG. Specifically, FDG delays the challenging task of resolution of function calls until all the information is available, e.g., the value of an unresolved function pointer is passed as an argument of another function call.

We now describe FDGs and how bottom-up abstract interpretation creates FDG, call graphs and intra-procedural control-flow graphs.

Functional Dependency Graph (FDG). A functional dependency graph is a graph whose nodes represent functions or function identifiers (e.g., pointers) and whose edges represent different types of dependencies among those nodes. Given two nodes v_1 and v_2 in the graph, an edge (v_1, v_2) represents the fact that v_2 is resolvable after v_1 is resolved with a call edge. The FDG captures in a concise way the different ways in which JavaScript calls functions and the dependencies between functions and function identifiers. This allows FAST to accurately add call edges when the function identifiers are resolved during abstract interpretation. In particular, FAST uses the paths in the FDG to propagate the resolution of function calls when the callee is known. For instance, function `executor` in Figure 1 contains a function call via a function pointer `fn` in Line 7. The function pointer is passed as a parameter to function `promisify`, which in turn is called by `execProcess`. FDG models a dependency edge (called lookup dependency below) between `promisify`/`execProcess` and `executor`. Then, when another function calls `promisify`/`execProcess`, FAST traverses dependency edges in FDG (i.e., Figure 8 (a)) to resolve `fn` and add corresponding call edges for `executor`.

We categorize FDG dependencies into four main types covering all different scenarios in the ES6 specification [16].

- Lookup Dependency. A lookup dependency is caused by a function pointer lookup (such as `fn` in Figure 1) in a closure or an outer scope where the pointer is used for invocation. These dependencies are represented by edges labeled with *lookup* in Figure 8 (a). Generally, a lookup dependency is determined by a lookup path (LP), which is defined as a series of lookups like `a1[a2][a3]...[ak]`. A lookup path can be a straight line or a compound structure where each `ak = b1[b2]...[bk]`. We call a lookup path final when all objects that variables like `ak` and `bk` point to are either defined in a scope or passed as function parameters. Then, FAST creates a lookup dependency between the function pointer location and the functions with the parameters. For example, Line 15 of Figure 1 shows a relatively complex lookup path `childProcess[method]` (which is also shown as LP2 in Figure 5) where `childProcess` is defined in an outer scope at Line 2 and `method` is passed as a parameter at Line 14 of the `execProcess` function. FAST then creates a lookup dependency between Line 7 of `executor` function and `execProcess` at Line 14.
- Callback Dependency. A callback dependency (called a "trigger") is caused by a callback function invocation, e.g., `cb` at Line 7 of Figure 1, where a function is the parameter of another undecided or asynchronous function call. That is the undecided or asynchronous function triggers this callback function. If the former is undecided, FAST will determine call edges after the former is resolved just like Line 7 of Figure 1; if the former is asynchronous, FAST puts the invocation of latter callback after the former to the event queue of the abstract interpreter because the callback is only registered after the former's execution.
- Return Dependency. A return dependency is caused by an invocation of a function returned by another function. Line 44 of Figure 1 shows such an example: The return value of `execProcess` is invoked as a function at Line 44 with a parameter `command`. That is, FAST determines the call edge of `compress` after `execProcess` is analyzed.
- Promise Dependency. A promise dependency is caused by a `Promise` object. FAST creates a special `Promise` node in the functional dependency graph after the `new` operation and a "new" edge between the node and the creator function. The created `Promise` node has incoming dependencies from the functions that call `resolve` and `reject` and outgoing dependencies caused by `then`. Note that `await` is syntactic sugar of the `then` representation. That is, FAST will create a "then" dependency edge to the statement immediately after the `await` statement.

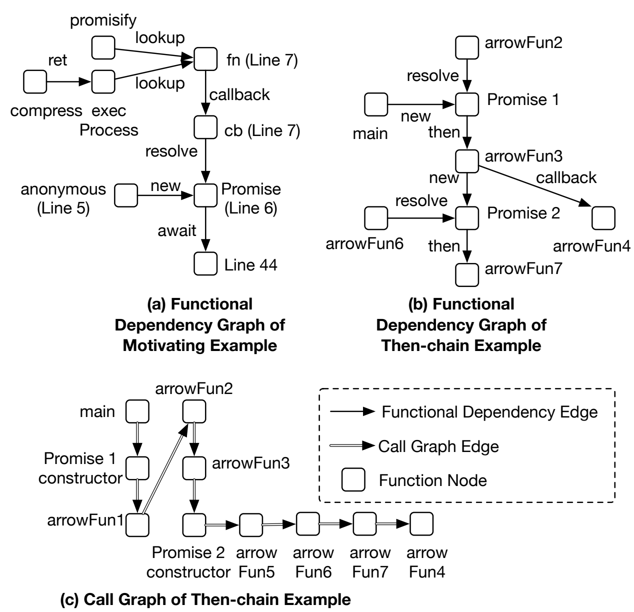

Figure 8: An illustration of functional dependency graphs.

```
1  const myPrms = new Promise((resolve, reject) => { //
   ↪ arrowFun1
2    setTimeout(()=>{ // arrowFun2
3      resolve("done");
4    }, 300)
5  });
6  myPrms.then(value => { // arrowFun3
7    setTimeout(()=>{}/*arrowFun4*/);
8    return new Promise((resolve, reject) => {// arrowFun5
9      setTimeout(()=>{ // arrowFun6
10         resolve(value);
11     }, 300)
12   })
13 }).then(value => { //arrowFun7
14   console.log(value);
15 });
```

Figure 9: An illustration of `then` chaining in `Promise`-related call graph construction (The global scope of this example is called "main" later in the paper).

To better illustrate `Promise` dependencies, we also show a `then` chain example in Figure 9. The example creates a new `Promise` at Line 1 and then two `then` functions that are chained together at Line 6 and Line 13. The example has seven arrow functions that are annotated as comments in the figure. Figure 8 (b) shows the functional dependency graph of this `then` chain example. The main scope creates `Promise 1`, which is resolved by `arrowFun2`. Then, `Promise 1` triggers the `then` function `arrowFun3`. `arrowFun3` triggers `arrowFun4` as an asynchronous function and also creates another `Promise 2`. `Promise 2` is resolved in `arrowFun6` and triggers `arrowFun7`.

Graph Creation. We describe how FAST uses bottom-up abstract interpretation to generate functional dependency, call edges, and intra-procedural control-flow edges, as well as to resolve the dependencies. We describe the generation based on different types of statements. A detailed algorithm can also be found in Appendix A.

- Function calls. There are four types of function calls: directly resolvable, pending, return-related, and callbacks (parameter-related). FAST adds corresponding call or dependency edges to the FDG based on the type. If the function is immediately resolvable, e.g., a direct function call, FAST adds the corresponding call edge. Otherwise, FAST adds a dependency edge and waits for the dependent function for adding a call edge.
- Function definitions. There are three types of function definitions: callback, return function, and function expression. FAST pushes newly defined functions onto the stack for further abstract interpretation. At the same time, FAST also tries to resolve functions that are dependent on the newly defined function. For example, if a function is defined as a return value, FAST follows dependency edges, finds its invocation location, and adds call edges.
- Promise-related statements. There are four types of Promise-related statements: new, then, await, and resolve/reject. FAST adds dependency edges based on the statement type. If the statement is a resolve/reject, FAST will resolve the corresponding the corresponding Promise and then trigger the "then" function if it is present.

Having captured all the dependencies between possible function calls in the FDG, when FAST encounters dependency edges, it is able to execute a *resolve-and-trigger* strategy. In particular, once a single node is resolvable, FAST will follow paths formed by dependency edges to resolve all the pending call edges related to those paths. Let us review our motivating example in Figure 1 and its functional dependency graph in Figure 8 (a) again. When the parameter value of `execProcess` becomes available in the `compress` function, FAST resolves `fn` and then `cb` and then the Promise and `await` in a chain. Similarly, if we look at our `then` chain example in Figure 9, Figure 8 (c) shows the call graph generated from Figure 8 (b), where `arrowFun2` triggers a chained call edge until `arrowFun7`.

4.2. Stage II: Data-flow Path Generation

In this stage, FAST finds a data-flow path between a source and a sink following a specific control-flow path. Details of such control-flow path discovery after bottom-up abstract interpretation can be found in Appendix B. We describe three components here: (*i*) intra-procedural backward slicing, (*ii*) top-down abstract interpretation and (*iii*) data-flow search and vulnerability detection.

First, we describe intra-procedural backward slicing. FAST generates intra-procedural data flow for each function and then performs a backward slicing based on the intermediate sink function, i.e., the next function call in the control-flow path, to skip unrelated statements. Let us look at our motivating example again. Figure 6 shows the backward slicing results (highlighted statements) of the `compress` function of our motivating example in Figure 1 following a control-flow path leading to the final sink Line 7. We marked all the intra-procedural data-flow edges related to the intermediate sink at Line 44: Anything unrelated to `command` (e.g. Line 43) or not on the control-flow path (e.g., Line 39) is filtered. This intra-procedural data-flow slice is used for our top-down abstract interpretation.

Second, FAST follows a specific control-flow path and an intra-procedural slice selected based on the control-flow path to abstractly interpret a subset of program statements. Such a procedure is called a top-down abstract interpretation because it follows the call sequence, especially the caller-callee relations. We describe two substeps of top-down abstract interpretation.

Step 1. Object-level data-flow generation. First, FAST generates data flow between different objects (i.e., object-level data flows). Specifically, consider the following two statements: (1) `p = a;` and (2) `o = p + b;`. Both `p` and `a` point to the same node, which solves the points-to information. Then, FAST creates a data flow between the node that `p` and `a` point to and the one that `o` points to. So FAST does for `o` and `b`. The plus operator is also annotated atop of the object-level data-flow edge for the third stage to generate exploits. Note that similar data-flows are created for template strings (e.g., `` `string${var}` ``) and built-in function (e.g., `Array.prototype.join`) and operations are annotated on the edge as well.

Step 2. Path-sensitivity information collection. FAST stores path-sensitivity information as an object in the object-level data-flow graph and pushes the object onto a so-called branch stack. Consider an `if` statement with a condition `a && b`. FAST creates an object node to denote the result of `a && b` that both object nodes of `a` and `b` have a data-dependency upon. Later on, when an object is created under a certain branch, the object is attached with a tag that represents the current stack, i.e., all the path-sensitivity related objects in the stack. Then, when FAST finishes the abstract interpretation of the branch, FAST pops the corresponding path-sensitivity object out of the branch stack.

Lastly, FAST performs a data-flow path search to determine the connectivity between sources and sinks. FAST takes in input the list of sources and sinks and performs a Depth First Search (DFS) over the interprocedural DFG. The final result of this step is a set of source-sink data-flow paths to indicate a possible vulnerability.

4.3. Stage III: Exploit Generation and Validation

The goal of this stage is to generate an exploit based on the extracted data-flow path and the detected vulnerability.

If a path is exploitable, FAST considers the vulnerability as exploitable. Otherwise, FAST repeats Stage I to try another control-flow path. Stage III is composed of three steps: type inference, constraint generation, and exploit code generation.

Type Inference. One challenge in using constraint solving with is that of translating instructions into the language of the constraint solver. Specifically, the main issue is that JavaScript is weakly and dynamically typed but constraint solvers (such as the Z3) are strongly, statically-typed. Therefore, when FAST generates constraints from JavaScript, it also needs to provide type information to the solver. To address this issue, FAST incorporates methods for inferring variable types from known types. Particularly, FAST uses two specific inference methods: *forward* and *backward*. Forward inference follows the data flow from an object to its uses in built-in functions and derives the type based on the specific built-in. For example, if an object is used in `childProcess.exec`, FAST can infer that this object is a string type. Second, backward inference is that FAST follows the data-flow in backward from an object and iterates through all the objects related to the object in the data flow. For example, say, FAST is inferring the type of `b` in `b = a + "str"`. When FAST goes backward and finds that "str" is of a string type, FAST then infers both `a` and `b` are of a string type for the solver.

Constraint Generation. The second step is to generate constraints from the data-flow path extracted from Stage II. We classify constraints in FAST into three categories. (i) Sink object constraints, which are converted from the sensitive sink object, e.g., parameters of the sink function. Such constraints have two parts: constraints on the sink object itself, and constraints on the sink object and source objects. The former is vulnerability specific: for example, if the vulnerability is command injection, FAST may add a constraint based on a vulnerable dictionary like `(str.contains o "; touch exp #")`. The latter is based on a backward traversal of the sink object in the data-flow graph to reach sources. (ii) Path constraints, which are converted from path objects stored in the branch stack as discussed in Section 4.2. FAST loops through all the objects in the stack and generates such constraints. The generation process is similar to sink object without the vulnerability-specific constraint. (iii) Constant constraints, which are generated during the former two when FAST can determine the value of a certain object from a constant value. A detailed algorithm can be found in Appendix C.

Exploit Code Generation. The third step is to generate exploit code based on the constraints extracted from the second step. FAST feeds all the constraints into a solver (such as Z3) and obtains a solution. Then, the next step is to generate an exploit code, which has two sub-steps: function call preparation and exploit validation. First, when the solver gives values for each source object, FAST needs to first find the correct way to call the function. Specifically, FAST finds the definition of the function object and then searches through its parent object (e.g., `parent.child`) until it finds an external object, such as `module.exports`. Next, it adds the solution for a source object as the parameter to the function call. Second, FAST validates the generated exploit by running the exploit code. Take command injection for example. FAST checks whether an exploit file is created under the current directory if the exploit code is to touch a new file.

4.4. Implementation

We implemented FAST with 4,166 Lines of Code (LoC) in Python and 274 LoC in JavaScript. Our open-source implementation is available at this GitHub repository: https://github.com/fast-sp-2023/fast. The abstract syntax tree (AST) generation is based on Esprima [17]. The graph representation reuses the graph component from the open-source project ODGen [18] and the graph library NetworkX [19]. The constraint solving is based on Z3 Theorem Prover [20], which includes Z3-str, now an official component of Z3. Note that all third-party code is excluded from the above LoC.

5. Evaluation

Our evaluation answers five Research Questions (RQs):

- RQ1 [Zero-day]: How many zero-day vulnerabilities can FAST detect but state-of-the-art approaches cannot?
- RQ2 [FP&FN]: What are FAST's false negatives (FNs) and false positives (FPs) in detecting vulnerabilities?
- RQ3 [Scalability]: How scalable is FAST in detecting vulnerabilities in large-scale packages?
- RQ4 [Call Graph]: How many new call graph edges can FAST generate compared with state of the art?

5.1. Experimental Setup

Datasets. We collect and form three datasets. (i) Real-world Node.js packages with the first 100,000 NPM Node.js packages ranked by number of dependencies. (ii) Vulnerability benchmark with 391 vulnerable Node.js packages with 391 taint-style vulnerabilities from three types, i.e., OS command injection, arbitrary code execution and path traversal. The packages in this benchmark come from the ODGen repository [18], the Nodest paper [10], and legacy CVEs in 2021 and 2022 (which is after the ODGen paper). (iii) Scalability benchmark with 13 vulnerabilities in eight packages-version pairs with more than 10K LoC (excluding third-party code). We collect this dataset by surveying popular CMSes [21] in JavaScript and finding their in-scope, taint-style vulnerabilities with confirmed exploit code.

Experimental Environment. All our experiments are performed on a server with 192 GB memory and Intel Xeon E5-2690 v4 2.6GHz CPU with 14 cores. We run 16 threads of FAST at the same time for the real-world Node.js packages to speed-up the analysis. We evaluate the following tools in our experiment. First, there are two variations of FAST: FAST-det and FAST-exp. FAST-det, the default version of FAST, detects a vulnerability if a data-flow path is found between a source and a sink. FAST-exp reports a vulnerability found by FAST-det as exploitable if it can successfully generate an exploit. Second, we also include

TABLE 2: [RQ1] A breakdown of confirmed zero-day vulnerabilities found by FAST but not state-of-the-art approaches (SOTAs), i.e., neither ODGen [8] nor CodeQL [7] detects them, on 100k real-world Node.js packages.

Vulnerability	FAST-det& ¬SOTA	FAST-exp& ¬SOTA	FAST-det& SOTA
Command Injection	113	92	177
Arbitrary Code Exec.	68	39	29
Path Traversal	61	51	24
Total	242	182	230

TABLE 3: [RQ2] False Positive and Negative Rate Comparison between FAST and ODGen.

	FAST-det	**FAST-exp**	ODGen	CodeQL
False Negative Rate	16.6%	58.3%	43.7%	35.3%
False Positive Rate	11.8%	0%	23.3%	27.8%

two state-of-the-art (SOTA) tools: (i) ODGen [8], the SOTA abstract interpretation tool, and (ii) CodeQL [7], the SOTA, industry-level syntax-directed tool.

5.2. RQ1: Zero-day Vulnerabilities

In this subsection, we answer the research question on how many zero-day vulnerabilities FAST can detect while four SOTA approaches (mentioned in our experimental setup) cannot. We run all the tools upon our real-world Node.js packages. Then, we consider a detected vulnerability as zero-day if a human expert confirms the vulnerability with a generated exploit and we cannot find any information about the vulnerability online. We also have responsibly reported all zero-day vulnerabilities to corresponding developers and gave them 45 days for fixes. So far we have obtained 21 CVE identifiers; we anonymize them for the purpose of double-blind submission. Table 2 shows a list of zero-day vulnerabilities that is broken down by vulnerability type. In total, FAST detects 242 zero-day vulnerabilities and exploits 182 of them.

A Case Study. We use fastboot-gcloud-storage-downloader@1.0.0., which is a downloader for the FastBoot App Server to download and unzip deployed applications from Google Storage, as an example. The package uses "exec" to download and unzip deployed applications but fails to sanitize inputs potentially controlled by an adversary, thus leading to an OS command injection vulnerability. FAST-det successfully detects this package as vulnerable and then FAST-exp automatically generates an exploit. By contrast, neither ODGen nor CodeQL detects this vulnerability because of the heavy use of `Promise` and template string, leading to missing control- or data-flow paths.

5.3. RQ2: False Negatives and Positives

Overview. Table 3 shows an overview of the comparison between all four approaches. False negatives (FNs) are evaluated on the vulnerability benchmark (because we have the ground truth information) and false positives (FPs) on the first 10K Node.js packages in the real-world dataset (because it contains non-vulnerable packages and we do not have any ground truth).

TABLE 4: [RQ2-FN] False negative comparison on vulnerability benchmark between two variations of FAST, ODGen [8] and CodeQL [7] on vulnerability benchmark.

	Cmd Injection		**Code Execution**		**Path Traversal**		**Total**	
	TP	**FN**	**TP**	**FN**	**TP**	**FN**	**TP**	**FN**
FAST-det	169	18	42	12	115	35	326	65
FAST-exp	86	101	13	41	65	85	164	228
ODGen	107	80	24	30	89	61	220	171
CodeQL	122	65	21	33	110	40	253	138

On one hand, FAST-det outperforms all SOTAs with the lowest FP and FN rates. FAST-det outperforms existing abstract interpretation (i.e., ODGen) because of our improvement on scalability. FAST-det outperforms existing syntax-driven approaches (e.g., CodeQL) because abstract interpretation can solve dynamic JavaScript features like dynamic object lookups using bracket syntax. On the other hand, FAST-exp has zero false positives but relatively high false negatives because it generates exploits by solving all the constraints. In many cases, FNs are because Z3-solver does not come up with a solution while our human being can solve them manually. Note that we count packages with intended functionalities as true positives of analysis but not zero-day vulnerabilities because we are calculating true positives of our program analysis, which is performing correctly. The total number is also small, i.e., only nine arbitrary code execution among all vulnerable packages.

False Negative Breakdown. Table 4 shows a breakdown of false negatives of different tools. FAST-det outperforms existing works in all vulnerability categories. The main reason of FN for FAST-det is that there are some unmodeled sources or sinks, leading to missing data flow. ODGen's FNs are mainly because of code coverage, i.e., much vulnerable code may not be even reached during the analysis. CodeQL's FNs are due to dynamic JavaScript features, such as function calls related to bracket syntax.

False Positive Breakdown. Table 5 shows a breakdown of FAST's false positives by vulnerability types and its comparison with SOTAs. FAST outperforms SOTAs on all types of vulnerabilities. The main reason of FPs of FAST-det is that many applications contain either control- or data-flow sanitizations, which make the detected vulnerability unexploitable. This also shows that we need FAST-exp to help the exploitation. As a comparison, CodeQL's FPs are higher than FAST-det because there are over-approximations of control- and data-flows due to lack of abstract interpretation in a syntax-driven approach.

5.4. RQ3: Scalability

In this subsection, we evaluate the scalability of FAST in detecting vulnerabilities of the scalability benchmark. There are two things worth noting here. First, although there is only one CVE identifier for strapi@4.0.8, there are two

TABLE 5: [RQ2-FP] False positives of two variations of FAST in analyzing 10k real-world Node.js packages.

	Cmd Injection		Code Execution		Path Traversal		Total	
	TP	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
FAST-det	56	4	16	5	3	1	75	10
FAST-exp	35	0	6	0	3	0	44	0
ODGen	17	5	13	5	3	0	33	10
CodeQL	52	13	12	8	1	4	65	25

TABLE 6: [RQ3] Detection and exploitation status of FAST and ODGen of the scalability dataset (>10K LoC).

Package	Version	LoC	Vulnerability	FAST	ODGen
strapi	4.0.8	196,338	CVE-2022-0764†	2/2	0/2
strapi	3.0.0-beta.17.7	85,520	CVE-2019-19609†	2/2	0/2
ghost	4.3.0	71,041	GHSA-wfrj-qqc2-83cm‡ CVE-2021-29484	1/1 0/1	0/1 0/1
NodeBB	1.4.0	70,950	npm:nodebb:20161120‡	1/1	0/1
NodeBB	0.6.1	46,092	npm:nodebb:20150413‡	1/1	0/1
total.js	3.4.5	40,593	CVE-2020-28494	1/1	0/1
			CVE-2021-23344	1/1	0/1
			CVE-2021-23389* CVE-2021-32831*	1/1	0/1
total.js	3.2.2	33,109	CVE-2019-8903 CVE-2019-10260	0/1 0/1	0/1 0/1
hapi	0.15.9	12,681	npm:hapi:20130320‡	1/1	0/1
Total	–	–	–	11/14	0/14

†: The CVE maps to two vulnerabilities with two sinks and two different exploits.
‡: We use snyk-id because there are no CVE identifiers.
∗: These two CVEs map to the same vulnerability.

vulnerable sinks and two different exploits, i.e., two vulnerabilities. The same applies to strapi@3.0.0-beta.17.7. Second, interestingly, although there are two CVE identifiers (CVE-2021-23389 and CVE-2021-32831) for total.js@3.4.5, there is only one vulnerability. Note that these two CVEs are follow-ups of CVE-2021-23344, because the patch of CVE-2021-23344 is also vulnerable.

Table 6 shows the detection results of FAST and ODGen. FAST is able to detect ten out of 14 vulnerabilities, even in strapi with almost 200K LoC. By contrast, ODGen detects none of these vulnerabilities after analyzing each vulnerability for one day (sometimes it may crash). FAST also misses the detection of three vulnerabilities. The main reason is the lack of modeling of corresponding sources or sinks by FAST. Take CVE-2021-29484 [22] in ghost@4.3.0 for example. It is client-side vulnerability starting from a `postMessage` channel as a source, which was not modelled by FAST. The two vulnerabilities of total.js@3.2.2 are similar. We do not find sources and sinks modeled by FAST that are related to the vulnerabilities. Furthermore, we cannot reproduce either vulnerability, which prevents us from understanding the real source or sink.

At the same time, we also show the total finish time of FAST vs. the number of Abstract Syntax Tree (AST) Nodes of our scalability and vulnerability benchmark combination in Figure 10. When the number of AST node increases, the finish times of both FAST-det and FAST-exp increase. The increase is linear as we show the trend in a line fit (both x- and y-axes are in log scale). The finish time of FAST-exp is slightly higher than FAST-det because of the additional exploitation time.

We also show a cumulative distributional function (CDF) graph of the performance overhead of both FAST and FAST-det on our vulnerability benchmark in Figure 11. The median performance overheads are 26.3 seconds and 31.6 seconds for FAST-det and FAST-exp respectively. There are three things worth noting here. First, FAST finishes analyzing most packages with one minute. Second, the performance overheads of FAST-det and FAST-exp are similar, i.e., exploit generation is relatively fast. Lastly, the largest overhead is 3,401 seconds (almost an hour) for FAST-exp (not shown in figure) in analyzing api@0.15.9 with 12K Lines of Code because of the heavily uses of dynamic calls. We also have a performance breakdown by stages in Appendix D.

5.5. RQ4: Call Edges

In this research question, we compare call edges produced by FAST and existing approaches, namely the open-source implementations of (i) ODGen [8], [18], an abstract interpretation approach, and (ii) JS Call Graph [23], [24], a syntax-directed approach. Note that we choose JS Call Graph because some follow-up works are either entirely closed source [14] or does not provide a call graph for comparison [7].

Our methodology is as follows. We run all three approaches on our vulnerability benchmark, produce call edges and then compare the results produced by three approaches. We then manually inspect all the edges produced by three approaches for correctness. The inspection of all the edges takes a graduate student approximately 230 hours. Lastly, we show the breakdown of false positive and negative edges of each approach.

First, Table 7 shows false positives and negatives of call edges produced by all three approaches. FAST outperforms both ODGen and JS Call Graph (JSCG) in terms of FPs and FNs. Let us start from FPs, i.e., incorrect call edges. The FPs of JS Call Graph are the highest because it adopts a syntax-based, name-driven matching. Therefore, JS Call Graph often mismatches a function call to a definition under an incorrect scope. Say there are two functions called `foo` under different scopes and JS Call Graph often chooses a wrong one. By contrast, the FPs of ODGen and FAST are relatively smaller. The main reasons are unsupported features. For example, if FAST cannot recognize whether a callback function is synchronous or asynchronous, FAST will default it as synchronous, leading to potential FPs.

We then discuss FNs, i.e., missing call edges. ODGen misses many call edges because of code reachability in the analysis. Specifically, ODGen often has an exponential number of nodes during analysis, leading to a scalability issue as shown in the motivating example of Figure 1. The FNs of JS Call Graph are also mostly caused by scope mismatch, i.e., it chooses the wrong function under a different scope.

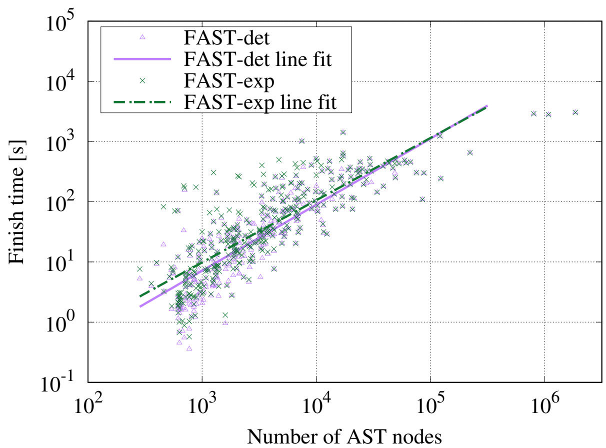

Figure 10: [RQ3] Detection Finish Time vs. the Number of Abstract Syntax Tree (AST) Nodes on a Combination of Vulnerability and Scalability Benchmarks

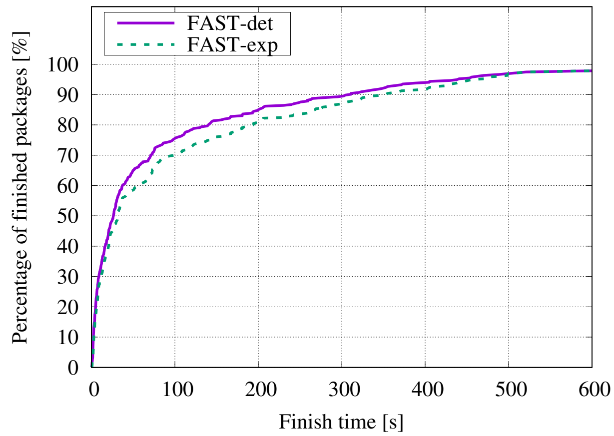

Figure 11: [RQ3] Cumulative Distribution Function (CDF) of Performance Overhead on a Combination of Vulnerability and Scalability Benchmarks

TABLE 7: [RQ4] Call edge breakdown of ODGen, JS Call Graph, and FAST (%Edges for True Positives: TP/(TP+FP); %Edges for False Positives: FP/(TP+FP); %Edges for False Negatives: FN/(TP+FN))

	ODGen		JS Call Graph		**FAST**	
	#Edges	%Edges	#Edges	%Edges	#Edges	%Edges
True Positives	4,137	89.4%	3,617	78.6%	6,831	92.8%
False Positives	492	10.6%	985	21.4%	531	7.2%
Scope mismatch	0	0%	985	21.4%	0	0%
Unsupported feature	380	8.2%	0	0%	479	6.5%
Implementation bug	112	2.4%	0	0%	52	0.7%
False Negatives	3,059	42.5%	3,579	49.7%	365	5.1%
Reachability	1,824	25.3%	0	0%	0	0%
Function pointers	0	0%	280	3.9%	0	0%
Scope mismatch	0	0%	2,190	30.4%	0	0%
Unknown objects	137	2.0%	0	0%	137	2.0%
Implementation bug	1,098	15.2%	1,109	15.4%	228	3.1%

There are two reasons of FNs for FAST. On one hand, FAST cannot create call edges for an unknown object, e.g., one passed through a function parameter without a formal definition. On the other hand, our current implementation still has an engineering bug in creating call edges for some function calls in embedded ternary operator. We will fix this bug in the future.

Second, we also show a Venn graph of all the edges produced by three approaches in Figure 12. On one hand, the overlaps between ODGen and FAST are large because both are based on abstract interpretation. The missing part from ODGen, as described, is mostly because of reachability. On the other hand, JS Call Graph has many unique edges compared with FAST and ODGen. The false positive rate for the unique edges is very high and the main reason is scope mismatch as described above.

6. Discussion

Ethics. We have contacted vulnerable Node.js package developers and given them 45 days for a fix if we can find their contact. At the same time, we are also working with a

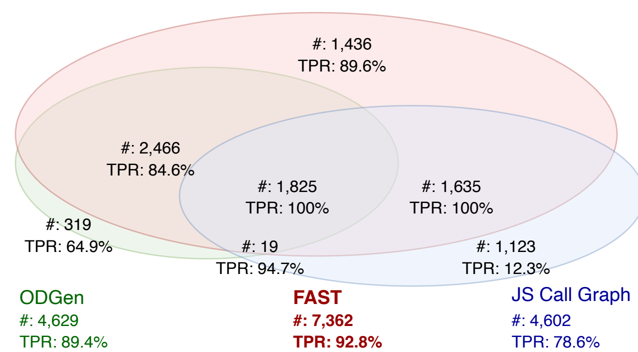

Figure 12: A Venn diagram showing the overlaps among call edges produced by three approaches (TPR: TP/(TP+FP)).

CVE Numbering Authority (CNA) to not only obtain CVE identifiers but also contact corresponding developers for fixes. Our practice follows industry standard in vulnerability disclosure [25] and our organization's policy.

Loops. Loops are also a major challenge leading to scalability issues in prior works. FAST is able to reduce the number of abstractly interpreted loops due to two reasons. First, the bottom-up abstract interpretation only analyzes loops that are related to function calls, e.g., function pointer lookups and invocations in a loop, thus skipping many loops related to data operations. Second, the top-down abstract interpretation only analyzes loops that have control- or data-dependencies with the sink, thus skipping those that do not. The loop analysis follows two strategies: if the looping number is known (e.g., a constant array), FAST extensively loops through every element; if unknown, FAST uses a threshold, i.e., three, for the loop.

Vulnerability Exploitation. The purpose of FAST-exp is to filter packages that can be automatically exploited, thus reducing human efforts in confirming vulnerabilities. The current implementation can reduce the amount of human works by about half, while still leaving the rest as human work. The major reason of failures is that Z3 solver fails to produce a solution and times out based on provided constraints, but a human being can come up with a solution with the constraints. We leave this as our future work.

Analysis Soundness. While FAST significantly improves the scalability of existing abstract interpretation, we would like to point out that FAST—just like all existing static analysis—is unsound [15]. Our manual inspection shows that unsoundness, particularly False Negatives, is primarily caused by three reasons in practice: (i) lack of modeling of built-in functions (>90%), (ii) AST parsing errors from Esprima (e.g., public class field that is supported by many browsers and Node.js [26] and to be included in ES2023 [27]), and (iii) the pruned path in the second phase is still heavyweight to analyze. In theory, such unsoundness may also be caused by dynamically introduced code especially when user inputs are involved. At the same time, we would like to point out that functions related to dynamic code are often sinks of taint-style vulnerabilities (e.g., `eval` [28], [29] for arbitrary code execution). Therefore, such unsoundness in call graph construction often does not affect FAST's ability in detecting vulnerabilities.

7. Related works

We discuss the related work in this section.

Node.js Vulnerability Detection. On one hand, we start from dynamic analysis. Jalangi [30] uses a selective record-replay method to analyze front- and back-end JavaScript programs dynamically. Arteau [31] proposes a dynamic fuzzer to detect prototype pollution vulnerabilities. On the other hand, we describe static analysis. ODGen [8] proposes object dependence graph to detect vulnerabilities based on graph queries. DAPP [32] uses AST and control-flow patterns to detect prototype pollution vulnerabilities. ObjLupAnsys [11] detects prototype pollution vulnerabilities by expanding and mapping two clusters during the abstract interpretation. Nodest [10], a project based on TAJS [12], introduces an efficient method to detect command injection vulnerability. Both Ocular [33] and CodeQL [7] are industry-level, graph query-based vulnerability detection tool. As a comparison, FAST scales to large, complex Node.js applications to detect taint-style vulnerabilities.

Other than taint-style vulnerabilities, in the past, researchers have studied various security issues or non-taint-style vulnerabilities in the Node.js eco-systems, which include supply chain security [34], [35], Regular Expression Denial of Service (ReDoS) [36]–[38], privilege reduction [34], debloating [39], hidden property abuse [40], and prototype pollution [41]–[43]. As a comparison, FAST is targeting a different problem from those work, and may be able to help them in the future if static analysis is used.

JavaScript Symbolic Execution. JavaScript symbolic execution also has two general types: dynamic [44], [45] and static [46]. On one hand, dynamic symbolic execution, such as ExpoSE [45], relies on an existing JavaScript engine, to propagate symbolic values. On the other hand, static symbolic execution, such as Cosette [46], uses a symbolic interpreter to propagate symbols and extract constraints to find specification-driven bugs. FAST-exp is a static symbolic execution engine and it is the first that generates exploit code statically for JavaScript vulnerability.

Client-side JavaScript Security. We also start from dynamic analysis. Melicher et al. [47] and Steffens et al. [48] both use dynamic taint analysis to find DOM-based XSS. Deemon [49] adopts dynamic analysis and property graphs to detect CSRF vulnerability. CSPAutoGen [50] enforces a template following Content Security Policy to defend against client-side XSS. PathCutter [51] cuts off the propagation paths of XSS worms. Black Widow [52] introduces a black box data-driven approach to crawl and scan web applications. JSObserver [53] investigates the client-side JavaScript code integrity problem caused by JavaScript global identifier conflicts. Next, we describe static analysis. JStap [5], HideNoSeek [54], JaSt [55] and JShield [56], [57] adopt signature matching or static analysis to detect malicious JavaScript programs. DoubleX [6] analyzes the taint flow to detect browser extension vulnerabilities. JSIsolate [58] uses the dependency relationship of different components of the JavaScript programs to prevent the functionalities from interfering with each other. COP [59] proposes a configurable origin policy to isolate JavaScript in a more fine-grained pattern. Cao et al. [60] studied a new protocol of single sign-on for client-side JavaScripot. New browser architectures, such as virtual browser [61] and deterministic browser [62], have also been proposed. JAW [63] models browser objects in a Hybrid Property Graph for client-side CSRF vulnerabilities. Researchers have also studied client-side browser fingerprints [64]–[66] or web tracking [67] in general. As a comparison, the target of FAST, i.e., Node.js vulnerability, is different from prior works.

Some existing works [68]–[72] adopt Automated Exploit Generation (AEG) to exploit client-side XSS vulnerabilities based on dynamically collected traces or dataflows. Kudzu [44] uses dynamic symbolic execution and a constraint solver to detect and exploit client-side XSS and code injection vulnerabilities. Song et al. [73] and Kang et al. [74] exploit the underlying JIT compiler, instead of JavaScript itself, which could be applied to other JIT-compiled languages.

JavaScript Static Analysis Frameworks. TAJS [12] and JSAI [75] adopt abstract interpretation to analyze JavaScript programs for type inference. SAFE [9] and its follow-up work SAFEWAPI [76] covert JS to an Intermediate Representation (IR) for abstract interpretation. PageGraph [77] and AdGraph [78] model the relations between different browser objects. SAFE$_{DS}$ [79] adopts Jalangi, a dynamic analysis tool, to build dynamic shortcuts on top of SAFE to accelerate the static analysis to large packages such as official tests of Lodash. As a comparison, FAST does not need any dynamic execution, which need setup of both inputs and environments to deploy. Furthermore, none of these frameworks are used for vulnerability detection or exploitation.

JavaScript call graph construction [80]–[85] has been studied for a long time, which may use static [81], dynamic [82], or hybrid [80] analysis. For example, Nielsen et al. [14] scan Node.js application to construct modular (e.g., inter-file) call graph graph. Feldthaus et al. [23] design field-based flow analysis for constructing call graphs. Existing static call graph construction traditionally faces challenging issues for dynamic features, such as bracket syntax and `Promise`. Existing dynamic call graph construction often faces issues like code coverage and practical deployment (e.g., some Node.js packages may not run without a proper environment setup). Hybrid analysis leverages benefits of both static and dynamic analysis but also inherits drawbacks of both. As a comparison, FAST is the first static abstract interpretation based call graph construction, which tackles call edges related to many dynamic JavaScript features.

Vulnerability Detection or Program Analysis Techniques. Yamaguchi et al. introduce Code Property Graph (CPG) [86] to detect C/C++ vulnerabilities. Built upon CPG, Backes et al. [87] adapt CPG to PHP to detect PHP vulnerabilities. Randoop [88] produces unit tests for Java via feedback-directed random test generation. Program slicing [89], a concept proposed in 1980s, has been widely used for program analysis and vulnerability detection. Previous works [90], [91] proposed to use abstract interpretation to facilitate program slicing. As a comparison, the pruning process, i.e., program slicing adopted by FAST is used to scale abstract interpretation.

8. Conclusion

In this paper, we propose a novel two-phase abstract interpretation, called FAST, for detection and exploitation of Node.js taint-style vulnerabilities. The first phase (bottom-up abstract interpretation) generates a control-flow path between source and sink. Then, the second phase (top-down abstract interpretation) follows the control-flow path to only analyze statements with control- and data-dependencies with the sink. Compared with state-of-the-art abstract interpretation, such a pruned analysis significantly reduces the states in the abstract domain and scales the analysis. After two phases, FAST also collects and solves data- and control-flow constraints along the target control-flow path to automatically generate exploits. Our evaluation shows that FAST outperforms the state-of-the-art approach in reducing false negatives and detects 242 zero-day vulnerabilities with 21 CVE identifiers.

Acknowledgement

We would like to thank Isaac Chang, Jianjia Yu, Junmin Zhu, and Zhengyu Liu for their help with manual verification and exploitation of zero-day vulnerabilities. We also would like to thank Snyk for vulnerability disclosure and CVE assignment, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006 and a DARPA Young Faculty Award (YFA) under Grant Agreement D22AP00137-00 as well as an Amazon Research Award (ARA) 2021. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, DARPA, or Amazon.

References

[1] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 430–441.

[2] "Static exploration of Taint-Style vulnerabilities found by fuzzing," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/shastry

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

[4] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and automatically preventing injection attacks on NODE.JS," in *NDSS*, 2018.

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

[6] A. Fass, D. F. Somé, M. Backes, and B. Stock, "DoubleX: Statically detecting vulnerable data flows in browser extensions at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1789–1804. [Online]. Available: https://doi.org/10.1145/3460120.3484745

[7] GitHub. CodeQL. https://codeql.github.com/.

[8] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining Node.js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-song

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

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

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

[12] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. 16th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 5673. Springer-Verlag, August 2009.

[13] Promise - JavaScript — MDN. https://developer.mozilla.org/en-US/docs/Web/{JavaScript}/Reference/Global_Objects/Promise.

[14] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–41. [Online]. Available: https://doi.org/10.1145/3460319.3464836

[15] F. Al Kassar, G. Clerici, L. Compagna, F. Yamaguchi, and D. Balzarotti, "Testability tarpits: the impact of code patterns on the security testing of web applications," 2022.

[16] ECMAScript 2015 language specification. https://262.ecma-international.org/6.0/.

[17] Esprima: ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/.

[18] S. Li. ODGen source code. https://github.com/Song-Li/ODGen/.

[19] NetworkX: Network analysis in python. https://networkx.org/.

[20] Z3 thereom prover. https://github.com/Z3Prover/z3.

[21] N. James. Best Node.js CMS platforms for 2022. https://blog.logrocket.com/best-node-js-cms-platforms-2022/.

[22] P. Gerste. Ghost CMS 4.3.2 - cross-origin admin takeover. https://blog.sonarsource.com/ghost-admin-takeover.

[23] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript ide services," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 752–761.

[24] Field-based call graph construction for JavaScript. https://github.com/Persper/js-callgraph.

[25] A. Manion. Vulnerability disclosure policy. https://vuls.cert.org/confluence/display/Wiki/Vulnerability+Disclosure+Policy.

[26] [MDN] public class fields. https://developer.mozilla.org/en-US/docs/Web/{JavaScript}/Reference/Classes/Public_class_fields.

[27] ECMAScript 2023 language specification. https://tc39.es/ecma262/.

[28] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 34–44. [Online]. Available: https://doi.org/10.1145/2338965.2336758

[29] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval begone! semi-automated removal of eval from JavaScript programs," *SIGPLAN Not.*, vol. 47, no. 10, p. 607–620, oct 2012. [Online]. Available: https://doi.org/10.1145/2398857.2384660

[30] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 488–498. [Online]. Available: https://doi.org/10.1145/2491411.2491447

[31] O. Arteau, "Prototype pollution attack in NodeJS application," NorthSec, 2018.

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

[33] Ocular interpreter. https://docs.shiftleft.io/ocular/interpreter.

[34] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on Node.js via rwx-based privilege reduction," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1821–1838.

[35] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," *arXiv preprint arXiv:2002.01139*, 2020.

[36] C.-A. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 361–376.

[37] Z. Bai, K. Wang, H. Zhu, Y. Cao, and X. Jin, "Runtime recovery of web applications under zero-day redos attacks," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1575–1588.

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

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

[40] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the Node.js ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2951–2968.

[41] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent Spring: Prototype pollution leads to remote code execution in Node.js," 2023.

[42] Z. Kang, S. Li, and Y. Cao, "Probe the Proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites," in *Network and Distributed System Security Symposium (NDSS 2022)*, 2022.

[43] H. Y. Kim, J. H. Kim, H. K. Oh, B. J. Lee, S. W. Mun, J. H. Shin, and K. Kim, "DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules," *International Journal of Information Security*, vol. 21, no. 1, pp. 1–23, 2022.

[44] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[45] B. Loring, D. Mitchell, and J. Kinder, "ExpoSE: Practical symbolic execution of standalone JavaScript," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 196–199. [Online]. Available: https://doi.org/10.1145/3092282.3092295

[46] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic execution for JavaScript," in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3236950.3236956

[47] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting," in *Network and Distributed System Security Symposium (NDSS)*, 2018, https://doi.org/10.14722/ndss.2018.23309.

[48] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild," in *Network and Distributed System Security Symposium (NDSS)*, 2019, https://publications.cispa.saarland/id/eprint/2744.

[49] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting csrf with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1757–1771. [Online]. Available: https://doi.org/10.1145/3133956.3133959

[50] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "Cspautogen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA, 2016.

[51] Y. Cao, V. Yegneswaran, and Y. Chen, "Pathcutter: Severing the self-propagation path of xss JavaScript worms in social web networks." in *NDSS*, 2012.

[52] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1125–1142.

[53] M. Zhang and W. Meng, "Detecting and understanding JavaScript global identifier conflicts on the web," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 38–49. [Online]. Available: https://doi.org/10.1145/3368089.3409747

[54] A. Fass, M. Backes, and B. Stock, "HideNoSeek: Camouflaging malicious JavaScript in benign asts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1899–1913. [Online]. Available: https://doi.org/10.1145/3319535.3345656

[55] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JaSt: Fully syntactic detection of malicious (obfuscated) JavaScript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Giuffrida, S. Bardin, and G. Blanc, Eds. Cham: Springer International Publishing, 2018, pp. 303–325.

[56] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "Jshield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 466–475. [Online]. Available: https://doi.org/10.1145/2664243.2664256

[57] Y. Cao, X. Pan, Y. Chen, J. Zhuge, X. Qian, and J. Fu, "Malicious code detection technologies," Dec. 15 2015, US Patent 9,213,839.

[58] M. Zhang and W. Meng, "Jsisolate: Lightweight in-browser JavaScript isolation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 193–204. [Online]. Available: https://doi.org/10.1145/3468264.3468577

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

[60] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen, "Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham: Springer International Publishing, 2014, pp. 276–298.

[61] Y. Cao, Z. Li, V. Rastogi, and Y. Chen, "Virtual browser: A web-level sandbox to secure third-party JavaScript without sacrificing functionality," ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 654–656. [Online]. Available: https://doi.org/10.1145/1866307.1866387

[62] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic browser," ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 163–178. [Online]. Available: https://doi.org/10.1145/3133956.3133996

[63] S. Khodayari and G. Pellegrino, "JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2525–2542. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/khodayari

[64] Y. Cao, S. Li, E. Wijmans *et al.*, "(cross-) browser fingerprinting via os and hardware level features." in *NDSS*, 2017.

[65] S. Wu, P. Sun, Y. Zhao, and Y. Cao, "Him of many faces: Characterizing billion-scale adversarial and benign browser fingerprints on commercial websites," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[66] S. Wu, S. Li, Y. Cao, and N. Wang, "Rendered private: Making glsl execution uniform to prevent webgl-based browser fingerprinting." in *USENIX Security*, 2019.

[67] X. Pan, Y. Cao, and Y. Chen, "I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser," in *NDSS*, 2015.

[68] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild." 2019.

[69] S. Bensalim, D. Klein, T. Barber, and M. Johns, "Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 27–33.

[70] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Dexterjs: Robust testing platform for dom-based xss vulnerabilities," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 946–949. [Online]. Available: https://doi.org/10.1145/2786805.2803191

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

[72] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1709–1723.

[73] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation." in *NDSS*, 2015.

[74] X. Kang and S. Debray, "A framework for automatic exploit generation for jit compilers," in *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks*, 2021, pp. 11–19.

[75] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 121–132. [Online]. Available: https://doi.org/10.1145/2635868.2635904

[76] S. Bae, H. Cho, I. Lim, and S. Ryu, "Safewapi: Web api misuse detector for web applications," ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 507–517. [Online]. Available: https://doi.org/10.1145/2635868.2635916

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

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

[79] J. Park, J. Park, D. Youn, and S. Ryu, "Accelerating JavaScript static analysis via dynamic shortcuts," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1129–1140. [Online]. Available: https://doi.org/10.1145/3468264.3468556

[80] G. Antal, Z. Tóth, P. Hegedűs, and R. Ferenc, "Enhanced bug prediction in JavaScript programs with hybrid call-graph based invocation metrics," *Technologies*, vol. 9, no. 1, p. 3, 2020.

[81] G. Antal, P. Hegedus, Z. Tóth, R. Ferenc, and T. Gyimóthy, "Static JavaScript call graphs: A comparative study," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 177–186.

[82] T. R. Toma and M. S. Islam, "An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application," in *2014 International Conference on Informatics, Electronics & Vision (ICIEV)*. IEEE, 2014, pp. 1–6.

[83] J. Dijkstra, "Evaluation of static JavaScript call graph algorithms," Ph.D. dissertation, Software Analysis and Transformation, 2014.

[84] D. Seifert, M. Wan, J. Hsu, and B. Yeh, "An asynchronous call graph for JavaScript," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 29–30.

[85] M. Chakraborty, R. Olivares, M. Sridharan, and B. Hassanshahi, "Automatic root cause quantification for missing edges in JavaScript call graphs," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

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

[87] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017, pp. 334–349.

[88] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[89] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[90] I. Mastroeni and D. Zanardini, "Abstract program slicing: An abstract interpretation-based approach to program slicing," *ACM Trans. Comput. Logic*, vol. 18, no. 1, feb 2017. [Online]. Available: https://doi.org/10.1145/3029052

[91] H. S. Hong, I. Lee, and O. Sokolsky, "Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking," in *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*. IEEE, 2005, pp. 25–34.

Appendix A. Control-flow Graph Generation

Algorithm 1 shows a high-level overview of procedure of generation of functional dependency and call edges. The algorithm accepts a `stack` holding all the functions to analyze and an initially empty `callDepGraph`, which combines the known call graph (with resolved call edges) with the functional dependency graph. Specifically, FAST first pops up a function from the stack (Line 3), analyzes the function and pushes all function definitions in this function scope onto the stack for future analysis (Line 4). Then, FAST loops through all statements in the function in the abstract domain (Line 5) and adds edges based on statement types (Lines 6–29).

After bottom-up abstract interpretation, FAST searches for interprocedural control-flow paths between sources and sinks of taint-style vulnerabilities. We remind readers that the FAST also constructs intra-procedural control flow graphs of each function but have omitted the description of this standard step for space reasons. The search has two steps: locating sources and searching paths to sinks from sources.

Algorithm 1 Bottom-up Abstract Interpretation

```
1:  procedure BottomUp(stack← init, callDepGraph← init)
2:    while stack is not empty do
3:      func ←stack.POP() , addEdge←callDepGraph.addEdge
4:      Push func.scope.fns onto stack, update callDepGraph
5:      for stmt in func.stmts do
6:        switch stmt do
7:          case resolvable-fn-call:
8:            addEdge(func —call→ stmt.fn)
9:            stmt.fn.args.foreach(arg => resolve(
10:             arg.func —lookup→ *)
11:         case pending-fn-call:
12:           stmt.fn.lookupPath.args.foreach(arg =>
13:             addEdge(arg.func —callback→ stmt.fn))
14:         case ret-fn-call: addEdge(func —ret→ stmt.fn)
15:         case param-fn (callback):
16:           stack.push(stmt.fn)
17:           x ← isKnown?call:trigger
18:           target ← isSync?stmt.caller-fn:top
19:           addEdge(target —x→ stmt.fn)
20:         case return-fn:
21:           stack.push(stmt.fn), resolve(stmt.fn —ret→ *)
22:         case function expression: stack.push(stmt.fn)
23:         case new Promise:
24:           stack.push(exec←stmt.executor)
25:           addEdge(func —new→ Promise, func —call→ exec)
26:         case then: addEdge(stmt.prms —then→ stmt.then)
27:         case await: addEdge(stmt.prms —await→ stmt.await)
28:         case resolve/reject:
29:           addEdge(func —resolve/reject→ stmt.defFunc.prms)
30:           resolve(stmt.defFunc.prms)
31:       end switch
32:     end for
33:   end while
34: end procedure
```

TABLE 8: A list of sources and sinks that are broken down by vulnerability types.

Vulnerabilities	Sources	Sinks
Command Injection	Arguments of functions in `module.exports`, command line arguments, environment variables, HTTP* requests	functions in `child_process`
Arbitrary Code Exec.		`eval`, `Function`
Path Traversal	HTTP* requests	file systems → HTTP* responses

*: "HTTP" includes HTTPS and third-party server packages such as Express.

Appendix B. Source and Sink Discovery and Path Search

We describe how FAST discovers sources and sinks and then finds a control-flow path between sources and sinks. Note that a list of sources and sinks can be found in Table 8.

Here are the details. First, FAST finds sources as the start of a control flow path. There are generally two source types: specific API calls and functions defined in `module.exports`. The former can be found via pattern matching; the latter needs a search on all the defined functions. Specifically, FAST adopts a breadth first search (BFS) to loop all possible objects starting from `module.exports` to all properties and subproperties that are defined under `module.exports`. That is, FAST finds functions like `module.exports.foo()` and `module.exports.foo().bar()` as sources.

Second, FAST adopts a depth first search (DFS) to

Algorithm 2 Extracting constraints from a data-flow path

```
1: map ← a map from object nodes to symbols
2: procedure GETSYMBOLS(constraints, type, o_0, o_1, o_2, ...)
3:   for every o_i in o_0, o_1, o_2, ... do
4:     if o_i is in map then
5:       if map[o_i] has the same type as type then
6:         s_i ← map[o_i]
7:       else
8:         try type conversion
9:       end if
10:    else
11:      s_i ← map[o_i] ← MAKESYMBOL(type)
12:      if o_i has a concrete value then
13:        constraints.ADD(MAKECONSTRAINT(=, s_i, o_i.value))
14:      end if
15:    end if
16:  end for
17:  return s_0, s_1, s_2, ...
18: end procedure
19: procedure DFCONSTRCONV(constraints, sinkObj, conditions)
20:  queue ← [sinkObj] + conditions
21:  while queue is not empty do
22:    head ← queue.POP()
23:    for each incoming edge e to head do
24:      e_0, e_1, e_2, ... ← all edges in the same group with e
25:      o_0, o_1, o_2, ... ← e_0.tail, e_1.tail, e_2.tail, ...
26:      op ← operation of the edge group
27:      switch type of op do
28:        case string operations:
29:          s_0, s_1, s_2, ... ← GETSYMBOLS(string, o_0, o_1, o_2, ...)
30:        case number operations:
31:          s_0, s_1, s_2, ... ← GETSYMBOLS(number, o_0, o_1, o_2, ...)
32:      end switch
33:      constraints.ADD(MAKECONSTRAINT(=, head,
           MAKECONSTRAINT(op, s_0, s_1, s_2, ...)))
34:    end for
35:  end while
36: end procedure
```

find a control-flow path from sources to sinks. The search follows the timing sequence of call edges on a specific statement. For example, say we have `func1(func2())` or `func2().func1()`. In both cases, FAST searches through `func2()` first and then reaches `func1()` to ensure the feasibility of the following data-flow path generation stage. FAST also limits the number of times that a statement can be visited to avoid loops in the control-flow path. Note that this is unrelated with the follow-up top-down abstract interpretation, which can still explore functions recursively.

Appendix C. Constraint Generation

Algorithm 2 shows a simplified algorithm of constraint generation. Given an object, FAST loops through all the incoming edges to the object (Line 23), obtain objects related to incoming edges (Line 25) and the operator (Line 26). Then, FAST obtains symbols for this operator based on the type (Line 27) and then adds the constraint to the pool (Line 33). The symbol generation and lookup process is shown in Lines 2–18. FAST maintains a `map` between symbols (which are acceptable by constraint solvers) and object nodes (Line 1). When FAST accepts an operator and their operands, FAST tries to lookup or generate symbols (Line 11). Note that if there are type issues, FAST will attempt to perform type conversion (Line 8) and if FAST encounters constants, FAST adds a corresponding constant constraint.

TABLE 9: A breakdown of performance overhead of FAST by different stages.

	strapi@4.0.8	strapi@3.0.0-beta.17.7	total.js@3.4.5
Stage I: CF Path	1,298 ± 588	301 ± 41.8	1,534 ± 85.3
Stage II: DF Path	22.4 ± 1.59	3.20 ± 0.67	146 ± 110
Stage III: Exploit	72.3 ± 14.0	41.7 ± 33.5	280 ± 342

TABLE 10: A list of CVE identifiers assigned to zero-day vulnerabilities detected by FAST.

CVE-2022-24431	CVE-2022-25855	CVE-2022-25908
CVE-2022-24377	CVE-2022-25923	CVE-2022-21191
CVE-2022-25906	CVE-2022-25171	CVE-2022-25916
CVE-2022-21129	CVE-2022-25853	CVE-2022-21810
CVE-2022-25962	CVE-2022-25890	CVE-2022-25350
CVE-2023-25805	CVE-2022-25926	CVE-2020-7735
CVE-2020-7730	CVE-2020-15123	CVE-2020-15362

Appendix D. Performance Breakdown Evaluation

We break down the performance overhead of three packages with more than 10K LoC by three different stages. Table 9 shows the breakdown. Stage I is the slowest because FAST needs to analyze all the function. Stage II is faster than Stage I because FAST follows a subset of program with control- and data-flow dependencies with the sink. Lastly, Stage III is also relatively slow (much faster than Stage I but slower than Stage II), because its takes time for Z3 solver to find a solution given constraints.

Appendix E. A List of CVE Identifiers for Zero-day Vulnerabilities

Table 10 lists 21 CVE identifiers that are assigned to zero-day vulnerabilities found by FAST.