Test Suites Guided Vulnerability Validation for Node.js Applications

Changhua Luo*
Wuhan University
Wuhan, China
The Chinese University of Hong Kong
Hong Kong SAR, China
chdeluo@gmail.com

Penghui Li†
Zhongguancun Laboratory
Beijing, China
lipenghui315@gmail.com

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Chao Zhang
Tsinghua University
Beijing, China
chaoz@tsinghua.edu.cn

ABSTRACT

Dynamic methods have shown great promise in validating vulnerabilities and generating Proof-of-Concept (PoC) exploits of Node.js applications. They typically rely on dictionaries or specifications to determine the values of request parameters and their relationships. However, they still struggle to generate complex inputs from the provided dictionaries or specifications.

This work introduces a novel approach that utilizes existing test suites to automatically generate end-to-end application inputs for vulnerability validation. Our key observation is that Node.js applications often provide comprehensive test suites—in our study, the unit testing code can cover an average of 85% of application code—which can hardly be achieved by existing dynamic methods. We thus design a new system, JSGo, that leverages test suites to construct end-to-end test inputs. Since test suites directly invoke application code instead of issuing requests from client-accessible entry points, we cannot simply transform test suites into application inputs. We instead propose a novel trace-guided mutation mechanism based on concolic execution.

Our evaluation demonstrates that JSGo could reproduce 20 out of 26 known vulnerabilities, which significantly outperformed the state-of-the-art methods Restler, Miner, Witcher, and Burp by 10, 12, 11, 10 more cases, respectively. We also applied JSGo to validate static analysis results in popular Node.js applications such as hexo. It successfully validated seven vulnerabilities, two of which have been patched because of our reports.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Vulnerability Validation; Test Suites; Node.js

ACM Reference Format:
Changhua Luo, Penghui Li, Wei Meng, and Chao Zhang. 2024. Test Suites Guided Vulnerability Validation for Node.js Applications. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690332

1 INTRODUCTION

Node.js applications have powered a large number of services on the web. A survey conducted by the Node.js Foundation in 2024 revealed that 36.42% of developers are using Node.js for their projects [66]. Renowned companies like Netflix, PayPal, and LinkedIn build their server-side applications using Node.js for its scalability and speed in handling concurrent connections [11]. However, alongside its popularity, Node.js applications suffer from severe security vulnerabilities such as prototype pollution and cross-site scripting. As an example, in the 2019 Equifax data breach, vulnerabilities in a Node.js application led to the exposure of personal information of approximately 147 million people [17].

Dynamic testing has been widely used to detect vulnerabilities in web applications, especially Node.js applications. A recent work Witcher [64] applies AFL to the Node.js applications to enable mutational grex-box fuzzing by instrumenting the runtime. Black-box web scanners such as Burp [49] and Black Widow [20] also have consistently revealed security issues in Node.js apps. In addition, REST API fuzzers [5, 19, 22, 40] generate diverse RESTful requests to test web applications that support REST APIs, including those built upon Node.js. Indeed, due to the efficacy and reliability, testing has been a standard practice for vulnerability detection in Node.js applications [20, 47, 50].

However, generating valid end-to-end application inputs (*i.e.*, HTTP requests) to trigger deep vulnerabilities in Node.js applications has been a challenge. Since Node.js applications take as input HTTP requests, simple random input mutation falls short of

*This work was done when the author was at the Chinese University of Hong Kong.
†Corresponding author.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690332

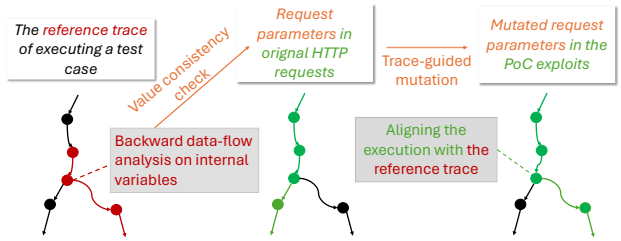

Figure 1: The overview of execution alignment. The red and green colors represent the execution traces of test suites and (mutated) HTTP requests, respectively.

generating high-quality HTTP request parameters that satisfy the required logic of the application. Therefore, most dynamic tools rely on external resources like fixed input dictionaries [24] or OpenAPI specification [62] that configure request parameters (which are key-value pairs) and relationships between application endpoints. However, the dictionaries or specifications can be unreliable and incomplete. Prior solutions have difficulty in producing valid inputs that diverge from the external resources. For example, Witcher designs an HTTP mutator to diversify inputs [64], but its seeds of request parameters are still derived from the dictionary, hence the diversity of mutated inputs remains limited.

This work aims to address the above-mentioned limitations and enhance vulnerability validation for Node.js applications. We made a key observation that Node.js applications usually (if not always) come with test suites that dynamically execute parts of application code. Test suites are sets of unit testing code specifically designed to verify that applications function as intended. Our studies in §2 demonstrate that 55 out of 58 real-world Node.js applications are packaged with test suites. Besides, the test suites are mostly comprehensive. Since test suites directly execute target code units rather than starting from the client-accessible entry points through issuing HTTP requests, they usually can achieve a high code coverage. We recorded the code coverage achieved by executing the bundled test suites and found they could cover an average of 85% of code across the applications.

Since test suites could already achieve a high coverage, this work proposes to leverage test suites to construct end-to-end inputs for Node.js applications. The rationale is that test suites can execute the code that web fuzzers usually struggle to reach. However, a test suite cannot directly be used as an end-to-end input for a Node.js application vulnerability. Security vulnerabilities triggered by a (mutated) test suite can be false positives because the test suite does not execute from the application's entry point. Therefore, we propose to extend the test suites into normal client-initiated end-to-end application inputs (*i.e.*, HTTP requests).

To fill the gap, we design *trace-guided mutation*, a novel approach that constructs and mutates HTTP requests based on the execution information of a test suite. We mutate the requests so that the application takes execution paths that align (connect) with the ones of the test suites, thus reaching deep vulnerable code and then possibly triggering the vulnerabilities in the code.

Constructing and mutating HTTP requests based on test suites is challenging. An intuitive approach is to utilize the runtime information obtained from test suite execution as guidance for input mutation. However, it is unclear how such guidance can overcome the disparity incurred by the differences between test suites (JavaScript code invocations) and HTTP requests. A test suite directly assigns values to internal variables within the tested code. To trigger the same (vulnerable) code from application entry points, specific HTTP requests need to be constructed to satisfy the required internal variable values along the possible paths. To align these internal variable values and thus executions, one might utilize symbolic execution [38] to compute symbolic expressions for the internal variables. However, according to our experiments, Node.js applications contain intricate code and a sophisticated event-driven mechanism, often rendering full concolic execution infeasible.

We design and implement JSGo to generate end-to-end inputs for vulnerability validation of Node.js applications. JSGo can be used to reproduce known vulnerabilities and validate static analysis results. Figure 1 illustrates the high-level idea. JSGo takes a code location as the target and generates end-to-end inputs to trigger the vulnerability in the target code. The target can be a vulnerable code location that dynamic tools fail to reach because of its deep location, or they can reach it but cannot provide the specific execution context required to trigger the vulnerability. JSGo generates inputs by performing *trace-guided mutation* on HTTP requests produced by existing fuzzers. Specifically, JSGo takes the execution trace of a test case reaching the target code as the *reference trace*. Starting from an application entry point, it checks the HTTP requests generated by a web fuzzer and selects one if its trace intersects with the reference trace.[1] It then mutates the HTTP request to align its trace with the reference trace.

To achieve trace alignment and trigger the security vulnerability, JSGo mutates the HTTP requests to manipulate the values of two types of internal variables. The first type is the one used in the conditional statement where the execution trace of an HTTP request diverges from the reference trace. JSGo manipulates their values to match those used in the reference trace. This ensures that the executions of HTTP requests take the branch in reference trace and progressively approach the target code. The second type is the variables used in the target code. JSGo manipulates their value to be the prepared attack payloads (*e.g.*, `'__proto__'` of object index variables for prototype pollution vulnerabilities). This ensures that the executions of HTTP requests meet the data-flow conditions for triggering injection vulnerabilities.

We propose an under-constrained approach to manipulate the values of internal variables. Specifically, JSGo conducts backward data-flow analysis from an internal variable until identifying variables (referred to as the *source variables*) that constantly have a runtime value identical to request parameters (such as cookies, query parameters, *etc.*). We consider the source variables to be user-supplied inputs because of their value consistency and symbolize them during symbolic execution. This allows the symbolic execution engine ExpoSE [38] to skip much early code, such as the code parsing structured HTTP requests into program variables. We then introduce an artificial constraint $Equals(internal\_variable, value)$ into the program, where the first operand represents a symbolic expression of an internal variable based on source variables, and the

[1] In a few cases, no execution of HTTP requests intersects with any reference trace. We discuss this situation in §7.

second operand is the desired value of the internal variable. By solving this expression, we determine the values of source variables and thus the request parameters used in the HTTP requests. Note that our approach can handle complex conditional statements within the application, as we do not solve any program path constraints but focus on the artificial constraint to align execution traces.

Our current implementation of JSGo can generate validation payloads for XSS, SQLi, and prototype pollution vulnerabilities. It automatically validates prototype pollution by checking if the inserted object property exists. For XSS and SQLi vulnerabilities, we validate them following common practices [19, 23].

We evaluated JSGo from two perspectives—reproducing known vulnerabilities and validating static detection results. In total, we collected 26 known vulnerabilities in 15 Node.js web applications. We selected the applications 1) evaluated in prior works [56, 72] and 2) intentionally designed to contain vulnerabilities. JSGo could generate end-to-end inputs for reproducing 20 out of 26 vulnerabilities, achieving 10 more than the second-best dynamic tool in our evaluation. To evaluate JSGo's capabilities in validating suspicious vulnerabilities, we set vulnerable locations provided by the static detection tools [29, 34, 56] as targets. Static detection tools usually report vulnerable functions or paths, which are often insufficient as evidence of valid vulnerabilities. JSGo validated 7 vulnerabilities in the latest version of Node.js applications with several thousand stars on Github, including derby and hexo. We have reported our findings to the developers and 2 vulnerabilities have been patched at the time of writing.

In summary, this paper makes the following contributions.

- We presented an in-depth study to demonstrate the value of test suites for reaching deep code.
- We developed JSGo, a new tool to generate end-to-end inputs for validating vulnerabilities in Node.js applications, leveraging our novel trace-guided input mutation technique.
- Our evaluation demonstrated that JSGo can generate complex inputs to test deep Node.js application code, which is difficult for previous works.
- We open-source JSGo and the associated artifacts publicly at https://github.com/WHU-seclab/JSGo.

2 A SURVEY ON TEST-SUITES

Test suites are a collection of test cases that verify the behavior of specific application functionalities. In Node.js, test suites are typically written in JavaScript or TypeScript, utilizing testing frameworks such as Mocha [41], Jasmine [26], and Jest [28]. A test suite usually comprises two main components: 1) a `'Describe'` section that outlines the functionality being tested, and 2) `'it'` blocks, each containing an individual test case. Within an `'it'` block, the test code is executed to perform a particular action, and an associated `assertion` (*e.g.*, `expect` statement) verifies whether the observed outcome aligns with the expected behavior. Listing 1 shows a test suite in parse-server [45] that validates `'Parse.Push'` functionality.

Test suites are usually included in the same Node.js application code repositories. We conducted a preliminary survey of Node.js web applications by searching keywords like 'nodejs' and 'framework' on GitHub. We finally collected 58 Node.js web applications.

```
describe('Parse.Push', () => {
  // a test case whose execution can reach the sink operation of a prototype vulnerability.
  it('should properly send push', async () => {
    const { sendToInstallationSpy } = await setup();
    const pushStatusId = await Parse.Push.send({
      where: {
        deviceType: 'ios',
      },
      data: {
        badge: 'Increment',
        alert: 'Hello world!',
      },
    });
    await pushCompleted(pushStatusId);
    // an assertion
    expect(sendToInstallationSpy.calls.count()).toEqual(10);
  });
  // other test cases related to the 'Parse.Push' functionality
  ...
});
```

Listing 1: A test suite verifying the "Parse .Push" functionality in parse-server 5.0.0-alpha.13.

Table 1: Statistics on the test suite availability in Node.js applications.

	Apps w/o Test-Suites	Apps with Test-Suites			
		Jest	Mocha	Jasmine	Others
Num.	3	15	14	6	20
Proportion	5.17%	25.86%	24.14%	10.34%	34.48%
Example	chat.io	Next.js	Nest.js	parse-server	vitest

The dataset included 8 popular applications based on usage statistics in 2023 [61]. The number of web applications we studied is substantial compared to those evaluated in prior works [56, 64]. A Node.js application installs test frameworks like `Mocha` for testing through `package.json` file. Therefore, we check the `package.json` file and find if the general testing frameworks are included and listed there under the 'devDependencies' key and the 'scripts' key. Table 1 shows the proportion of Node.js applications providing test suites. Out of 58 Node.js applications surveyed, the majority (55) provide test suites.

We further investigate the comprehensiveness of test suites. We randomly sampled 14 applications (including 4 popular ones according to [61]) that provide test suites. We utilized code coverage tools nyc [25] and c8 [8] to record the code coverage while running the test suites. The average code coverage of test suites in 14 applications ranges from 55% (Rocket.Chat) to 100% (Ant Design, *etc.*), with an overall average of 85.0%. To compare the coverage of test suites and web fuzzers, we utilized Restler [5] to test the 14 applications within a 24-hour timeframe. We generated OpenAPI/Swagger specifications based on the application usage tutorials. We do not run other tools like Witcher [64] or Black Widow[20] because they are unable to test certain Node.js applications (*e.g.*, parse-server [45]) that interact programmatically with client-side APIs. The results demonstrated that Restler often achieved lower coverage compared to the unit testing code. *In conclusion, test suites have the potential to complement web fuzzers.*

```
1  function sanitizeDatabaseResult(originalObject, result
       ) {
2      const response = {};
3      if (!result) {
4          return Promise.resolve(response);
5      }
6      Object.keys(originalObject).forEach(key => {
7          const keyUpdate = originalObject[key]; //
       determine if that was an op
8          if (keyUpdate && typeof keyUpdate === 'object'
        && keyUpdate.__op && ['Add', 'AddUnique', '
       Remove', 'Increment'].indexOf(keyUpdate.__op) >
       -1) {
9              expandResultOnKeyPath(response, key,
       result);
10         }
11     });
12     ...
13 }
14 function expandResultOnKeyPath(object, key, value) {
15     if (key.indexOf('.') < 0) {
16         object[key] = value[key];
17         return object;
18     }
19     const path = key.split('.');
20     const firstKey = path[0];
21     const nextPath = path.slice(1).join('.');
22     object[firstKey] = expandResultOnKeyPath(object[
       firstKey] || {}, nextPath, value[firstKey]);
23     delete object[key];
24     return object;
25 }
```

Listing 2: A prototype pollution in parse-server 5.0.0-alpha.13.

3 MOTIVATION AND CHALLENGES

In this section, we motivate our research using an example. We further illustrate the challenges of using test suites to guide input generation with this example.

Listing 2 shows a code snippet of parse-server. The function `sanitizeDatabaseResult()` calls `expandResultOnKeyPath()` in line 9. Before that, there are two conditions in line 3 and line 8. The function `expandResultOnKeyPath()` traverses nested objects represented by the parameter `object` along the path specified by the parameter `key`, updating the `object` with the value provided in the parameter `value` at the specified path.

Prototype pollution occurs if the parameter `key` contains properties that modify the prototype of the `object`. Specifically, if the `key` includes properties such as `'constructor.prototype'`, the function might inadvertently alter the prototype of objects, leading to unexpected and potentially malicious behavior. To trigger the vulnerability, the function `expandResultOnKeyPath()` should be executed multiple times, *i.e.*, the execution should reach line 22 (which is one of the target code locations to trigger this vulnerability). The other target location is line 16, in which the prototype pollution occurs when the value of variable `'object'` is the prototype object.

3.1 Limitation of Existing Works

Existing dynamic tools cannot trigger the above vulnerability. Listing 3 illustrates the HTTP requests (lines 1-8) generated by a fuzzer called Restler [5], as well as the end-to-end input in lines 10-13

```
1  // HTTP requests produced by Restler
2  curl -X POST http://localhost:1337/parse/classes/
       GameScore \
3  -H ... \
4  -d '{"score":1.23, "playerName":"fuzzstring","
       cheatMode":true}'
5
6  curl -X PUT http://localhost:1337/parse/classes/
       GameScore/dt5eSOZvj9 \
7  -H ... \
8  -d '{ "score":1.23, "playerName":"fuzzstring", "
       cheatMode":true}'
9
10 // An HTTP request produced by JSGo
11 curl -X PUT http://localhost:1337/parse/classes/
       GameScore/dt5eSOZvj9 \
12 -H ... \
13 -d '{"score": 2.23, "playerName": "fuzzstrini", "
       cheatMode": false, "KybeTDkRgO.constructor.
       prototype.dummy": {"__op": "Increment", "amount":
        -1}}'
```

Listing 3: An end-to-end input produced by JSGo.

crafted by JSGo. We list two key limitations of previous techniques that hinder their capabilities. The two limitations also apply to other dynamic tools.

Composite Type Data. While existing dynamic tools can generate request parameter values in basic types (line 8 in Listing 3), they cannot generate request parameter values in composite types. For instance, the HTTP request in line 13 demonstrates a query parameter, which is an *object* with two properties `'__op'` and `'amount'`.

The inability of existing techniques to generate data in composite types impedes their capacity to produce inputs required for reaching deep code. According to our evaluation, these tools cannot trigger vulnerabilities in applications like Ghost and juice-shop because of this reason. In this example, the condition statement in line 8 of Listing 2 checks if the variable `keyUpdate` (a program variable corresponding to a query parameter) is of type *object*. While most tools can generate inputs leading to executions taking the false branch, they cannot execute the code in the true branch.

Complex Request Parameter Values. Existing tools depend on the user-supplied dictionary or specification to generate request parameter values. However, the dictionary or specification might not be complete. For example, we collected the sample usages[2] and the official document[3] as the specification of Restler for testing parse-server. Despite this, none of the produced HTTP requests could execute code in line 22 of Listing 2. In fact, to reach line 22, the expected key of a query parameter in the HTTP request should be a string 'sentPerType.ios' or 'sentPerType.android'. This is a value automatically generated by particular types of devices making the HTTP request rather than directly supplied by users.

3.2 Research Goals and Challenges

In this section, we discuss our research goals and challenges.

3.2.1 Research Goals and Scope. Our research goal is to overcome the above-mentioned limitations and to better generate end-to-end

[2]https://github.com/bhtz/parse-server-swagger
[3]https://docs.parseplatform.org/parse-server/guide/

inputs for Node.js applications. The key idea is to mutate HTTP requests based on the application's test suites. For example, parse-server includes 19 test suites that can directly execute code in line 22. Listing 1 is one among them. By mutating an HTTP request to align its execution with that of this test suite, we can generate an HTTP request reaching the deep code in line 22 and further test it.

We generate end-to-end inputs to trigger vulnerabilities in the given target code, which can be either sink operations of a known vulnerability or vulnerable code identified by static analysis tools [34, 56]. We believe that it is reasonable to assume knowledge of the vulnerable code when generating test inputs. For example, we can obtain the target location from the CVE database if we aim to reproduce known vulnerabilities or from static detection tools if we validate statically detected vulnerabilities. Generating end-to-end inputs to trigger vulnerabilities is also important in other domains such as C/C++ applications [10, 39] and OS kernels [63].

In this work, we generate end-to-end test inputs for validating three types of vulnerabilities—cross-site scripting (XSS), SQL injection and prototype pollution—in Node.js applications. Note that our objective does not involve automated exploitation through methods such as generating specific values for the polluted properties [37, 56] or ensuring the dynamic execution of the XSS payloads [20]. Instead, we aim to automatically generate HTTP requests capable of reaching sinks and manipulating variables within the sink operations to serve as the prepared attack payloads (*e.g.*, `'__proto__'` in prototype pollution).

3.2.2 Challenges. It is not trivial to generate an end-to-end HTTP request (*e.g.*, lines 11-13 in Listing 3), even using a test suite. We summarize two challenges below.

Different Execution Entry Points. We aim to generate HTTP requests such that their execution reaches some deep code. To this end, we mutate HTTP requests to align their executions with those of test cases. To meet the path constraints, certain variable values need to be satisfied while serving an HTTP request. For example, to align the execution of an HTTP request with a test case execution reaching line 22 in Listing 2, the variable `keyUpdate` in line 8 in the execution of HTTP requests needs to take the same value as that in test suite execution. However, we cannot directly assign particular values to internal variables as the applications start from entry points when processing HTTP requests. One common approach to achieving this is by symbolizing the external inputs (*i.e.*, HTTP requests) and representing internal variables using symbolic expressions. This approach is infeasible due to the complex language features and application semantics that the current concolic execution engine [38] has limitations in handling.

Data-flow Conditions. The second challenge is triggering the vulnerabilities after the executions of HTTP requests reach the target location. Note that purely relying on information from test suites is not sufficient because test suites are designed for verifying normal functionalities. In our research, our goal is to inject attack payloads into variables within sink operations even if these variables are the results of user inputs and data operations. This task presents challenges because of the data flow from the user-supplied input data to program variables.

4 JSGO

We propose JSGo to address the above-mentioned challenges. JSGo leverages test suites to mutate HTTP requests for validating or reproducing security vulnerabilities in Node.js applications. In this section, we discuss the components of JSGo.

4.1 Workflow

Since test suites can achieve high coverage, JSGo mutates HTTP requests generated by web fuzzers, facilitating their penetration into the deeper code covered by test suites. JSGo achieves this by aligning HTTP requests' execution traces with those of the selected test cases. In particular, given the target code (*e.g.*, the code containing a suspicious injection vulnerability), JSGo first selects base test cases and sequences of HTTP requests for alignment. This selection is based on the trace intersection of test cases and HTTP request sequences (§4.2). We design strategies to mutate specific HTTP requests without disrupting the data dependencies within sequences. Later, JSGo employs trace-guided mutation to mutate an HTTP request (§4.3). In this step, it identifies and manipulates request parameters that correspond to two types of internal variables in the application code. This enables JSGo to generate end-to-end HTTP requests that reach and test the target code. We finally employ a vulnerability validation component (§4.4) to ensure that the mutated HTTP requests trigger the vulnerabilities.

4.2 Selecting Base Test Suites and HTTP Requests

We aim to generate end-to-end HTTP requests for triggering some (deep) vulnerabilities by aligning execution traces with those of known valid test suites. In this step, we execute the existing test suites and monitor their execution traces, aiming to find *reference traces* that can already reach the target locations. After that, we generate base HTTP requests using an existing web fuzzer called Restler [5] and select the ones whose execution traces intersect with the reference traces.

4.2.1 Identifying Reachable Test Cases. We select test cases based on whether their executions reach the target code. Since most Node.js applications offer test suites, we execute their testing code and analyze the corresponding code coverage. Our analysis is conducted at the granularity of test cases (*e.g.*, `'it'` blocks), as we have observed that test cases are typically *independent* units. We consider the execution trace (*i.e.*, a sequence of executed statements) of reachable test cases as *reference trace*.

4.2.2 Generating and Selecting HTTP Request Sequences. We discuss how we select the basis HTTP requests for mutation.

Generating HTTP Request Sequences. JSGo leverages an existing web fuzzer, Restler, to produce HTTP requests. We use Restler for its latest implementation of state-of-the-art testing techniques [19, 40]. For instance, Restler can reuse prior response contents to extend request sequences (*e.g.*, the `PUST` request reuses `'dt5eSOZvj9'` in Listing 3). Restler can also test various types of Node.js applications as it generates HTTP request sequences that are acceptable to most web applications. Specifically, it generates each new sequence by progressively adding a new request to the end of an existing sequence. Restler has a garbage collection mechanism

that is responsible for recycling allocated resources post-fuzzing. We disable it since we need to operate on them later.

Selecting HTTP Requests to Mutate. In each sequence, we select only the last request whose execution trace, upon replaying, intersects with the reference trace. We focus on the last intersecting request because the prior intersecting requests in this sequence have been handled when we analyze shorter sequences. We will mutate the last intersecting request in §4.3.

An HTTP request can intersect with multiple reference test cases. We prioritize in the test case that has *not* been used by other HTTP requests as the reference test case of that HTTP request. This enables us to obtain diversified reference traces and generate HTTP requests exercising the target code from different paths. Not all paths leading to sink operations may implement the appropriate sanitization. By testing diverse HTTP requests reaching the sinks, we have more opportunities to trigger the target vulnerabilities.

Before mutating the selected HTTP requests for the application to serve, we need to address a challenge. The mutated requests might introduce potential data dependency issues when processed by the application. The issues arise because the dependencies encoded in the mutated requests might become outdated within the contexts of the sequences where they are located. For instance, if we mutate any DELETE operation without updating the object IDs it deletes, errors will occur when serving this request as the same objects will be attempted to be deleted twice. Below, we introduce how we overcome the data dependency issues based on the different types of the selected HTTP requests.

① **PUT, PATCH, POST, and GET Methods.** For the request that uses the `PUT` or `PATCH` method, we mutate the request parameters of it. While the `PUT` or `PATCH` method can update an existing resource, the way it updates will not influence future update operations. For example, in Listing 3, we can mutate the request parameter of `PUT` request (lines 6-8) only. The mutated `PUT` requests (lines 11-13) can update the same object repeatedly.

We also mutate the parameters of the HTTP request that uses the `POST` or `GET` method, Mostly, `GET` requests are used for retrieving data and POST requests are for submitting data to be processed. Mutating the request parameters usually does not lead to data dependency issues. However, it is possible to use such requests to alter existing data, as shown in [12]. For example, developers can delete a resource by sending one such request to an endpoint designated for deletion. We design the strategy below to mitigate such issues.

② **DELETE Method.** The most complex situation occurs when the last request of a sequence is a DELETE one. A `DELETE` request generally operates on resources created by the prior requests within the sequence. However, the original sequence execution has already deleted these resources. The resources cannot be deleted again using the unmodified resource identifiers.

Our current approach to addressing this problem involves manual efforts. To process the mutated request with the DELETE method, we recover and replay the entire sequence in which the `DELETE` request is located. We then manually modify critical parameter fields related to resources such as object IDs. We envision an automated solution by leveraging Restler's dependency analysis to update the resource identifiers accordingly during the recovery process. Given that manual intervention only involves copy-paste operations, we consider such an adjustment acceptable.

4.3 Trace-Guided Mutation

In this section, we discuss how we mutate the last intersecting request of the HTTP request to trigger the target vulnerabilities. We propose a novel technique called *trace-guided* mutation for mutating HTTP requests. In the following sections, we will first describe our high-level idea of trace-guided mutation, then the two types of variables we manipulate, and finally explain how we manipulate variables using concolic execution.

4.3.1 Overview. The core technique of trace-guided mutation involves mutating user-supplied inputs based on the reference values. These reference values are either runtime values of certain variables during the execution of the reference test case or our prepared attack payloads used to trigger security vulnerabilities. The difference between trace-guided mutation and other mutation techniques is that we have obtained the reference value to mutate towards. Specifically, we mutate HTTP requests to manipulate certain types of program variables (detailed in §4.3.2), enabling their execution values to match the reference values. By doing this, we can control the program behaviors and generate HTTP requests to reach and test the target code.

4.3.2 Identifying Variables for Manipulation. We manipulate two types of variables to align executions and trigger vulnerabilities. In particular, we consider the reaching conditions and data-flow conditions during variable manipulation.

Reaching Conditions. The HTTP requests must execute the target code before triggering the vulnerability there. The reaching conditions are often determined by the variables in conditional statements on the path. However, not all condition variables in reference traces are important. We identify and manipulate the variables that cause misalignment between HTTP requests and reference test cases.

Specifically, JSGo conducts a control-dependency program slicing from the intersecting code to the target code. This slicing extracts the conditions on which the target code is control-dependent. JSGo then identifies these reaching conditions for further manipulation, focusing on their comparison expressions. For instance, consider the execution of the test case depicted in Listing 1, where divergence occurs from the target code (specifically, at line 22 in Listing 2) due to the condition at line 8 in Listing 2. In this case, JSGo identifies the variable `keyUpdate` for manipulation.

Data-flow Conditions. Upon reaching the target code, the next step is to trigger potential vulnerabilities there. Specifically, we focus on manipulating critical parameters used in sink operations. In the case of prototype pollution, there might be two sink operations. We consider both sink operations as the targets to trigger that vulnerability. In Listing 2, for example, lines 16 and 22 are considered as the targets to trigger a prototype pollution vulnerability. To identify data-flow conditions, we analyze the vulnerability-specific sinks (targets) and extract the related variables.

4.3.3 Manipulating Variables by Mutating Request Parameters. After identifying the two types of critical variables, we propose to manipulate them and ensure they take desired runtime values. Specifically, JSGo monitors the runtime values of reaching conditions. If discrepancies exist between serving the HTTP request and executing its reference test case, JSGo intervenes by manipulating the variables involved in these comparison expressions. For example, JSGo first records the value of `keyUpdate` during the execution of the reference test case as the reference value in Listing 2. The value of `keyUpdate` during the execution of HTTP requests, if found equivalent to the reference value, can result in alignment at line 8 with the reference trace. Similarly, for data-flow conditions, JSGo monitors their runtime values and aligns them with prepared attack payloads per vulnerability type.

We propose a novel method to enhance the existing concolic execution tool ExpoSE [38] so that it can generate inputs needed to satisfy both reaching and data-flow conditions. With the help of the concrete values (the original HTTP requests), concolic execution is able to focus on related program paths and states [21, 54] instead of blindly exploring all. When conducting concolic execution, an intuitive approach to controlling the runtime value of a program variable VAR is to symbolize user inputs, derive symbolic expressions of VAR, and then solve the expressions to obtain the desired user input values. However, it is non-trivial to apply concolic execution for Node.js applications due to the JavaScript language features and the complexities of web applications. First, JavaScript supports compound types like objects, making it difficult for existing concolic execution tools to generate runtime variable values in the correct types. Second, web applications are complex and have much code, which cannot be handled well by existing concolic execution tools.

To address the first challenge, we do not directly solve symbolic expressions involving JavaScript objects. Instead, if an object variable V_o is one of the two types of internal variables in §4.3.2, we identify the request parameter of that object variable V_o by comparing the request parameter value with the V_o's runtime values. We then calculate the required request parameter values that would lead to the alignment of object values. To mitigate the second challenge, we simplify the concolic execution analysis to handle only operations changing the variable values. As a result, JSGo only needs to perform analysis on reduced application code. In the following, we describe the details, *i.e.*, how we generate end-to-end HTTP requests to manipulate a given variable, VAR.

Symbolizing Internal Variables. We first determine the variables that need to be symbolized. Directly symbolizing user inputs involves high complexity and is unaffordable. An internal variable VAR may be data dependent on some variables, which directly take values from user inputs and we denote them as *source variables* of VAR. The source variables have shorter data flow paths to the target internal variables, and lower complexity for the concolic execution engine. We thus propose to symbolize and mutate the source variables for helping manipulate the internal variable VAR. We specify two general conditions for source variables. First, they must have data dependencies with VAR. Second, they must *directly* take values from the user-supplied inputs. Our key observation is that while there is much code handling the user-supplied HTTP request, the variables storing request parameter values might not be updated in an early stage and their values are mostly directly reused in the program.

However, it is difficult to identify such variables using static analysis as the applications can use diverse and complex custom input processing logic. We observe some existing works use parameter keywords [13, 19] to infer internal variables directly controlled by user requests. We find the heuristics to be unreliable in the context of Node.js applications. Instead, we set different special request parameter values and monitor such values during the application execution. If some internal variables always take the exact special parameter values, they are highly likely to directly take values from the user-set request parameter values.

To identify source variables of a variable VAR, we conduct backward static data-flow analysis starting from VAR. We stop backward data-flow analysis once we identify the first source variables of VAR because we prefer variables that are closer to VAR for simplicity. We discuss the details of static analysis in §5.1.

Concolic Execution. We employ concolic execution on the applications using the selected HTTP request in §4.2.2 as input. Specifically, we introduce an artificial constraint $Equals(VAR, VAL)$, where $Equals()$ compares the equivalence of two expressions, and VAL denotes the corresponding reference value of VAR (see §4.3.2). The concolic execution engine can produce a symbolic representation of the target variable VAR, and solve the artificial constraint to assign the desired value VAL to VAR. Such an artificial constraint is generated and solved just before 1) the point of divergence between the HTTP request and its corresponding reference test case or 2) the target code if the execution reaches that stage.

We mitigate a challenge in concolic execution related to the numerous program paths that emerge during this phase. Since our main objective is to solve the artificial constraint on a single path, many of the program paths become unnecessary. Consequently, we modify ExpoSE [38] to execute only the program path towards the target code location. Specifically, we halt the concolic execution analysis once ExpoSE reaches the next program location for solving the artificial constraint.

We use the example in Listing 2 to illustrate the idea. To reach the target code, the reference value of variable `keyUpdate` should be an object. Through the value consistency check, JSGo discovered that `keyUpdate` is from a user-supplied request parameter value. Therefore, `keyUpdate` is not only the variable VAR we want to manipulate because it is used in the condition, but also the source variable. JSGo assigns the reference value VAL (a JavaScript object) of `keyUpdate` to a request parameter value after solving the artificial constraint $Equals(VAR, VAL)$. We discuss how we assign object values to request parameters in §5.2.

Mutating HTTP requests. Finally, we utilize the value V_{new} generated by ExpoSE to modify the HTTP requests. V_{new} represents the value of a source variable that results in the reference value in VAR. This value should be assigned to the corresponding user-supplied request parameter. We identify the request parameter linked to each source variable based on the consistency between the source variable's runtime value and the request parameter value. We then set the request parameter value to take V_{new}. However, it is not straightforward if V_{new} has a different type from those of

existing parameter values. This discrepancy can arise when V_{new} corresponds to a new request parameter shown in the reference test case but absent in the HTTP request. In such cases, simply mutating existing parameters may not suffice.

To solve this problem, our mutation approach introduces a new parameter to the HTTP request. This parameter value is V_{new}, which can be in a simple type like an integer or a compound type such as an object. This new parameter's key is a randomized string. We select strings as the key because HTTP request parameter keys must adhere to this data type. Subsequently, we dispatch the mutated HTTP requests to the application and scrutinize the execution trace. If the new execution path still deviates before the condition as in the prior execution, it could be attributed to the randomized parameter key. In such cases, the reference value of that parameter key must also be a string and can be similarly obtained from the reference trace. Hence, we further mutate the parameter key to align it with the reference value.

In Listing 2, since there are no request parameter values in the object type (lines 6-8 in Listing 3), JSGo generates a new query parameter (the last parameter of line 13 in Listing 3) and uses a random parameter key. It issues an HTTP request and further mutates its parameter key based on the reference test case execution. This results in an HTTP request *directly* reaching the target code. Subsequently, JSGo identifies the input fields (the last parameter key of line 13 in Listing 3) that correspond to the source variables affecting the critical variables used in the target code. Since parts of the parameter key are used as the object variable in the target code, JSGo assigns the prepared payloads to the corresponding fields in the parameter key.

4.4 Vulnerability Validation

While the aforementioned techniques enable us to generate HTTP requests that reach the target code and set prepared attack payloads to the variables in the target code, there remains a gap in triggering vulnerabilities. Even if we manage to assign XSS payloads to critical variables in an XSS sink location, we may still be unable to trigger the XSS vulnerability if the payload is not executed.

Therefore, we conduct further validation of vulnerabilities after crafting the mutated HTTP requests. For XSS and SQL injection vulnerabilities, we send the HTTP requests and observe if the attack payloads are executed successfully. Specifically, we monitor the runtime values of variables used in XSS or SQLi sink locations and check if they contain special attack payloads.

Regarding prototype pollution, we implement automatic validation by verifying if the inserted property (*e.g.*, 'dummy') exists in the JavaScript `'Object'`. Additionally, according to some CVEs [14, 15, 58], developers consider it dangerous if one object index variable is set to `'__proto__'` or if two-dimensional object index variables are set to `'constructor'` and `'prototype'`. Therefore, if the runtime values of index variables take these special values, we also consider the HTTP requests as dangerous inputs.

5 IMPLEMENTATION

We implemented all techniques in a tool named JSGo. JSGo was built at the top of a number of existing tools, including ODGen [34], Restler [5], and ExpoSE [38]. The users need to provide code location and vulnerability type. This is available from various sources such as static analysis tools like ODGen and Fast. In this section, we delve into several important implementation issues.

5.1 Static Data-Flow Analysis

In §4.3.3, we perform static data-flow analysis to identify the variables that need to be symbolized. We utilized ODGen [34], a JavaScript taint-analysis tool for this task. We also attempted other alternative solutions such as Fast [29] and CodeQL [16], and found that ODGen outperformed them for our needs.

ODGen generates object dependence graphs illustrating the data dependencies among different objects in the application. It provides functions to traverse the data dependency edges in a backward manner using depth-first search. This perfectly aligns with our requirements to identify the data flows affecting internal variables. We thus directly leverage these functions to identify and export data dependencies.

However, a limitation of ODGen is its inability to complete analysis within a reasonable time for large Node.js applications. To address this challenge, we adopt a strategy to simplify the program based on our specific needs. Specifically, we profile the application when executing the HTTP requests. Since all required functions are reserved, the semantics and functionality per case are also preserved. Subsequently, ODGen analyzes this reduced codebase to reduce its analysis time.

5.2 Type Consistency

We find that although JavaScript supports dynamic typing, the values of parameters should maintain the same type. In other words, when we mutate request parameters, the modified values should retain the original type to avoid invalidating the HTTP request; otherwise, this could lead to errors in the Node.js application. When identifying parameter keys or values corresponding to the program variables based on the consistency of their values, we also need to consider their types.

JavaScript variables may exist in singular types such as integers or compound types like objects. Regardless of their types, we can capture their values. Specifically, we employ the `JSON.stringify()` function to log the runtime values of specific variables during the execution of reference test cases. Subsequently, we parse the file contents to perform type conversion. For instance, JavaScript objects typically start with `{`, while strings begin with `"` or `'`. We ensure the logged runtime values are correctly typed before utilizing them in request parameters. For example, in the input generated by JSGo in Listing 3, the value assigned to `keyUpdate` is of the object type, and we directly incorporate the logged object (*e.g.*, `{...}`) into the request.

6 EVALUATION

We evaluate JSGo's capability of vulnerability validation in two aspects: 1) reproducing known vulnerabilities and 2) validating static detection results.

6.1 Dataset and Setup

In this section, we discuss the dataset we use and how we set up the experiments.

6.1.1 Dataset. To evaluate the efficacy of JSGo and other tools, we include 15 Node.js web applications (including 19 versions) that contain known vulnerabilities in our dataset, as shown in Table 2. The dataset includes 11 real web applications (including 13 versions) and 4 synthetic ones (including 6 versions, marked with † in Table 2) intentionally designed to be vulnerable. We believe this dataset is diverse and representative as it includes seven applications evaluated by prior work [29, 56, 64], ordinary applications like pdx-parks, and popular applications like Ghost. Note that some applications like Express.js were evaluated by prior works [56], but were not included in our dataset because we could not find any known vulnerabilities in them. Nevertheless, JSGo has indirectly analyzed Express.js because it is included by other applications (*e.g.*, parse-server) as a component.

To reproduce known vulnerabilities, we require the corresponding sink operations as targets. We searched for known vulnerabilities in Node.js programs using CVE databases, GitHub issues and commits, blogs, and Snyk [67]. However, we found limited public information on these vulnerabilities. For instance, some applications do not disclose PoC exploits and CVE details. We therefore identified vulnerable code locations by searching for keywords like XSS in commits. Some CVEs were reported by static analysis tools [29, 34]. To obtain more detailed information, we reviewed the vulnerabilities reported by these tools and reran them on the affected application versions to gather further specifics. In total, we collected 26 known vulnerabilities in these 15 Node.js applications.

Besides reproducing known vulnerabilities, we also used JSGo to validate the static analysis results of the *latest application versions*. We applied static analysis tools to all applications in Table 2, except for the synthetic ones. We also validated static alerts in 8 additional applications (more details in §6.3) selected from §2, as they have comprehensive test suites according to our study.

6.1.2 Setup. We discuss the experiment setup in this part.

Comparison with Related Tools. Since we cannot find any directed fuzzing tools for Node.js applications, we include state-of-the-art web application fuzzers and scanners in the evaluation. We do not compare JSGo with validators or monitors that only monitor the application's runtime behaviors and cannot produce test inputs, because JSGo proactively generates HTTP requests and is not a (reactive) validator. Specifically, we compare with Restler [5], Witcher [64], Miner [40] and the web scanner Burp [49]. Witcher operates in two steps: 1) initially crawling the application and collecting a set of entry URLs, and 2) separately fuzzing each URL. Since certain Node.js applications interface with clients programmatically via REST APIs and their index web pages (if accessible) contain no useful information like hyperlinks, they are incompatible with crawler-based testing tools like Witcher and Burp. Additionally, Witcher operates by instrumenting the Node.js runtime, whose version is not compatible with certain applications. Consequently, we do not evaluate Witcher and Burp on these applications. To use REST API fuzzers, we reuse existing application specifications or make a new one based on their documents if none is available. We customize the dictionaries and specifications in other tools to ensure that our collected attack payloads are applied to all applications with each tool. We apply the methods in §4.4 to validate if the generated HTTP requests are true positives.

Target Code. Our evaluation consists of two main components: reproducing known vulnerabilities in older versions of applications and generating test inputs to validate new vulnerabilities. Generating inputs for reproducing known vulnerabilities is useful because such inputs are sometimes unavailable. In our evaluation, we identified 8 instances of reflected Cross-Site Scripting (XSS), 6 instances of SQL Injection (SQLi), and 12 instances of prototype pollution vulnerabilities in these applications. 12 out of 26 known vulnerabilities do not provide public PoC exploits. For validating new vulnerabilities, we select the vulnerable locations identified by existing static analysis tools [34, 56] as the target locations for JSGo.

Time Limit. JSGo produces application inputs by modifying the HTTP requests generated by Restler. It has two steps: first, JSGo relies on Restler to generate seed inputs; second, it selects and adjusts these requests to approach specific target locations. We execute Restler for one hour before transitioning to JSGo's mutation process. This one-hour duration is sufficient for Restler to generate an ample number of seeds for JSGo. In the mutation phase, JSGo can exit automatically, for example, when it completes the analysis or encounters a failure in selecting test cases.

For Witcher, we limit its crawler to run for four hours and fuzz each URL for 20 minutes, following the practice in its paper. We set a time limit of six hours for other dynamic analysis tools. To facilitate a fair comparison with previous techniques, we allow all other tools to run until they reach the time limit.

6.2 Reproducing Known Vulnerabilities

We present the efficiency of JSGo in reproducing known vulnerabilities in this subsection.

6.2.1 Overall Results. We list the results of reproducing known vulnerabilities in Table 2. JSGo, Restler, Miner, Witcher, and Burp reproduced 20, 10, 8, 9, and 10 cases, respectively. We explain the performance of the other tools first. We will discuss JSGo in detail in subsequent sections.

Restler and Miner. Despite Miner's objective of enhancing Restler's capabilities, our experiments showed that it used more time than Restler to reproduce five cases. The reason is probably that Restler, which is actively maintained by Microsoft, has better integrated the techniques introduced in Miner. For instance, Restler could generate APIs that reuse contents from prior responses, whereas Miner, despite initially proposing this technique, did not achieve this design in certain applications (*e.g.*, parse-server). Since JSGo reused Restler for the first hour of fuzzing, it used the same time frame as Restler for reproducing the vulnerabilities within 1 hour. Note that Restler and Miner heavily relied on swagger/OpenAPI specifications. They could not reproduce the vulnerabilities (*e.g.*, case 25) in the endpoints not specified in the specifications.

Burp. Burp used the shortest time to trigger vulnerabilities in eight cases (*e.g.*, case 3). Unlike other tools, Burp did not require compilation phases (required by REST API fuzzers) or fuzzing phases

Table 2: Evaluation results on Node.js applications. † denotes synthetic applications. PP denotes prototype pollution vulnerability type. N/A means the tool is not applicable to test this application. - means that the tool cannot reproduce the vulnerability (within the time limit). Target code is only required by JSGo.

Application			Vulnerability Information			Vulnerability Triggering Time (hours)					Concolic Execution	
ID	Name	Stars	Type	Target Code	PoC Availability	Restler	Miner	Witcher	Burp	JSGo	# Iteration	Time (minutes)
1	thinkjs-3.2.4/thinkjs-helper	20	PP	index.js:11	No	-	-	N/A	-	-	-	-
2	thinkjs-2.2.3	5.3K	SQLi	src/model/base.js:366	No	0.58	-	N/A	-	0.58	0	0
3	parse-server-6.4.0	20.6K	SQLi	src/Adapters/Storage/Mongo/MongoTransform.js:354	Yes	0.22	0.23	N/A	0.05	0.22	0	0
4	parse-server-5.0.0-alpha.13	20.6K	PP	src/Controllers/DatabaseController.js:262,268	Yes	-	-	N/A	-	3.67	2	<1
5	parse-server-5.0.0-alpha.13	20.6K	PP	src/Adapters/Storage/Mongo/MongoTransform.js:514	Yes	-	-	N/A	-	2.80	3	2
6	parse-server-5.0.0-alpha.13	20.6K	PP	src/RestWrite.js:1618	Yes	-	-	N/A	-	-	-	-
7	parse-server-5.0.0-alpha.13	20.6K	PP	src/triggers.js:113	Yes	-	-	N/A	-	2.39	1	<1
8	juice-shop-16.0.0†	9.5K	XSS	routes/trackOrder.ts:17	Yes	0.12	0.12	1.30	0.08	0.12	0	0
9	juice-shop-16.0.0†	9.5K	SQLi	routes/trackOrder.ts:20	Yes	-	-	1.82	-	3.60	4	6
10	juice-shop-16.0.0†	9.5K	SQLi	routes/updateProductReviews.ts:17	Yes	-	-	-	-	5.55	4	11
11	juice-shop-16.0.0†	9.5K	SQLi	routes/login.ts:36	Yes	0.05	0.04	0.60	0.01	0.05	0	0
12	juice-shop-16.0.0†	9.5K	SQLi	routes/search.ts:23	Yes	0.05	0.05	0.87	0.34	0.05	0	0
13	NodeGoat/marked@0.3.5†	1.8K	XSS	app/views/memos.html:31	No	-	-	1.30	-	1.74	2	6
14	NodeGoat/marked@0.3.6†	1.8K	XSS	app/views/memos.html:31	No	-	-	1.35	-	1.63	2	6
15	NodeGoat/marked@0.3.8†	1.8K	XSS	app/views/memos.html:31	No	-	-	1.31	-	1.63	2	6
16	NodeGoat/forever@2.0.0†	1.8K	PP	forever/index.js:31	No	-	-	-	-	-	-	-
17	NodeGoat/nconf@0.10.0†	1.8K	PP	lib/nconf/stores/memory.js:96	No	-	-	-	-	-	-	-
18	totaljs/framework-3.4.5	4.3K	PP	utils.js:6612	No	2.70	-	N/A	N/A	4.72	2	<1
19	pdx-parks	3	PP	node-jquery-deparam.js:75	Yes	0.46	0.46	0.34	0.16	0.46	0	0
20	Ghost-1.19.2	45.6K	PP	core/server/api/invites.js:224	No	-	-	-	-	3.88	4	5
21	ember.js-4.8.0	22.4K	PP	packages/@ember/-internals/metal/lib/property_get.ts:115	Yes	-	-	N/A	N/A	-	-	-
22	moleculer-0.14.33	6K	PP	src/utils.js:413	No	-	-	-	-	-	-	-
23	NodeBB-0.6.1	14K	XSS	src/topics/posts.js:149	No	0.35	0.20	N/A	0.03	0.35	0	0
24	hapi-0.15.9	14.6K	XSS	lib/payload.js:236	Yes	0.17	0.32	N/A	0.03	0.17	0	0
25	Ghost-4.3.0	45.6K	XSS	core/server/web/admin/views/preview.html:6	No	-	-	0.56	0.22	1.77	1	<1
26	docsify-4.12.0	27.1K	XSS	src/plugins/search/search.js:212	Yes	0.47	0.50	0.65	0.01	0.47	0	0

(required by Witcher), which consumed additional time. However, Burp was unable to exercise applications in a stateful manner. For example, to reproduce certain vulnerabilities related to updating resources (*e.g.*, case 2 and case 10), it is necessary to construct a PUT request operating on the resource identifiers of a POST request. Burp was not aware of the dependencies, thus could not reproduce deep vulnerabilities.

Witcher. Witcher was able to test only 15 out of 26 vulnerabilities. The reason arose from its reliance on a web crawler to fetch application URLs and its dependency upon a specific version of the Node.js runtime (§6.1.2). In the 15 cases that Witcher could test, it successfully reproduced 10 cases within the time limit. It is worth noting that Witcher is not designed as a directed fuzzer. It took some time to crawl before conducting the fuzzing experiments, and thus, it took more time.

6.2.2 Advantages of JSGo. JSGo reproduced 6 vulnerabilities that other tools could not reproduce, because it enhances vulnerability validation in two aspects: it generates HTTP requests 1) reaching deep code locations and 2) exploring diverse execution contexts. They are explain the false negatives of existing approaches.

Reaching Deep Code Locations. Reaching the code is necessary to trigger the vulnerabilities there. JSGo can generate complex inputs that reach deep code locations, even when the input semantics are not explicitly encoded in the specifications or dictionaries. In parse-server, JSGo is the only tool capable of reproducing three prototype pollution vulnerabilities. This is because JSGo could generate complex parameters (such as those in object type). Testing other applications also necessitates complex inputs. For instance, to reach the target code and trigger SQL injection vulnerabilities in juice-shop (case 10), we need to craft a `PATCH` request containing representations of the JSON Patch operations.

Exploring Diverse Execution Contexts. Reaching the vulnerable code does not necessarily indicate successful vulnerability reproduction, as certain vulnerabilities can only be triggered in specific contexts. Case 20 is the example where other dynamic tools reach the vulnerable location (*i.e.*, `fetchLoggedInUser()` function) but cannot trigger the vulnerabilities. The reasons for this might include: 1) vulnerabilities are sanitized in some paths but not all, and 2) the inputs to sink operations are untrusted inputs in only a few paths to the sink functions.

6.2.3 Factors Influencing Vulnerability Reproduction Efficiency. The time required to trigger different vulnerabilities varies widely, ranging from less than a minute to several hours. There are mainly two factors affecting JSGo's efficiency in reproducing a vulnerability.

Application Code Complexity. JSGo typically spent most of its time on the static data-flow analysis phase. Due to variations in application sizes, we could discern significant differences in the time taken to construct data dependency graphs and trigger vulnerabilities. As shown in Table 2, dynamic symbolic execution did not consume much time. This demonstrated the effectiveness of our strategies outlined in §4.3.3, as only one path was concolically executed and one symbolic constraint was solved in an iteration.

Diversity of Reachable Test Suites. JSGo cannot identify the test suites that execute specific paths leading to vulnerabilities, which can lead to its low efficiency when it finds many reachable test suites that exercise the target code. Case 18 is an example of this challenge. The vulnerability is located in a `set()` function of `utils.js` in `totaljs`. This function is widely called. It is not only directly invoked by its relevant test cases but also indirectly called through test cases that invoke other components, which further invoke the function. As a result, test cases reaching the vulnerable code are quite common while only the ones calling `U.set()` are useful for triggering the vulnerabilities. JSGo spent time mutating requests based on the test cases invoking the vulnerable function

even if these test suites safely invoked that function. It was less efficient in some cases because of it.

6.2.4 Failure in Vulnerability Reproduction. In this subsection, we investigate why JSGo could not reproduce certain vulnerabilities. The reasons include 1) JSGo has false negatives in reproducing vulnerabilities and 2) some vulnerabilities are unexploitable.

Lack of Suitable Test Suites. JSGo had false negatives when it could not find any suitable test case for aligning HTTP requests. For example, Restler—that JSGo uses to produce initial HTTP requests—failed to generate inputs that can reach the sink operation in case 21. When searching for reachable test cases that reach the sink operation, JSGo also found none. Consequently, it stopped attempting to adjust HTTP requests as it was unable to handle this case.

Interestingly, although the developers of ember.js did not include test code invoking the vulnerable function (specifically, the `setProperties()` function) in version 4.8.0, they added test cases to execute the function in the latest version. The Node.js community has increased its recognition of the importance of providing test suites for their applications [53]. We believe that our assumption of using test suites is valid in many cases.

Limitations of Implementations. JSGo failed to reproduce certain known vulnerabilities due to limitations in current prototype implementation. In case 6, JSGo encountered issues when attempting to execute concolic execution using the ExpoSE. Despite our efforts to address errors (*e.g.*, the inclusion of Jalangi2 with different versions in parse-server and ExpoSE), the tool still could not compute constraints because of some unsupported application code between the source variables and the artificial condition. Consequently, JSGo could not obtain the desired values for source variables and thus request parameters. Also, the static tool we used might not be able to discover the variables whose values are *constantly* identical to the user inputs. JSGo was unable to analyze such cases if it did not locate source variables to symbolize.

Unexploitable Vulnerabilities. Our evaluation encompasses certain known vulnerabilities that, upon confirmation, were found to be invalid. Consequently, it is impossible to reproduce these vulnerabilities. Below are the reasons.

First, some vulnerabilities within functionalities or components are unexploitable through HTTP requests. For instance, in case 16, the prototype pollution vulnerability is present in forever@2.0.0, a vulnerable package used within NodeGoat. NodeGoat does not utilize the vulnerable function to receive user inputs, so the reported vulnerabilities are not exploitable in the application. The vulnerabilities in case 17 are invalid for the same reason.

Second, for some reported vulnerabilities, their source inputs originate from trusted actors rather than end users. Therefore, these vulnerabilities are non-exploitable and have not been addressed by developers in the newer versions. Case 22 in moleculer is such an example. As we confirmed with the developers, since the sink operation of a reported vulnerability was located in an internal function that processed environment variables instead of user inputs, the vulnerability was considered invalid.

6.2.5 False Positives. JSGo might produce test inputs that are rejected by client-side checks. As discussed in §4.4, JSGo monitors whether the runtime values of server-side sink variables contain prepared attack payloads. This method is insufficient for validating reflected XSS vulnerabilities as it does not consider client-side protections such as Content Security Policy enforcement and client-side sanitizations. To evaluate false positives, we replayed the inputs produced by JSGo, then observed if they could trigger an alert on the client side. The results showed that JSGo did not have any false positives in our experiments. The reason is that the tested applications did not employ proper client-side protections, leading to exploitable XSS vulnerabilities. It is possible to integrate automated client-side validation with JSGo to improve detection precision. We leave this as a future work.

The other two types of vulnerabilities are server-side threats. The capability to manipulate sink variable values could lead to successful exploitation. Specifically, the tampered sink variable values either altered SQL query structures (SQLi) or allowed modifications to the prototype of an object (prototype pollution). JSGo did not produce any false positives in our manual validation.

6.2.6 Concolic Execution's Contribution to Vulnerability Validation. To generate a test input, JSGo performs multiple iterations of concolic execution. Each iteration corresponds to one (artificial) constraint. In the last two columns of Table 2, we detail the iteration number of concolic execution and the time used to solve all constraints for producing each vulnerability-triggering input. In summary, JSGo required 1-4 iterations of concolic execution per vulnerability validation. The time taken to solve constraints usually spanned several minutes for one vulnerability.

JSGo solved artificial constraints rather than path constraints to streamline symbolic constraint solving. During each iteration, JSGo generates an artificial constraint directly or employs backward data-flow analysis, depending on whether the runtime value of the internal variable aligns with any request parameter. We explain this process in detail and present several case studies below.

Base Case. If the value of the internal variable is consistent with a request parameter, the variable can be directly symbolized and takes that value. Take case 4 as an example, where in its first iteration, JSGo introduces an artificial constraint $keyUpdate == \{\text{“}\_\_op\text{”} : ...\}$, where $keyUpdate$ is the internal variable and $\{\text{“}\_\_op\text{”} : ...\}$ is the reference value. ExpoSE solves a constraint $(= \ keyUpdate \ \{\text{“}\_\_op\text{”} : ...\})$ by assigning $\{\text{“}\_\_op\text{”} : ...\}$ to the fresh symbol $keyUpdate$. Note that while this artificial constraint is trivial, the real condition that uses $keyUpdate$ in the application code is not.

Data-flow Analysis. If an internal variable and the request parameter take different values, JSGo employs backward data-flow analysis to find source variables to symbolize. The artificial constraint is a bit more complex. For example, in the second iteration of case 20, Formula 1 shows the constraint represented with source variables, where the properties of an object variable *options* are source variables.

$$\begin{aligned} & options.method == \text{“}update\text{”} \wedge options.email == \text{“}test@test.com\text{”} \\ & \wedge AuthenticatoinConstraint \wedge (options.newPassword \neq null) \\ & \wedge (options.newPassword \neq undefined) \\ & \wedge (options.newPassword.length > 10) \\ & \wedge \underline{options.newPassword.toLowerCase() == \text{“}sl1m3rson99\text{”}} \end{aligned} \tag{1}$$

It is worth highlighting two points here. First, ExpoSE collected all real path conditions until it met the artificial

one (underlined above). The value of an internal variable was *options.newPassword.toLowerCase*(), which was different from the user-supplied input *options.newPassword*. ODGen detected their data flow, and JSGo symbolized *options.newPassword*. Note that JSGo omitted format checks on *options.newPassword.toLowerCase*() in the application code but utilized the reference value ("*sl1m3rson99*") for producing this artificial constraint.

Second, we have observed that other cases, such as the first iteration in case 13, resemble case 20 and do not involve complex constraints from source variables to internal variables. However, this does not imply that internal variables can always be represented by source variables using simple constraints. On the contrary, this issue primarily occurs because ExpoSE lacks full support for complex operations, including many built-in functions and asynchronous processes [38]. We discuss a possible solution in §7

6.3 Validating Static Detection Results

In this section, we apply JSGo to validate static vulnerability detection results in the latest versions of Node.js applications in our dataset. We tested 19 web applications, including 11 real web applications from Table 2 and 8 additional ones (the applications listed in Table 3, `Rocket.Chat/Rocket.Chat`, and `meteor/meteor`). We leverage static analysis tools, including ODGen [34], Fast [29], and silent-spring [56] to identify potential XSS, SQLi and prototype pollution vulnerabilities. Note that Fast detects XSS and SQLi; silent-spring detects only prototype pollution; and ODGen is known to have a scalability issue [34]. We set a 6-hour time limit for each static analysis tool per application version.

6.3.1 Results. We applied JSGo to validate all 75 unique alerts reported by the static analysis tools. We used the vulnerable code locations reported by static analysis as the testing targets. JSGo stopped analysis in a few cases where there were no suitable test cases (the reasons are discussed in §7). In total, it took approximately 44 hours to analyze all the alerts. To improve efficiency, object dependence graph construction, the most time-consuming part in JSGo, can be done once per application and reused for data-flow queries when validating multiple static alerts.

For each validated vulnerability, we list in Table 3 the static analysis tools that reported the alerts, the total number of static alerts of the same vulnerability type reported by these tools, and the reference test case. It can be observed that static analysis tools can report alerts in the latest versions of popular applications, and JSGo can leverage test suites to validate the alerts.

Among the static analysis results, JSGo validated 7 as true-positive vulnerabilities in 6 popular applications like `derbyjs/derby` and `hexojs/hexo`. To validate the 7 static alerts, JSGo required at most 5 iterations of concolic execution. We acknowledge that JSGo might have false negatives in vulnerability validation because of reasons discussed in §6.2.4. However, we did not evaluate false negatives as proving a case unexploitable is difficult and orthogonal to our research goal. Additionally, all the new vulnerabilities validated by JSGo were instances of prototype pollution. This may be attributed to the fact that prototype pollution is a relatively new type of vulnerability, whereas XSS and SQL injection (SQLi) vulnerabilities have been extensively tested and studied in prior works [29, 34].

We further discuss the advantages of JSGo in validating static detection results. It uses test suites and can test undocumented code. In `derbyjs/derby`, attackers need to manipulate the value of a specific attribute "`as`" in template files. JSGo identified the critical input field "`as`" by examining the consistency between input data (provided in test suites) and values of the sink variables reported by the static analysis tool. Without leveraging test suites, generating an end-to-end input is difficult because the OpenAPI specifications do not account for the practice of users supplying malicious tag values within these templates.

Our concolic execution guided by reference values contributed significantly to validating static alerts. For example, in `keystonejs/keystone`, some internal variables represent the length of attacker-controllable inputs (*e.g.*, `propPath` and `shortcut`). To validate two reported static alerts, the length variables must satisfy certain conditions before the executions reach the reported sink location. The reference test cases provide direct solutions (*e.g.*, "`propPath.length==3`" and "`shortcut.length==2`") to satisfy these checks. The artificial constraint expressed by JSGo is shown in Formula 2. The SMT solver solved the constraints and generated input values (*e.g.*, "`propPath : ['block', '', '']`", where it added two items of empty string `''` to increase the array length) required to reach the reported sink locations.

$$\begin{aligned}&(and\ \ (=\ \ (to\_real\ \ propPath\_Array\_Length)\ \ 3.0)\\&\qquad(=\ \ (to\_real\ \ shortcut\_Array\_Length)\ \ 2.0)\ )\end{aligned} \tag{2}$$

Disclosure. We reported all the 7 validated vulnerabilities to the developers for confirmation. It is worth noting that some developers patch vulnerabilities only if they believe the vulnerabilities pose a legitimate risk to their users. For instance, JSGo validated a vulnerability in `hexojs/hexo`. The developers were uncertain whether end users had the capabilities to manipulate the source data in the way demonstrated in our test input. The same applied to `hapijs/hapi`. The developers believed that exploitation cannot be carried out by non-privileged users, while JSGo cannot determine the identity of users supplying the inputs. At the time of writing, the developers confirmed and fixed two vulnerabilities we reported.

6.3.2 Comparison. We compare JSGo with Restler because Restler could test all Node.js web applications in Table 3. JSGo performed better as it validated seven static alerts yet Restler could *not* validate any of them.

Three applications (`Rocket.Chat/Rocket.Chat`, `hexojs/hexo`, and `meteor/meteor`) in our dataset were previously analyzed by silent-spring [56]. Its authors manually analyzed 32 reported alerts for these applications and identified 1 alert in `Rocket.Chat/Rocket.Chat` as exploitable. JSGo confirmed another vulnerability in `hexojs/hexo` that was missed by the authors. This demonstrates that JSGo could assist humans in validating more vulnerabilities.

6.3.3 Case Studies. In this section, we disclose the details of two vulnerabilities validated by JSGo and have been patched by developers.

Prototype Pollution in Derby. Derby is a popular Node.js web framework with 4.7K stars on GitHub. JSGo validated a prototype pollution vulnerability in Derby, which prompted a fix in its latest

Table 3: New vulnerabilities validated by JSGo. - denotes no static alerts are validated. `Static tools` denote the static tools that report the validated vulnerability. `S` denotes silent-spring, `O` denotes ODGen, and `F` denotes Fast. `# Alerts` denotes the number of unique static alerts in a type of "Type".

Applications	Github Stars	Type	Static tools	# Alerts	Reference Test Case	# Iteration	State
derbyjs/derby-4.0.0	4.7K	PP	S, O	4	"ignores DOM mutations in components' create()"	5	Patched
primefaces/primereact-10.6.2	5.7K	PP	S	5	"when input is using keyfilter for alphabetic accept paste of alphabetic values"	3	Patched
hapijs/hapi-21.3.7	14.5K	PP	S	2	"exposes an api (drops scope by default)"	2	Reported
hexojs/hexo-7.1.1	38.4K	PP	S	1	"write config"	2	Reported
keystonejs/keystone-5e078c	8.8K	PP	S, O	2	"child field in nested array"	4	Reported
keystonejs/keystone-5e078c	8.8K	PP	S, O	2	"defaults limits to 100KiB"	4	Reported
windyfancy/webcontext-25b80c	153	PP	S, O	1	"5.test read and write session "	1	Reported

version. Restler could not reproduce this vulnerability because it failed to reach the reported sink location.

To validate the vulnerability, JSGo first identified a test case listed in Table 3 that reached this vulnerable function. While comparing the traces of HTTP requests and the reference trace, JSGo identified the important input field, *i.e.*, the `"as"` attribute set to HTML templates. Indeed, the functionality of handling HTML templates is using the vulnerable function. When the users supply HTML templates to the applications, they can send requests to manipulate the `"as"` attribute to be the payload `__proto__` and conduct prototype pollution attacks.

Prototype Pollution in Primereact. Primereact is a rich set of open-source UI Components for React, with 5.7K stars on GitHub [51]. Its code implementation in handling locale is vulnerable to prototype pollution. A locale is a set of parameters that defines the user's language preferences that the user wants to see in their user interface. The client-side users can request to set or update the locale data they wish to use, which is stored in the server-side application. The user input ultimately influences the parameters of `updateLocaleOption()`, a function identified as vulnerable to prototype pollution. After we reported the feasibility of controlling parameters of `updateLocaleOption()`, the developers implemented security checks to filter out values of `'__proto__'` and `'prototype'` used in the object index variables of this function. Interestingly, they also added the same checks in other locale functions that were not reported as vulnerable by the static tools.

7 DISCUSSION

Lack of Intersecting Traces. JSGo mutates the HTTP requests whose traces intersect with reference traces. It is possible that reference test cases do not exhibit traces intersecting with the executions of HTTP requests. One possible approach to addressing this is chaining test suites. Suppose we aim to adjust the HTTP request based on the test case T_b, yet their executions show no intersection. If we can identify another test case T_a that shares intersecting execution with both the HTTP request and T_b, we can then align the HTTP request with T_a to ensure that its execution also intersects with T_b. We further adjust the mutated HTTP request to align its execution with that of T_b.

Static Symbolic Execution. In this work, we use concolic execution to help align execution traces and generate end-to-end test inputs. However, concolic execution could not solve some complex artificial constraints (see §6.2.6). This can potentially be mitigated through static symbolic execution solutions like Fast [29]. In particular, we believe static symbolic execution and concolic execution approaches have their own advantages and disadvantages. On the one hand, static symbolic execution tools can generate reference values for request parameters even if no reference test case is available. On the other hand, using the selected HTTP requests as initial inputs, concolic execution can effectively test related program paths that depend on particular input values. Concolic execution can also handle Node.js programs' dynamic behaviors as it executes the code and collects constraints at runtime. Integrating static symbolic execution into JSGo thus could further improve JSGo, especially when there is no reachable test suite. We leave this as a future work.

8 RELATED WORK

Node.js and NPM Security. Many works are proposed to study, detect, and exploit vulnerabilities in Node.js applications and the NPM ecosystem. Zimmermann *et al.* demonstrate that the NPM ecosystem leads to high risk to the software supply chain due to NPM packages' dependencies on malicious code [74]. Xiao *et al.* discovered a novel feasible attack called "hidden property abusing" in the Node.js applications [71]. Some work uses static or hybrid analysis to detect vulnerabilities (*e.g.*, injection vulnerabilities and supply chain attacks) in NPM packages [29, 34, 60], JavaScript bundling [52], and Node.js applications [29, 34, 56]. Liu *et al.* [37] and Shcherbakov *et al.*[57] further detect and chain prototype pollution gadgets to exploit prototype pollution. In addition to injection vulnerabilities, Oz *et al.* detect file upload vulnerabilities on Node.js [43]. Staicu *et al.* find that regular-expression denial-of-service (ReDoS) vulnerabilities could compromise the availability of JavaScript-based web servers [59], and several works are proposed to detect ReDoS vulnerabilities [35, 36, 52, 70] or recover the performance of web services against ReDoS attacks [6]. To facilitate evaluating vulnerabilities, Bhuiyan *et al.* [9] and Oliver *et al.* [42] propose benchmarks containing different types of vulnerabilities for Node.js.

Besides vulnerability detection and exploitation, some works propose or investigate defense mechanisms for JavaScript. Mir [65] and Hodor [69] protect Node.js applications by restricting code privilege. JSLIM [73] and Mininode [31] debloat Node.JS applications to reduce attack surfaces. Synode [60] rewrites the source code to enforce a safe mode for preventing injection vulnerabilities on Node.js. BreakApp [32] compartmentalizes JavaScript applications at the boundaries of untrusted modules to enhance security. NatiSand [1] and JSand [2] sandbox JavaScript; and SANDDRILLER [3] and ADsafe [48] verify JavaScript sandboxing properties.

JSGo particularly focuses on validating security issues in Node.js applications. It uses test suites to generate HTTP requests for reproducing known vulnerabilities and validating static analysis results.

Test Suites and Software Maintenance. Test suites are commonly utilized in testing compilers or interpreters. In JavaScript, many researchers use test suites to fuzz JavaScript engines. Superion [68], DIE [44], and Montage [33] implement the AST level mutations using the JavaScript test code as seed inputs. In particular, Superion finds bugs in JavaScript engines by conducting crossover on the AST sub-trees of the testing code. DIE advances the mutation by restricting the types of sub-trees [44].

Test suites are also used in other domains. Jeong *et al.* [27] discovered that popular GitHub applications commonly include test suites, which they utilize to generate API call sequences for fuzzing. Shamshiri *et al.* [55] and Bavota *et al.* [7] investigate how test suites impact software maintenance.

This work uses test suites to generate end-to-end test inputs for Node.js applications, and is orthogonal to these prior works.

Dynamic Web Application Testing. Dynamic techniques generate specific inputs for testing web applications [18, 20, 64]. In Enemy of the State [18], server-side states are inferred by comparing client-side response differences. Alternatively, Navex [4] monitors the server side by tracking session creation and database queries. Moreover, jÄk [46] and Black Widow [20] also consider client-side events like form submissions. ProbetheProto [30] uses dynamic taint analysis to detect client-side prototype pollution among one million real-world websites. JSGo, on the other hand, leverages the comprehensive test suites into a more directed testing.

9 CONCLUSION

This paper has introduced JSGo, a tool that leverages test suites to generate end-to-end application inputs for validating vulnerabilities in Node.js applications. Unlike existing dynamic tools, JSGo benefits from test suites, allowing it to identify which input fields to mutate and the values to mutate towards, even if this information is not encoded in specifications. Our investigation into existing Node.js applications reveals that test suites offer valuable insights into enhancing the effectiveness of existing vulnerability validation approaches. We use under-constrained analysis to identify source variables and address challenges encountered when symbolically executing modern Node.js applications. In our evaluation, JSGo reproduced 20 out of 26 known vulnerabilities and validated 7 static alerts in recent versions of 23 server-side Node.js applications.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14209323).

REFERENCES

[1] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. NatiSand: Native code sandboxing for JavaScript runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*.

[2] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*.

[3] Abdullah Alhamdan and Cristian-Alexandru Staicu. 2023. SandDriller: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, USA.

[4] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. Montréal, Canada.

[6] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime recovery of web applications under zero-day redos attacks. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[7] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE.

[8] bcoe. 2024. c8 - native V8 code-coverage. https://www.npmjs.com/package/c8.

[9] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench. js: An executable security benchmark suite for server-side JavaScript. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, USA.

[11] Brainhub. 2023. Famous Node JS Apps. https://brainhub.eu/library/famous-apps-built-with-nodejs.

[12] Stefano Calzavara, Mauro Conti, Riccardo Focardi, Alvise Rabitti, and Gabriele Tolomei. 2019. Mitch: A machine learning approach to the black-box detection of CSRF vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy*.

[13] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event.

[14] code intelligence. 2024. CVE-2023-36665. https://www.code-intelligence.com/blog/cve-protobufjs-prototype-pollution-cve-2023-36665.

[15] code intelligence. 2024. CVE-2024-39853. https://gist.github.com/mestrtee/840f5d160aab4151bd0451cfb822e6b5.

[16] codeql. 2024. Discover vulnerabilities across a codebase with CodeQL. https://codeql.github.com/.

[17] dev.to. 2023. Securing Your Node.js Apps by Analyzing Real-World Command Injection Examples. https://dev.to/lirantal/securing-your-nodejs-apps-by-analyzing-real-world-command-injection-examples-1ll6.

[18] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA, USA.

[19] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue. 2024. Vulnerability-oriented Testing for RESTful APIs. (Aug. 2024).

[20] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

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

[22] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Sacramento, CA, USA.

[23] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, USA.

[24] infosecinstitute. 2024. Dictionary attack using Burp Suite. https://www.infosecinstitute.com/resources/hacking/dictionary-attack-using-burp-suite/.

[25] istanbuljs. 2024. Istanbul's state of the art command line interface. https://www.npmjs.com/package/nyc.

[26] Jasmine. 2024. Jasmine is a Behavior Driven Development testing framework for JavaScript. https://github.com/jasmine/jasmine.

[27] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. Utopia: Automatic generation of fuzz driver using unit tests. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[28] Jest. 2024. Jest is a delightful JavaScript Testing Framework with a focus on simplicity. https://jestjs.io/.

[29] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. 2023. Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[30] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites.. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[31] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[32] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. Sok: Taxonomy of attacks on open-source software supply chains. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[33] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.

[34] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining node. js vulnerabilities via object dependence graph and query. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA, USA.

[35] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. 2022. RegexScalpel: Regular Expression Denial of Service (ReDoS) Defense by Localize-and-Fix. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA, USA.

[36] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[37] Zhengyu Liu, Kecheng An, and Yinzhi Cao. 2024. Undefined-oriented Programming: Detecting and Chaining Prototype Pollution Gadgets in Node. js Template Engines for Malicious Consequences. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[38] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*.

[39] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[40] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. 2023. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, USA.

[41] Mocha. 2024. Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser. https://mochajs.org/.

[42] Philip Oliver, Jens Dietrich, Craig Anslow, and Michael Homer. 2024. CrashJS: A NodeJS Benchmark for Automated Crash Reproduction. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE.

[43] Harun Oz, Abbas Acar, Ahmet Aris, Güliz Seray Tuncay, Amin Kharraz, and Selcuk Uluagac. 2024. (In) Security of File Uploads in Node. js. In *Proceedings of the Web Conference (WWW)*. Singapore.

[44] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2023. Fuzzing javascript engines with aspect-preserving mutation. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[45] parse community. 2024. Parse Server for Node.js / Express. https://github.com/parse-community/parse-server.

[46] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jäk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Kyoto, Japan.

[47] Andrey Petukhov and Dmitry Kozlov. 2008. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University* (2008).

[48] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *Proceedings of the 20th USENIX Security Symposium (Security)*. San Francisco, CA, USA.

[49] portswigger. 2024. Burp Suite - Application Security Testing Software. https://portswigger.net/burp.

[50] portswigger. 2024. Small to mid-size business cybersecurity solutions. https://portswigger.net/organizations/small-business-security.

[51] primereact. 2023. The Most Complete React UI Component Library. https://github.com/primefaces/primereact.

[52] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark.

[53] Ville Santala. 2022. Automated Testing in a CI/CD pipeline: node. js and react software project. (2022).

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

[55] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. 2018. How do automatically generated unit tests influence software maintenance?. In *2018 IEEE 11th international conference on software testing, verification and validation (ICST)*. IEEE.

[56] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent spring: Prototype pollution leads to remote code execution in Node. js. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, USA.

[57] Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. 2024. Unveiling the Invisible: Detection and Evaluation of Prototype Pollution Gadgets with Dynamic Taint Analysis. In *Proceedings of the Web Conference (WWW)*. Singapore.

[58] snyk.io. 2024. CVE-2023-26158. https://security.snyk.io/vuln/SNYK-JS-MOCKJS-6051365.

[59] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: a study of ReDoS vulnerabilities in JavaScript-based web servers. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, USA.

[60] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. 2018. Understanding and automatically preventing injection attacks on Node. js. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[61] statista.com. 2023. Most used web frameworks among developers worldwide, as of 2023. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/.

[62] swagger. 2024. OpenAPI Specification. https://swagger.io/specification/.

[63] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark.

[64] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[65] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing dynamic library compromise on node. js via rwx-based privilege reduction. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. Virtual Event, Korea.

[66] W3Techs. 2023. Node.js Statistics: The Updated Guide on Node.js Usage and Trends. https://www.bacancytechnology.com/blog/nodejs-statistics.

[67] W3Techs. 2024. Snyk security. https://snyk.io/.

[68] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. Montréal, Canada.

[69] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. 2023. Hodor: Shrinking attack surface on node. js via system call limitation. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark.

[70] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, et al. 2023. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

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

[72] Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. 2022. Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-platform Ecosystem. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, USA.

[73] Renjun Ye, Liang Liu, Simin Hu, Fangzhou Zhu, Jingxiu Yang, and Feng Wang. 2021. JSLIM: Reducing the known vulnerabilities of JavaScript application by debloating. In *International Symposium on Emerging Information Security and Applications*. Springer.

[74] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA, USA.