- 背景介绍和动机
- what is endpoint 、payload data、query data
- 实验是如何收集数据的
- 实验结果的含义

Statically Checking Web API Requests in JavaScript

Erik Wittern*, Annie T. T. Ying*, Yunhui Zheng*, Julian Dolby, Jim A. Laredo
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
Email: {witternj, aying, zhengyu, dolby, laredo}@us.ibm.com

***Abstract*—Many JavaScript applications perform HTTP requests to web APIs, relying on the request URL, HTTP method, and request data to be constructed correctly by string operations. Traditional compile-time error checking, such as calling a non-existent method in Java, are not available for checking whether such requests comply with the requirements of a web API. In this paper, we propose an approach to statically check web API requests in JavaScript. Our approach first extracts a request's URL string, HTTP method, and the corresponding request data using an inter-procedural string analysis, and then checks whether the request conforms to given web API specifications. We evaluated our approach by checking whether web API requests in JavaScript files mined from GitHub are consistent or inconsistent with publicly available API specifications. From the 6575 requests in scope, our approach determined whether the request's URL and HTTP method was consistent or inconsistent with web API specifications with a precision of 96.0%. Our approach also correctly determined whether extracted request data was consistent or inconsistent with the data requirements with a precision of 87.9% for payload data and 99.9% for query data. In a systematic analysis of the inconsistent cases, we found that many of them were due to errors in the client code. The here proposed checker can be integrated with code editors or with continuous integration tools to warn programmers about code containing potentially erroneous requests.**

***Keywords*-Static analysis; JavaScript; Web APIs**

I. Introduction

Programmers write applications using a growing variety of publicly accessible web services. Catalogs such as IBM's API Harmony [1], [2], Mashape's PublicAPIs [3], or ProgrammableWeb [4] list thousands of *web Application Programming Interfaces* (web APIs) exposed by these services. Applications invoke web APIs by sending HTTP requests to a dedicated URL using one of its supported HTTP methods; required data is sent as query or path parameters, or within the HTTP request body. The URL, HTTP method, and data to send are all basically strings, constructed by string operations within the applications. Figure 1 shows an exemplary excerpt of such a JavaScript application performing these actions.

When a request targets a URL that does not exist or sends data that does not comply with the requirements of the web API, a runtime error occurs. This prevalent calling mechanism for web APIs—which relies on a string URL, a HTTP method, as well as string input and output—does not allow type-safety checking. In other words, checks for traditional compile-time errors are not available for programmers writing code calling web APIs. A recent study found that a significant number of analyzed mobile applications will fail in light of changes to the web APIs they consume [5]. The situation is worsened as (web) applications are increasingly developed using dynamic languages like JavaScript, which generally also have minimal static checking.

As an example of resulting errors, we found code in GitHub that mistakenly attempts to make a request to `https://api.spotify.com/v1/seach`, as opposed to invoking the correct URL ending with `/search`. Another example we found (Figure 1) attempts to invoke the deprecated Google Maps Engine API.[1] A programmer wishing to avoid these errors can manually assess the correctness of web API requests by consulting the API's (online) documentation or formal web API specifications. Such specifications, like the OpenAPI Specification [6] (formerly known as *Swagger*, the name we will use for the rest of the paper) can be created by API providers or third parties to document valid URLs, HTTP methods, as well as inputs and outputs that a web API expects.

Tools that can automate this manual—and thus error-prone and tedious—checking should have two desirable features:

1) Such tools should statically analyze JavaScript source code to automatically identify HTTP requests and retrieve the related URL, HTTP method, and data, which are all encoded as strings and created using typical string operations like concatenation. In addition, the analysis must be inter-procedural as the strings can be assembled across functions.
2) As input, such tools should make use of available specifications, like Swagger, for the definitions of valid URLs, HTTP methods, and data.

Such tools can report errors either real-time as a programmer is writing the application, or during continuous integration. In addition, they can help API providers to monitor usages of their APIs in publicly available code.

In this paper, with these two features in mind, we propose an approach that takes as input Swagger specifications and statically checks whether the web API requests in JavaScript code conform to these specifications. Our approach first extracts the URL string, HTTP method, and the corresponding data from a request, using an inter-procedural static program analysis capable of extracting strings [7], and then checks whether the request conforms to publicly available web API specifications. For the initial implementation, we chose to handle requests

* The author names were sorted alphabetically. The authors contributed equally to the work.

[1] https://mapsengine.google.com/about/index.html

written using the jQuery framework due to its popularity – reportedly, 70% of websites use the jQuery framework [8]. The main contribution of our approach is in leveraging existing work in making static whole-program analysis possible for framework-based JavaScript web applications (i.e., [7], [9]–[11]) and applying it to a new problem of checking whether a request is consistent with a web API specification.

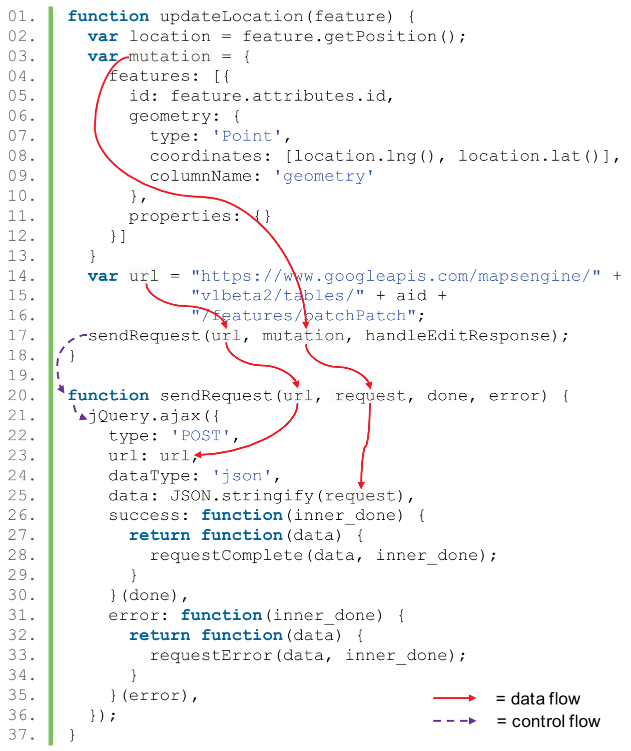
We evaluated our approach by checking whether web API requests from over 6000 JavaScript files on GitHub[2] were consistent or inconsistent with publicly available web API specifications provided by the APIs Guru project [12]. From 6575 requests for which we had web API specifications available, our analysis achieved a precision of 96.0% in correctly extracting and determining that the URL and HTTP method in a request is consistent or inconsistent with the corresponding web API specification. Our approach also correctly extracted and determined whether the request data was consistent or inconsistent with the data requirements with a precision of 87.9% for payload data and 99.9% for query data. We systematically examined all the URLs and payload data that were inconsistent with any specification (1477 cases) and found that many of these inconsistencies were due to errors in the client code, including calls to deprecated APIs, errors in the URLs, and errors in data payload definitions. In only five of the 1477 cases, limitations in our static analysis affected the matching of a request to a URL endpoint in the specification to the point of incorrectly flagging requests as mismatches. These limitations also extended to four out of 18 cases where request payloads were mistakenly flagged as mismatches and two out of 41 cases where query parameters were mistakenly flagged as mismatches. These results show that the static analysis is precise enough to be used in our proposed checker, for checking an application under development, or for checking the validity of web API usage in existing source code in case a web API undergoes changes.[3]

The remaining of this paper is organized as follows: After illustrating an example (Section II), we describe our approach: the static analysis (Section III) and the checker (Section IV). We then present the evaluation (Section V), related work (Section VI), threats (Section VII), and conclusion (Section VIII).

II. Background and Example

In this section, we first introduce concepts and terms regarding web APIs and their specifications. We then demonstrate through an example the two steps of our approach: how we use static analysis to extract the strings constructing a web API request (Section II-A) and how we check the results of the static analysis against Swagger specifications (Section II-B).

Web APIs are programmatic interfaces that applications invoke via HTTP to interact with remote *resources*, such as data or functionalities. Resources are identified by URLs while the type of interaction (e.g., retrieval, update, deletion of a resource) depends on the HTTP *method*. Following previous



```
function updateLocation(feature) {
  var location = feature.getPosition();
  var mutation = {
    features: [{
      id: feature.attributes.id,
      geometry: {
        type: 'Point',
        coordinates: [location.lng(), location.lat()],
        columnName: 'geometry'
      },
      properties: {}
    }]
  }
  var url = "https://www.googleapis.com/mapsengine/" +
            "v1beta2/tables/" + aid +
            "/features/batchPatch";
  sendRequest(url, mutation, handleEditResponse);
}

function sendRequest(url, request, done, error) {
  jQuery.ajax({
    type: 'POST',
    url: url,
    dataType: 'json',
    data: JSON.stringify(request),
    success: function(inner_done) {
      return function(data) {
        requestComplete(data, inner_done);
      }
    }(done),
    error: function(inner_done) {
      return function(data) {
        requestError(data, inner_done);
      }
    }(error),
  });
}
```

Fig. 1: Code excerpt of a request to the Google Maps Engine API; Source: https://raw.githubusercontent.com/jmcgill/plexernet/6bec8004e28813469bd9516c71d33b5b27e0f0ae/map_playground/playground.js

work, we refer to the combination of a URL and HTTP method as an API *endpoint* [13]. To be successfully invoked, some endpoints depend on additional data, for example the ID of a resource being sent as a path parameter within the URL or a new/updated state of a resource being sent in the body of an HTTP request.

Application developers can learn the usage of the endpoints of an API either by consulting its online documentation, typically presented in HTML, or by relying on a formal web API specification. Specifications define, among other things, an API's endpoints as well as the data required for and returned by requests. The OpenAPI specification (Swagger) is one of these specifications, which enjoys broad industry support [6]. Figure 2 shows an excerpt of a Swagger document describing the Instagram API. It defines, for example, the `schemes` of the API, its `host` and `basePath`, which together form the API's *base URL*, in this case `https://api.instagram.com/v1`. Swagger defines the different endpoints of an API in the `paths` property, using URL templates (possibly containing path parameters, i.e., `{tag-name}` in the path `/tags/{tag-name}/media/recent`) and supported HTTP methods. Per endpoint, Swagger provides a human-readable `description`, definitions of the `parameters` (path and query parameters as well as required HTTP bodies), definitions of possible `responses`, as well as `security` requirements.

[2]https://github.com/
[3]Consider the high number of changes reported for various APIs at https://www.apichangelog.com/

Entries in the `definitions` property describe the structure of data to send to or receive from endpoints using JSONschema notation [14] or a XML Object notation that is specific to Swagger. Data definitions can be referenced from endpoint definitions, as is exemplary shown for the `TagMediaListResponse` definition in Figure 2.

A. Determining the content of a request in JavaScript code

The first step of the checker is to extract the specifics of web API requests in the code. Our approach employs an inter-procedural static analysis to extract URL and input strings from a web API request in JavaScript code. Recall the code in Fig. 1 as an example. Focusing on the `url` variable, we can see that it is composed from two constant strings and the `aid` variable in the function `updateLocation`. The value of `url` is then passed to `sendRequest`, where it flows into the `jQuery.ajax` call. The value of `aid` is a parameter and could be different in multiple runs. Hence, when we aim to extract the URL used in this request, we denote `aid` as a symbolic value {`aid`} using curly braces, indicating that the value is not known statically. Overall, the URL extracted for the shown request is `https://www.googleapis.com/mapsengine/v1beta2/tables/{aid}/features/batchInsert`.

As this example shows, a simple textual search like `grep` would not be effective because the call site of the request (e.g., `sendRequest` in Figure 1) can be different from the definition of the URL string (`updateLocation` in Figure 1). In addition, given a URL string can be assembled across multiple functions and lexical scopes (e.g., the URL `https://api.instagram.com/v1/tags/{tag-name}/media/recent` which our static analysis correctly extracts from the code excerpts in Figure 3), resolving such an URL string requires non-trivial data flow analysis. The same holds for the HTTP method or request data values, which may be created within multiple functions.

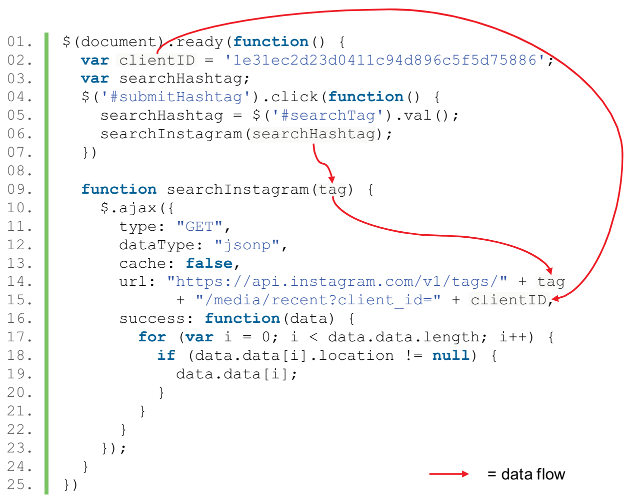
B. Checking a request against a web API Specification

The second step of our approach checks whether the extracted information from the web API requests conforms with web API specifications. Consider, for example, the URL `https://api.instagram.com/v1/tags/{searchHashtag}/media/recent?client_id=1e3...` extracted by our static analysis from the code excerpt in Figure 3. Our check would start by determining whether it - together with the associated method - targets an actual endpoint defined in the Swagger specification of the Instagram API, including the `searchHashtag` path parameter. In addition, we can check whether the `client_id` parameter is expected by the endpoint or if there are other query parameters required, which are missing in the URL. Finally, we can check if the data sent in the request body adheres to the data definitions in the Swagger specification.

Another option to check whether a web API request is correct would be to perform a dynamic analysis [15]. However, invocations of web APIs often require authentication (for example, using API keys), so that a system using dynamic analysis would need to provision keys to register and even agree to the terms of service. Typically, terms of service are not easy to understand by a layman, and are much less likely to be encoded in a machine readable form to allow a program to decide whether or not to comply with the terms. Finally, even if the key provisioning issue was addressed, ensuring dynamic analysis has the proper code coverage is challenging.

```
swagger: '2.0'
schemes:
  - https
host: api.instagram.com
basePath: /v1
paths:
  '/tags/{tag-name}/media/recent':
    get:
      description: Get a list of recently tagged media.
      parameters:
        - description: The tag name.
          in: path
          name: tag-name required: true type: string
        - description: Count of tagged media to return.
          in: query
          name: count
          required: false
          type: integer
      responses:
        '200':
          description: List of media entries with this tag.
          schema:
            $ref: '#/definitions/TagMediaListResponse'
definitions:
  TagMediaListResponse:
    properties:
      data:
        description: ...
        type: array
        items:
          $ref: '#/definitions/MediaEntry'
```

Fig. 2: Excerpt of a Swagger specification for the Instagram API, highlighting the `GET /tags/{tag-name}/media/recent` endpoint



```
$(document).ready(function() {
  var clientID = '1e31ec2d23d0411c94d896c5f5d75886';
  var searchHashtag;
  $('#submitHashtag').click(function() {
    searchHashtag = $('#searchTag').val();
    searchInstagram(searchHashtag);
  })

  function searchInstagram(tag) {
    $.ajax({
      type: "GET",
      dataType: "jsonp",
      cache: false,
      url: "https://api.instagram.com/v1/tags/" + tag
        + "/media/recent?client_id=" + clientID,
      success: function(data) {
        for (var i = 0; i < data.data.length; i++) {
          if (data.data[i].location != null) {
            data.data[i];
          }
        }
      }
    });
  }
})
```

Fig. 3: Code excerpt of a request to the Instagram API; Source: https://github.com/codeforamerica/mdc-instagram/blob/ master/app/assets/js/app.js

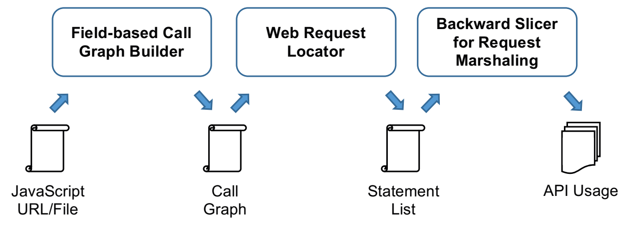


Fig. 4: Web API Usage Extractor Overview.

```
"request": {
  "cache": false,
  "success": "anonymousFunctionID",
  "dataType": "jsonp",
  "type": "GET",
  "url": "https://api.instagram.com/v1/tags/{searchHashtag}/" +
         "media/recent?client_id=d1e31ec2d23d0411c94d896c5f5d75886"
}
```

Fig. 5: Extracted API usage for the example in Figure 3.

III. Web API Usage Extraction

A fundamental part of our approach is to detect and extract web API usages from JavaScript source code. Figure 4 shows the web API usage extractor pipeline. The input is a JavaScript file. The output is web API usage including URLs and request payloads in JSON format. The decision to have a single JavaScript file as input, i.e., setting the analysis scope to the file-level as opposed to the program-level, is based on our initial observations that the input strings are often in the same file as the requests. This decision also supports an analysis light-weight enough to be used repeatedly during development. For the rest of this section, we describe the three main components in the pipeline:

Field-based Call Graph Builder: The extractor takes a JavaScript file as input and parses the script, excluding files with syntax errors. The analysis then translates the script into the intermediate representation and builds an approximate call graph, called a *field-based call graph* [7]. A field-based call graph is a statement-level call graph that uses one abstraction for all instances of each property used in the program, as opposed to one abstraction for each property of each abstract object as in traditional call graphs; this has been shown to scale well for framework-based JavaScript web applications even in the presence of JavaScript's dynamic features [9]. In our implementation, we used the field-based call graph construction available in WALA [16]. For optimization, this call graph construction in WALA used to ignore all data flow that does not involve functions. To support our string analysis, we extended the data flow analysis in the call graph construction to also track data flow of strings in the program. We take all functions in the script as entry points for the call graph. Standard approaches take event handlers and top-level blocks as entry points. However, if we were to use the same approach, our analysis with the scope at the file-level could miss entry points if functions are registered as event handlers in a script beyond the analysis scope.

Web Request Locator: To identify API invocations, we look for framework-specific patterns in the call graph. For jQuery, we handle the most common patterns, i.e., function calls to `$.ajax`, `$.get`, and `$.post`. We note instructions that make such calls and use them as the seeds for the inter-procedural data flow analysis, the next component in the pipeline. When a script does not contain a matched invocation statement, our analysis does not produce any output and the pipeline terminates.

Backward Slicer for Request Marshaling: In this step, we extract the statements that contribute to the input of the web API invocations. Starting from each request function call captured in the previous step, we apply standard inter-procedural backward slicing [17] to narrow down the subset of statements of the program that affect the statement containing the request. In our implementation, we used the WALA backward slicer [16]. To get the actual strings pertaining to the URL and other parameters in a request, we recover all possible data flows that lead to the request. String values that cannot be determined until runtime are represented by symbolic values (e.g., the value of `searchHashtag` in Figure 3 is retrieved from the front-end in line 5). For strings with symbolic values, we model common string operators (i.e., concatenations and `encodeURI`). For constant strings, we model additional string operators including `substring`, `replace`, and `indexOf`. Currently, research on robust and scalable modeling of string operators with symbolic values is still ongoing [18]–[21]. However, we believe these cases are not significant in the web API usage extraction. This assumption is also supported by the observations in the experiment: we only found two cases where this limitation led the checker to incorrectly flag a string URL as a mismatch to the specification (Section V-B).

We assume all execution paths leading to the request are feasible and we thus perform path-insensitive data flow analysis. It is possible that a variable has multiple definitions from different paths. For example, Figure 6 shows two common patterns where multiple URLs can be extracted from a request. In Figure 6a, variable `query` in line 7 can have different values depending on the predicate in line 2. In Figure 6b, function `changeDisplayStuffs` can be invoked with different parameter values in line 8 and 9. For such cases, we take the union of all possible values. Finally, we output the analysis result in JSON format. The analysis output for the example shown in Figure 3 is presented in Figure 5 as an example. The extracted data contains the retrieved URL and HTTP method as well as all other properties passed to the `$.ajax` function.

IV. Checking Procedure

The goal of the checking procedure is to match the information produced by the static analysis against formal web API specifications. The procedure reveals inconsistencies between the request implementation and the specification. In general, the information about each request consists of (1) one URL of the web API to invoke, including the path identifying the endpoint and possibly a query string, (2) the HTTP method, and when required (3) data being sent in the payload body

```javascript
var query = "track:" + ... + "artist" + ...;
if (query.split("#").length > 1) { // forbidden character
  query = query.split("#")[0] + query.split("#")[1];
}

$.ajax({
  url: "http://api.spotify.com/v1/search?q=" + query + "&type=track",
  ...
});
```

(a) Multiple paths; Source: https://github.com/CasualKyle/CasualKyle.github.io/blob/master/script.js

```javascript
function changeDisplayStuffs(trackID) {
  $.ajax({
    url: "https://api.spotify.com/v1/tracks/" + trackID,
    ...
  });
}

changeDisplayStuffs("1HbcclMpw0q2WDWpdGCKdS");
changeDisplayStuffs("79qlNzUpZzZeXLior0zdtz");
```

(b) Multiple callers; Source: https://github.com/soojee/2016-Hack Rice-DemocraticDJ/blob/master/Song%20Viewer/js/create_search.js

Fig. 6: Code excerpts exemplifying the extraction of multiple URLs for one request.

of the request. Due to the nature of the analysis, though, in practice multiple URL values, HTTP methods, and payload data can be retrieved for a single request because the static analysis considers all possible execution paths to the invocation (as mentioned previously in Section III). We designed the checking procedure to compare any possible combinations of these data points against a web API specification. If any combination matches a specification, no error is reported so as to not annoy users with false positives.

A. Checking endpoints

The first part of the procedure aims to match a request to an endpoint defined in an API specification. The procedure starts by checking whether any of the URLs for a given request begins with any of the base URLs of the known API specifications. If an API specification contains more than one base URL, for example as it defines schemes `HTTP` and `HTTPS`, all versions are checked. Furthermore, it is possible that multiple specifications are found to match to a request, for example because multiple specification versions exist for the same API or because the static analysis reports multiple URLs for the request. If no specification can be matched, the procedure reports an error, instructing users to check whether the base URL is defined correctly.

Next, the procedure attempts to match a request's URLs to paths defined in the API specifications. To match a path, the procedure takes every URL of a request and compares it against the path definitions in every specification previously matched to that request. Every path definition of the request's URL strings is retrieved by truncating the base URL defined in the specification and the query string, if it exists. The remaining path strings are then compared against the path definitions in the specifications by checking whether every path component (separated by `"/"`) matches. This matching considers that both, the path strings from the URLs and the Swagger path definitions, may contain variables denoted by curly brackets. The procedure treats these path components as wild cards. Multiple path definitions may be matched to a single request, because multiple specifications can be matched and because a request may contain multiple URLs.

Finally, the procedure determines if the HTTP method matches the specification. If the static analysis does not report an HTTP method, the method is assumed to be `GET`. Any method determined by the static analysis is checked against all methods defined in all matched specifications and all matched paths.

B. Checking request data

For requests that can be matched to an endpoint definition in an API specification, the procedure additionally checks the validity of request data (if it exists). Request data is either the data sent in the payload body of a request (typically of `POST`, `PUT`, or `PATCH` requests) and the data sent within a query string.

1) Checking payload data: Data sent in the payload body of an HTTP request can be in any format. As the static analysis focuses on JavaScript, and because its the prevalent data format in web APIs [22], we focus on data in the JavaScript Object Notation (JSON). Swagger specifications allow the expected payload data to be defined, either for certain paths (across all methods) or for specific endpoints (an endpoint-level definition overrules a path-level one). Payload data definitions can be specified in place, or by referencing definitions in the central `definitions` section of the specification. The procedure considers all these ways to define payloads and, if needed, resolves conflicts of definitions on different levels. If any of the matched specifications defines a payload schema in any of the matched endpoints, the procedures determines if the payload data reported in the request information adheres to that schema or not. A possible violation is that a property marked as `required` in the schema is not present in the data.

2) Checking query parameters: The query data is encoded in key-value pairs. Within API specifications, query parameters can be defined as either optional or required. The checking procedure, then, can determine whether all required query parameters are present in a request. Again, the procedure considers definitions of query parameters from different locations in a specification and to resolve possible conflicts between definitions on different levels. To check the query parameters, the procedure parses the query strings of all URLs reported for a request. It then checks whether any of the found parameter sets matches the parameter definitions found in any of the endpoint definitions matched for the request.

V. Evaluation

To evaluate the web API request checker described in Sections III and IV, we applied it to the problem of identifying and checking whether JavaScript web APIs requests are consistent or inconsistent with a API specifications. The input of the checker is a set of JavaScript code mined from GitHub as well as a set of Swagger specifications (Section V-A).

For the evaluation, we are interested in two research questions:

- **RQ1**: Given JavaScript code describing a web API request, to what degree can the analysis correctly determine whether the request is consistent with an endpoint in given Swagger specifications? (See Section V-B)
- **RQ2**: For a request consistent with an endpoint in the Swagger specifications, to what degree can the analysis correctly determine whether the request data (the payload and the query parameters) is consistent with specifications? (See Section V-C and Section V-D)

For both of these questions, we first determined *quantitatively* how consistent were the information extracted from the static analysis compared to Swagger specifications. To obtain this quantitative information, we count positive matches as well as errors reported from our checking procedure.

For the set of requests that does *not* match endpoints or request data requirements of a specification, we determined if each of the inconsistent instances was legitimate or due to deficiencies in the approach. To ascertain the cause of each of these inconsistencies, we performed a *qualitative* analysis. For example, the analysis determined that the endpoint `https://api.instagram.com/v1/subscriptions` is inconsistent with the Swagger specification of the Instagram API because the specification does not contain a `/subscriptions` path. The qualitative analysis determined that this particular path was deprecated from Instagram's API. With the result of the qualitative analysis, we tabulated the number of instances where the approach correctly identifies endpoints and request data as consistent or inconsistent.

A. Data Collection

The evaluation requires two types of input data: First, we obtained web API specifications to compare against information produced by the static analysis using the checking procedure. Second, we mined JavaScript source code that (likely) contains requests to the APIs for which we have specifications.

1) API specifications from APIs Guru: For the Swagger specifications, we made use of a community-maintained collection of specifications from the APIs Guru repository [12]. The repository contains specifications either provided by API providers or third-parties, or generated using dedicated scripts. At the time of performing the experiments, we collected 260 specifications, which pertain to 230 APIs (with some APIs having specifications for multiple versions). These specifications act as a source of ground truth to indicate whether requests in the source code invoke the API correctly. We discuss threats to this ground truth in Section VII.

2) JavaScript files from GitHub: To increase the generalizability of the results, we aimed to collect a large set of JavaScript files containing web API calls. We obtained such a set by querying GitHub using its search capabilities.[4] Each search query targets the domain name of an available Swagger specification from APIs Guru and JavaScript instances that send requests using the jQuery function `$.ajax()` (though we handle additional ways to make requests, i.e., `$.post` and `$get`). We thus used the search queries of the form `extension:js "$.ajax" "{domainName}"`. To automate the data collection, we used Selenium [23] to invoke the GitHub code search and crawled the search result to obtain links to JavaScript files. The files returned by the search may not necessarily contain requests to the target domain. For example, the domain name may be in a comment and the script still matches the search criteria. From these queries, we obtained 6746 JavaScript files, from which we extracted 19668 web API requests.

[4]https://github.com/search

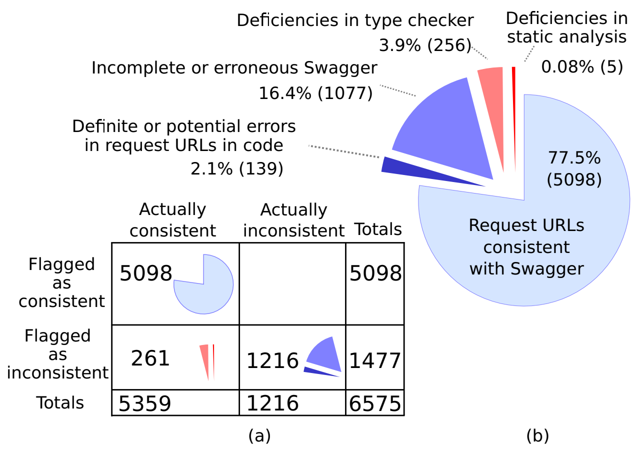
For this evaluation, we focused on the 6575 requests that matched a specification (i.e., a request URL matching the base URL of a specification, including matching the schemes, domain, and the basepath). We removed the remaining 13093 requests from our evaluation, with 9915 of the requests were safe to remove: These requests neither contained web API calls (4926)[5] nor were the requests the GitHub search intended to target (i.e., 4989 requests that did not match the domain of any of the evaluated APIs). The remaining 3178 requests corresponded to URLs that could not be resolved by our static analysis, e.g., containing symbolic values in the base URL. Some of the symbolic values were global variables that could have been resolved if the analysis scope were expanded beyond the file-level, while other symbolic values were not typically possible to be resolved by static analysis (e.g., values provided to the application at run-time by a user or a configuration file). The strength of our approach is that our analysis does not raise a false alarm on these 3178 cases.

B. Endpoint Results

For RQ1, the goal was to determine what percentage of the request endpoint URLs extracted from code was correctly flagged as consistent or inconsistent with the Swagger specifications, i.e., the *precision* of the approach. The JavaScript files obtained from GitHub as described in Section V-A contained 6575 requests in which the endpoint of a URL matched one of the Swagger specifications, i.e., matches the base URL, including matching the defined schemes (`http` or `https`), domain, and the basepath.

Overall, we found the precision of matching the endpoints of requests to be 96.0%, which was tabulated from requests that were flagged consistent and were actually consistent (◔ - 5098 requests in the upper left cell in Figure 7a), and requests that were flagged inconsistent and were actually inconsistent (◭ - 1216 requests in Figure 7a). In addition to matching the base URL in a Swagger specification, a request is a valid endpoint when it satisfies two conditions: (1) the URL matches an endpoint path, which can contain path variables (e.g., `/repos/{owner}/{repo}`) and (2) the HTTP method matches (e.g., `GET`).

[5]Of the 4926 requests, 4177 requests contained URLs targeting internal endpoints (including relative URLs, localhosts, IP addresses), not web APIs; 567 requests only contained empty strings, null values, or numeric values; and 182 requests contained URLs matching the domain but not the base path, likely to be targeting static web pages or data.



	Actually consistent	Actually inconsistent	Totals
Flagged as consistent	5098		5098
Flagged as inconsistent	261	1216	1477
Totals	5359	1216	6575

Fig. 7: Distribution of the 6575 extracted endpoint invocations

For the 1477 requests that did not match to a valid endpoint (and , the "Flagged as inconsistent" row in Figure 7a), we qualitatively determined if the static analysis checker correctly identified a legitimate inconsistency or not. We found that a significant fraction of these requests were true negatives due to errors in the actual URLs from the JavaScript code (139, 2.11%) and incomplete, missing, or erroneous Swagger specifications (1077, 16.4%), together pertaining to 1216 requests (in Figure 7a). The remaining 261 requests were false positives (in Figure 7a) due to deficiencies in the static analysis (5, 0.08%) and in the checking procedure (256, 3.9%). For the rest of Section V-B we present the qualitative analysis of the results on these true negatives and false positives.

Endpoint Results: True Negatives -

Requests that were correctly flagged as inconsistent fall into two categories: definite or potential errors in the code, and erroneous and incomplete specifications.

For the first category, we found that 139 requests contained an erroneous URL from the JavaScript code that we correctly flagged as inconsistent, accounting for 2.1% of the 6575 requests (Figure 7b). Of the 139 requests, 23 were attributed to deprecated calls and programming errors that were definitely erroneous:

- **Deprecated APIs**: 16 requests were inconsistent because a URL corresponded to a call to an API that were deprecated entirely, or calls to subset of an API that happened to be deprecated. For example, the request to `https://www.googleapis.com/freebase/v1/text/en/bob_dylan` was to the Google's Freebase Internet Marketing API, which has been deprecated since June 30, 2015.[6]
- **Documented Programming Errors**: In two cases, we found evidence (e.g., in the form of a question posted on a forum) that the URLs in the code were erroneously constructed because of errors in interpreting API documentation.[7]
- **Typographical errors**: In five cases, the requests contained obvious typographical errors. For example, `https://api.spotify.com/v1/seach` did not match the endpoint `search` in the Swagger specification of the Spotify API because of the typographical error in "seach". In another case, the checker reported that the extracted URL string `'https://api.spotify.com/v1/users/'+userID+/playlists/+playlistID+'/tracks'` (a string that looked like a syntax error because of the absence of quotes surrounding `/playlists/`) did not match any endpoints in the Spotify API, even though there was an endpoint `http://api.spotify.com/v1/users/{userId}/playlists/{playlistId}/tracks` in the Spotify's Swagger specification. One could argue that the checker flagging this case as a mismatch was a mistake, because at runtime, `/playlists/` without the quotes actually evaluates to the string `'/playlists/'` as `/playlists/` is interpreted as a regular expression. However, we argue that marking this case as a mismatch is legitimate because it is likely that the author of the code intended to include the quotes `'/playlist/'` but this potential error was not caught by the JavaScript interpreter nor testing.

In addition, we found 116 requests with potential errors:

- **HTTP method**: In 112 cases, the URL was using the wrong HTTP method, e.g., using GET instead of POST as specified in a Swagger specification. Of these cases, 74 were GET requests to `https://www.googleapis.com/oauth2/v1/tokeninfo` that should be POST according to the Swagger specification and online API documentation, even though the server accepts the GET call. We categorized these cases as potential errors because even when an API provider may accept such calls, it is still worth-while to warn a programmer that the call is not consistent with the definition of the API.
- **Port number**: We observed four requests in which a URL contained port numbers. These cases may be problematic because port numbers are seldom in any publicly advertised base URLs. For the four cases involving port numbers, one could argue that it was worth issuing a warning to a programmer because port numbers are unlikely to be in a legitimate base URL. With the static checker that could flag that a URL in the requests were consistent with a basepath but not any of an endpoint, a programmer authoring code containing these 155 cases could potentially made aware of these bugs.

For the second category of the true negatives, we found

[6]https://developers.google.com/freebase/v1/topic-overview

[7]http://stackoverflow.com/questions/11606101/how-to-get-user-email-from-google-plus-oauth

that 1077 requests (or 16.4%; Figure 7b) corresponded to invocations that were not matched with any endpoint of the corresponding Swagger specifications. From a manual inspection on the documentation on the API being invoked in the requests, we found that the mismatch was due to erroneous and incomplete Swagger specifications:

- **Errors in Swagger specifications**: We determined that 867 cases due to errors in the Swagger specifications. These erroneous requests were because a base path had been refactored to include a version number (e.g., the `1.0` path segment in `https://mandrillapp.com/api/1.0/messages/send.json`) but the Swagger specification was not updated.
- **Missing endpoints in Swagger specifications**: For 210 requests in which a URL found by the static analysis and verified by us as valid calls, a corresponding endpoint definition with the assumed path was missing from the Swagger specifications. These requests corresponded to 18 endpoints across four APIs: Reactome, Slack, Trello, and Google APIs.
- **Missing authorization URLs in Swagger specifications**: Eight URLs relate to endpoints for authentication purposes, e.g., `https://slack.com/api/oauth.access` and `https://trello.com/1/appKey/generate`. A complete Swagger specification should include such necessary authentication URLs. However, in these 8 cases those authentication URLs were missing in the specifications.

As it turned out, the 1077 requests flagged by the checker as inconsistent with the Swagger specification were actual errors in the Swagger specification. Upon informing APIs Guru of these errors, we learned that these specifications were already corrected.[8] This scenario demonstrates a potential use case for using the checker on a large repository of API usage for identifying missing or erroneous information on a given set of Swagger specifications.

Endpoint Results: False Positives -

There were two sources of mistakingly flagging a request to be a mismatch: deficiencies in the static analysis and deficiencies in the checking procedure. For the first source of errors, we found that only five out of the 1496 cases were due limitations in the static analysis – which is surprising, given the challenges in analyzing JavaScript and that the analysis scope was within a file. There were two types of deficiencies:

- **Limitation of the analysis scope**: In three cases, because the analysis scope was within a file, the analysis failed to construct a valid URL because the code contains variables or function calls defined outside the file.
- **Handling string library functions**: In two cases, the requests used the library function `split`. In the static analysis, we explicitly model other more common string operations, i.e., string concatenations and `encodeURI`, as we described in Section III. Currently, handling a more complete set of string library functions with symbolic values is an active research topic [18]–[21]. We could leverage such research and model more string operators in the future.

The few deficiencies caused by the static analysis show that our technique and the chosen analysis scope are feasible for the problem of extracting requests.

Due to deficiencies of the checking procedure, our approach mistakenly determined that 256 requests were inconsistent with Swagger specifications. There are two main causes of these mistakes:

- **Conservative Matching**: We designed to checker be confident in flagging requests as "consistent with the Swagger", at the risk of flagging legitimate URLs as potentially inconsistent. In consequence, we found 251 requests that the procedure mistakenly flagged as inconsistent with any of the Swagger specifications. For example, the code `page.config.baseUrl+'tags/'+term+'media/recent?client_id='+page.config.clientId` should have matched the endpoint `tags/{tag}/media/recent`. However, the checking procedure requires that the variable is constrained to a single path segment, i.e., characters without a '/' in it.
- **Missing authorization URLs**: In five cases, the requests were for authorization. These URLs are defined in a Swagger file but the checker does not currently check for such URLs.

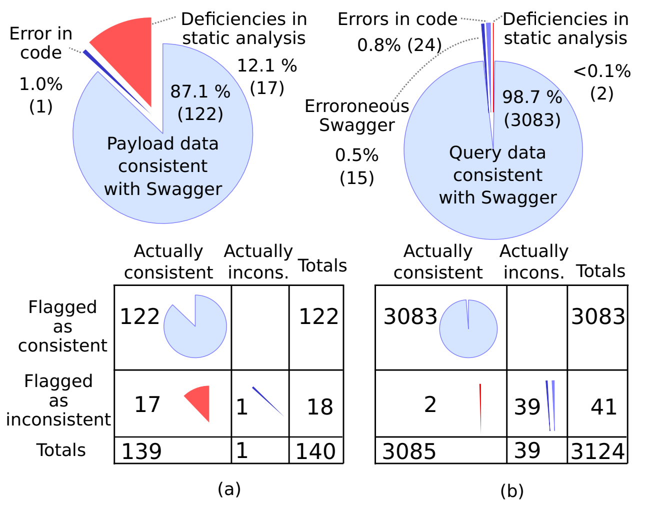
Overall, using the checker for determining whether an invocation of an web API endpoint corresponds to a valid one in a Swagger specification yielded a promising result, with 96.0% of the endpoint invocations correctly flagged as consistent or inconsistent with the Swagger specifications.

C. Payload Data Results

For RQ2, we focused on how well the analysis correctly determines whether the request data is consistent with web API specifications. We present the results for the payload data in this Section V-C and query data in Section V-D.

To assess the approach's ability to check for correct payload data, we first determine how many of the 5098 requests for which we can match an endpoint have a payload schema definition in any of the corresponding API specifications. We only consider requests for which the payload definition is mandatory, i.e., if the request matches multiple specifications or multiple endpoints as described in Section IV, they all need to denote a payload definition. We found that overall 140 requests have a mandatory payload definition (see Figure 8a). Out of these 140 cases, we found that 122 requests (◔) contained data extracted from the static analysis that adheres to

[8]http://www.apiful.io/intro/2016/05/16/challenges-in-maintaining-specs.html



	Actually consistent	Actually incons.	Totals
Flagged as consistent	122		122
Flagged as inconsistent	17	1	18
Totals	139	1	140

	Actually consistent	Actually incons.	Totals
Flagged as consistent	3083		3083
Flagged as inconsistent	2	39	41
Totals	3085	39	3124

Fig. 8: RQ2 results for payload (a) and query data (b)

the required payload. Of the 18 payloads that did not adhere to any of the specifications, we qualitatively ascertained whether the approach correctly determined the mismatches (i.e., true negatives which correspond to ╲ in Figure 8a) or not (i.e., false positives which correspond to ◥). Tabulating the matched cases (◯) with the true negatives (╲) yields a precision of 87.9%.

Payload Data Results: True Negatives (╲)

The single true negative case is caused by an error in the code: While the analysis reports a required data property to be sent in a query parameter, the specification requires it to be sent in the payload body.

Payload Data Results: False Positives (◥)

The 17 false positive cases are explained by deficiencies in the static analysis. In 13 cases, the static analysis does actually report data that, upon manual inspection, does match with the data schema in the specification. However, in these cases, the data is present as a JSON-encoded string in the request source code (rather than a JSON object), so our checking procedure fails to correctly match it. In four other cases, the analysis reports a variable to be replaced by a global JSON string, which cannot be resolved statically.

D. Query Parameter Results

Regarding the query parameters, we first determined how many of the 5098 requests for which we could match an endpoint have query parameters defined in the API specifications. Like in the case of the payload data, we only considered requests that match an endpoint that has a mandatory query parameter definition, of which we found 3124. Figure 8b presents the breakdown of these requests. We found that 3083 of the requests (◯) complied with their corresponding query parameter definitions. We qualitatively analyzed the 41 cases in which our approach reported a mismatch, of which 39 were true negatives (‖) and 2 were false positives (|). The precision is 99.9%, taking into account the matches (◯) and the true negatives (‖).

Query Parameter Results: True Negatives - (‖)

The 39 cases in which our approach correctly detects an inconsistency between source code and specification fall into two categories:

- **Errors in the specification**: 15 cases concern `GET` requests to the `../repos/{owner}/repo/contributors` path of the GitHub API. According to the API specification, a query parameter `anon` is required to indicate whether to list anonymous contributors. Invoking this endpoint test-wise reveals that requests also succeed without that parameter in the request, pointing to an error in the specification.
- **Errors in the code**: We find 24 cases where required data is sent in the wrong place. In 11 cases, `GET` requests to the `../artists/{id}/top-tracks` path of the Spotify API miss a `country` query parameter, which is required according to the API's specification and online documentation. Interestingly, though, all 11 requests send a `country` data property in their payload body. Similar cases can be observed for five `GET` requests to the `../artists` path and two `GET` requests to the `../me/following/contains` and `../me/tracks/contains` paths, where, again, required query parameters are sent in the payload body instead. Similarly, in four `POST` requests to the Instagram API, one `GET` request to the Slack API, and one `GET` request to the Buffer API, query parameters, which are required based on the specification, are sent in the payload body instead.

Query Parameter Results - False Positives (|)

We observe two cases where the static analysis misses to report an inconsistency because the extracted URLs end with a variable. The variable spans across parts of the endpoint path and possibly the query parameters. Thus, in this cases, the analysis fails to report required query parameters.

Overall, out of the 39 reported query parameter mismatches, 15 cases are explained by errors in the specification - required parameters are actually not necessary for a successful request. In 24 cases, query parameters that are required according to the specification are sent in the payload body of the request instead, which is, in most of these cases, in conflict with both, the API specification and online documentation. Only in two cases do we find mismatches due to our approach's failure to report needed information.

VI. Related Work

Our work on checking JavaScript code with respect to web API specifications is similar in spirit to a range of work on checking and bug finding approaches, such as TypeScript [24] and JSHint [25], respectively. Due to the dynamic nature of JavaScript, and the extreme difficulty of providing precise analysis, such languages and tools tend to be lenient; rather than attempting to be complete, they work by partially enforcing type rules or using a set of patterns that can be tuned to provide some level of feedback without overwhelming their users with a large number of false reports. Our work shares that approach: our analysis is biased to only report issues that are fairly likely to be real. However, our approach differs from other bug finding tools for JavaScript by being based on inter-procedural static analysis, and relying on traditional techniques such as inter-procedural slicing and string analysis.

Halfond et al. introduce static analysis techniques for understanding web API usage in Java applications [26], [27]. One technique focuses on how APIs are used in HTML code that is dynamically generated as part of a Java web application [26]. The work focuses on first approximating the HTML and then extracting invocations from it. This approach works well when the logic creating the request is on the server side; it does not target API calls generated in JavaScript on the client side, which is our focus. Other work introduces symbolic execution to improve results, once again focusing on Java web application [27].

In the context of JavaScript, related work has shown that understanding API specifications can make dynamic testing more effective [28]. In contrast to this work, we focus on static analysis rather than on dynamic testing. Our work also relates to checking JavaScript function calls. SAFE$_{\text{WAPI}}$ [29] analyzes JavaScript and checks function calls against the web IDL specifications of those functions. SAFE$_{\text{Wapp}}$ [30] models the web application execution environment (e.g., DOM) and checks function invocations against ECMAScript rules. In comparison, we check web API requests, which is not a language construct in JavaScript. In addition, resolving the targets (endpoints) and parameters requires string analysis.

Multiple works have been proposed to check web APIs for compliance with best-practices [22], [31]. These works currently take as input observed web API requests [22] or human-readable documentations [31], but could be adapted to work on top of API specifications. In contrast to checking specifications, we here propose to check the adherence of source code with given specifications.

VII. Threats to Validity

By considering the URL, HTTP method, and request data, our approach covers important parts that determine whether a request contains any errors or not. However, our approach currently does not examine request headers, which can, for example, contain authentication information that affects the validity of a request. Despite this limitation, our approach is still a valuable first step towards a wider coverage of errors on web API requests.

A threat to validity of our evaluation is in the way we retrieved source code from GitHub (see Section V-A). Our data collection relies on the code search facilities provided by GitHub.[9] GitHub only provides limited insights into the search algorithm, for example, that characters like ".", "/", or "\" will be ignored and that only small repositories (less than 384 KB and less than 500,000 files) are indexed.[10]

The API specifications we used in our experiments may contain errors, similar to any other type of documentation. However, we consider APIs Guru to be one of the most reliable sources of API specifications. They have a policy in place to validate specifications that are not updated within 48 hours. In addition, APIs Guru reports the origin of specifications (which typically stem from API providers or the developer community).

Finally, a threat to the validity of our experiments is that we manually analyze the errors reported by our checking method. We performed this analysis to shine a light on the sources of errors and took care to cross-validate among multiple sources as much as possible.

VIII. Conclusion

In this paper, we have leveraged existing research in static analysis scalable to framework-based JavaScript web applications and created an analysis capable of extracting strings pertaining to web APIs requests. We used these extracted request data as input to a checker that determines whether the requests are consistent or inconsistent with formal web API specifications. A qualitative analysis of the results from our checker on 6575 requests shows that most of reported inconsistencies were due to errors in the client code (calls to deprecated APIs, errors in the URLs, errors in data payload definitions) and incomplete Swagger specifications, as opposed false positives. Quantitatively, we found that the approach can correctly determine whether a request is consistent or inconsistent with web API specification with a high precision of 96.0% for endpoint checking, 87.9% for payload data checking, and 99.9% for query parameter checking.

These results point to the promise in creating tools that are capable of warning programmers of source code containing inconsistent web API requests that can be potentially erroneous. As such, this approach can be integrated with existing tools that support developers in using web APIs [1]. Our proposed checker can also be employed with continuous integration for checking the validity of web API usage in case a web API undergoes changes. Furthermore, our work can lead to tools to help API providers to monitor usages of their APIs in publicly available code or integrate with third-parties change monitoring sites such as API ChangeLog.[11] As for future work on the checker itself, we aim to extend the scope of our static checking method to consider additional aspects of web API requests like header information, HTTP response codes, or the structure of returned data. In addition, based on the positive results with jQuery, we want to extend our implementation to handle other web frameworks.

[9]https://github.com/search
[10]https://help.github.com/articles/searching-code/
[11]https://www.apichangelog.com/

References

[1] IBM API Harmony. https://apiharmony-open.mybluemix.net/.

[2] E. Wittern, V. Muthusamy, J. A. Laredo, M. Vukovic, A. Slominski, S. Rajagopalan, H. Jamjoom, and A. Natarajan, "API Harmony: Graph-based search and selection of APIs in the cloud," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 12:1–12:11, March 2016.

[3] PublicAPIs — Directory of public APIs for web and mobile. https://www.publicapis.com/.

[4] ProgrammableWeb - APIs, Mashups and the Web as Platform. http://www.programmableweb.com/.

[5] T. Espinha, A. Zaidman, and H.-G. Gross, "Web API Fragility: How Robust is Your Mobile Application?" in *Proceedings of the IEEE MOBILESoft*, 2015, pp. 12–21.

[6] Open API Initiative. https://openapis.org/specification.

[7] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient Construction of Approximate Call Graphs for JavaScript IDE Services," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 752–761.

[8] Usage Statistics and Market Share of JavaScript Libraries for Websites, August 2016. https://w3techs.com/technologies/overview/javascript_library/all/.

[9] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 165–174.

[10] E. Andreasen and A. Møller, "Determinacy in static analysis for jquery," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 17–31.

[11] Y. Ko, H. Lee, J. Dolby, and S. Ryu, "Practically tunable static analysis framework for large-scale JavaScript applications (T)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 541–551.

[12] APIs.guru - Wikipedia for Web APIs. https://apis.guru/.

[13] P. Suter and E. Wittern, "Inferring Web API Descriptions From Usage Data," in *Proceedings of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015.

[14] JSON Schema and Hyper-Schema. http://json-schema.org/.

[15] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering and the Foundations of Software Engineering*, 2013, pp. 488–498.

[16] The T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[17] Y. N. Srikant and P. Shankar, *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, Inc., 2007.

[18] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering and the Foundations of Software Engineering*, 2013, pp. 114–124.

[19] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions," in *Proceedings of the 26th International Conference on Computer Aided Verification*, 2014, pp. 646–662.

[20] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective search-space pruning for solvers of string equations, regular expressions and length constraints," in *Proceedings of the 27th International Conference on Computer Aided Verification*, 2015, pp. 235–254.

[21] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "Progressive reasoning over recursively-defined strings," in *Proceedings of the 28th International Conference on Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds., 2016, pp. 218–240.

[22] C. Rodrıguez, M. Baez, F. Daniel, F. Casati, and J. Carlos, "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," in *Proceedings of the International Conference on Web Engineering*, 2016.

[23] Selenium - Web Browser Automation. http://www.seleniumhq.org/.

[24] TypeScript - JavaScript that scales. https://www.typescriptlang.org/.

[25] JSHint, a JavaScript Code Quality Tool. http://jshint.com/.

[26] W. G. J. Halfond and A. Orso, "Automated identification of parameter mismatches in web applications," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 181–191.

[27] W. G. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 285–296.

[28] C. S. Jensen, A. Møller, and Z. Su, "Server Interface Descriptions for Automated Testing of JavaScript Web Applications," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 510–520.

[29] S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFEWAPI: Web API Misuse Detector for Web Applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 507–517.

[30] C. Park, S. Won, J. Jin, and S. Ryu, "Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 552–562.

[31] F. Palma, J. Gonzalez-Huerta, and N. Moha, "Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti) Patterns," in *Proceedings of the International Conference on Service-Oriented Computing*, 2015, pp. 171–187.