NodeMedic-FINE: Automatic Detection and Exploit Synthesis for Node.js Vulnerabilities

Darion Cassel*†§, Nuno Sabino*‡§, Min-Chien Hsu*, Ruben Martins* and Limin Jia*
*Carnegie Mellon University
darion.cassel@gmail.com, {nsabino,minichieh,rubenm,ljia}@andrew.cmu.edu
† Work done prior to joining Amazon
‡Instituto Superior Técnico, Universidade de Lisboa, and Instituto de Telecomunicações

***Abstract*—The Node.js ecosystem comprises millions of packages written in JavaScript. Many packages suffer from vulnerabilities such as arbitrary code execution (ACE) and arbitrary command injection (ACI). Prior work has developed automated tools based on dynamic taint tracking to detect potential vulnerabilities, and to synthesize proof-of-concept exploits that confirm them, with limited success.**

One challenge these tools face is that expected inputs to package APIs often have varied types and object structure. Failure to call these APIs with inputs of the correct type and with specific fields leads to unsuccessful exploit generation and missed vulnerabilities. Generating inputs that can successfully deliver the desired exploit payload despite manipulation performed by the package is also difficult.

To address these challenges, we use a type and object-structure aware fuzzer to generate inputs to explore more execution paths during dynamic taint analysis. We leverage information generated by the taint analysis to infer the types and structure of the inputs, which are then used by the exploit synthesis engine to guide exploit generation. We implement NodeMedic-FINE and evaluate it on 33,011 npm packages that contain calls to ACE and ACI sinks. Our tool finds 2257 potential flows and automatically synthesizes working exploits in 766 packages.

I. Introduction

The Node.js ecosystem is vast and ever-growing, with millions of JavaScript packages available through the package management system npm alone [1]. Each package serves as a building block for developers to create their own applications. Each package typically has a set of public APIs, functions that can be called from other packages, called *entry points*. As its popularity increases, the Node.js ecosystem has become an attractive target of attackers [2, 3, 4, 5, 6, 7, 8]. Prior work has shown that many packages in the Node.js ecosystem contain security vulnerabilities [9, 10, 11, 12, 13, 14, 15, 16, 17, 18]. The most serious vulnerabilities are Arbitrary Command Injection (ACI) and Arbitrary Code Execution (ACE) vulnerabilities, which allow an attacker to execute code or commands on the system that runs the application [19, 20].

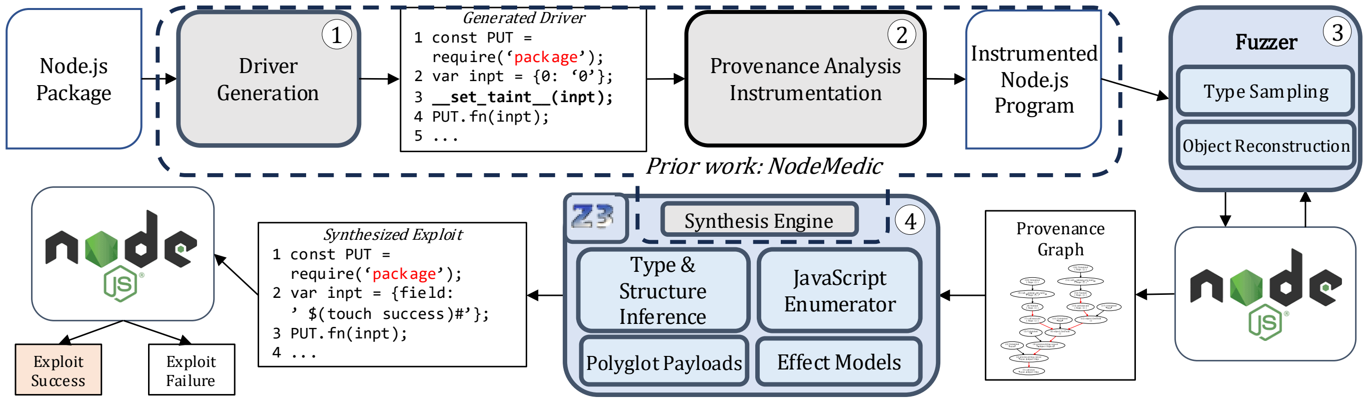
Prior work has developed automated analyses to detect potential ACI and ACE vulnerabilities in JavaScript programs [9, 10, 12, 14, 21, 22, 23, 24, 25, 26, 27] and to synthesize proof-of-concept exploits to confirm them [21, 22, 23, 24, 25, 26, 27]. Several of these tools implement dynamic taint tracking to identify ACI and ACE vulnerabilities at run time. At a high level, dynamic taint analyses aim to find a *flow* of information from attacker-controlled inputs to a package's entry point to sensitive APIs such as `eval`, called *sinks*.

Dynamic analysis alone, without fuzzing the inputs or leveraging path conditions, can only observe one execution path of the program, leading to missed vulnerabilities. Another drawback of such an analysis is false positives [9, 10, 21]; the tool may report many potentially dangerous flows, but not indicate which flows can truly be exploited. To reduce false-positive rates, prior work [21] has explored using SMT string synthesis to automatically generate functional proof-of-concept exploits from output of the dynamic taint analysis for Node.js packages: NodeMedic was able to automatically confirm 155 ACI and ACE flows in a sample of 10,000 packages. However, the limitations of the approach lead to a failure to confirm 23% of ACI flows and 73% of ACE flows [21]. The static analysis tool FAST has also used synthesis to generate proof-of-concept exploits. Unlike dynamic analysis tools, it collects control-flow constraints via abstract interpretation [16].

One fundamental challenge for dynamic taint analysis is that inputs to package APIs often have varied types and structures. If the dynamic taint analysis does not call these APIs with inputs of the correct type, with specific fields, it may miss vulnerabilities. Generating exploits faces similar challenges. A second challenge to generating viable proof-of-concept exploits is that the algorithm has to consider operations performed on the tainted inputs before they reach the sinks. A third challenge, particularly for generating exploits for ACE vulnerabilities, is that they must be syntactic and semantically valid JavaScript to deliver the payload.

§Shared first authorship.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241636
www.ndss-symposium.org



Generated Driver
```
1 const PUT =
  require('package');
2 var inpt = {0: '0'};
3 __set_taint__(inpt);
4 PUT.fn(inpt);
5 ...
```

Synthesized Exploit
```
1 const PUT =
  require('package');
2 var inpt = {field:
  ' $(touch success)#'};
3 PUT.fn(inpt);
4 ...
```

Fig. 1: NODEMEDIC-FINE's end-to-end pipeline for vulnerability detection and exploit generation. Blue components in 3, 4 are novel. Components (1, 2, Synthesis Engine) inside the box with dashed outline are from prior work, NODEMEDIC [21].

To address these challenges in the context of dynamic taint analysis, we propose to leverage runtime information generated from dynamic taint tracking to (1) help a fuzzer to generate nontrivial inputs to explore more execution paths during dynamic taint analysis and (2) to infer the type and structure of the inputs, which are then used by the exploit synthesis engine to guide the generation of exploits. In addition to type and structure information, we also propose to incorporate the semantics of operations performed on tainted data in the synthesis algorithm to increase the success rate of exploit generation. Finally, we explore generating valid completions of JavaScript code string prefixes to synthesize ACE exploits.

Building on top of NODEMEDIC's driver[1] generation (①) and dynamic provenance (taint) analysis (②) [21], we implement NODEMEDIC-FINE (Fuzzer, INference, Enumerator) for automatically detecting ACI and ACE flows and synthesizing proof-of-concept exploits to confirm them (Figure 1). First, we implement a novel type- and structure-aware fuzzer to explore the Node.js package (③). Second, we extend the prior synthesis engine with new components (④) that infer input types and structure, implement an Enumerator to produce valid completions of JavaScript, and incorporate additional constraints based on effects of JavaScript operations. These methodologies can be applied to any dynamic taint analysis engine that can produce a *provenance graph* [21].

We evaluate NODEMEDIC-FINE on 33,011 npm packages in active use that contain calls to ACI and ACE sinks. NODEMEDIC-FINE finds 2257 potential flows and automatically synthesizes exploits that confirm 766 flows. The type- and structure-aware fuzzer found 1.7x the number of potential flows that NODEMEDIC uncovered. The new synthesis components were pivotal in confirming an additional 62 confirmed flows, for a total 1.6x confirmed flows compared to NODEMEDIC. We have open-sourced NODEMEDIC-FINE; please see the Appendix B for details.

[1]For NODEMEDIC, a *driver* is a Node.js program that imports the package-under-test and calls its public APIs with provided inputs [21].

```
1  module.exports = {
2    execute: function(params, callback, error) {
3      var exec = require('child_process').exec;
4      var cmd = 'rsync';
5      if(params.flags !== undefined) {
6        cmd += ' -' + params.flags;
7      }
8      if(params.options !== undefined) {
9        cmd += ' ' + params.options;
10     }
11     if(params.source !== undefined) {
12       cmd += ' ' + params.source;
13     }
14     if(params.destination !== undefined) {
15       cmd += ' ' + params.destination;
16     } else {
17       console.log('Err: ...');
18     }
19     exec(cmd, function(error, stdout, stderr) {
20       if(error !== null) { error(error);}
21       else { callback(stdout); }});
22     }};
```
Fig. 2: An example ACI vulnerability

Responsible disclosure. We follow a coordinated vulnerability disclosure process (i.e., responsible disclosure) [28] for the vulnerabilities discovered in our evaluation. We are in the process of triaging and responsibly disclosing our confirmed flows; see Section V-G for details. Thus far, 1 high severity CVE [29] has been assigned.

II. BACKGROUND

We show an example ACI vulnerability and briefly review NODEMEDIC's dynamic taint analysis algorithm and output provenance graph, which NODEMEDIC-FINE takes as input.

Motivating example. The code snippet of a function with a confirmed ACI vulnerability is shown in Figure 2. The encompassing package exports the `execute` function, making it public to other packages. This function is a wrapper around `rsync`, and it looks for several attributes in the first argument `param`. If `param` has a `flags` attribute, the package concatenates its value to the final command that is executed using `exec` (lines 6-7). This package has an ACI vulnerability when an attacker is able to control the first argument of `execute`. For instance, if an attacker calls `execute` with the following arguments, all files on the server hosting the execution of this package could be deleted.

```
execute({"flags": "$(rm -rf /)"}, function(){}, function(){})
```

The attacker can execute any arbitrary command by setting the `flags` attribute appropriately.

Dynamic taint analysis. Dynamic taint analysis, or *taint tracking*, is a runtime mechanism for tracking information flows from *sources*, e.g., the inputs to package entry points to sensitive *sinks* like the `exec` function (c.f. [30]). Certain program values, such as the above-mentioned sources, are labeled as *tainted* and these labels are then *propagated* by program operations. For example, `params` in Figure 2 is labeled as tainted and is used in an assignment and concatenation operation on line 7 then `cmd` becomes tainted. Dynamic information flow analysis has been particularly effective for analyzing code-injection vulnerabilities, such as ACE and ACI, in JavaScript (c.f. [31]).

We call a discovered flow from an attacker-controllable source to sensitive sink a *potential flow*, because it is unknown if the flow can be exploited. Once a flow has been determined to be exploitable—an input to the package results in successful execution of an exploit payload—we call it a *confirmed flow*. Not every confirmed (exploitable) flow is a *vulnerability*, which is a flow that does not correspond to a legitimate behavior of a package's API, e.g., executing arbitrary commands.

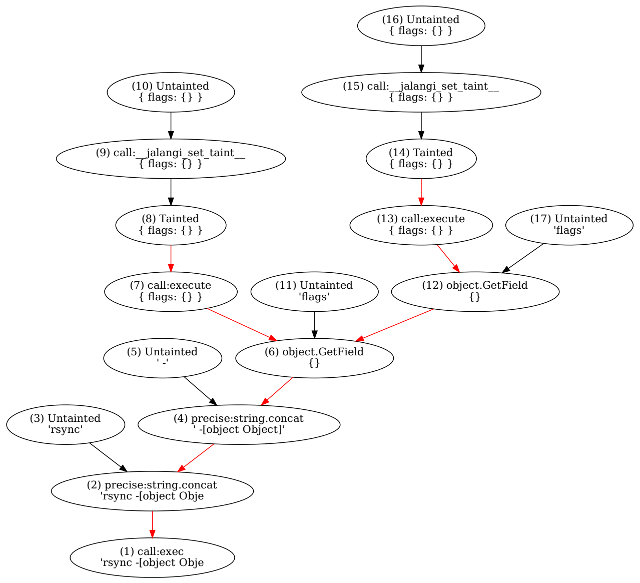
NODEMEDIC: Provenance graphs and naive synthesis. NODEMEDIC-FINE builds on top of NODEMEDIC [21], which is a dynamic taint analysis tool for identifying Arbitrary Code Execution (ACE) [19] and Arbitrary Command Injection (ACI) [20] in Node.js packages. To analyze a package, NODEMEDIC automatically generates a simple driver program that imports the package and executes its public APIs with *fixed* values for all arguments, that are marked as tainted (potentially attacker-controllable). NODEMEDIC instruments the code to implement the dynamic taint analysis. The instrumented code is run with Node.js and outputs potential flows from tainted inputs to sinks as a *provenance graph*.

The provenance graph captures a runtime trace of how tainted data flowed through the program. An example provenance graph for the code in Figure 2 is shown in Figure 3. Each node has a numeric identifier, an operation, and a value (a truncated, stringified representation of the data at that node). The leaf nodes are program inputs or constants. The remaining nodes are operations that data passes through, terminating at a sink. For example, node (14) taints the input parameter; a concatenation is shown in node (4); and node (1) is the sink call. The flow of tainted data is indicated by red edges.

Using the provenance graph, NODEMEDIC synthesizes a candidate exploit, generates a driver to call the package with the exploit, and executes it. It then checks for the desired effect of the exploit (e.g., creation of a file). However, NODEMEDIC was not able to synthesize an exploit for this example, even though it reports a potential flow.

III. MOTIVATION AND OVERVIEW

Automatically generating exploits for packages like the one shown in Section II is challenging. We identify key challenges



(16) Untainted { flags: {} }
(10) Untainted { flags: {} }
(15) call:__jalangi_set_taint__ { flags: {} }
(9) call:__jalangi_set_taint__ { flags: {} }
(14) Tainted { flags: {} }
(8) Tainted { flags: {} }
(13) call:execute { flags: {} }
(17) Untainted 'flags'
(7) call:execute { flags: {} }
(11) Untainted 'flags'
(12) object.GetField {}
(5) Untainted ' -'
(6) object.GetField {}
(3) Untainted 'rsync'
(4) precise:string.concat ' -[object Object]'
(2) precise:string.concat 'rsync -[object Obje
(1) call:exec 'rsync -[object Obje

Fig. 3: Example provenance graph for code in Figure 2

in improving the completeness of ACE and ACI vulnerability detection and exploit synthesis based on dynamic taint tracking and present an overview of NODEMEDIC-FINE to explain how we address these challenges.

Challenges. Three key challenges we face (also noted in prior work [21]) are: 1) Dynamic analysis of Node.js packages needs inputs that satisfy specific type and structure requirements. NODEMEDIC only executes the package using a single fixed constant input. For example, to call the entry point shown in Figure 2 and trigger a flow, the driver has to call it with an object with the `flags` attribute. 2) Confirming flows also requires synthesized inputs to have a particular type and structure, such as the example input object containing an exploit payload in its `flags` field. 3) The confirmation methodology needs to generate string payloads that have semantically valid completions of JavaScript strings for ACE vulnerabilities. These challenges are not specific to NODEMEDIC; they apply broadly to confirming vulnerabilities found by JavaScript dynamic taint analysis tools [22, 24, 32].

Overview. NODEMEDIC-FINE implements novel fuzzing and synthesis methodologies to address these challenges. To address the first challenge, we introduce a coverage-guided, type-aware fuzzer that can generate inputs with diverse types and object structure. To address the second challenge, we enhance the exploit synthesis methodology to generate inputs with types and structure inferred from provenance graphs, and to support JavaScript coercion and common string operations. For the last challenge, we incorporate an *enumerator* component in the synthesis methodology that produces syntactically-valid completions of JavaScript strings.

The overview of NODEMEDIC-FINE is shown in Figure 1. NODEMEDIC-FINE takes as input Node.js packages. NODEMEDIC-FINE generates a driver that imports the instrumented package and calls its public entry points with inputs.

The driver generation is straightforward, except that the inputs used are from the fuzzer. The fuzzer is coverage-guided and can generate inputs from a variety of types and dynamically reconstruct attributes that are expected from object inputs (more details in Section IV-A). NodeMedic-FINE directly utilizes NodeMedic's dynamic taint provenance analysis to produce a provenance graph when a potential flow is discovered. Any Node.js dynamic taint tracking tool would be usable, as long as it generates a provenance graph.

The next few components of NodeMedic-FINE synthesize an exploit, taking the provenance graph as input. To generate exploits of the correct type, NodeMedic-FINE includes a type inference component, which infers the types of the input, including its inner structure, based on operations performed on the input present in the provenance graph. For instance, upon seeing the `getField` operation in node (6), we can infer the input is an object with a field `flags`; seeing the `concat` operation in node (4) we can infer the flag field's value is of type string (more details in Section IV-C). To aid generation of exploits for ACE vulnerabilities, we implement an Enumerator component, which takes the prefix of the exploit to be generated as input, and returns a list of templates, each of which is a syntactically valid JavaScript expression that starts with the prefix and will execute the intended statement (more details in Section IV-E). Building on NodeMedic's synthesis engine, the inference algorithm and Enumerator create an SMT formula encoding the above-mentioned constraints. By solving for symbolic variables representing package API input, Z3 [33] generates a satisfying instantiation of these variables, forming a candidate exploit.

IV. NodeMedic-FINE Design

This section explains NodeMedic-FINE's novel fuzzing and synthesis components.

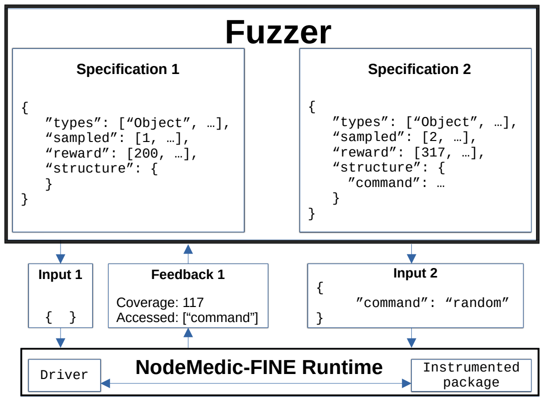
A. Fuzzing Types and Structure

To explore more execution paths, we implement a coverage-guided, type- and object-structure–aware fuzzer for Node.js packages, which iteratively refines its internal weights for generating inputs of different types based on coverage information. The fuzzer can refine the structure of the generated objects based on field access information from the runtime.

Fuzzing loop. The fuzzer's interactions with the rest of NodeMedic-FINE is shown in Figure 4. The fuzzer takes an input specification for the entry point parameter being analyzed, generates inputs based on the specification, and sends them to be executed by NodeMedic-FINE.[2] NodeMedic-FINE returns coverage information and the attributes accessed via instrumented field access operations (*getField* [35]). The fuzzer takes this feedback and refines its input specification to start the next round of fuzzing, until a time budget is exhausted.

Input specification. Inputs are specified hierarchically by the following elements: a list of types that the input can have (`types`); a list of number of samples taken for each type, where the ith element specifies how many times the fuzzer sampled

[2]The fuzzer utilizes the npm package Hasard [34] for generating random values according to a rigorous specification of the characteristics of the value.



```
{
   "types": ["Object", …],
   "sampled": [1, …],
   "reward": [200, …],
   "structure": {
   }
}
```

```
{
   "types": ["Object", …],
   "sampled": [2, …],
   "reward": [317, …],
   "structure": {
     "command": …
   }
}
```

```
{ }
```

Coverage: 117
Accessed: ["command"]

```
{
    "command": "random"
}
```

Fig. 4: Fuzzer loop

an input of the ith type in the types list (`sampled`); a list of coverage data for inputs of each type; where the ith element represents the accumulated number of lines of code triggered by generated inputs of the ith type in the types list (`reward`); and a recursive specification of the structure of the final input (`structure`). The "Specification" boxes in Figure 4 are example specifications. The first box states that the first type in the list is an "Object", not yet sampled by the fuzzer. It sets the initial reward for Objects to 200 and defines its structure as empty.

Weight adjustment. Our fuzzer is coverage-guided: the amount of code executed using the previous inputs influences future input generation. The `reward` and `sampled` data in the specification contribute to the adjustable weight used for tuning input generation. We provide an initial weight for each type, based on the observation that some types are more likely to trigger flows in Node.js package APIs than others. We aim to choose weights that increase the likelihood of generating inputs that trigger a potential flow. We performed a small scale analysis on 12k packages sampled from npm to identify the frequency of each JavaScript type that resulted in a potential flow. In this experiment, we started fuzzing with equal weight for all types and analyzed the reported potential flows. We found that object inputs are most likely to result in potential flows, followed by strings, booleans, and functions. We seed the reward field in initial input specifications to reflect the above observation.

These weights are dynamically adjusted after each fuzzing iteration based on coverage. The fuzzer only knows how effective each type is at improving coverage after it has tried them all. There is often a tradeoff between continuing to explore inputs of types that have already shown promise in the past and trying out inputs of types that have not been explored much. This is known as the exploration-exploitation dilemma [36].

When deciding which new type to explore, we employ a straightforward yet effective method. We start by obtaining an array representing the expected coverage $\frac{reward_t}{sampled_t}$ for each type t. This array is then normalized, and its elements are used as probability weights. The `sampled` list is initialized with all 1's, since initial values for `reward` are also given. Though somewhat standard, this fuzzing method is a necessary groundwork for our novel contributions: object reconstruction and type-awareness.

Using this approach, it is more likely for input types that were effective in the past to have higher expected coverage values and therefore to be chosen more frequently, while still making it possible for types that were not effective in the past to still be chosen again eventually.

Object reconstruction. The initial specification of objects contains no attributes. For the fuzzer to generate objects with useful structure, we extended NODEMEDIC's taint instrumentation to keep track of the field names whenever a *getField* operation is performed. This information is given as feedback to the fuzzer. At the end of each iteration, the input specification is updated to include newly discovered attributes. For example, in Figure 4 "Feedback 1" from the first run of the fuzzer states that it covers 117 lines of new code and access the field `"command"`. The input specification is updated to "Specification 2": with new coverage data and more detailed object structure. The fuzzer then generates a new input with the field `"command"` set to a random input.

B. Handling Trivially-Exploitable Flows

Many packages with potential flows could be exploited using the following *polyglot* input strings, designed to handle multiple scenarios simultaneously: For ACI: `$(touch /tmp/success) #" || touch /tmp/success #' || touch /tmp/success` accounts for single quotes and double quotes contexts, or when certain shell metacharacters are sanitized. For ACE: `global.CTF();//" +global.CTF();//' +global.CTF();// ${global.CTF()}` executes `global.CTF` even if the payload is injected in double or single quotes or backticks.

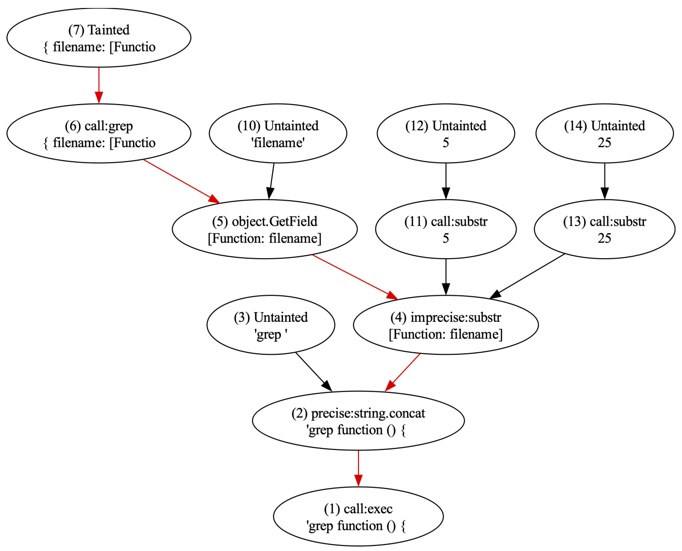
For ACI flows, the shell expansion meta characters `$(touch /tmp/success)` already handle most contexts. The payload may be injected inside a shell string with double quotes or backticks and it will still execute, even if some parts of the command are not syntactically valid. Therefore, the ACI polyglot is typically not needed.

For ACE, carefully crafting the payload is crucial because the final argument to ACE sinks needs to be syntactically valid JavaScript; otherwise none of payload statements will execute. Unlike the ACI polyglot, the ACE polyglot is highly effective in confirming flows (Sections V-D-V-E).

C. Type and Structure Inference

Inputs generated by the fuzzer may have varied types and structures (Section IV-A). However, there is no guarantee that these randomly generated inputs have the correct type or structure to exploit the vulnerability. For example, an input generated by the fuzzer that results in the flow in Figure 2 is `{"flags": {"RF<bWD c^G;wmo?S": ""}}`, but an input that *exploits* the flow must have structure `{"flags": "payload"}`.

To address this, we extend the synthesis methodology to infer required input types and structures and integrate this information into the process of constraint-based exploit synthesis. The key idea is that the provenance graph is a record of all operations performed at runtime on the package API



(7) Tainted { filename: [Functio
(6) call:grep { filename: [Functio
(10) Untainted 'filename'
(12) Untainted 5
(14) Untainted 25
(5) object.GetField [Function: filename]
(11) call:substr 5
(13) call:substr 25
(3) Untainted 'grep '
(4) imprecise:substr [Function: filename]
(2) precise:string.concat 'grep function () {
(1) call:exec 'grep function () {

Fig. 5: Provenance graph for toy example API.

input, and thus it can be used to infer the types and structure of the input. For example, if the package API performs a `substr` operation on its input, then we can infer that the type of the input is *string*. Similarly, if the package performs a field access operation on its input, then we can infer that the input is a JavaScript datatype that supports field access such as objects, arrays, maps, and sets.

We first present a motivating example and give an overview of the technique (Section IV-C1). Then we describe the type inference algorithm (Section IV-C2) and the structure inference algorithm (Section IV-C3). Finally, we describe how the inferred information is integrated into the exploit synthesis process (Section IV-C4).

1) Motivating Example and Overview: The `grep` package API is shown in Figure 6a. The `query` argument has the type *object* with a field `filename`, which is a string that has the operation `substr` applied to it. The resulting string is passed to the `exec` sink, leading to an ACI vulnerability. Figure 5 shows the provenance graph generated by our tool.

The inference algorithm traverses the provenance graph (Figure 5) from the leaf nodes towards the root and extracts information about the type and structure of attacker-controllable inputs, refining its *abstract value* (c.f. Figure 6b); a data structure that stores a set of possible types for the input–its *types*–as well an abstract *structure* that recursively stores abstract values for discovered properties (fields) of the input. The initial abstract value is shown in Figure 6b; `"Bot"` (*Bottom*) represents any JavaScript type. The presence of the `GetField` operation allows the inference to refine the type-set of the `query` input from *{Bottom}*, to *{Object, Array, Map, Set}*. Furthermore, the algorithm examines the field that was accessed in the `GetField` operation, `"filename"`, and determines that it is not numeric. This further refines the type-set to *{Object}*. The algorithm also notes that the string value `"filename"` is part of the structure of the input. Finally, the algorithm reaches the

```
function grep(query) {
  exec("grep " + query["filename"].substr(5, 25));
}
```

(a) Toy example package API.

```
{ "id": "",
  "types": ["Bot"],
  "structure": {} }
```

(b) Initial abstract value for toy example API.

```
{ "id": "",
  "types": ["Object"],
  "structure": {
    "filename": {
      "id": "47341750",
      "types": ["String"],
      "structure": {} }}}
```

(c) Inferred abstract value for toy example API.

```
(declare-fun SymbolicField_1 () String)
(assert (str.contains
  (str.++ "grep "(str.substr SymbolicField_1 5 25))
   " $(touch success);#"))
(check-sat)
(get-model)
```

(d) SMT constraints for the toy example API with node IDs.

```
{ "id": "", "types": ["Bot"], "structure": {
    "filename": {
      "id": "47341750",
      "types": ["String"],
      "structure": {},
      "concrete": "BCDEA$(touch success);#G" }}}
```

(e) Concretized abstract value for the toy example API.

```
{ "filename": "BCDEA$(touch success);#G" }
```

(f) Candidate exploit for the extended toy example API.

Fig. 6: Generating an exploit for a toy example.

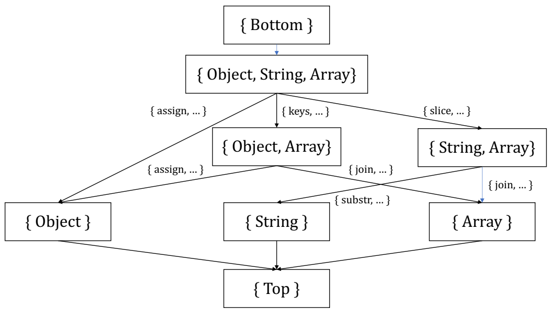


Fig. 7: Type lattice for object, string, and array types. Only a subset of edge labels are included for readability.

root of the provenance graph, the sink `exec`. At this point, the algorithm has inferred that the `query` input is an object with a field `"filename"` of string type. This is sufficient information for the synthesis algorithm to generate SMT constraints as shown in Figure 6d and eventually generate a successful exploit payload.

2) Inferring Types and Structure: Using the provenance graph and the fact that JavaScript imposes restrictions on what operations may be performed on a value of a particular type, we can infer types of values appearing in the graph.

Type lattice for type inference. We use a type lattice to represent knowledge of provenance graph value types. In Figure 7, we present a simplified type lattice graph for the JavaScript types `object`, `string`, and `array`. A type lattice is a partially ordered set where each subset is a collection of JavaScript types. Subsets are related to each other via a partial order relationship: *type compatibility*, which also represent refinement of our knowledge of a value's type. If we are at $\{String, Array\}$ because we have observed an operation that can be performed on both strings and arrays, then we see an operation that can only be performed on strings, we can then refine our knowledge of the type to $\{String\}$. We make two additional adjustments: 1) we generalize operations to *fields* to include type-specific properties, e.g., the `length` property of strings and arrays; 2) we label the edges of the type lattice graph with the list of operations that, if seen, would cause us to transition from one subset to another.

We have developed an algorithm to automatically derive the type lattice for JavaScript types. This type lattice computation is done once for a JavaScript language version. Details can be found in our technical report [37].

Algorithm 1 Types and Structure Inference

```
 1: $\mathcal{T} \leftarrow$ getTypeLattice(), $\mathcal{G} \leftarrow$ getProvenanceGraph()
 2: $\mathcal{P} \leftarrow$ getPaths($\mathcal{G}$), $\alpha \leftarrow$ {types: [], structure: { }}
 3: for path $p$ in $\mathcal{P}$ do
 4:     $\tau \leftarrow \bot$
 5:     for node $n$ in $p$ do
 6:         if $n.operation$ is builtin then
 7:             $\tau \leftarrow \mathcal{T}$.transition($\tau$, builtin($n.operation$))
 8:         else if $n.operation$ is GetField then
 9:             $\tau \leftarrow \mathcal{T}$.transition($\tau$, field($n.value$))
10:             $\alpha' \leftarrow$ {types: [], structure: []}
11:             $\alpha.structure[n.value] \leftarrow \alpha'$
12:             $\alpha \leftarrow \alpha'$, $\tau \leftarrow \bot$
13:         else if $n.operation$ is sink then
14:             $\tau \leftarrow \mathcal{T}$.transition($\tau$, sink($n.operation$))
15:         end if
16:         $\alpha.types \leftarrow \alpha.types \cup \tau$
17:     end for
18: end for
```

Traverse paths from the provenance graph. We extract a set of *paths* in the provenance graph from the package input nodes to the sink node. There is only one runtime path from each input to the sink; execution stops when a sink is reached. We define Algorithm 1 for inferring package API input types, taking as input the type lattice and the extracted paths. Our type for the leaf starts as *Bottom*. Along the way, we extract the field f of each visited node. We then consult the lattice and possibly perform a transition, depending on f, to a new refined type set. Transitions are labeled with either the field (for built-in operations), a wildcard (for other operations), or an exclamation point (for sink operations). We then continue until we reach the sink node, at which point we have obtained the most refined inference possible for the type of the input.

For example, the inferred type starts as `Bot` (*Bottom*); shown in Figure 6b. When we reach the access (`GetField`) of the field `"filename"` (node 5 in Figure 5) the type of the input is refined to `Object`, as non-numeric bracket field access is only supported for Objects. After the `GetField` operation, the type is reset to `Bot` because we are now inferring the type of the `"filename"` field. Once we reach the substr field (node 4 in Figure 5) the inferred type transitions to `String`, which is the correctly refined type of the `"filename"` field.

3) Inferring Structure: In addition to inferring that the `query` input is an object, we need to reconstruct its fields. This requires analyzing the field access operation in the provenance graph and reconstructing the fields and integrating their inferred types.

Inferring structure along provenance graph paths. The algorithm for inferring structure walks the provenance graph path, checking for field access operations (e.g., `GetField`). When a field access operation is found, the field's name is extracted, and then the remaining path is recursively analyzed. The result of the recursive call will be a new abstract value; in Figure 6c, this is the object assigned to the `"filename"` field.

Abstract values are only computed for tainted leaf nodes of the provenance graph (the attacker-controllable inputs). The example contains a single such input, `query`, as a result there is just one abstract value in the result. Currently, we do not support inference with multiple values (Section V-D).

For the toy example, as shown in Figure 6c the structure of the `query` input is inferred to be an object with a field `"filename"`, that is a string (which is structureless). This is a sufficient structure for the `query` input, given the behavior of the package API captured in the provenance graph.

4) Integration with Synthesized Payloads: To use inferred types and structure in the exploit synthesis process, we first augment the provenance graph with type information for each node, which we extract from the inferred abstract value (Section IV-C3). If the operation is a field access, we extract the inferred types of the field from the abstract value and add it as an annotation to the node. We label such a node as a `SymbolicField`, to be used in the SMT formula.

The synthesis algorithm will then generate SMT constraints from the augmented provenance graph. Solutions to the resulting formula are a set of strings corresponding to parts of the package inputs. As a final step, we insert these strings into the inferred abstract value to generate the final exploit. We only need to match the ID of the provenance node and the ID of abstract values, which are preserved across all the operations. The generated SMT constraints for our example is shown in Figure 6d. The SMT constants are prefixed with the provenance node ID, e.g., `SymbolicField_47341750`, which corresponds to the ID `47341750` of the `"filename"` field of the `query` as shown in Figure 6c. We solve the SMT statement with Z3 as described in Section IV-D and process the output of Z3 into `{'47341750': 'BCDEA$(touch success);#G'}`. Then, we can insert the solved strings into the abstract value as the field `"concrete"`. The resulting abstract value is in Figure 6e.

Finally, we *concretize* an abstract value by traversing the structure and replacing the abstract value with concrete ones from the SMT solutions. The final concretized result for our example is: `{'filename': 'BCDEA$(touch success);#G'}`.

D. Fine-grained Constraints

NODEMEDIC's synthesis algorithm derives SMT constraints from provenance graphs, which are then solved to generate candidate exploits. However, it does not handle the semantics of common JavaScript string operations (e.g., negative indices in `string.slice`), nor coercion operations (e.g., `"1"+ 2`), nor potential sanitization of the exploit payload. One of the areas that NODEMEDIC-FINE improves upon NODEMEDIC is to extend the synthesis algorithm with 1) additional models for JavaScript operations; 2) robust handling of JavaScript coercion; and 3) variations of exploit payload.

SMT models for JavaScript operations. SMT models for JavaScript operations are necessary to generate the SMT constraints to be solved for generating exploit payloads. For example, when NODEMEDIC encounters a concatenation operation in the operation tree, NODEMEDIC calls the Z3 concatenation operation with rewritten ASTs of the subtrees. We extended this approach to handle additional common JavaScript string operations such as `string.slice` and `string.replace` found in our dataset (Section V-A). The complexity of modeling these operations includes: 1) matching JavaScript semantics to Z3 operations and 2) storing additional constraints in a *context* to generate the final SMT formula.

Handling implicit coercion. JavaScript will implicitly coerce non-string values to strings in a number of cases, such as when an array is joined into a string, or when any non-string value is concatenated with a string. Without taking into account when values are converted to strings, the SMT formulas will be ill-formed, limiting our capability to generate exploits. The cause of this limitation is that NODEMEDIC does not have access to native (i.e., within the JavaScript engine) operations performed on values and thus does not include coercion operations in the provenance graph. NODEMEDIC-FINE improves upon NODEMEDIC by 1) transforming the provenance graph by inserting coercion operations explicitly; and 2) by providing SMT models for these coercion operations.

First, we traverse the graph and insert *coercion nodes* where we identify an implicit coercion would happen in JavaScript. For example, if we see a `string.concat` operation with a non-string argument, we insert a coercion node to convert the non-string argument to a string. This must be handled on a case-by-case basis. Second, we define SMT models for these coercion operations. For example, we model the coercion of a number to a string using the z3 `IntToStr` operation.

Variations of exploit payloads. To generate exploits, we need to find a compound string: $s_{\texttt{pre}} + s_{\texttt{pay}} + s_{\texttt{suf}}$, where $s_{\texttt{pre}}$ completes what comes before it, $s_{\texttt{pay}}$ delivers the exploit payload, and $s_{\texttt{suf}}$ causes whatever comes after it not be executed. Selections of $s_{\texttt{pre}}$, $s_{\texttt{pay}}$, and $s_{\texttt{suf}}$ are dictated by

```
module.exports = {
  evaluate: function(expr) {
    var out = new Function(
      "return 2*(" + expr + ")");
    return out();
  }
};
```

Fig. 8: Vulnerable entry point of a synthetic example with an arbitrary code execution vulnerability

the vulnerability type and sourced from known exploits. NODEMEDIC has one fixed string for each. NODEMEDIC-FINE instead allows the synthesis algorithm to pick from a set of variations, increasing its capability to generate valid exploits. We encode in SMT constraints a disjunction of variations.

E. Generating Valid JavaScript Payloads

Another area that NODEMEDIC-FINE improves over NODEMEDIC is its novel Enumerator component for synthesizing syntactically valid JavaScript payloads, which is a key challenge for confirming potential ACE flows. As seen in Section IV-B, ACI flows can be consistently confirmed by using payloads with shell meta-characters that escape most contexts. This does not apply to ACE flows; the final argument to the sink needs to not only be valid JavaScript, but also execute the intended payload. Figure 8 shows a synthetic example that demonstrates these challenges.

This example shows an entry point where the expected functionality is to return a number corresponding to the double of the result of evaluating the given argument as a mathematical expression. If we import the package and use it like so: `evaluate('1+1')` it returns 4. Notice that the expression to evaluate is given as a string which is interpreted as JavaScript.

A naive solution is `evaluate('1); console.log("VULN FOUND") //')`, with `1);` being the breakout sequence to finish the current expression. However, the exploit fails. The problem is that once JavaScript executes the instruction `return 2*(1);` it ignores what comes next, as the return statement just finishes the execution of the current function. A successful exploit injects the payload *before* closing the current expression, like: `evaluate('console.log("VULN FOUND")) //')`. Note that the final argument to the `Function` sink in this case is `return 2*(console.log("VULN FOUND")) //)`. We close the parenthesis context right after the payload and before the `//` comment start, otherwise an error would be thrown complaining that the expression is syntactically invalid, as the open parenthesis would never be closed.

Enumerator. We use Enumerator to construct an *objective payload*, which is the final string that will be passed to `eval` or the `Function` constructor. It is capable of constructing a final payload that obeys all syntactic constraints and executes the intended statement. The Enumerator is given a prefix, such as `return "(` and outputs a number of alternative payload templates, each with a placeholder for a statement to execute.

A payload template is a list where each element has one of the following types:

Literal: A constant string, usually with syntactic connectors.
Payload: The placeholder for the payload.
Identifier: This can be replaced with a valid variable name. It is important that the final JavaScript expression does not use undefined variables.
FreshIdentifier: This can be replaced with a valid variable name that was not used before, as some JavaScript expressions have to use fresh variables.
GetField: This can be replaced with any valid attribute.

An example payload template that the Enumerator outputs for the package and prefix described above is: `[Literal("return 2*("), Payload(), Literal(")"]`. Next, we discuss how payload templates are generated.

Graph representation. The Enumerator internally uses a graph representation for JavaScript syntax. Each node is a symbol representing a JavaScript syntactic category, such as variable names and elements for the template described above. The root is a node that represents the start of a new JavaScript expression. Collecting all symbols on a path from a node to the root yields a valid payload template, which together with the prefix string can be instantiated to a valid JavaScript program. Thus the transition between node A and B is only allowed if going to node B allows for a valid completion. To use the graph, the Enumerator starts from the beginning of the prefix, and finds the node matching the first symbol of the prefix, then follows the transition based on the next symbol. When the last symbol of the prefix is reached, the Enumerator uses the graph edges to generate the template. It performs a reachability analysis and outputs all paths that can reach the root of the graph from the current nodes. Each path is a valid template to complete the prefix. Details of how Enumerator keeps track of additional context to ensure the validity of the generated payload can be found in Appendix A-C.

Connection with SMT synthesis. To leverage NODEMEDIC-FINE's ability to handle sanitization measures and other constraints in the package, each element in a chosen template payload is turned into a symbolic variable by the synthesis algorithm (except Literals which are constant strings). Our synthesis infrastructure proceeds to synthesize an SMT statement where the argument to the sink is constrained to be equal to the concatenation of each element in a payload template where each variable has its own constraints, e.g., FreshIdentifier elements are unique.

Approach feasibility. JavaScript is a context-sensitive language, so it is impossible to represent all syntax in this way [38]. We found that the current primitives supported by the Enumerator are sufficient to complete most prefixes that we found in the wild under 0.1 seconds with negligible memory consumption.

V. EVALUATION

We evaluate the effectiveness of NODEMEDIC-FINE in detecting and automatically confirming ACI and ACE flows and compare it with prior Node.js dynamic taint analyses and a state-of-the-art static analysis tool FAST [16], which also supports proof-of-concept exploit generation. For ACI flows we focus on the effect of the inference methodology (Section IV-C) that is needed to generate the rich structures seen in ACI, but not ACE flows, which expect string inputs. For ACE flows we investigate the effect of the Enumerator component (Section IV-E) that generates completions of JavaScript code needed for ACE sink inputs, but not ACI sinks, which expect shell code. Finally, we evaluate the ability of NodeMedic-FINE to discover previously unidentified vulnerabilities in npm packages. We answer the following research questions:

RQ1: How effective is type-aware fuzzing (Section IV-A) at uncovering potential ACE, ACI flows?

RQ2: Does inference (Section IV-C) improve synthesis for confirming ACI flows?

RQ3: Is synthesis with the Enumerator (Section IV-E) effective for confirming ACE flows?

RQ4: How does NodeMedic-FINE compare to FAST [16] in the SecBench.js [39] dataset?

A. Experiment Setup and Dataset

Experiment setup. Experiments were deployed via Docker containers on two Ubuntu 20.04 VMs, each with 12 cores. Packages were analyzed in parallel; one container per instance of NodeMedic-FINE analyzing a package, restricted to using 4GB of RAM. We repeated this process with several variants of NodeMedic-FINE configured with key components disabled to evaluate the effect of each component. The workflow for analyzing each package is as follows: First, a driver is generated. The fuzzer (Section IV-A) is used in the driver depending on the variant. Next, the driver executes until it either times out, crashes, or finds a potential flow. The timeout for fuzzing is set to 2 minutes (Appendix A-B). If a flow is found, a second driver (no fuzzer) that only calls the API that triggers the flow is generated and executed to collect a minimal provenance graph. Next, we generate proof-of-concept exploits. We first test the polyglots as discussed in Section IV-B. Finally, if the polyglot is unsuccessful, we then run our synthesis algorithm (Section IV-C-IV-E).

Datasets. The first dataset consists of packages from npm. We gathered *all* packages from npm with at least 1 weekly download; 1,732,536 packages in total. From this set, we analyzed *all* 33,011 packages that contained calls to sinks NodeMedic supports (Section II). We describe the gathering process in detail in Appendix A-A. Package sizes range from 56 bytes to 236 MB, download counts are between 1 and 171,158,063 weekly downloads, and the number of dependencies is between 1 and 1366. We also evaluated NodeMedic-FINE against the 40 ACE and 101 ACI vulnerabilities available in SecBench.js, an increasingly popular dataset for server-side JavaScript.

Evaluation baseline. We include NodeMedic-MC, which is NodeMedic [21] enhanced with additional SMT models and support for implicit coercion (Section IV-D), as the baseline for comparisons with NodeMedic-FINE. NodeMedic-FINE's synthesis engine works on potential flows reported by using the fuzzer, which contributes to a large number of potential flows and thus indirectly increases confirmed flows as compared to NodeMedic. To make a fair comparison to NodeMedic, we use NodeMedic-MC +*Fuzzer* as the baseline for synthesis, which simply adds the fuzzer on top of the previous baseline.

TABLE I: Overall evaluation results and comparison to prior Node.js dynamic taint analysis tools.

		NodeMedic-FINE	*NodeMedic-MC*	*NodeMedic [21]*	*Ichnaea [9]*	*AFFOGATO [10]*
	Packages	33011	33011	10000	22	21
Potential	Total	**2257**	1338	155	15	17
	ACI	1788	1163	133	9	-
	ACE	469	175	22	6	-
Auto-conf.	Total	**766**	463	108	-	-
	ACI	612	396	102	-	-
	ACE	154	67	6	-	-

B. Overall Evaluation Results

The overall evaluation results broken down by type of flow (Section II) is shown in Table I. We compare the number of potential and automatically confirmed flows found by NodeMedic-FINE to those found by NodeMedic [21], and by two contemporary Node.js dynamic taint analysis tools, Ichnaea [9] and AFFOGATO [10]. Their scale [9, 10] is limited because they lack an automated analysis pipeline; they require manual driver creation, analysis invocation, and exploit confirmation. To our knowledge, the evaluation performed for NodeMedic-FINE is the largest-scale dynamic taint analysis of ACI and ACE flows in the Node.js ecosystem to date. In 33,011 packages, NodeMedic-FINE finds 2257 potential flows, among which 1788 are ACI flows and 469 are ACE flows. NodeMedic-FINE automatically confirms 766 flows, among which 612 are ACI flows and 154 are ACE flows. Among all confirmed flows found by NodeMedic-FINE, 1 ACE and 25 ACI are already-disclosed unpatched vulnerabilities. To date, we have been assigned 1 ACI CVE (Section V-H) and received acknowledgment from 54 developers that the reported vulnerabilities were real (48 ACI + 6 ACE).

In 33,011 Node.js packages, NodeMedic-FINE uncovers 2257 potential flows and confirms 766 of them automatically; 1.7x potential and 1.6x auto-confirmed flows compared to NodeMedic-MC.

C. RQ1: Fuzzer Performance

We evaluate the fuzzer's impact on identifying potential flows (Table III). The first column indicates the fuzzer's configuration: default (NodeMedic-FINE) also referred to as the *full* fuzzer; disabling object reconstruction (*No ObjRecon*); disabling type-aware fuzzing (only generating strings) (*No Types*); compared to NodeMedic-MC, which does not use fuzzing. Additional and missing potential flows compared to the full fuzzer are in the second and third columns, respectively.

The full fuzzer performs much better than no fuzzer, resulting in 919 additional flows. Type-awareness in the fuzzer is responsible for finding 391 extra potential flows compared to a fuzzer that only generates strings. Disabling type-aware fuzzing makes the fuzzer faster at finding flows that require strings, yielding 35 extra flows, most of which can be found by the normal fuzzer given a sufficiently long timeout, except for 5 that crash due to out of memory.

Object reconstruction contributed to finding 228 extra potential flows. These were cases where the packages required inputs to be objects having a certain structure, similar to our example in Section II. Disabling object reconstruction also allows the fuzzer to find 34 extra flows. The limited time budget for fuzzing causes this; 26 of these 34 flows can be found by the full fuzzer with longer timeouts while the remaining 8 cases crash due to out of memory. Sometimes the coverage-guidance that object reconstruction uses leads the fuzzer away from generating inputs that trigger potential flows. For example, one package prints an error and does not call the sink if a certain attribute is present in the user input. The object reconstruction will generate these attributes as coverage would increase; however, the absence of those attributes is needed for triggering the potential flow. A flow is found in this package by the fuzzer with object reconstruction capability disabled.

Result 1a: Type- and object-structure aware fuzzing uncovers 2257 potential flows; 1.7x the flows of NODEMEDIC-MC. Object reconstruction is necessary to find 228 flows. Generating diverse types yields 391 more flows compared to generating only strings.

We examine the impact of generating each input type during fuzzing and summarize the results in Table II. Each row indicates how many flows we miss by running a version of NODEMEDIC-FINE where the fuzzer can not generate a specific type.

Most flows can be triggered by more than a single input type, due to how loosely typed JavaScript is. This is why the total of flows missed only goes up to 1085 instead of the total 2257 potential flows found. Clearly, strings and objects are the most important to be generated, otherwise we would miss 609 and 243 flows, respectively. The ability of the fuzzer to generate other types also contribute to the discovery of a reasonable number of flows. Take the 34 cases where functions need to be passed to the packages as an example, most of those packages take, as argument, a callback that is called before the sink, and would crash if that argument is not a function.

Result 1b: By generating a variety of types the fuzzer has the ability to discover flows that it would otherwise miss.

D. *RQ2: Inference Performance*

We present evaluation results on the impact of the ACI polyglot (Section IV-B) and type and structure inference (Section IV-C), and discuss their limitations.

Impact of ACI polyglot and inference. Table IV reports extra, missing, and total counts of automatically confirmed ACI flows across six conditions: NODEMEDIC-FINE: inference of types and structure, ACI polyglot enabled; *No polyglot*: instead of using the ACI polyglot, we use NODEMEDIC's input `$(touch /tmp/success)`; *No type inf.*: inference of types disabled; *No inference*: inference of types *and* structure disabled; NODEMEDIC-MC *+Fuzzer*: the baseline condition with the fuzzer; and NODEMEDIC-MC. Below, we discuss the impact of each condition.

The ACI polyglot contributes 8% to the increase in confirmed flows over NODEMEDIC-MC *+Fuzzer*. The increase is due to the polyglot being able to bypass weak shell expansion sanitization in the package (Section IV-B). The three extra flows found when disabling the polyglot break down to two cases where the polyglot leads to invalid shell code (e.g., a bash loop), and one case where SMT solving with the polyglot times out.

Inferring (non-string) types yields a 15% increase in confirmed flows over NODEMEDIC-MC *+Fuzzer*. Through manual examination, we find that inferring array types is the key factor in the increase. Finally, inference of types and structure together contributes 77% to the increase in confirmed flows over NODEMEDIC-MC *+Fuzzer*. In these cases, the common pattern is that a structured object is required as input and a field of the object is used in the call to the sink. Without inferring this structure and inserting the synthesized exploit payload at the correct field, the payload fails to reach the sink.

The confirmed ACI flows missed by NODEMEDIC-FINE without inference of types and structure and the polyglot are the same flows missed by NODEMEDIC-MC *+Fuzzer*. The extra flows found by NODEMEDIC-MC *+Fuzzer* and NODEMEDIC-MC are due to disabling the polyglot.

A case study of a real package mirroring our example in Section IV-C can be found in the Appendix A-D. Inference of types and structure increases the complexity of the SMT formulae and synthesized package input, but does not introduce a performance bottleneck on average (Appendix A-E).

Result 2: NODEMEDIC-FINE's improvements to ACI confirmation are attributable to the ACI polyglot (8%), inferring non-string types (15%), and inferring structure (77%), yielding a total increase of 39 flows over baseline. These flows correspond to packages requiring specific types (avg: 1.3 types synthesized) and structure (avg: 1.2 fields synthesized).

Synthesis limitations for ACI. We manually triaged the top 100 packages, ranked by weekly downloads, where our infrastructure found a potential flow but failed to generate an exploit, yet the package was exploitable.

Of these, 79% were `spawn` sinks. In the majority (61%) of cases, failure was due to the lack of support for synthesizing multiple (two) inputs to the package; exploiting the `spawn` sink requires control of a command string *and* an options object. Furthermore, it is only possible to exploit `spawn` if the `shell` flag is set to true in the options object, the lack of which resulted in 15% of failures. The remaining cases for spawn required additional type information (4%), synthesis of a specific payload (3%), or a package specific issues (5%; e.g., requiring a particular input for a git command, executing Python code).

TABLE II: Potential flows missed by the fuzzer when it can not generate inputs of a given type.

Type Removed	*Strings*	*Objects*	*Arrays*	*Functions*	*Numbers*	*Regexes*	*Booleans*	*BigInts*	*Nulls*	*Undefined*	*Symbols*	*Total*
Flows missed	609	243	62	34	25	24	20	19	17	16	16	1085

TABLE III: Potential flows found by the fuzzer with varied configurations. Extra and missing flows are relative to the ones found by NODEMEDIC-FINE.

Condition	**Extra**	**Missing**	**Total**
NODEMEDIC-FINE	-	-	2257
No ObjRecon	34	228	2063
No Types	35	391	1901
NODEMEDIC-MC	0	919	1338

TABLE IV: Impact of inference of types and structure on ACI confirmed flows. Fuzzer with object reconstruction enabled except for NODEMEDIC-MC. Extra and missing flows are relative to NODEMEDIC-FINE.

Condition	**Extra**	**Missing**	**Total**
NODEMEDIC-FINE	-	-	612
No polyglot	3	6	609
No type inf.	0	6	606
No inference	0	35	577
NODEMEDIC-MC +*Fuzzer*	3	39	576
NODEMEDIC-MC	2	218	396

TABLE V: Confirmed ACE flows found while enabling or disabling several components of synthesis. Fuzzer with object reconstruction was enabled for all of these. Extra and missing flows are relative to the ones found by NODEMEDIC-FINE.

Condition	**Extra**	**Missing**	**Total**
NODEMEDIC-FINE	-	-	154
No Enumerator	0	27	127
No Polyglot	0	26	128
NODEMEDIC-MC +*Fuzzer*	3	54	103
NODEMEDIC-MC	1	88	67

For `exec`-like sinks, which only require control over a single string input and have shell evaluation on by default, synthesis failed for 21% of the packages. These cases generally required a more complex nested structure for the input along a path not found by the fuzzer (52%), synthesizing a specific non-payload input string, e.g., a valid file path or a specific character (14%) bypassing sanitization (10%), additional type information (5%), or a combination of these (14%).

We provide more details for the limitations described above, as well as details for the remaining corner cases for both `spawn` and `exec` sinks, in Appendix A-F.

E. RQ3: Enumerator Performance

We report on the effectiveness of our ACE polyglot and the Enumerator in confirming ACE flows. Table V summarizes the impact of disabling several NODEMEDIC-FINE components individually in the confirmation of ACE flows. We report the number of extra, missing and total counts of automatically confirmed ACE flows across five conditions: NODEMEDIC-FINE: uses polyglot and Enumerator; *No Enumerator*: uses polyglot; *No Polyglot*: uses Enumerator; NODEMEDIC-MC +*Fuzzer* and NODEMEDIC-MC.

ACE polyglot. Our ACE polyglot (Section IV-B) is more effective than using a simpler exploit `global.CTF();//`, increasing the number of confirmed flows from 128 to 154. The improvements are in situations where the payload is injected inside a string value and insufficient sanitization measures allow an attacker to escape that context.

Impact of completing prefixes. The Enumerator was called for 328 prefixes (205 unique) and came up with a valid prefix completion for 191 (58%) of those cases.

The Enumerator contributes to 27 confirmed ACE flows. All 27 cases required a complex payload to be constructed, involving the insertion of the payload in the right place, escaping the necessary contexts at the right time and, in some cases, an extra suffix concatenated after the prefix and our payload. An example is given in Appendix A-C.

We manually inspected 8 out of the 153 packages that we completed the prefix but could not automatically exploit. Four were not exploitable. Of the remaining 4, 2 had such intricate constraints that Z3 timed out, as they involved solving for inputs that passed through a JavaScript parser called *jsep* before reaching the sink or exploiting a stack machine; 1 package required a model for the **slice** operation where the length is symbolic to successfully construct the SMT statement for Z3; and 1 package required a call to the function returned by the entry point with an object argument. In all these cases, the Enumerator synthesized a valid completion but there were additional challenges that NODEMEDIC-FINE would need to overcome to create a working exploit.

Result 3: The Enumerator helped NODEMEDIC-FINE complete the majority of real world prefixes that we found in ACE flows, increasing the number of total confirmed ACE flows by 21%.

Limitations of the Enumerator. The Enumerator failed to complete the prefix for 137 packages with ACE flows. This was most commonly due to the need to complete JavaScript code that contained primitives not supported by our Enumerator. Lacking support for loops, nested objects, boolean expressions and the `+=` operator caused 63 out of these 137 failures. There were 12 cases where the prefix could not be completed even by a perfect Enumerator, because our synthesis algorithm does not handle multiple inputs (Section A-F). The argument to the sink was a combination of constant strings from the package and several attacker controlled inputs. When the prefix was

TABLE VI: SecBench.js eval results comparing NODEMEDIC-FINE (NM-F) with FAST in terms of potential (Pot.) and confirmed (Conf.) flows. Valid packages are downloadable, have a main executable file defined, and the vulnerability fits in the attacker model that we share with FAST. Executable are packages that are valid and can be installed and run.

Type	Valid	Executable	Pot. NM-F	Pot. FAST	Conf. NM-F	Conf. FAST
ACI	91	87	51	65	44	41
ACE	34	25	17	10	5	0
Total	125	112	68	75	49	41

passed from the synthesis algorithm to the Enumerator, it was already impossible to be completed. The remaining 62 cases needed a diverse set of JavaScript primitives to be supported by the Enumerator, including but not limited to class definitions, try/catch statements and generator functions.

Anomalous cases. NODEMEDIC-MC +*Fuzzer*, having no inference, confirmed 3 extra flows for all of which NODEMEDIC-FINE's inference generated a malformed SMT formula (Appendix A-F). The fuzzer was needed to find the potential flow in two of them, but the third one was found by NODEMEDIC-MC too, which resulted in its 1 extra flow.

F. RQ4: Comparison with FAST

Table VI compares NODEMEDIC-FINE with FAST [16] by reporting the flows found by running each tool on the SecBench.js dataset comprised of 101 ACI and 40 ACE real world vulnerabilities. We ran FAST and NODEMEDIC-FINE against 91 ACI and 34 ACE that fit our attacker model[3] and were downloadable and had a main executable file defined in package.json. From those, only 87 ACI and 25 ACE could be installed and run. As a dynamic analysis tool, NODEMEDIC-FINE can only analyze those cases, but being executable is not a prerequisite for FAST to find potential flows so we report all results of FAST on valid packages.

The most frequent reason why FAST finds a higher number of potential flows is because unlike NODEMEDIC-FINE it does not need to come up with an input that follows the potentially vulnerable path. There were also 3 cases where FAST exclusively found a flow because NODEMEDIC-FINE suffered from an undertainting issue. NODEMEDIC-FINE performs well with respect to confirmed flows, specially ACE vulnerabilities. FAST generated candidate exploits for 4 ACEs that ended up not executing the payload because the final argument to the sink was not valid JavaScript. Our enumerator allowed us to get past that problem in those cases. FAST failed to synthesize the right type for one of the arguments of the vulnerable entry point of the package *macaddress@0.2.8*, which needed to be a function. NODEMEDIC-FINE eventually generated a function for that argument and was able to create a working exploit.

[3]One discarded vulnerability required the command-line arguments to be attacker-controlled. The remaining vulnerabilities were not exploitable from the main package entry points but rather from an internal library's entry points.

TABLE VII: True and false positive rates for both confirmed flows and potential flows NODEMEDIC-FINE fails to confirm.

	Confirmed		Un-Confirmed	
Sink Type	TP	FP	TP	FP
ACI	64 (50 new)	40	0	14
ACE	6 (6 new)	3	4 (3 new)	11

Result 4: NODEMEDIC-FINE is comparable to state-of-the-art tool FAST in automatically detecting and synthesizing real-world vulnerabilities. NODEMEDIC-FINE excels at confirming ACE flows, which are typically hard to confirm.

G. Developer responses

So far, we have triaged 622 confirmed flows (567 ACI + 55 ACE). We emailed the developers of all vulnerabilities that we considered to be new true positives (270 ACI + 19 ACE). As of the time of writing, we received 56 responses (50 ACI + 6 ACE). 2 developers said they did not agree it was a true vulnerability because the attacker model did not apply to them, as they considered impossible for an attacker to control the entry point's arguments. The remaining 54 developers agreed that the reported vulnerabilities were real (48 ACI + 6 ACE). So far, 35 of these have been patched and a new version of the package is published. In 12 other cases, developers asked for more time to fix the vulnerability. For the remaining 7 vulnerabilities, the developers agreed it should be patched but said they do not have time to do so.

H. Previously Unidentified Vulnerabilities

We report on NODEMEDIC-FINE's true and false positive rates of identifying true vulnerabilities. A vulnerable flow is an exploitable, truly illegitimate behavior according to the package functionality.

We sample 113 flows automatically confirmed to be exploitable by NODEMEDIC-FINE and 29 flows from the most popular packages where a potential flow was identified but not automatically confirmed, and we manually examine whether they are vulnerable. Results are summarized in Table VII. The number for the true positives in the parenthesis is previously unreported new vulnerabilities.

In all, 70 out of the 113 flows are truly vulnerable. Two of the false positives are in packages that warn users not to pass unsanitized inputs to vulnerable entry points. Two other packages were vulnerable, but deprecated. The remaining 39 cases were packages that exposed a sink directly or the vulnerable entry point was intended for arbitrary command execution. 3 packages had real vulnerabilities in a different entry point, which NODEMEDIC-FINE did not explore.[4]

Most of the vulnerabilities are due to a lack of sanitization. Two have inadequate sanitization, which is bypassed by inputs generated by NODEMEDIC-FINE. We were assigned 1 CVE [29]. Out of 54 packages with acknowledged vulnerabilities, 25 have weekly downloads in the range $(0, 10]$, 12 in

[4]This was because NODEMEDIC-FINE stops at the first potential flow it finds, which in these cases was not the ideal flow to exploit

$(10, 100]$, 7 in $(100, 1000]$, 3 in $(1000, 3000]$ and the remaining 7 with >3K weekly downloads were submitted to Snyk, by whom the developers are being contacted. We are in the process of responsibly disclosing the remaining true positives.

Among the 4 ACE vulnerabilities, 1 needs a more sophisticated exploit driver with multiple interactions with the API to exploit the flow; 1 has complex SMT constraints and Z3 outputs `unknown`; and 2 packages needed the Enumerator to support class definitions and passing object arguments in function calls.

The ACI false positives were discussed in Section V-D. For ACE false positives, 1 was due to overtainting; 5 had proper sanitization; 2 packages were deprecated; and 1 package called the function constructor but the resulting function was never used. In the remaining 2 packages the inputs to the package API are a boolean or a number which can not contain a command or code to be injected in the sink.

VI. Limitations and Future Work

In this section we discuss limitations of our analysis and future work to improve NodeMedic-FINE.

Fixing vulnerabilities While our tool is designed to automatically exploit vulnerabilities, it can not automatically fix them. An effective mitigation for ACI is to use the more secure function *execFile*, which allows developers to properly separate the binary or command to execute from its potentially user-influenced flags. For ACE, avoiding calling dynamic code execution functions like *eval* with user-controlled arguments is best, otherwise proper input sanitization is paramount.

Missing information from instrumentation-based analysis. The inference methodology is limited by the underlying instrumentation-based dynamic analysis [21, 40] because it relies on the provenance graph, constructed by the underlying analysis. Imprecise or incomplete information typically result from uninstrumented code, which can appear in native operations not implemented in JavaScript or functions imprecisely analyzed by the underlying analysis for scalability concerns. Leveraging information from static analysis could further improve NodeMedic-FINE.

SMT models of JavaScript operations. An inherent limitation of constraint-based synthesis is its dependence on bespoke SMT models for JavaScript operations, which are time-consuming and error-prone to create due to quirks in the JavaScript language semantics. For instance, JavaScript's implicit coercion must be added to the SMT models on a per-operation basis because these coercions happen within the JavaScript engine and not visible to the instrumentation-based analysis. A related limitation, shared by prior work that applies SMT-solving techniques towards JavaScript analysis [41, 42, 43], is that the SMT solver may fail to find a solution within a reasonable time limit. Regular expression operations are known to be challenging to solve [41].

Multi-input synthesis. The inference methodology works poorly when more than one tainted inputs are given to the package API due to the following two limitations of the current infrastructure. First, the dynamic taint analysis infrastructure does not distinguish between multiple *kinds* of taint; thus, tainted paths from different inputs are indistinguishable. Second, the inference does not handle merging of abstract values from multiple tainted paths. As future work, we will include support for multiple kinds of taint by modifying the underlying taint map and propagation. We will also extend the inference to distinguish abstract values from different inputs and only merge those from the same input.

Shell string completion. To handle all cases (e.g., including sanitization) associated with synthesizing ACI shell code payloads that complete a shell string prefix or suffix, we would need a methodology similar to the Enumerator for ACE.

More complex drivers. NodeMedic-FINE does not generate sophisticated drivers needed for confirm flows where an exploit is only triggered if sequences of package API calls are performed, or handlers or external interactions (e.g., with the network, a database, or the file system) are executed. Prior client and server-side JavaScript taint analysis work has encountered similar limitations [9, 10, 22, 23, 32]. Beyond improving driver generation, one could analyze instead, packages that have simpler driver requirements and calls entry points of those packages that require complex drivers.

Multiple flows in the same package. NodeMedic-FINE stops after finding the first flow for each package, causing the analysis to miss vulnerabilities in a package if the package has multiple flows and the first one is a false positive. This is not a fundamental limitation of NodeMedic-FINE; we can implement an iterative pipeline to analyze all flows.

VII. Related Work

NodeMedic-FINE uses NodeMedic's underlying dynamic taint analysis engine to identify potential flows and to output important runtime information used for synthesizing proof-of-concept exploits. In the domain of detecting code-injection vulnerabilities in Node.js packages, some tools have used similar dynamic taint tracking techniques [9, 10, 40], while others used static approaches [11, 12, 13, 14, 15, 16, 17, 44, 45]. The synthesis algorithm depends on the output from the dynamic taint analysis, which can be obtained by other tools in the same category [9, 10, 40]. Thus, NodeMedic-FINE's synthesis methodology is generally applicable and can be implemented for these tools as well.

The dynamic taint tracking is not a contribution of NodeMedic-FINE, so we focus on closely related work in fuzzing and synthesis in the context of JavaScript.

General-purpose fuzzers adapted for Node.js. Fuzzing tools like AFL [46] have been adapted for Node.js fuzzing [47]. These general-purpose tools predominantly generate byte sequences or strings, lacking intrinsic knowledge of JavaScript's rich type system. While effective in many scenarios, searching the string space only is not sufficient to uncover a significant number of vulnerabilities. NodeMedic-FINE's fuzzer is type- and structure-aware and can generate inputs of a variety of types and with complex structure, like objects with specific attributes that have to be themselves objects.

JavaScript-specific fuzzers. Some approaches for input generation rely on package tests or even tests from its dependents to improve coverage in Node.js packages [48]. However, these tests do not always exist. JsFuzz [49] attempts to create coverage-guided JavaScript-specific fuzzing tools by facilitating the generation of inputs more suitable for JavaScript environments. However, their approach still heavily leans on string-based input generation and a manual creation of a fuzz target. This may not effectively explore the breadth of JavaScript's type system, which includes objects, arrays and function types. We observed through manual triage of the found potential flows that there is a considerable number of cases of vulnerable entry points that expect a function as one of the arguments. These would never be found fully automatically by state of the art fuzzers without knowing beforehand that one of the generated sequence of bytes would have to be transformed or replaced into a function.

SMT-based JavaScript exploration. While fuzzing helped NodeMedic-FINE to explore more execution paths of a JavaScript program, another commonly used method for program exploration is symbolic execution [30]. Several works perform symbolic execution for JavaScript [18, 41, 42] but the technique's limited scalability [50] conflicts with our goal of performing a large scale analysis on npm packages.

Synthesis. Several works use the JavaScript grammar to generate syntactically valid code [51, 52, 53] for fuzzing JavaScript interpreters. In comparison, our synthesis technique works at a finer granularity of syntactic constructions using SMT constraints: rather than generating numerous code chunks that are valid syntactically and semantically, but are arbitrary in their content, we need to synthesize specific sequences that bypass manipulation and deliver the payload.

Most prior work on JavaScript exploit synthesis targets cross-site scripting vulnerabilities [22, 23, 24, 25, 26]. They parse the AST of the statement reaching the sink to construct an exploit [22, 23, 24]. While feasible for webpages because *global* input sources (e.g., URL parameters) are accessed near the sink; it does not work for Node.js packages, where inputs are local and are often transformed before reaching the sink.

Several works use SMT solvers to synthesize exploits [27, 54, 55]. FAST [16] first generates a control-flow and an object dependence graph through abstract interpretation and finds a path between entry points and sink functions. It then generates a data flow that follows that control flow path, and solves constraints collected from both the data flow, the control flow path and the object dependence graph. FAST's synthesis uses only information from static analysis and thus the synthesis constraints may miss important dynamic information, e.g., more than 90% of FAST's false negatives are due to the lack of modeling of built-in functions, which come for free in NodeMedic-FINE since it executes the package.

In the domain of JavaScript synthesis, PMForce [27] synthesizes ACE exploits for the `postMessage` API's `event` object. PMForce gathers and uses *path* constraints to fill exploit templates used for `event.data`. Like the underlying NodeMedic [21], Our analysis also uses templates, but these encode ACE or ACI-specific breakouts, and in the case of ACE are produced by the syntactic analysis of the Enumerator. Moreover, the provenance graph encodes constraints *on operations* (not path constraints) that solve for structured inputs and ensure the exploit payload reaches the sink.

Closer to our approach, but applied to PHP, is the work of NAVEX [54], which uses a constraint-based approach to generate exploits. NAVEX is similar in that it uses constraints to select exploit payloads, but unlike our work, it does so by detecting uses of sanitization that would filter out certain attacks in an attack dictionary. NAVEX is also different from our approach in that it does not use dynamic provenance information, rather it uses path constraints to model vulnerable paths in a PHP application; it leverages Z3 to solve for inputs that jointly satisfy path constraints and constraints on the input to contain acceptable strings from the attack dictionary.

VIII. Conclusion

By leveraging type and object-structure information gathered at runtime, NodeMedic-FINE is able to explore more execution traces to identify more potential flows. The type- and structure-inference together with the Enumerator component, which is capable of completing prefixes to valid Javascript syntax, significantly improve the performance of the proof-of-concept exploit generation.

Acknowledgment

This work is supported in by the Future Enterprise Security Initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab), Carnegie Mellon CyLab, and Fundação para a Ciência e a Tecnologia (UIDB/50008/2020, Instituto de Telecomunicações, and PhD grant SFRH/BD/150692/2020).

References

[1] "Npm passes the 1 millionth package milestone! What can we learn?" 2021, http://tinyurl.com/npm-1-millionth.

[2] P. Muncaster, "Open Source Supply Chain Attacks Surge 430%," 2020, https://www.infosecurity-magazine.com/news/open-source-supply-chain-attacks/.

[3] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[4] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[5] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An Empirical Study of Information Flows in Real-World JavaScript," in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019.

[6] L. Gong, "Dynamic analysis for javascript," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2018.

[7] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.

[8] 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," in *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021.

[9] R. Karim, F. Tip, A. Sochurkova, and K. Sen, "Platform-Independent Dynamic Taint Analysis for JavaScript," *IEEE Transactions on Software Engineering*, 2018.

[10] F. Gauthier, B. Hassanshahi, and A. Jordan, "AFFOGATO: Runtime detection of injection attacks for Node.js," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.

[11] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven Node.js JavaScript applications," *ACM SIGPLAN Notices*, 2015.

[12] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting Taint Specifications for JavaScript Libraries," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[13] S. Li, M. Kang, J. Hou, and Y. Cao, *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*, 2021.

[14] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS," in *NDSS*, 2018.

[15] 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)*, 2022.

[16] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling JavaScript abstract interpretation to detect and exploit node.js taint-style vulnerability," in *IEEE Symposium on Security and Privacy*, 2023.

[17] M. Kluban, M. Mannan, and A. Youssef, "On detecting and measuring exploitable JavaScript functions in real-world applications," *ACM Transactions on Privacy and Security*, 2024.

[18] 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)*. USENIX Association, 2021.

[19] T. M. Corporation, "CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.3)," 2020–, https://cwe.mitre.org/data/definitions/94.html.

[20] ——, "CWE - CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection') (4.3)," 2020–, https://cwe.mitre.org/data/definitions/77.html.

[21] D. Cassel, W. T. Wong, and L. Jia, "NodeMedic: End-to-end analysis of node.js vulnerabilities with provenance graphs," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023.

[22] 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.

[23] 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*, 2015.

[24] 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.

[25] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, "Towards automated generation of exploitation primitives for web browsers," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[26] Y. Frempong., Y. Snyder., E. Al-Hossami., M. Sridhar., and S. Shaikh., "Hijax: Human intent javascript xss generator," in *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT,*, 2021.

[27] M. Steffens and B. Stock, "PMForce: Systematically analyzing postMessage handlers at scale," in *ACM Conference on Computer and Communications Security*, 2020.

[28] CERT, "The CERT guide to coordinated vulnerability disclosure," 2023, https://vuls.cert.org/confluence/display/CVD.

[29] "CVE-2024-21488," Available from Snyk, Snyk-ID SNYK-JS-NETWORK-6184371, Jan. 2024, https://security.snyk.io/vuln/SNYK-JS-NETWORK-6184371.

[30] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*, 2010.

[31] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A Survey of Dynamic Analysis and Test Generation for JavaScript," *ACM Computing Surveys*, 2017. [Online]. Available: https://doi.org/10.1145/3106739

[32] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-patching DOM-based XSS at scale," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[33] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[34] piercus, "Hasard," https://www.npmjs.com/package/hasard, 2020, npm package version 1.6.1.

[35] K. Sen and M. Sridharan, "Jalangi2," 2014–, https://github.

com/Samsung/jalangi2.

[36] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[37] D. Cassel, N. Sabino, M.-C. Hsu, R. Martins, and L. Jia, "NodeMedic-FINE: Automatic detection and exploit synthesis for node.js vulnerabilities (technical report)," *Carnegie Mellon Kilthub*, 2024, DOI:10.1184/R1/27901461.

[38] C. Martín-Vide, V. Mitrana, and G. Păun, *Formal languages and applications*. springer, 2013, vol. 148.

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

[40] 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*, 2013.

[41] B. Loring, D. Mitchell, and J. Kinder, "ExpoSE: Practical symbolic execution of standalone JavaScript," in *SPIN 2017*, 2017.

[42] 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*, 2018.

[43] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "JaVerT 2.0: Compositional symbolic execution for JavaScript," *Proceedings of the ACM on Programming Languages*, 2019.

[44] N. Patnaik and S. Sahoo, "Javascript static security analysis made easy with JSPrime," in *Blackhat USA*, 2013.

[45] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[46] M. Zalewski, "American Fuzzy Lop (AFL)," 2024, software available from http://lcamtuf.coredump.cx/afl/.

[47] AFLFuzzJS, "afl-fuzz-js: A JavaScript Port of the American Fuzzy Lop Fuzzer," 2014, https://github.com/tunz/afl-fuzz-js.

[48] H. Sun, A. Rosà, D. Bonetta, and W. Binder, "Automatically assessing and extending code coverage for npm packages," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 40–49.

[49] JSFuzz, "Jsfuzz," GitHub repository, 2020, available at: https://github.com/fuzzitdev/jsfuzz.

[50] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[51] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[52] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security*, 2016.

[53] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *Network and Distributed System Security*, 2019.

[54] A. Alhuzali, R. Gjomemo, B. Eshete, and V. N. Venkatakrishnan, "NAVEX: precise and scalable exploit generation for dynamic web applications," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18, 2018.

[55] S. Park, D. Kim, S. Jana, and S. Son, "{FUGIO}: Automatic exploit generation for {PHP} object injection vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

Appendix A
Additional Evaluation Details

A. Gathering of Evaluation Dataset

From the (>2M) packages in npm as of October 2023, we gathered those that have at least 1 weekly download (1,732,536 packages). In Figure 9, we show the number of packages that get filtered out at each stage of the gathering pipeline, until we are left with 33011 packages; our evaluation dataset. **setupPackage** ensures the package can be downloaded. **filterByMain** filters out packages that can not be imported because they do not define a main file. **filterBrowserAPIs** filters out packages that depend on browser APIs. The **filterSinks** discards packages that do not contain calls to ACE or ACI sinks visible to static analysis. Note that we also check if any of the dependencies have calls to sinks. **setupDependencies** filters out packages whose dependencies fail to download or install. **getEntryPoints** discards packages that do not have any public entry points defined. We gather metrics and annotate dependencies to not instrument in the **annotateNoInstrument** stage. Finally, in **runJalangiBabel** we instrument the package code using Jalangi.

B. Analysis Timeout

We have a hard timeout of 2 minutes for fuzzing. Figure 10 shows that after 30 seconds we start to have diminishing returns on the number of total potential flows found. We would not expect to find a large enough number of new potential flows if we increased the timeout further.

C. Enumerator Graph and Example

A section of the Enumerator graph representing the JavaScript language is shown in Figure 11. Several nodes are shown, including *Root*. Edges between nodes are such that we can confidently build syntactically valid JavaScript statements by traversing the graph. Note that a graph traversal is stateful and following an edge changes the state. State changes

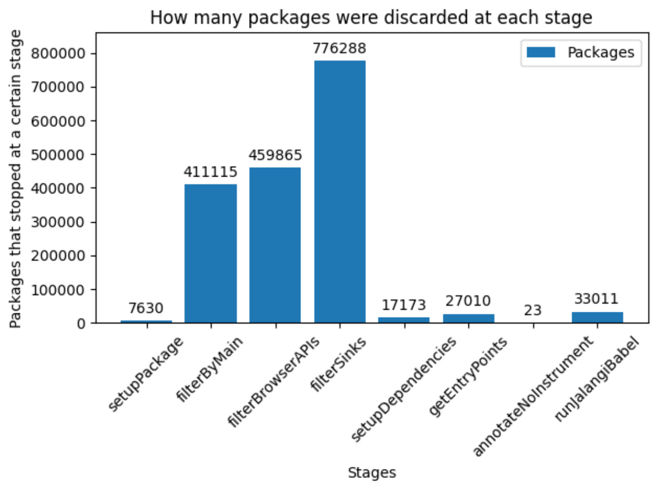


Fig. 9: How many packages were filtered out, by stage.

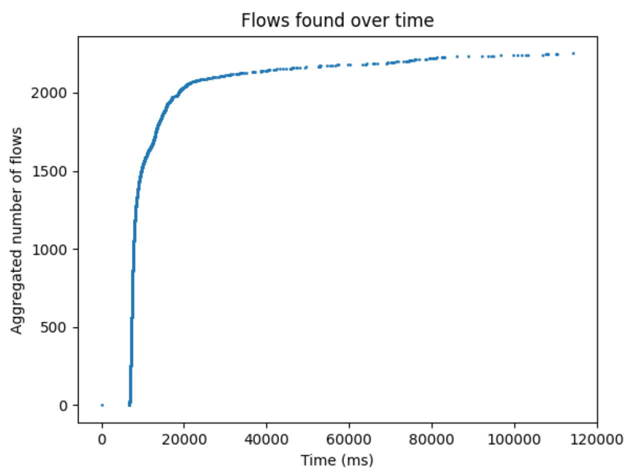


Fig. 10: How many flows would be found (y-axis) if we set the fuzzing timeout to (x-axis in miliseconds).

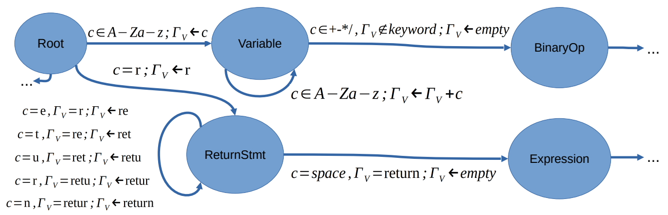

Root
Variable
BinaryOp
ReturnStmt
Expression
$c \in A-Za-z ; \Gamma_V \leftarrow c$
$c \in$ +-*/ $, \Gamma_V \notin keyword ; \Gamma_V \leftarrow empty$
$c =$ r $; \Gamma_V \leftarrow$ r
$c \in A-Za-z ; \Gamma_V \leftarrow \Gamma_V + c$
$c =$ e $, \Gamma_V =$ r $; \Gamma_V \leftarrow$ re
$c =$ t $, \Gamma_V =$ re $; \Gamma_V \leftarrow$ ret
$c =$ u $, \Gamma_V =$ ret $; \Gamma_V \leftarrow$ retu
$c =$ r $, \Gamma_V =$ retu $; \Gamma_V \leftarrow$ retur
$c =$ n $, \Gamma_V =$ retur $; \Gamma_V \leftarrow$ return
$c = space , \Gamma_V =$ return $; \Gamma_V \leftarrow empty$
...
...
...

Fig. 11: A section of the graph representation of JavaScript syntax used by the Enumerator. Edges have labels $C;U$ where C is a condition over the current character in the prefix c and the context Γ_V. U is a context update.

```
// Code showing the sink call
return new Function("x",
  "with (x) { return " + user_input + " } ")
// Prefix
with (x) { return
// Completion
[[  <payload>, <literal: '}'> ]]
// Exploit
global.CTF()} //
```

Fig. 12: Prefix, completion and the final exploit synthesized for a real world prefix

```
exports.process = function(node, tree, cb) {...
  childproc.exec(
  "coffee -o " + node.out + " -c " + node.files,
  function() { e = arguments[0],
    out = arguments[1], err = arguments[2];
    return cb(e, out + '\n' + err); }); ...}
```

Fig. 13: *b****@0*** code vulnerable to ACI.

```
{"id": "",
"types": ["Object"],
"structure": {
    "out": {
        "id": "c0a0f881",
        "types": ["Bot"],
        "structure": {}},
    "files": {
        "id": "bb7d142f",
        "types": ["Bot"],
        "structure": {}}}}
```

Fig. 14: Abstract value inferred for *b****@0***.

are labeled in Figure 11 on top of the edges, and encode constraints that would be hard to represent with a simple graph. For example, *function* is an invalid variable name, therefore we can not transition from the node *Variable* to node *BinaryOp* if the currently parsed variable name Γ_V is *function* or any other in a list of JavaScript keywords, thus we need the condition $\Gamma_V \notin keyword$ on that edge.

To illustrate how the Enumerator works, we show in Figure 12 an example of a prefix adapted from one of the 27 cases that the Enumerator successfully completed, together with the final synthesized exploit. Note the closing brackets after the main payload, without which the exploit would be a syntactically invalid statement and would not execute.

D. Inference ACI Case Study

We present a case study of a package, *b****@0***, sourced from our evaluation to illustrate the benefits of inference of types and structure. The package takes a list of source input files and allows one to build CoffeScript files and output them to a directory. Our taint analysis detected a potentially vulnerable flow in the package's `process` function, which accepts a `node` argument whose two fields, `out` and `files`, are passed unsanitized to the ACI sink `exec` as shown in Figure 13.

Running our inference methodology on the package, we infer the abstract value shown in Figure 14. We can see that the `out` and `files` fields are inferred to be present on the input, which is inferred to be an object. The fields themselves are not inferred to have any structure, indicating they are some non-extensible type. They are not specifically inferred to be strings because the package API does not perform any operations on

```
(declare-fun SymbolicField_bb7d142f () String)
(declare-fun SymbolicField_c0a0f881 () String)
(assert (str.contains (str.++ "coffee -o "
  SymbolicField_c0a0f881 " -c " SymbolicField_bb7d142f)
  " $(touch success);#"))
(check-sat)
(get-model)
```

Fig. 15: SMT formula generated for *b****@0***.

```
1  try {
2    var x0 = {"out": "B", "files": "$(touch success);#"};
3    var x1 = undefined;
4    var x2 = undefined;
5    new PUT["process"](x0,x1,x2);
6  } catch (e) { console.log(e); }
```

Fig. 16: Exploit driver for *b****@0***.

TABLE VIII: Characteristics of ACI SMT formulae and synthesized inputs generated by NodeMedic-FINE with inference of types and structure enabled.

Characteristic	**Measurement**
SMT formula size (bytes)	256
SMT symbolic input count	1.3
Z3 solving time (ms)	21.8
Synthesized field count	1.2
Synthesized value depth	0.9
Inferred type count	1.3

them that would require them to be strings. At the same time, the type string is a valid type for these fields so our synthesis methodology will treat them as strings.

Running our synthesis methodology on the package, we generate the SMT formula shown in Figure 15. We can see that the `out` and `files` fields are treated as strings, and the SMT formula encodes the constraints that the first string must be a completion of the prefix `"coffee -o "`, the second string must be a completion of the prefix `" -c "`, and the concatenation of the symbolic and literal strings must contain the payload `" $(touch success);#"`. Solving this with Z3, we obtain the satisfying assignments `SymbolicField_c0a0f881 = "B"` and `SymbolicField_bb7d142f = "$(touch success);#`. Matching the assignments to the abstract value, we can derive the candidate exploit input: `{"out": "B", "files": "$(touch success);#"}`.

Finally, we construct the exploit driver, which is shown in Figure 16; we can see that the driver simply constructs the candidate exploit input (line 2) and passes it to the package API (line 5). We run the exploit driver and confirm that the exploit is successful by checking for the presence of the file `success`, which is created.

E. Complexity of Synthesis with Inference

To understand the impact of inference of types and structure on complexity of the SMT formulae and synthesized package input, in Table VIII, we examine relevant characteristics for all flows *with* inference of types and structure enabled.

The measurements show that results of synthesis produce package inputs that are not trivial, having typically 1 or 2 distinct required fields and 1 or 2 different types. However, the resulting formulae are compact and could be solved in under a second on average.

```
1  function doSpawn(method, command, args, options) {
2    ...
3    var cpPromise = new ChildProcessPromise();
4    var reject = cpPromise._cpReject;
5    var resolve = cpPromise._cpResolve;
6    var successfulExitCodes = (options
7      && options.successfulExitCodes) || [0];
8    var cp = method(command, args, options);
```

Fig. 17: Code snippet from *c****@2***.

```
return new Promise<string>(resolve => {
  child_process.exec(`yarn why '${dep}' --json`,
                     (err, output) => { ...
```

Fig. 18: Code snippet from *d****@1***.

F. Limitations of ACI Synthesis

We provide additional details on limitations of synthesis with inference of types and structure for ACI flows.

Multi-input synthesis. Of the 100 exploitable flows, 48 of those packages had the `spawn` sink and accepted both a command string *and* an options object that were passed to `spawn`. Under these conditions it is possible to exploit the sink if the `shell` flag is passed in the options object. However, our synthesis methodology does not support synthesizing two inputs to a single sink (e.g., a payload as well as an options argument with the appropriate flag). Thus, we were unable to synthesize exploits for these packages.

To illustrate, consider the following example of the package *c****@2***. In Figure 17, we present a code snippet along the exploitable code path of the package. The procedure on line 1 is called by the package's entry point with the method to execute (which receives a reference a function that calls `spawn`), as well as a command, arguments, and options that get passed directly in the method call on line 8. NodeMedic-FINE synthesizes the command `$(touch /tmp/success);#`, but does not synthesize an options argument of the form `{'shell': true}`, thus causing the exploit payload's shell metacharacters to not be executed.

Infrastructure and synthesis bugs. We encountered 10 cases where an exploit failed to be synthesized due to bugs. Two of these cases were due to the generation of a malformed SMT formula, wherein the formula lacked a symbolic input to solve for, thus preventing the generation of a payload. The remaining two cases were due to bugs in processing synthesis results, leading to valid synthesized payloads being lost. In both cases, if the synthesized payload was used, the flow would have been automatically confirmed.