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

Darion Cassel
Carnegie Mellon University[*]

Nuno Sabino
Carnegie Mellon University

Ruben Martins
Carnegie Mellon University

Limin Jia
Carnegie Mellon University

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 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 1966 potential flows and automatically synthesized exploits to confirm 622 of them.

1 Introduction

The Node.js ecosystem is vast and ever-growing, with millions of JavaScript packages available through the package management system npm alone [2]. 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 [13, 18, 21, 33, 45, 51, 54]. Prior work has shown that many packages in the Node.js ecosystem contain security vulnerabilities [17, 23, 24, 25, 27, 28, 30, 43, 44, 50]. 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 [10, 11].

Prior work has developed automated analyses to detect potential ACI and ACE vulnerabilities in JavaScript programs [7, 8, 15, 16, 17, 24, 26, 35, 43, 44, 46] and to synthesize proof-of-concept exploits to confirm them [7, 8, 15, 16, 26, 35, 46]. 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 [8, 17, 24]; the tool may report many potentially dangerous flows, but not indicate which can truly be exploited. To reduce the false-positive rates, prior work has explored using synthesis techniques to automatically generate proof-of-concept exploits for Node.js packages: NODEMEDIC was able to automatically confirm 102 ACI and 6 ACE flows in a sample of 10,000 packages. However, their approach has several limitations leading to failure to confirm 31 ACI flows and 16 ACI flows [8].

One fundamental challenge 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 implicates similar challenges. A second challenge to generating viable proof-of-concept exploits is that the algorithm has to take into consideration 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 in order to deliver the payload.

To address these challenges, we propose to leverage runtime information generated from dynamic taint tracking to (1) help a fuzzer to generate nontrivial inputs to explore more

[*]Work done prior to joining Amazon Web Services.

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. Prior work only took the initial steps in this direction and missed opportunities such as modeling implicit type coercion and JavaScript built-in string operations. Finally, we explore generating valid completions of JavaScript code string prefixes to synthesize ACE exploits.

We implement NODEMEDIC-FINE (Fuzzer, INference, Enumerator) for automatically detecting ACE and ACI flows and synthesizing proof-of-concept exploits to confirm them. We build on NODEMEDIC's dynamic taint analysis and naive exploit synthesis algorithm. First, we implement a novel type- and structure-aware fuzzer to explore the Node.js package. Second, we improve the synthesis algorithm by incorporating additional constraints based on effects of JavaScript operations and input types and structure. Finally, we implement an Enumerator component to synthesize valid completions of JavaScript code. These methodologies can be applied to any dynamic taint analysis engine.

We evaluate NODEMEDIC-FINE on 33,011 npm packages in active use that contain calls to ACE and ACI sinks. NODEMEDIC-FINE finds 1966 potential flows and automatically synthesizes exploits that confirm 622 flows. The type- and structure-aware fuzzer found 2.6x the number of potential flows that NODEMEDIC uncovered. The new synthesis component were pivotal in the confirmation of an additional 28 challenging confirmed flows, for a total 2.5x confirmed flows compared to NODEMEDIC. We plan to open source our tool upon publication.

Responsible disclosure. We follow a coordinated vulnerability disclosure process (i.e., responsible disclosure) [9] for the vulnerabilities discovered in our evaluation. We are in the process of triaging and responsibly disclosing our confirmed flows. Thus far, 1 high severity CVE [1] has been assigned.

2 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 1. 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 run 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

```
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     }};
```

Figure 1: An example ACI vulnerability

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*, e.g., the `exec` function (c.f. [40]). Certain program values, such as the above-mentioned sources, are labeled as *tainted* and the labels on these values are then *propagated* by program operations. For example, `params` in Figure 1 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. [5]).

We call a discovered flow from an attacker-controllable source to sensitive sink a *potential flow*, because it is not yet known if the flow can be exploited. Once a flow has been determined to be exploitable—meaning that an input can be provided to the package that results in an exploit payload successfully executing—we call the flow 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 [8], which is a dynamic taint analysis tool for identifying Arbitrary Code Execution (ACE) [11] and Arbitrary Command Injection (ACI) [10] 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

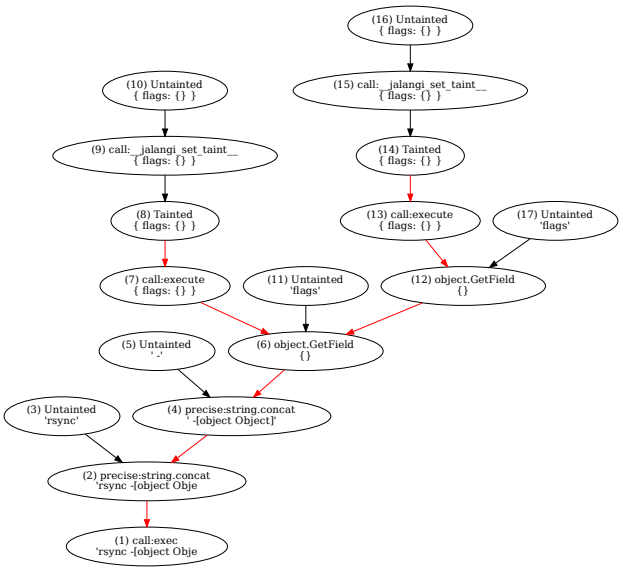

Figure 2: Example provenance graph for code in Figure 1

(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 1 is shown in Figure 2. 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 (i.e., creation of the file `success`). However, NODEMEDIC was not able to synthesize an exploit for this example, even though it reports a potential flow.

3 Motivation and Overview

Automatically generating exploits for packages like the one shown in Section 2 is challenging. We identify key challenges 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 [8]) 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 1 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 [7, 26, 34].

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 3. NODEMEDIC-FINE takes as input Node.js packages. To analyze the package and call its entry points, 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 4.1). 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 synthesizes 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 4.3). 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 4.5). The type inference algorithm and Enumerator create an SMT formula encoding the above mentioned constraints. By solving for symbolic variables rep-

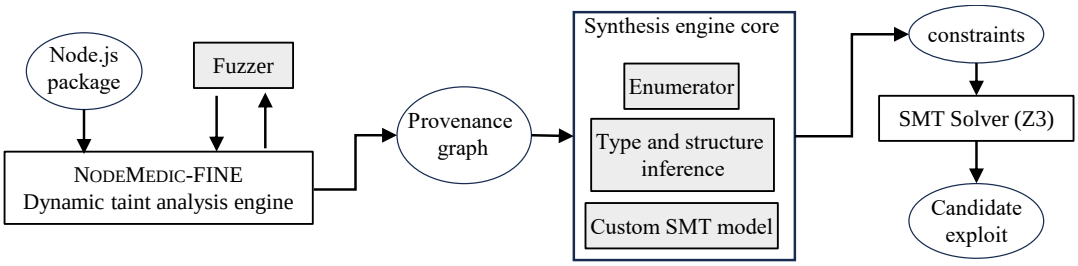

Figure 3: NodeMedic-FINE end-to-end pipeline for vulnerability detection and exploit generation

resenting package API input, Z3 [12] generates a satisfying instantiation of these variables, forming a candidate exploit.

4 NodeMedic-FINE Design

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

4.1 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. The fuzzer generates inputs based on the specification and sends them to be executed by NodeMedic-FINE.[1] NodeMedic-FINE returns coverage information and the attributes accessed via instrumented field access operations (*getField* [42]). The fuzzer takes this feedback and refines its input specification to start the next iteration of fuzzing, continuing 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 *i*th element specifies how many times the fuzzer sampled an input of the *i*th type in the types list (`sampled`); a list of coverage data for inputs of each type; where the *i*th element represents the accumulated number of lines of code triggered by generated inputs of the *i*th 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.

[1]The fuzzer utilizes the npm package Hasard [38] for generating random values according to a rigorous specification of the characteristics of the value.

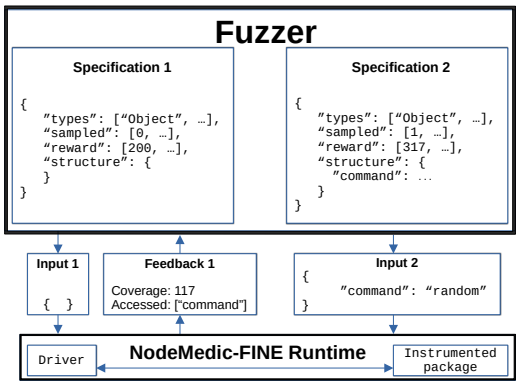

Figure 4: Fuzzer loop

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 commonly expected by 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 observe 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 [31].

When deciding which new type to explore, a natural way to model this decision is using a Poisson process [53]. In our

case, we consider each type t, with $\lambda_t = \frac{reward_t}{sampled_t}$. We consider the execution of a block of code to be the *independent* event Each type distribution estimates the expected number of lines of code that we will execute, if we choose this type.[2] We sample from each type distribution and choose the type with the largest sample value. Using this approach, it is more likely for input types that were effective in the past to have higher sample 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 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"`. Then, 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.

4.2 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 5.4-5.5).

4.3 Type and Structure Inference

Inputs generated by the fuzzer may have varied types and structures (Section 4.1). 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 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, e.g., objects, arrays, maps, sets, etc.

We first present a motivating example and give an overview of the technique (Section 4.3.1). Then we describe the type inference algorithm (Section 4.3.2) and the structure inference algorithm (Section 4.3.3). Finally, we describe how the inferred information is integrated into the exploit synthesis process (Section 4.3.4).

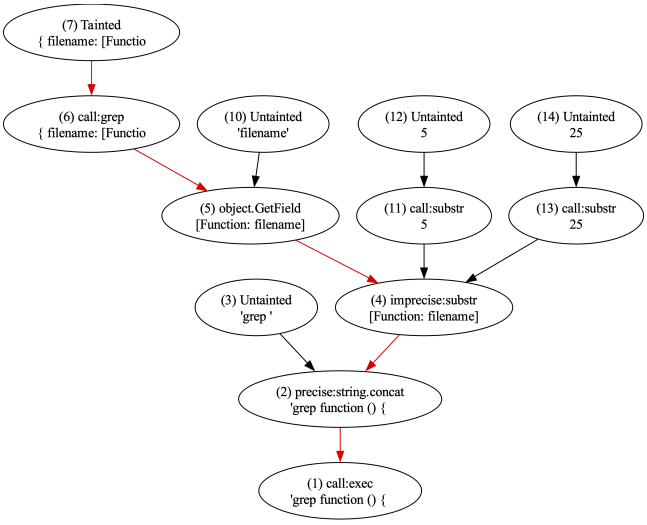

Figure 5: Provenance graph for toy example API.

4.3.1 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 struc-

[2] In reality, executed blocks of code are not independent due to control flow dependencies, but we assumed so for simplicity. Also, Hasard only supports sampling integers from Poisson distribution.

```
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_47341750 () String)
(assert (str.contains
    (str.++ "grep "
        (str.substr SymbolicField_47341750 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.

Figure 6: Generating an exploit for a toy example.

ture 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"` represents any JavaScript type. The presence of the `GetField` operation allows the inference to refine the type-set of the `query` input from *{Bottom}*, representing any JavaScript type, 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, Map}*. The algorithm also notes that the string value `"filename"` is part of the structure of the input. Finally, the algorithm reaches the 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.

4.3.2 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

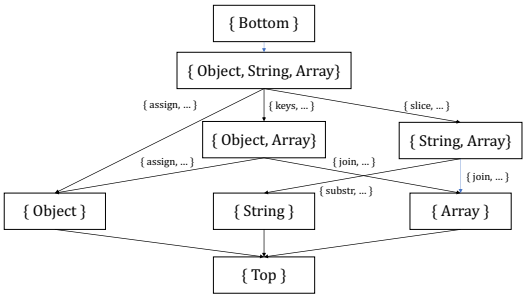

Figure 7: Type lattice for object, string, and array types. Only a subset of edge labels are included for readability.

knowledge of the type to $\{String\}$. We make two additional refinements: 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 also 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 Appendix B.1.

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 an algorithm (pseudo code in Appendix B.2) 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`. 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.

4.3.3 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 5.4).

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"`, which 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.

4.3.4 Integration with Synthesized Payloads

Next, we describe how inferred types and structure are used in the constraint-based exploit synthesis process. We first augment the provenance graph with type information for each node, which we extract from the inferred abstract value (Section 4.3.3). 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 4.4 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'}`.

4.4 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. NODEMEDIC-FINE extends NODEMEDIC's 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, NODEMEDIC models string concatenation as follows.

```
1  def generate_operation_tree(tree: OperationTreeNode):
2    return z3.Concat(*[
3      _generate(child) for child in tree.children])
```

When we encounter a concatenation operation in the operation tree, we call 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 5.1). 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. Appendix B.3 shows two models to illustrate these complexities.

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

```
1  module.exports = {
2    evaluate: function(expr) {
3      var out = new Function(
4        "return 2*(" + expr + ")");
5      return out();
6    }
7  };
```

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

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. Appendix B.4 explains the algorithm in detail.

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 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. Concretely, we encode in SMT constraints a disjunction of variations. Details are in Appendix B.4.1.

4.5 Generating Valid JavaScript Payloads

Synthesizing syntactically valid JavaScript payloads is a key challenge for confirming potential ACE flows. As seen in Section 4.2, ACI flows can be consistently confirmed by using payloads with shell meta-characters that escape most contexts. This does not apply for 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 now describe how Enumerator constructs an *objective payload*, which is the final string that will be passed to `eval` or the `Function` constructor. It differs from prior work like NODEMEDIC in its ability to construct 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 can have 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. We omitted details of how Enumerator keeps track of additional context to ensure the validity of the generate payload, in addition to the graph. These details can be found in Appendix Figure 28.

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 constrained to be unique.

Approach feasibility. JavaScript is a context-sensitive language, so it is impossible to represent all syntax in this way [32]. Still, 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.

5 Evaluation

We evaluate the effectiveness of NODEMEDIC-FINE in detecting and automatically confirming ACI and ACE flows:
RQ1: How effective is type-aware fuzzing (Section 4.1) at uncovering potential ACE, ACI flows?
RQ2: Does inference (Section 4.3) improve synthesis for confirming ACI flows?
RQ3: Is synthesis with the Enumerator (Section 4.5) effective for confirming ACE flows?

We also evaluate the ability of NODEMEDIC-FINE to discover previously unidentified vulnerabilities in npm packages, and compare it with prior Node.js dynamic taint analyses.

5.1 Experiment Setup and Dataset

Experiment setup. Experiments were deployed via Docker containers on two Ubuntu 20.04 VMs, each with 12 cores and 32GB of RAM. Packages were analyzed in parallel; one container per instance of NODEMEDIC-FINE analyzing a package. 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 4.1) 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.1). If a flow is found, a second driver (no fuzzer) that only calls APIs that trigger the flow is generated and executed to collect a minimal provenance graph. We test the polyglots that are effective for simple cases (Section 4.2). Finally, if the polyglot is unsuccessful, we then run our synthesis algorithm (Section 4.3-4.5).

Dataset. 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 2). We describe the gathering process in detail in Appendix 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.

Evaluation baseline. We include NODEMEDIC-MC, which is NODEMEDIC [8] enhanced with additional SMT models and support for implicit coercion (Section 4.4), as the baseline for comparisons with NODEMEDIC-FINE.

Table 1: Overall evaluation results and comparison to prior Node.js dynamic taint analysis tools.

		NODEMEDIC-FINE	*NODEMEDIC-MC*	*NODEMEDIC [8]*	*Ichnea [24]*	*AFFOGATO [17]*
	Packages	33011	33011	10000	22	21
Potential	Total	**1966**	752	155	15	17
	ACI	1673	690	133	9	-
	ACE	293	62	22	6	-
Auto-conf.	Total	**622**	246	108	-	-
	ACI	567	234	102	-	-
	ACE	55	12	6	-	-

5.2 Overall Evaluation Results

The overall evaluation results broken down by type of flow (Section 2) is shown in Table 1. We compare the number of potential and automatically confirmed flows found by NODEMEDIC-FINE to those found by NODEMEDIC [8], and by related Node.js dynamic analyses [17, 24]. 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 1966 potential flows, among which 1673 are ACI flows and 293 are ACE flows. NODEMEDIC-FINE automatically confirms 622 flows, among which 567 are ACI flows and 55 are ACE flows. Among all confirmed flows found by NODEMEDIC-FINE, 1 ACE and 21 ACI are already-disclosed vulnerabilities. To date, we have been assigned 1 ACI CVE (Section 5.6).

> In 33,011 Node.js packages, NODEMEDIC-FINE uncovers 1966 potential flows and confirms 622 of them automatically; 2.6x potential and 2.5x auto-confirmed flows compared to NODEMEDIC-MC.

Table 2: 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	-	-	1966
– *ObjRecon*	37	74	1929
– *Types*	50	195	1821
NODEMEDIC-MC	0	1214	752

5.3 RQ1: Fuzzer Performance

We evaluate the fuzzer's impact on identifying potential flows (Table 2). The first column indicates the fuzzer's configuration: default (NODEMEDIC-FINE) also referred to as the *full* fuzzer; disabling object reconstruction (– *ObjRecon*); disabling type-aware fuzzing (only generating strings) (– *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 column, respectively.

The full fuzzer performs much better than no fuzzer, resulting in 1214 additional flows. Type-awareness in the fuzzer is responsible for finding 195 extra potential flows compared to a fuzzer that only generates strings. Disabling type-aware fuzzing yields 50 extra flows, 25 of which can be found by the full fuzzer with longer timeout. The other 25 were lost in the second phase due to a serialization issue (Section 6).

Object reconstruction contributed to finding 74 extra potential flows. These were cases where the packages required inputs to be objects having a certain structure, similar to our example in Section 2. Disabling object reconstruction also allows the fuzzer to find 37 extra flows. The limited time budget for fuzzing causes this; 26 of these 37 flows can be found by the full fuzzer with longer timeouts while the remaining 11 cases crash due to out of memory. Sometimes 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. The fuzzer found a flow in this package when object reconstruction is disabled.

Result 1: Type- and object-structure aware fuzzing uncovers 1966 potential flows; 2.6x the flows of NODEMEDIC-MC. Object reconstruction is necessary to find 74 flows. Generating diverse types yields 195 more flows compared to generating only strings.

Table 3: 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	-	-	567
– *Inference*	0	21	546
NODEMEDIC-MC +*Fuzzer*	0	21	546
NODEMEDIC-MC	0	333	234

5.4 RQ2: Inference Performance

We present the evaluation results on the impact of type and structure inference and discuss its limitations.

Impact of types and structure inference. Table 3 reports extra, missing, and total counts of automatically confirmed ACI flows across four conditions: NODEMEDIC-FINE: inference of types and structure enabled; – *Inference*: inference of types and structure disabled; NODEMEDIC-MC +*Fuzzer*: the baseline condition with the fuzzer; and NODEMEDIC-MC. Disabling inference means that only string values are synthesized and synthesized values are used directly as inputs.

Disabling inference of types and structure results in 21 ACI flows missed, all of which are because a structured object is expected as input by the package API and only a field of the object is used in the call to the sink. The confirmed ACI flows missed by NODEMEDIC-FINE without inference of types and structure are the same flows missed by NODEMEDIC-MC +*Fuzzer*. This indicates that for ACI flows, inference of types and structure is the sole additional factor that contributes to the increase in confirmed ACI flows. A case study of a real package mirroring our example in Section 4.3 can be found in the Appendix C.1. 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 C.3).

Result 2a: Inference of types and structure leads to the automatic confirmation of 21 additional ACI flows. These flows correspond to packages whose inputs require specific types (2 on average) and structures (1.4 fields on average).

Limitations of extended synthesis for ACI. We manually triaged the top 25 packages, ranked by weekly downloads, where our infrastructure found a potential flow but failed to generate an exploit. Out of the 25, we found 11 exploitable ACI flows. The remaining 14 packages were false positives.

Of the 11 exploitable flows, synthesis failed for 6 due to the lack of support for synthesizing multiple (two) inputs to a single sink. All had the `spawn` sink and accepted a command string *and* an options object, both of which need to be synthesized. It is only possible to exploit the sink if the `shell` flag is set to

Table 4: 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	-	-	55
– *Enumerator*	0	7	48
– *Polyglot*	0	19	36
NODEMEDIC-MC +*Fuzzer*	3	7	51
NODEMEDIC-MC	1	39	17

true in the options object. One package required synthesis to complete a nontrivial string context to bypass shell expansion sanitization; this could be solved by extending the Enumerator methodology (Section 4.5) to apply to ACI flows. The other four cases are due to incomplete handling of corner cases in our implementation. A detailed breakdown and description can be found in Appendix C.4.

Result 2b: Inference of types and structure suffers from two fundamental limitations: (1) inability to synthesize multiple inputs to a single sink, and (2) inability to synthesize completions of shell strings.

5.5 RQ3: Enumerator Performance

We report on the effectiveness of our ACE polyglot and the Enumerator in confirming ACE flows. Table 4 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; – *Enumerator*: uses polyglot; – *Polyglot*: uses Enumerator; NODEMEDIC-MC +*Fuzzer* and NODEMEDIC-MC.

ACE polyglot. Our ACE polyglot (Section 4.2) is more effective than using a simpler exploit `global.CTF();//`, increasing the number of confirmed flows from 36 to 55. 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. We omit discussion of the ACI polyglot as it did not significantly change the number of confirmed flows (see Appendix C.2 for details).

Impact of completing prefixes. The Enumerator contributes to 7 confirmed ACE flows. All 7 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 B.5.

The Enumerator came up with a valid prefix completion for 101 unique packages total. We manually inspected 8 out of the 94 packages that we 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 3a: 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 15%.

Limitations of the Enumerator. The Enumerator failed to complete the prefix for 98 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 51 out of these 98 failures. There were 4 cases where the prefix could not be completed even by a perfect Enumerator, because our synthesis algorithm does not handle multiple inputs (Section C.4). The argument to the sink was a combination of constant strings from the package and several attacker controlled inputs. When the prefix was passed from the synthesis algorithm to the Enumerator, it was already impossible to be completed. The remaining 43 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 C.4). 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.

5.6 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 112 flows automatically confirmed to be exploitable by NODEMEDIC-FINE and 30 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 5. The number for the true positives in the parenthesis are previously unreported new vulnerabilities.

Table 5: True and false positive rates for both confirmed flows and potential flows NODEMEDIC-FINE fails to confirm.

Sink Type	Confirmed TP	Confirmed FP	Un-Confirmed TP	Un-Confirmed FP
ACI	63 (49 new)	40	1 (1 new)	14
ACE	6 (6 new)	3	4 (3 new)	11

In all, 69 out of the 112 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 cases (39) were packages that exposed a sink directly or the vulnerable entry point was intended for arbitrary commands execution. 3 packages had real vulnerabilities in a different entry point, which NODEMEDIC-FINE did not explore.[3]

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 [1]. 7 vulnerabilities in packages 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.

We failed to synthesize an exploit for the true ACI vulnerability due to a serialization issue (Section 6). 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 5.4. 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.

6 Limitations and Future Work

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

Missing information from instrumentation-based analysis. The inference methodology is limited by the underlying instrumentation-based dynamic analysis [8, 41] 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 [14, 29, 39], is that the SMT solver may fail to find a solution within reasonable time limit. Regular expression operations are known to be challenging to solve [29].

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 prorogation. 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 [17, 24, 26, 34, 35]. 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. Currently, NODEMEDIC-FINE stops after finding the first flow for each package. This will cause the analysis to miss vulnerabilities in a package if the package has multiple flows and the first one flow is a false positive. This is not a fundamental limitation of NODEMEDIC-FINE; we can implement an iterative pipeline to analyze all flows.

Serialization of inputs. Once an input that causes a vulnerable flow is found, we need to serialize it so that we can use it later to confirm the flow. We implemented basic serialization

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

for most JavaScript builtin types, but it is incomplete. Serializing JavaScript objects is known to be difficult, as a perfect implementation needs to handle a variety of edge cases like self referential objects. Bugs in our serialization of payload caused a few failures in confirming flows. However, this not a fundamental limitation in our approach. A more robust serialization is under development.

7 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, others tools have used similar dynamic taint tracking techniques [17, 24, 41], while others used static approaches [23, 25, 27, 28, 30, 37, 43, 44, 48]. The synthesis algorithm depends on the output from the dynamic taint analysis, which can be obtained by other tools in the same category [17, 24, 41]. 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 [52] have been adapted for Node.js fuzzing [3]. 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 [47]. However, these tests do not always exist. JsFuzz [22] 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 [40]. Several works perform symbolic execution for JavaScript [29, 39, 50] but the technique's limited scalability [6] conflicts with our goal of performing a mass scale analysis on npm packages.

Synthesis. Several works use the JavaScript grammar to generate syntactically valid code [19, 20, 49] 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 bypasses manipulation and delivers the payload.

Most prior work on JavaScript exploit synthesis targets cross-site scripting vulnerabilities [7, 15, 16, 26, 35]. They parse the AST of the statement reaching the sink to construct an exploit [7, 26, 35]. 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 [4, 36, 46]. In the domain of JavaScript synthesis, PMForce [46] 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 [8], 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 [4], 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.

8 Conclusion

We implemented NODEMEDIC-FINE an end-to-end analysis infrastructure for automatic detection and confirmation of ACI and ACE flows in Node.js packages. We applied NODEMEDIC-FINE to analysing popular packages with calls to sinks. NODEMEDIC-FINE automatically confirmed 680 exploitable flows. NODEMEDIC-FINE's fuzzer component

is more effective than prior work at finding potential flows. Our type inference algorithm and the Enumerator are shown to be capable of automatically confirming flows that were previously considered challenging for automated tools.

References

[1] Anonymized. CVE-2024-21488.

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

[3] AFLFuzzJS. afl-fuzz-js: A JavaScript Port of the American Fuzzy Lop Fuzzer, Year. Software available from URL.

[4] Abeer Alhuzali, Rigel Gjomemo, Birhanu 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*, SEC'18, 2018.

[5] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 2017.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[7] Souphiane Bensalim, David Klein, Thomas Barber, and Martin 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.

[8] Darion Cassel, Wai Tuck Wong, and Limin 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.

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

[10] The MITRE Corporation. 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.

[11] The MITRE Corporation. CWE - CWE-94: Improper Control of Generation of Code ('Code Injection') (4.3), 2020–. https://cwe.mitre.org/data/definitions/94.html.

[12] Leonardo De Moura and Nikolaj 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.

[13] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021.

[14] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 2019.

[15] Yaw Frempong., Yates Snyder., Erfan Al-Hossami., Meera Sridhar., and Samira Shaikh. Hijax: Human intent javascript xss generator. In *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT,*, 2021.

[16] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[17] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime detection of injection attacks for Node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.

[18] Liang Gong. *Dynamic Analysis for JavaScript*. PhD thesis, EECS Department, University of California, Berkeley, 2018.

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Network and Distributed System Security*, 2019.

[20] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[21] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav 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.

[22] JSFuzz. Jsfuzz. GitHub repository, 2020. Available at: https://github.com/fuzzitdev/jsfuzz.

[23] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. Scaling JavaScript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *IEEE Symposium on Security and Privacy*, 2023.

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

[25] Maryna Kluban, Mohammad Mannan, and Amr Youssef. On detecting and measuring exploitable JavaScript functions in real-world applications. *ACM Transactions on Privacy and Security*, 2024.

[26] Sebastian Lekies, Ben Stock, and Martin 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.

[27] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*. 2021.

[28] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[29] Blake Loring, Duncan Mitchell, and Johannes Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *SPIN 2017*, 2017.

[30] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 2015.

[31] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[32] Carlos Martín-Vide, Victor Mitrana, and Gheorghe Păun. *Formal languages and applications*, volume 148. springer, 2013.

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

[34] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Auto-patching DOM-based XSS at scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[35] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. DexterJS: Robust testing platform for DOM-based XSS vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[36] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. {FUGIO}: Automatic exploit generation for {PHP} object injection vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[37] Nishant Patnaik and Sarathi Sahoo. Javascript static security analysis made easy with JSPrime. In *Blackhat USA*, 2013.

[38] piercus. Hasard. https://www.npmjs.com/package/hasard, 2020. NPM package version 1.6.1.

[39] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, 2018.

[40] Edward J Schwartz, Thanassis Avgerinos, and David 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.

[41] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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.

[42] Koushik Sen and Manu Sridharan. Jalangi2, 2014–. https://github.com/Samsung/jalangi2.

[43] 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.

[44] Cristian-Alexandru Staicu, M. Pradel, and B. Livshits. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*, 2018.

[45] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei 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.

[46] Marius Steffens and Ben Stock. PMForce: Systematically analyzing postMessage handlers at scale. In *ACM Conference on Computer and Communications Security*, 2020.

[47] Haiyang Sun, Andrea Rosà, Daniele Bonetta, and Walter Binder. Automatically assessing and extending code coverage for npm packages. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 40–49, 2021.

[48] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[49] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, 2016.

[50] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[51] 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.

[52] Michal Zalewski. American Fuzzy Lop (AFL), 2024. Software available from http://lcamtuf.coredump.cx/afl/.

[53] Mingyi Zhao and Peng Liu. Empirical analysis and modeling of black-box mutational fuzzing. In *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings 8*, pages 173–189. Springer, 2016.

[54] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael 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.

A Gathering of Evaluation Dataset

Gathering consisted of collecting a list of packages and saving each and their dependencies locally using Verdaccio. This is done to save up bandwidth, as inevitably some packages will have the same dependencies as others we can get their code locally. From the (>2M) packages in npm currently, 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 usable packages, our finished dataset. We show all steps of our pipeline in the same order as they run. A package stops at the **setupPackage** if it can not be downloaded. The **filterByMain** stage filters out packages that can not be imported because they do not define a main file. A package stops in the **filterBrowserAPIs** stage when it is not intended for client-side usage as it is the case for the ones that require a browser. The **filterSinks** stage 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. We proceed to install the dependencies in the **setupDependencies** stage, which may error if we fail to download or install one of the dependencies. In the **getEntryPoints** stage we discard packages that do not have any public entry points defined. Finally, we gather useful metrics for characterizing the dataset in the **annotateNoInstrument** stage. The last stage is the **runJalangiBabel** where we instrument the package code using Jalangi, all remaining packages get through this stage and that constitutes the final dataset of packages that we are going to analyse. We store the package and dependencies, together with its instrumented counterpart for all popular packages with calls to sinks.

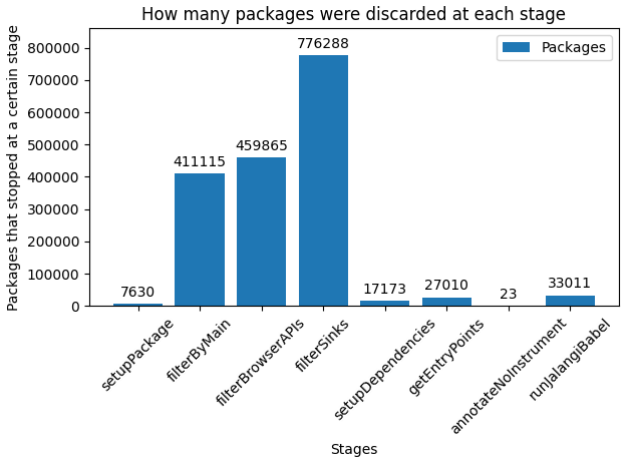

Figure 9: How many packages were filtered out, by stage.

A.1 Analysis Timeout

We have a hard timeout of 2 minutes for fuzzing. Figure 15 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, and 2 minutes is already quick enough that we can run our full pipeline against all packages that we found to contain calls to sinks over a day.

B Additional Synthesis Methodology Details

B.1 Type Lattice Derivation

While the type lattice presented in Figure 7 is relatively simple, an actual implementation of such a type lattice for the full gamut of JavaScript types is complex. JavaScript includes a set of seven primitive types as well as a large set of built-in types. Each of these types has a large set of fields–operations that can be performed on them, and properties they support. For example, the `string` type has 43 operations that can be performed on it (`substr`, `slice`, `replace`, etc.) This makes derivation of the type lattice time-consuming and error-prone to construct manually. Thus, we develop an algorithm to automatically derive the type lattice for JavaScript types. The algorithm does this dynamically in three steps: 1) enumeration of the built-in JavaScript types' fields; 2) attempted access of all fields; 3) derivation of unique fields; 4) lattice construction.

The approach for the first step is depicted in Figure 16. The algorithm iterates over each of the built-in JavaScript types as listed above, and for each type it enumerates the fields that are supported by the type. This is done by first enumerating the fields of the type itself, and then enumerating the fields of the type's prototype and constructor. The fields are then added to a list of fields for the type.

In step 1, the algorithm derived a list of all fields that are valid for particular types. However, this list is not complete. For example, the built-in `Map` type does not have a field `toString`, but it does support accessing the `toString` property because it is common to almost all JavaScript types. To address this, in step 2 we take the previous union of the lists of fields generated in step 1 (`allFields`), and attempt to access each field on each type. If the field is accessible, then we add it to the list of fields for that type. This is shown in Figure 17.

Another operation that is not captured in step 1 getting and setting of non-built-in properties. As expected, primitive types such as `string` and `number` do not support extension, e.g., `"foo"["a"]` will always result in an error, while `object["a"]` will not result in an error if `"a"` is defined on `object`. To capture this, as shown in Figure 18, we check if the type supports extension using the `Object.isExtensible` built-in JavaScript function.

Finally, in step 4, the algorithm constructs the type lattice (Figure 19). The algorithm iterates over *sorted valid set*, which is a list of all unique type sets and performs pair-wise comparison between these sets. For each pair it determines if there is a transition to be added. If there is a transition, then the algorithm adds an edge to the type lattice graph. These edges are labeled with the marker, `Field<${set}>`, which we can look-up in our previously derived list of fields to determine which fields cause the transition. In adding transitions, the algorithm considers two cases: 1) the transition is to a new type set; 2) the transition is to the same type set. In the first case, which happens when the second set is in a partial order relation with the first, the algorithm adds an edge from the first type set to the second type set indicating that a refinement is possible. In the second case, where there was overlap but not a partial order. The algorithm adds a self-loop to the first type set indicating no refinement. There are a few additions to the type lattice that are not shown in the pseudocode, such as adding transitions to *Top* and *Bottom*. Once all transitions have been added, the algorithm returns the type lattice. This type lattice computation is done once for a JavaScript language version. It is then is usable for analysis of any package using that JavaScript version.

B.2 Provenance Graph Path Traversal

With paths through the provenance graph and the type lattice, we define an algorithm for inferring package API input types. We show truncated pseudocode for the algorithm in Figure 20.

Starting from the leaf node of each provenance graph path, we perform a traversal to the sink node (the root of the tree). Our type for the leaf starts as *Bottom*. Along the way, we extract the field f of each visited node (denoted `node.field` in the psuedocode). We then consult the lattice and possibly perform a transition, depending on f. 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 the obtained the most refined inference possible for the type of the input.

B.3 Additional SMT Models

We extended NODEMEDIC to handle additional common JavaScript string operations found in our dataset (Section 5.1). We describe below the models for two representative models: 1) `string.slice`, which illustrates the complexity of matching JavaScript semantics to Z3 operations, and 2) `string.replace` which illustrates models that require storing additional constraints in a *context* used to generate the final SMT formula.

The semantics of `string.slice` are that the substring of the first argument starting at the index of the second argument and ending at the index of the third argument is returned. This corresponds to the semantics of the Z3 `Extract` operation. However, JavaScript also allows negative indices, which are interpreted as indexing from the end of the string. Thus, we must model this behavior in our SMT formula (Figure 22). We do this by first generating the ASTs of the first and third arguments, and then checking if the second argument is negative. If it is, we add the length of the first argument to the second argument before generating the AST of the second argument. We can then call the Z3 `Extract` after performing this transformation.

The model for the JavaScript `string.replace` operation is shown in Figure 23. The semantics of this operation are that the first argument is searched for the second argument, and if

it is found, it is replaced with the third argument. Since we are concerned with exploit generation, we are only interested in what is being *removed* by the replace operation. The character or substrings removed represent sequences that our exploit payload must *not* contain, or they could be stripped from the exploit payload, rendering it potentially unsuccessful. Thus, we model the `string.replace` operation by first generating the AST of the first argument, and then adding a constraint to our SMT context that the first argument does not contain the second argument, e.g., the substring that is being replaced. We then return the AST of the first argument, which represents the string that remains after the replace operation.

B.4 JavaScript Implicit Coercion Support

High-level selections of our implementation of modeling implicit coercions are shown in Figure 24. The function `_coerce` is called when a coercion node is encountered in the operation tree. This function first determines the type that the coercion is to, and then calls a corresponding helper function to generate the AST of the coercion. The helper function generates the AST of the coercion based on type. For example, the helper function `_generate_coerce_string` is called when the coercion is to a string. This function first checks if the coercion is from a number, boolean, or array, and then generates the AST of the coercion accordingly. In some cases this is straightforward; for example, if the coercion is from a number, the AST of the coercion is generated by simply calling the Z3 `IntToStr` method on the AST of the child node. The case of coercion of a boolean value to a string is also straightforward, but has no direct Z3 equivalent. Thus, we must manually add a constraint to the SMT context that the AST of the coerced value is either the string `"true"` or the string `"false"`.

Finally, the case of coercion of an array to a string is more complex. This is because the semantics of this coercion is that the elements of the array are joined into a string. In JavaScript, arrays can have any type of value, e.g., a mix of objects, strings and even subarrays. Naively representing coercion of each of these array elements would result in a blowup in formula size and solving time, potentially causing the solver to timeout within our time limit. Instead, we leverage the observation that in order to exploit an ACE or ACI vulnerability, the array must simply contain a single string that is attacker-controllable; if the array contains a correctly-constructed payload, it doesn't matter what the other elements of the array are. Thus, we model the coercion of an array to a string by simply selecting the first element of the array and assuming it is a string, as shown in the last branch in Figure 24. This is an unsound, but is sufficient for our purposes; if the array does not contain a string, then generation of a successful payload was never possible. As a result of this simplifying assumption, the model for the JavaScript `array.join(delimiter)` operation is straightforward, as shown in Figure 25. When generating the operation tree we insert a node representing a coercion of the array to a string. The `array.join` operation itself can then be modeled just as a concatenation of the coerced array (a string) to the delimiter string.

B.4.1 Exploit Payload Variations in SMT

Figure 26 illustrates generating varied ACI payloads. The function `construct_aci_payloads` enumerates a list of potential ACI payloads using different prefixes, payloads, and suffixes. Each of these are a possible exploit payload that the SMT solver can pick, given the other constraints in the SMT context. In the function `generate_sink`, we check if the SMT context is empty; if it is not, then we generate the above list of possible ACI payloads (if it is empty there are no constraints and any one of the payloads will work). This list of payloads is then converted into a list of Z3 constraints, each of which is a call to the Z3 `Contains` method, which checks if the AST of the child node contains the payload. These constraints are then combined into a single constraint using the Z3 `Or` method.

The SMT formula solving for an exploit payload generated by this procedure for the toy example is shown in Figure 27. As can be seen, the SMT formula contains a disjunction of constraints, each of which is a call to the Z3 `Contains` method, which checks if the AST of the child node contains a particular payload that was enumerated. Combined with other SMT constraints that may forbid particular characters, this will allow the solver to still find a satisfying assignment for the exploit payload.

B.5 Enumerator Applied to a Real Package

To illustrate the Enumerator performing in a real world scenario, we show in Figure 29 an example of a prefix adapted from one of the 7 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.

C Additional Evaluation Details

C.1 Inference ACI Case Study

Next, we present a case study sourced from our evaluation to illustrate the benefits of inference of types and structure. The case study is a package, *b****@0**, that 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 30.

Running our inference methodology on the package, we infer the abstract value shown in Figure 31. We can see that the `out` and `files` fields are inferred to be present on the input,

Table 6: Impact of ACI polyglot. Fuzzer with object reconstruction enabled for all conditions except NODEMEDIC-MC. Extra and missing flows are relative to NODEMEDIC-FINE.

Condition	Extra	Missing	Total
NODEMEDIC-FINE	-	-	567
– *Polyglot*	4	5	566
NODEMEDIC-MC +*Fuzzer*	0	21	546
NODEMEDIC-MC	0	333	234

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 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 32. 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 33; 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.

C.2 Effect of ACI Polyglot

In Table 6, we show the impact of the ACI polyglot on the number of confirmed flows: – *Polyglot*: instead of using the ACI polyglot, we use a simple shell expansion `$(touch /tmp/success)`; We see disabling the polyglot results in 4 extra and 5 missing flows.

Introducing the ACI polyglot did not significantly increase the number of confirmed flows, indicating that a simple shell expansion based payload is already powerful enough to cover most simple ACI cases. The four extra flows are cases where the package do not accept quotes in the payload, which the polyglot contains (Section 4.2). These cases could be addressed by using variations of the polyglot that do not contain quotes.

Table 7: 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)	244
SMT symbolic input count	1.2
Z3 solving time (ms)	24.43
Synthesized field count	1.4
Synthesized value depth	1.05
Inferred type count	2

Table 8: Causes of ACI synthesis failure along with the number of packages that failed due to each cause.

Cause	Count
Multi-input synthesis	6
String string completion	1
SMT malformed (missing symbolic input)	2
Infrastructure bug (results processing)	2
Total	**11**

C.3 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 7, we examine relevant characteristics for the 21 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 of at least 2 different types. However, the resulting formulae are compact and could be solved in under a second on average.

C.4 Limitations of ACI Synthesis

We provide additional details on limitations of the synthesis methodology extended by inference of types and structure for ACI flows.

Multi-input synthesis. Of the 11 exploitable flows, 6 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 34, 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.

Shell string completion. One package required the use of methodology similar to JavaScript completion the enumerator employs for ACE flows (Section 4.5). The package *d****@1** calls the `exec` sink with a command string that is constructed by concatenating a prefix string with the user input, `dep` (line 3 of Figure 35).

The prefix string includes the character `'` (a single quote). Within a single quote the shell treats all characters as literals, thus preventing evaluation of shell-metacharacters. Our polyglot exploit includes a single quote, but does not include a *suffix* to close the remaining single quote. In other words, NODEMEDIC-FINE produces the exploit payload: `' || touch /tmp/success`, but the payload `' || touch /tmp/success '` is needed. To handle all possible cases (e.g., taking into account sanitization) associated with synthesizing such a payload, we would need to employ a methodology similar to Enumerator's generation of completions for ACE flows.

Infrastructure and synthesis bugs. Finally, we encountered four cases where a exploit failed to be synthesized due to bugs in our infrastructure or synthesis implementation. 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 the results of synthesis leading to valid synthesized payloads being lost. In both cases, if the synthesized payload had been used, the flow would have been automatically confirmed.

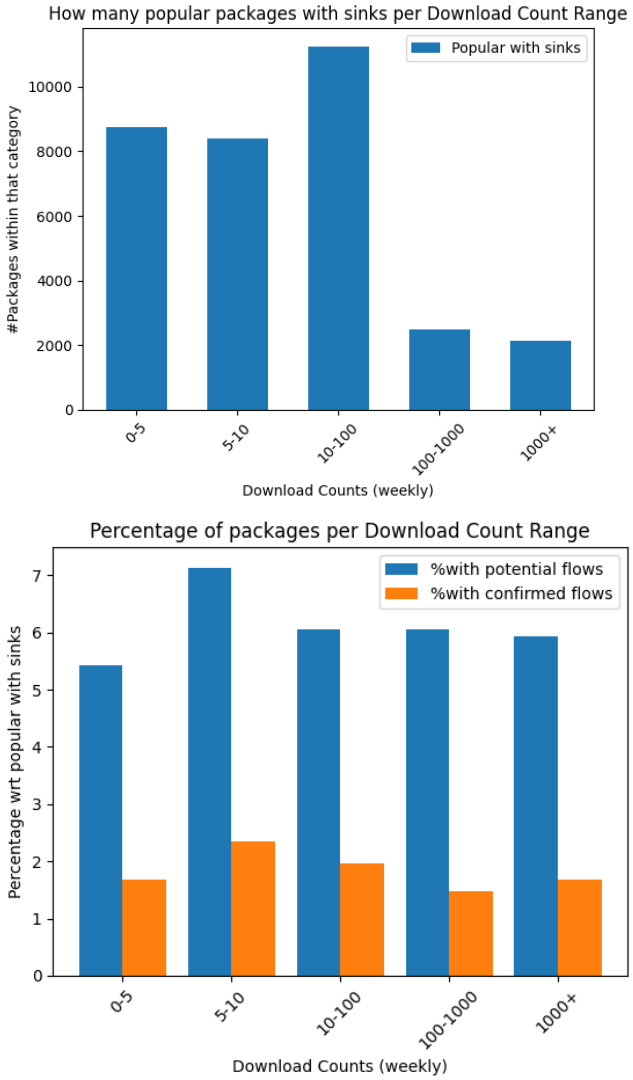

Figure 10: Frequency of packages within ranges of download counts, split into "with sinks", "with potential flows" and "with confirmed flows"

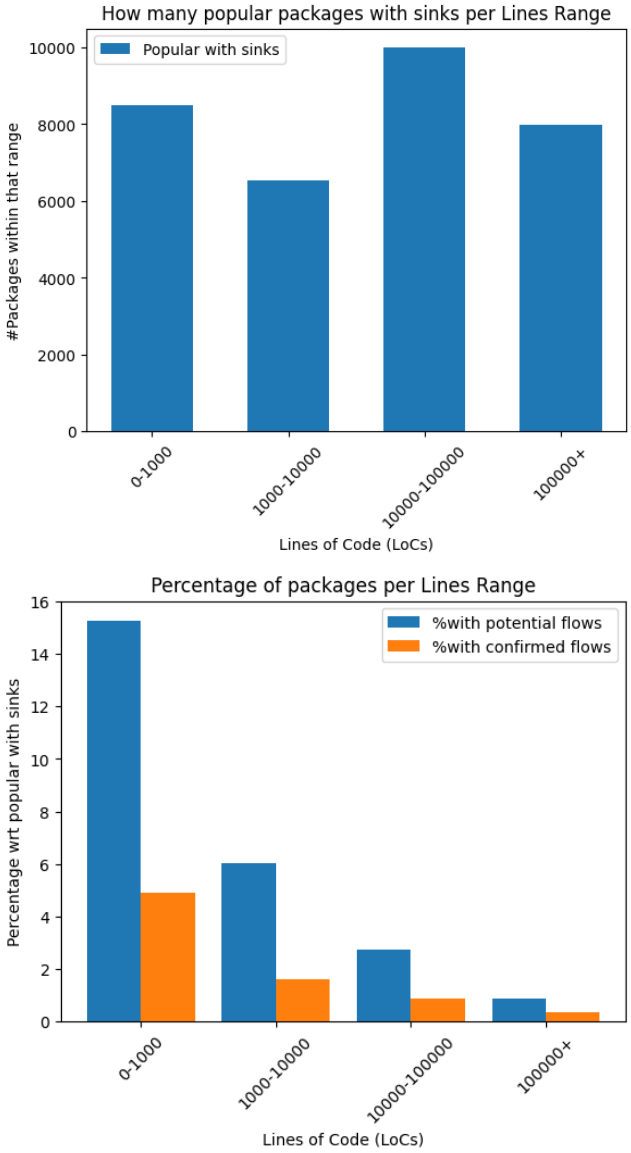

Figure 11: Frequency of packages within ranges of lines of code counts, split into "with sinks", "with potential flows" and with "confirmed flows"

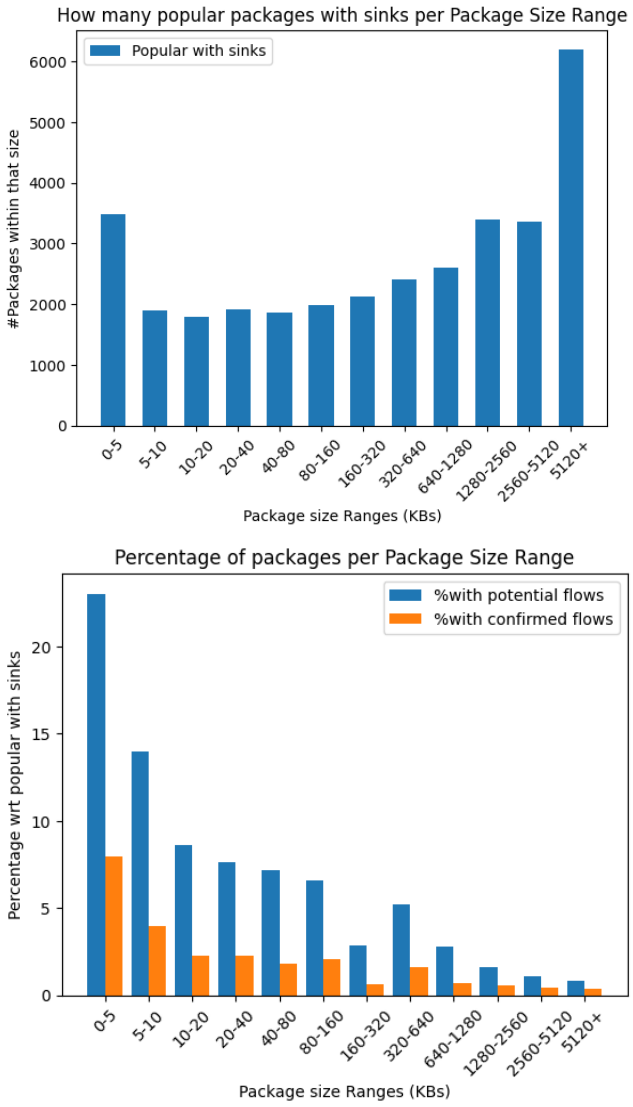

Figure 12: Frequency of packages within ranges of package size, split into "with sinks", "with potential flows" and with "confirmed flows"

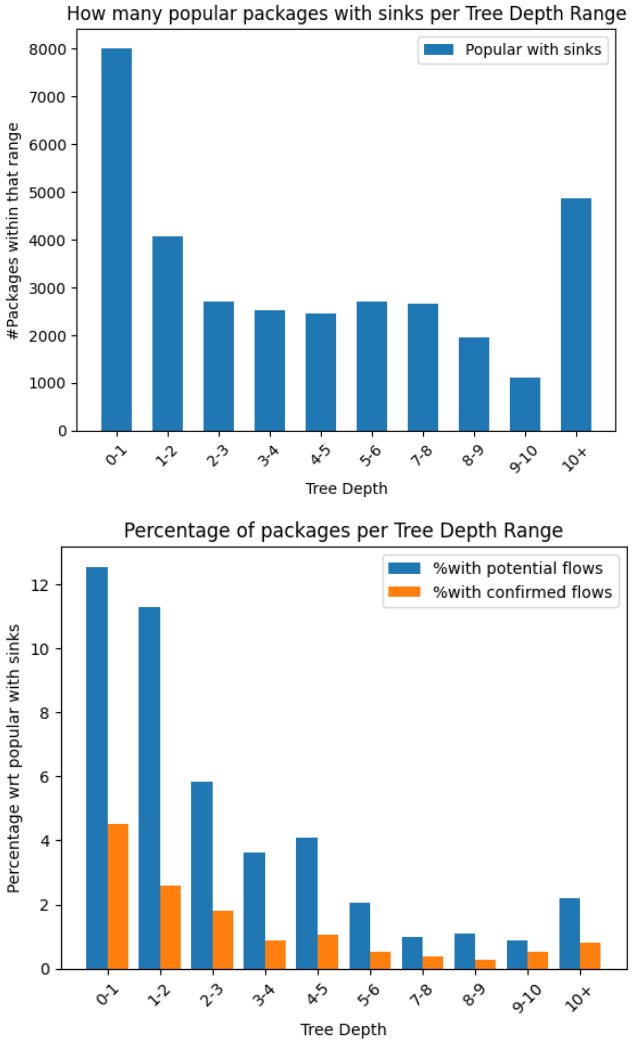

Figure 13: Frequency of packages within ranges of tree depth size, split into "with sinks", "with potential flows" and with "confirmed flows"

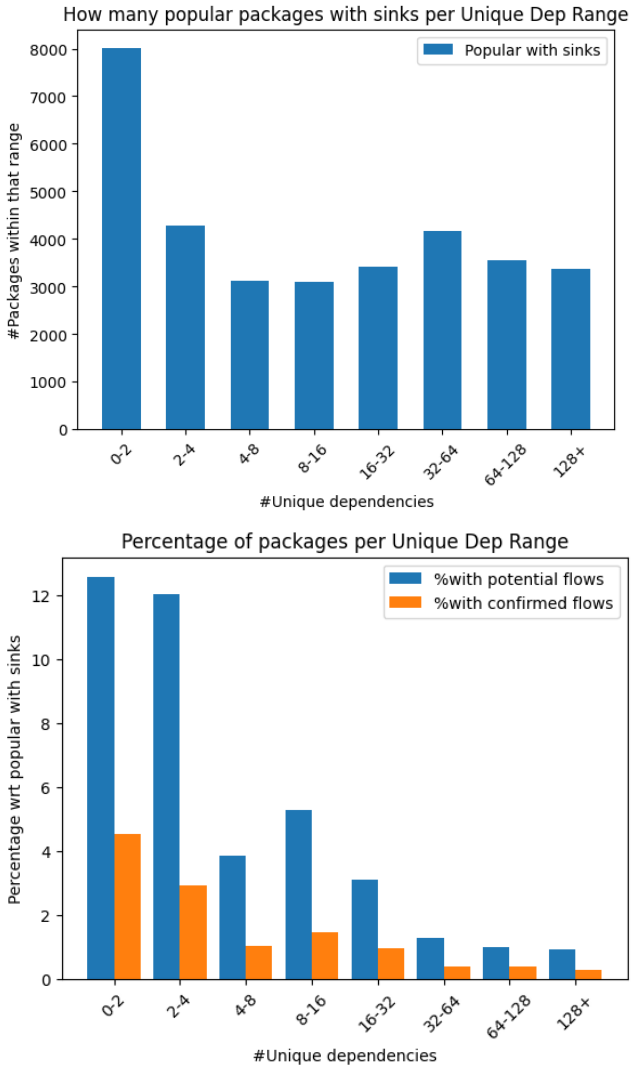

Figure 14: Frequency of packages within ranges of unique dependendy numbers, split into "with sinks", "with potential flows" and with "confirmed flows"

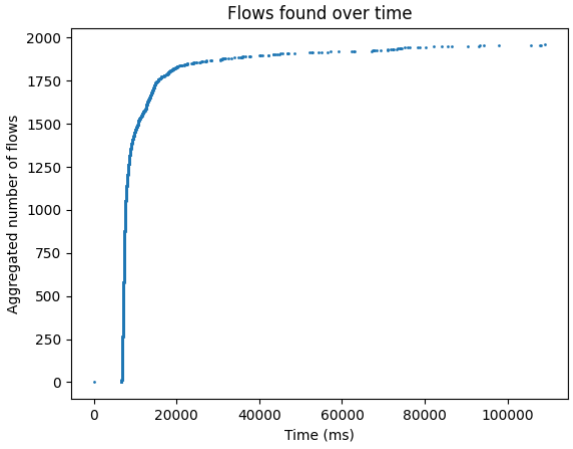

Figure 15: How many flows would be found (y-axis) if we set the fuzzing timeout to (x-axis in miliseconds).

```
for (const typeName in types) {
  const instance = types[typeName];
  // List of all supported fields
  let fields = Object.getOwnPropertyNames(instance);
  try {
    // Attempt to get the fields of the prototype
    let protoFields = Object.getOwnPropertyNames(
      instance.__proto__
    );
    fields.push(...protoFields);
  } catch (err) {}
  try {
    // Attempt to get the fields of the constructor
    let constrFields = Object.getOwnPropertyNames(
      instance.constructor
    );
    fields.push(...constrFields);
  } catch (err) {}}
```

Figure 16: Algorithm step 1: Deriving fields for each type.

```
for (const field of allFields) {
  for (const typeName in types) {
    const instance = types[typeName];
    // Try to see if we can access the field
    try {
      let result = instance[field];
      // If we can, add it to the fields of that type
      fieldSets.set(typeName,
        existingFields.add(field));
    } catch (err) {}}}
```

Figure 17: Algorithm step 2: Attempted access of fields.

```
for (const typeName in types) {
  const instance = types[typeName];
  try {
    // Check support for extension
    if (Object.isExtensible(instance)) {
      fieldSets.set(typeName,
        existingFields.add('Extension'));
    }
  } catch (err) {}}
```

Figure 18: Algorithm step 3: Checking extensibility.

```
for (const s1 of sortedValidSets) {
  for (const s2 of sortedValidSets) {
    // Case 1: Transition to a new type set
    if (subsetEq(s1, s2)) {
      lattice[s1][`Field<${s2}>`] = s2;
    // Case 2: Stay at the same type set
    } else if (intersect(s1, s2).size != 0) {
      lattice[s1][`Field<${s2}>`] = s1;
    }}}
```

Figure 19: Algorithm step for constructing the type lattice.

```
tl = TypeLattice()
leaf_types: LeafTypes = {}
for k, v in provenance_graph_paths_paths:
  inferred = "Bottom"
  for node in v[1:]:
    if isinstance(node.field, BuiltinOperationType):
      inferred = tl.transition(inferred, node.field)
    elif isinstance(node.field, OtherOperationType):
      inferred = tl.transition(inferred, "*")
    elif isinstance(node.field, SinkOperationType):
      inferred = tl.transition(inferred, "!")
    ...
    leaf_types[k].append(inferred)
```

Figure 20: Algorithm for inferring types along provenance graph paths.

```
def generate_operation_tree(tree: OperationTreeNode):
    return z3.Concat(*[
      _generate(child) for child in tree.children])
```

Figure 21: SMT model for string concatenation.

```
def model_js_string_slice(tree: OperationTreeNode):
  val = _generate(tree.children[0])
  length = _generate(tree.children[2])
  if int(str(length)) < 0:
    length = z3.Length(val) + length
  return z3.Extract(
    val, _generate(tree.children[1]), length)
```

Figure 22: SMT model for the `string.slice` operation.

```
def model_js_string_replace(tree: OperationTreeNode):
  _context.append(z3.Not(z3.Contains(
    _generate(tree.children[0]),
    _generate(tree.children[1]),
  )))
  return _generate(tree.children[0])
```

Figure 23: SMT model for the `string.replace` operation.

```
def _coerce(child: OperationTreeNode):
  # Ex: Function:*->String; to_type = "String"
  to_type = get_to_type(child.types)
  if to_type == "String":
    return _generate_coerce_string(child, child.types)
  ...
def _generate_coerce_string(
  child: OperationTreeNode, child_types: List[str]):
  if "Number" in child_types:
    return z3.IntToStr(self._generate(child))
  elif "Boolean" in child_types:
    val = self._generate(child)
    self._context.append(
      z3.Or(
        val == z3.StringVal("true"),
        val == z3.StringVal("false")))
    return val
  elif "Array" in child_types:
    return z3.Select(
      _generate(child),
      z3.IntVal(0))
  ...
```

Figure 24: Partial code for modeling JavaScript implicit coercion

```
def generate_operation_tree(...):
  if ptree.operation == "join":
    parents[0] = OperationTreeNode(
      "implicit", "coerce",
      [parents[0]],
      ["Function:*->String"])

def model_js_array_join(tree: OperationTreeNode):
    return z3.Concat(
        _generate(tree.children[0]),
        _generate(tree.children[1]))
```

Figure 25: SMT model for the `array.join` operation.

```
def construct_aci_payloads(self):
  prefix_list = [" ", "; ", "&& "]
  payload_list = ["touch success",
    "$(touch success)", "`touch success`"]
  suffix_list = [";#"]
  payloads: List[str] = []
  for prefix in prefix_list:
    for payload in payload_list:
      for suffix in suffix_list:
        payloads.append(f"{prefix}{payload}{suffix}")
  return payloads

def generate_aci_sink(self, tree: OperationTreeNode):
  sink_input = self._generate(tree.children[0])
  if self._context != []:
    return z3.Or(*[
        z3.Contains(sink_input, payload)
      for payload in self.construct_exec_payloads()])
```

Figure 26: ACI payload variation synthesis.

```
(declare-fun SymbolicField_b3e14a71 () String)
(assert (and (or (str.contains
  (str.++ "grep " SymbolicField_b3e14a71)
            " touch success;#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            " $(touch success);#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            " `touch success`;#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            "; touch success;#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            "; $(touch success);#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            "; `touch success`;#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            "&& touch success;#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            "&& $(touch success);#")
(str.contains (str.++ "grep " SymbolicField_b3e14a71)
            "&& `touch success`;#"))
(not (str.contains SymbolicField_b3e14a71 "$"))))
```

Figure 27: SMT formula with varied payloads.

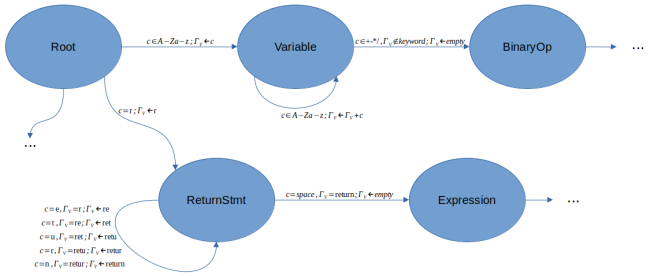

Figure 28: A section of the graph representation of JavaScript syntax used by the Enumerator. Edges have labels $C;U$ where C is a condition over the current character in the prefix c and the context Γ_C. U is a context update

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

Figure 29: 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);
  });
  ...
}
```

Figure 30: *b****@*0*** code vulnerable to ACI.

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

Figure 31: Abstract value inferred for *b****@0**.

```
(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)
```

Figure 32: SMT formula generated for *b****@0**.

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

Figure 33: Exploit driver for *b****@0**.

```
function doSpawn(method, command, args, options) {
  ...
  var cpPromise = new ChildProcessPromise();
  var reject = cpPromise._cpReject;
  var resolve = cpPromise._cpResolve;
  var successfulExitCodes = (options
    && options.successfulExitCodes) || [0];
  var cp = method(command, args, options);
  ...
```

Figure 34: Code snippet from *c****@2**.

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

Figure 35: Code snippet from *d****@1**.