Mining API Type Specifications for JavaScript

Shuai Wang[1,2,3], Wensheng Dou[1], Chushu Gao[1,3*], Jun Wei[1,2,3], Tao Huang[1,2]
[1]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3]University of Chinese Academy of Sciences, Beijing, China
{wangshuai, wsdou, gaochushu, wj, tao}@otcaix.iscas.ac.cn

***Abstract*—API specifications play an important role in software development. However, API specifications are often not well documented, especially for JavaScript. Many JavaScript API specifications lack of precise type information for API parameters and return values. In this paper, we propose a static approach for mining JavaScript type specifications automatically. We gather the usage information of return values and parameters statically, and infer types of return values based their usages, by identifying a known type which they are used most likely to be, and infer parameters by identifying the most used parameters. We evaluate the approach on the homepages of Alexa top 1000 websites, the experimental results show that our approach can gain high precision. Our case study on jQuery shows that our approach gains high precision and reasonable recall on jQuery, and we can use our inferred API type specifications to detect 2 jQuery misusage errors in real-world web sites, and 1 missing type error in jQuery documentations.**

Keywords—JavaScript; API; Type Specification

I. INTRODUCTION

JavaScript is the de facto dominated language in developing web application clients, and is increasingly used in server-side applications, desktop applications, and mobile phone applications [1]. JavaScript APIs are widely used in reusing libraries (e.g., jQuery [2]) and interacting with runtime environments (e.g., DOM [3]). API specifications are useful for software development. Developers always need to consult the API specifications during their developing process.

However, API specifications are always not well documented [4], especially for JavaScript. As a weakly-typed language, JavaScript allows developers to define functions with only parameter names, and ignore their types. JavaScript API specifications are always created based on types annotated manually, which are error-prone and less precise for complex objects (e.g., JSDoc [5]), or by using examples to illustrate the usages of APIs (e.g., KnockoutJS [6]). However, precise type specifications are very useful, such as auto-completion suggestions in IDEs (e.g., Aptana [7]), sample code and tutorial recommendation [8][9], and automated error detection [10][11].

In this work, we focus on mining API type specifications for JavaScript. JavaScript is a weakly-typed language, which brings great challenges in mining type specifications. First, JavaScript has no declarative type definition, and it is impossible to get the type of the return value of an API directly. Second, type conversion is widely used in JavaScript, and any types can be convert into expected types in runtime. Third, JavaScript has no constraints on parameters of function calls, arbitrary number and types of values can be used as parameters to any function calls, the missing parameters are treated as *undefined*, the extra parameters can be retrieved via *arguments* variable, and type-unmatched parameters are converted to excepted types in runtime implicitly.

This work is enlightened by the following observations: 1) if a value is mostly used like some type, it should be that type, and 2) the majority of API usages are correct in real world. This work uses three steps to mine JavaScript API type specifications, including the types of return values and parameters. First, we statically analyze the source code of JavaScript to collect the usage information of constant values and APIs, including involved operations, accessed properties, applied as parameters, etc. Second, we build a probability model based on usage information of constant values. This model presents the probability of operands of each usage being each type, and then we calculate the most like type for API return values based on their usage information using this probability model. Finally, we build a parameter type tree for parameters resulting in the same return values of APIs, and infer acceptable parameter types for the corresponding return values by identifying the most used parameters as true positives. The inferred types of return values and parameters make up the final API type specification.

We evaluate our approach on the homepages of Alexa [12] top 1000 websites. The evaluation shows that our approach can gain high precision in inferring the types for API return values and parameters. Moreover, we conduct a case study on the most popular JavaScript library, jQuery [2], and the result shows that the precision of our approach on jQuery is still high, while the recall in inferring parameter types of jQuery APIs and properties of jQuery object are not that high, because a lot of jQuery APIs (48.3%) or parameter inputs (57.3%) are seldom or never been used. The study also detected 2 jQuery misuses in real-world web sites, and 1 missing type in jQuery API documentation, which illustrates the potential usages of our approach in API misuse detection and document verification.

This paper makes the following main contributions:

- We leverage a probability model to infer types for API return values, and identify object types by merging similar objects based on a novel property-based similarity algorithm.

* Corresponding author

- We introduce a novel parameter type tree to summarize the occurrence for each parameter, and leverage a statistical method to infer possible API parameters.
- We perform a systematic experiment on Alexa top 1000 websites, and the result shows that our approach can gain high precision.
- We conduct a case study on the most popular JavaScript library, jQuery, and the result shows that our approach can gain high precision and reasonable recall. We can further detect API misuses and verify API documentations.

The rest of this paper is organized as follows. In Section II, we illustrate the challenges by a real-world example. In Section III, we describe our approach. We show how we have evaluated the approach in Section IV, discussion of our work in Section V, and related work in Section VI. Finally we give a conclusion in Section VII.

II. Example and Challenges

In this section, we illustrate the features of JavaScript and challenges of our work through a real-world example written in jQuery.

A. Example

JavaScript is widely used in adjusting presentation of web pages. List 1 shows an example, this code segment provides a preview of modifying font color and size in a web page. The functionality is triggered via two kinds of mouse events, *mouseover* and *mouseout*. When a user wants to see the effect of changing the font color, he/she should move the mouse to over the target color (an element with class named *colorPicker*), then the target color is retrieved and set to the content (Lines 12-13); when the mouse moves leaving the target, the color of content is restored (Line 15). The modification of font size is done in the same way (Lines 21-22 and 25).

The example illustrates some features of JavaScript.

1) Un-typed Variable Definition

JavaScript is a weakly-typed language, and variables are not defined with explicit type declaration, but with just a general keyword *var* instead (e.g., the variable *currentColor* defined in Line 10). Variables defined in this manner can be used to hold the values of any types. This fact makes it's hard to get the type of each variable from the source code directly.

2) Implicit Type Conversion

Due to lack of type definition, it's impossible to perform strict type checking in compiling time for JavaScript. As a result, in JavaScript programs one operator may encounter values of arbitrary types. In order to avoid throwing runtime type mismatch exceptions as much as possible, JavaScript converts values of unexpected types into that of expected types in runtime implicitly. For example, in line 3, the variable *value* is used as the expression of *if* statement, which is expected as a *Boolean* value. According to its context, we can know that the type of *value* in line 3 should be *String* or *undefined*, which is converted into *Boolean* in runtime. An empirical study shows that implicit type conversion is heavily used in real-world JavaScript applications [13].

```
1  function setContentStyle(name, value) {
2    var content = $('.content');
3    if (value) {
4      content.css(name, value);
5    } else {
6      content.css(name);
7    }
8  }

9  function init() {
10   var currentColor = 'black';
11   $('.colorPicker').mouseover(function() {
12     var color = $(this).css('color');
13     setContentStyle('color', color);
14   }).mouseout(function() {
15     setContentStyle('color', currentColor);
16   });
17   var currentSize = {
18     'font-size': '12px'
19   };
20   $('.sizePicker').mouseover(function() {
21       var size = $(this).css('font-size');
22       setContentStyle('font-size', size);
23   });
24   $('.sizePicker').mouseout(function() {
25       setContentStyle(currentSize);
26   });
27 }

28 $(init);
```

List 1. JavaScript Source Code of the Example

3) Arbitrary Types and Number of Parameters

JavaScript allows passing arbitrary types and numbers of parameters to functions. Various types of parameters may be processed in different ways so that the invoked functions can perform as polymorphic functions [14]. For example, as shown in List 1, the jQuery core function *$* can take String values (Lines 2, 11, 20, 24), object values (Lines 12, 21), and a function value (Line 28) as its parameters. Moreover, various number of parameters can be used to simulate polymorphic function. For example, the *setContentStyle* function defined in the example (Lines 1-8) can receive only one object parameter or two String parameters.

4) Multiple Types of Return Values

JavaScript has no constraints on types of the return values of functions, developers are allowed to return any values of any types at any times. Such as the *css* function of jQuery it returns a String value presenting the font color setting of target element in line 12, while in line 4 it returns a jQuery object which can be used to invoke other APIs continuously, just like the codes in lines 11-16 do.

B. Challenges

The goal of our work is to mine type specifications for JavaScript APIs from source code. However, the lack of type constraints brings highly dynamic features to JavaScript, which brings great challenges to mine type specifications.

1) Type Inference

As mentioned above, there is no explicit type definition in JavaScript, so it's impossible to retrieve the types of return values of API calls directly. The usage information of these unknown values can be used to infer their types, however

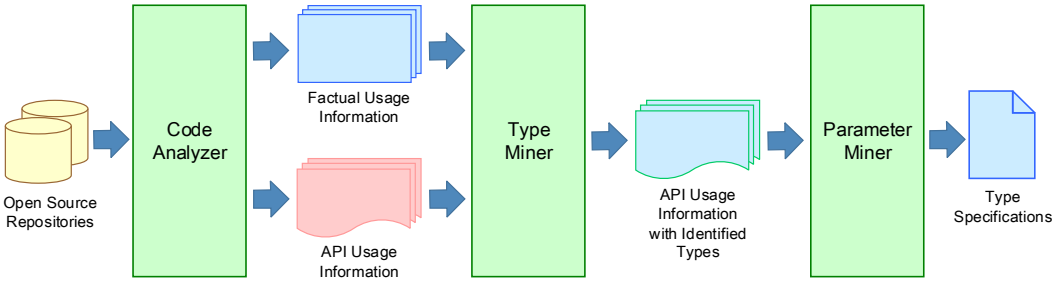


Fig. 1. Approach Overview

implicit type conversion is heavily used, which may bring a lot of confusing usage information. How to infer types of return values of APIs under the interference of implicit type conversion as precise as possible is the first challenge.

2) Object Unification

APIs in the same family always return objects of the same type, such as *mouseover* and *mouseout* in jQuery both return jQuery objects. So objects returned by different APIs but ought to be the same one must be identified and merged, this is called *object unification*. One potential method is to identify similar objects based on their properties. However, there are some property names widely used in different libraries, and properties accessed on return values of APIs invoked in longer path are always fewer than that in shorter path. How to design a similarity algorithm is the second challenge.

III. APPROACH

In this section we provide an detailed description of our statistical approach on the example described in section II.

A. Overview

Our approach aims to mine JavaScript API type specifications, which consist of API return value types and parameter types. If the type of a return value or a parameter is Object, its properties are included as well.

Our idea is to infer types of API return values based on their usages, if a return value is used mostly like some known type, we identify the known type as its type. Furthermore, we infer types of parameters based on their occurrence frequency, the most occurred parameters are identified as true positives.

Our approach consists of three components: a code analyzer, a type miner, and a parameter miner.

The code analyzer statically analyzes the source code of JavaScript, and extracts two kinds of usage information: factual usage information and API usage information. Factual usage information refers to usages of values of inferable types, including explicit types, such as constant, and inferred types, such as results of expressions. The API usage information includes usage information of their return values and corresponding parameters, which are used to infer types of return values and parameters respectively.

The type miner infers possible types for API return values, and built-in variables used in API calls, such as *document* used in *$(document)*. We firstly build a probability model based on the factual usage information, which illustrates the probability of known types used in each operation. Here we assume that API calls receiving the same types of parameters always return values of the same type. Then we summarize the usage information of the same API calls, and infer its return value type by selecting a known type which it is used most likely. The types of built-in variables are inferred in the same way.

The parameter miner infers possible types for API parameters. Here we assume that parameters resulting in same type of return values trigger the same functionality of an API. Then we group parameters resulting in the same types of return values together, and identify the most frequently occurred parameters for each group as true positives.

The overview of our approach is shown in Fig. 1. We describe the three components in detail below.

B. The Code Analyzer

The code analyzer performs an abstract interpretation [15] on input source code based on abstract of types. We abstract all the values into seven types, and trace the usages of them in totally nine kinds of operations.

Because JavaScript source code used in a single web page is always very large, and libraries which are hard to analyze statically are widely used, it's a challenge to analyze the whole web page as a single application at one time [24]. To facilitate the analyze process, we analyze JavaScript source code within each code file individually, which greatly brings down the scale of code to analyze for one time. This method also brings us another advantage, it supplies us an easy way to identify APIs, i.e., functions that are not defined in the same source file with analyzed code can be identified as potential APIs.

1) Types

We aim to leverage the least background knowledge of JavaScript to infer types as much as possible, so we decide to trace only seven basic types of JavaScript, including: *String*, *Number*, *Boolean*, *Object*, *Function*, *null* and *undefined*, and none of their properties are modeled. Other built-in types, such as Array, Date, etc. are classified as Object. Note that we define *null* as an individual type, because it acts differently from *Object* and *undefined*.

2) Operations

Operations refer to statements that values may be used in. The code analyzer traces all kinds of operations defined in JavaScript. We divide the operations into nine categories, according to their semantics and expected types of operands. For example, a category named *addition*, contains only one operator

"+", who has two semantics in JavaScript, *String concatenation* and *numeric addition*, which expect *String* and *Number* types respective. The details of all operations are shown in Table I.

3) Factual Usage Information

Factual usage information records all kinds of *scenarios* where values of inferable types are used. Here, the term *scenarios* refers to accesses on some operations with some operands, including operations with certain types, accesses on certain properties, application as parameters to certain APIs, etc.

The code analyzer abstracts values to their types. For primitive values, we record their types instead of their values; for complex values (e.g. object), we record their properties as well; for expressions, we record the types of evaluated results according on the JavaScript specification [20] (e.g. the expression *x * y* always results in a Number value). For example, we use keyword *"String"* (abbr. *"S"*) to present the 11 String constant values (i.e. *".content"*, *"black"*, etc.).

The factual usage information consists of a list of counters, each counter presents the number of occurrences that one type is involved into the corresponding scenarios. For example, in List 1, *String* values are used as the only parameter of API call *$* for 7 times, so the counter named *"<$, 1, 1>"* for String is 7, where the 3-tuple *<f, i, n>* means the *ith* parameter of function call to *f* with total *n* parameters, as shown in Fig. 2 (a).

The code analyzer collects the factual usage information in the following steps. Initially, the code analyzer creates empty usage counters for each type, i.e. *"String"*, *"Number"*, etc. Then the analyzer monitors the usages of all type-inferable values. When the analyzer encounters a statement listed in Table I and all the operands are type-inferable, then the analyzer increases the corresponding counter for each operand type by 1. For example, when the analyzer encounters a statement, a *String* added with a *Number*, such as *"this is " + 2*, then for the *String* type, its counter "*adding-with-Number*" is increased by 1.

4) API Usage Information

API usage information refers to information about how APIs are called, and how API return values are used, which is used to infer types of return values and parameters of APIs respectively. Each piece of API usage information presents one API call in the source code, which consists of three parts: name, parameter information, and usage information of the return value (if any).

The name is represented using its full call path along with the parameter types of preceding API calls, for example, the name of call to *css* in line 12 is *$(O).css*.

Parameter information refers to number, order, and types of parameters passed to the API call. We use the composition of name and parameter information as identifiers of API calls. Note that return values of API calls, or built-in variables, which are not type-inferable so far, may be used as parameters of other API calls, we use the their identifiers where their types are required. For example, parameter information of call to *css* in line 4 may be *(S, $(O).css(S))* because its second parameter may come from *$(O).css(S)*.

The usage information of the return value records all kinds of *scenarios* where it's used, similarly to factual usage information. For example, the return value of call to *css* (named *$(O).css(S)*) in line 12 is passed to *setContentStyle* as its second parameter, which is sequentially used as the expression of *if* in line 3 and the second parameter of call to css (named *$(S).css*) in line 4, so the counter named *"if"* is 1, and the counter named *"<$(S).css, 2, 2>"* is 1, as shown in Fig. 2 (b).

TABLE I OPERATIONS

Operations	Expected Types	Statements
Numerical	Number	$v_1 - v_2$, $v_1 * v_2$, v_1 / v_2, $v_1 \% v_2$, $v_1 >> v_2$, $v_1 << v_2$, $v_1 >>> v_2$, $v_1 \& v_2$, $v_1 \mid v_2$, $v_1 \wedge v_2$, $+v$, $-v$
Addition	Number, String	$v_1 + v_2$
Relational	Number, String	$v_1 > v_2$, $v_1 < v_2$, $v_1 >= v_2$, $v_1 <= v_2$
Equality	Any types	$v_1 == v_2$, $v_1 != v_2$, $v_1 === v_2$, $v_1 !== v_2$
Assignment	Type of *value*	v = *value*
Conditional	Boolean	if (v), ! v
Function Call	Function	v(), new v()
Parameter	Any type	*f*(v_1, v_2, …)
Property Access*	String, Number, Boolean, Object, Function	v.length, v.match, v.css …

v, v_1, v_2: operands

Because we cannot abstract an unknown value to a known type, we introduce *abstract values* to represent their values, which is enlightened by *abstract locations* [24]. *Abstract values* refer unknown values those are not type-inferable, such as API return values and built-in variables. *Abstract values* are designed with the ability of recording their usage information. In fact, an abstract value is a list of counters, similarly to the usage counters defined for constant usage information.

C. The Type Miner

The type miner infers the most likely types for API return values and built-in variables based on their usages. Firstly, we build a probability model based on the factual usage information, which presents the probability of each known type involved in each scenario. Then we calculate the probability for each return value/built-in variable by summarizing weighted probability of their involved scenarios, and identify the type with maximum probability as their initial types. Because different API calls may return values ought to be the same type, we unify API return values by merging objects identified as similar based on a property-based similarity algorithm. Finally, for return values identified as objects, we identify the most used properties as their own properties.

This work is accomplished via 4 sub components: a model trainer that trains probability models, a type inferer that infers types of API return values and built-in variables, an object unifier that unifies objects identified as similar, and a property identifies that identifies properties of inferred objects.

1) The Model Trainer

The model trainer builds a probability model based the factual usage information. The probability model presents the

* In JavaScript, accesses on properties of variables pointing to Number or Boolean values are allowed.

S:
if: 1
<$, 1, 1>: 7
<$(O).css, 1, 1>: 2
<$(S).css, 1, 2>: 3
<$(S).css, 2, 2>: 1
<$(S).css, 1, 1>: 3

(a) Factual Usage Information

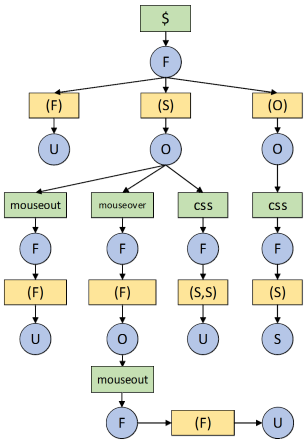
12 $(O)
.css: 1
12 $(O).css(S)
if: 1
<$(S).css, 2, 2>: 1
2 $(S)
.css: 4
4 $(S).css(S, $(O).css(S))
……

(b) API Usage Information

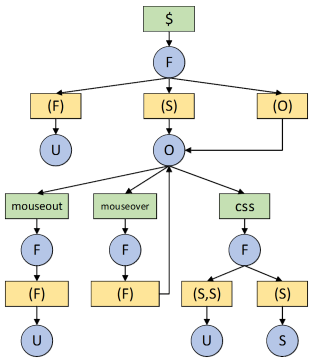
	S	N	B	O	F	L	U
if	1	0	0	0	0	0	0
<$,1,1>	0.7	0	0	0.2	0.1	0	0
<$(S).css,2,2>	1	0	0	0	0	0	0

S:String, N:Number, B:Boolean, O:Object, F:Function, L:null, U:undefined

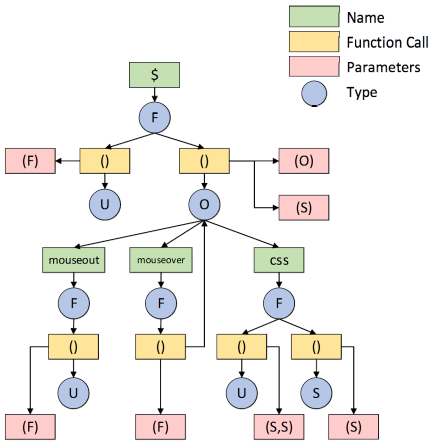
(c) Probability Model



(d) API Invoking Tree



(e) API Invoking Graph with Identified Types



(f) API Invoking Graph with Identified Parameters

Fig. 2. The Process Procedure

probability of a known type involved in a kind of scenarios. Take the *jQuery* core API *$* for example. In the above example, *$* is called 10 times with only 1 parameter, for 7 times the parameter is *String*, for 2 times is *Object*, and for 1 time is *Function*, so we conclude that when if a value is applied to a function call of *$* as its only parameter (i.e. *<$, 1, 1>*), this value has 70% probability of being a *String*, 20% probability of being an *Object*, and 10% probability of being a *Function*, as the second line shown in Fig. 2 (c). Note that the probabilities shown in Fig. 2 (c) are just the result trained based on the above example, which are quite different from the result on real-world applications, for example, the most possible type of expression of *if* statement in real-world is *Boolean*, rather than *String* shown in Fig. 2 (c).

To build the model, the model trainer summarizes all the collected counters for factual usage information, then calculates the probability of each type involved in each kind of scenarios. We use $P_m(t, s)$ to represent the probability of type *t* involved in scenarios *s*, where $t \in \{$ *String, Number, Boolean, Object, Function, null, undefined* $\}$.

2) The Type Inferer

The type inferer infers the most possible type for unknown values based on their usage information. We use *C(v, s)* to represent the occurrence of abstract value *v* involved in scenario *s*, *Pt(v, t)* to represent the probability of abstract value *v* being type *t*. Considering that the more an abstract value *v* is involved in an scenario *s*, the more possible *v* is of type used in *s*, we use the following formula to calculate *Pt(v, t)*:

$$P_t(v,t) = \sum_{s \in S} C(v,s) * P_m(t,s) \qquad (1)$$

where *S* stands for the set of all scenarios. Then the type with the maximum probability is identified as its type.

Note that formula (1) infers types of abstract values based on their usage information, while some APIs do have no return values, we collect no usage information of their return values. To handle this situation, we introduce a heuristic that handles *undefined*: for abstract values with no usage information, the type inferer infers their types as *undefined*; for abstract values having usage information, the type inferer infers their types using formula (1), and the inferred type should not be *undefined*. This heuristic is reasonable for two reasons: 1) it doesn't matter what type a never used return value is, 2) if the return value of a function is used in other call path, then the object unifier may merge them, and the identified *undefined* will be overlapped.

In the above example, API call *$(O).css(S)* appears 2 times, and their return values are used in scenarios *"if"* and *"<$(S).css, 1, 1>"* for 2 times. According to formula (3) and probabilities shown in Fig. 2 (c), we can conclude that *$(O).css(S)* is most likely to return a String value.

After the type inference, we link API calls with their inferred types, and inferred objects with their properties, which constructs an API invoking tree, as shown in Fig. 2 (d). An API invoking tree consists of 3 kinds of nodes: name nodes, function call nodes, type nodes. Name nodes present variables and properties; function call nodes present a kind of API calls, identified by their names and parameter; type nodes presents inferred types. The API invoking trees are input of sequential steps.

3) The Object Unifier

In the second step, we identify API calls with the same names and parameters as the same API calls, and infer one return

value type for each of them. This introduces a problem, i.e., one API may be identified as different ones, such as *$(O).css(S)* and *$(S).css(S)*. Furthermore, even different APIs may return the same type values, such as *$(S)* and *$(S).mouseover(F)* in the above example.

The object unifier is introduced to solve this problem, it identifies objects that ought to point to the same one and merges their properties. To accomplish this goal, we introduce a property-based similarity algorithm to calculate the similarity between objects, and use a threshold-based method to determine whether two objects should be identified as the same. We construct the similarity algorithm based on the following observations: 1) property accesses belonging to the same family should have higher weight, 2) common used property name should have lower weight, such as length, size, etc., 3) for some objects, only a small portion of properties are visited due to the long access path or the objects are rarely used, such objects should have higher weight. Based on these observations, we construct the following similarity algorithm:

$$sim(o_1, o_2) = W_f(o_1, o_2) * \frac{\sum_{p \in o_1, p \in o_2} W_p(p)}{min(|o_1|, |o_2|)} \quad (2)$$

where W_f is used to calculate the weight based on whether o_1 and o_2 belong to the same family, and W_p is used to get the weight for property p. W_f is defined as following:

$$W_f(o_1, o_2) = \begin{cases} w & o_1, o_2 \; belong \; to \; the \; same \; family \\ 1 & otherwise \end{cases} \quad (3)$$

here w is the weight for objects belonging to the same family, w and the threshold is predefined, and determined based on the experiments in Section IV.

Then we merge properties of identified objects. If a property belongs to only one of the objects, we just add it to the final object with its inferred type; if a property belongs to more than one objects, we add it to the final object, and summarize their usage counters, and reinfer its type. Note that some properties may be new objects, and are merged recursively.

After this step, the API invoking tree is translated into a graph, i.e. API invoking graph, as shown in Fig. 2 (e).

4) The Property Identifier

After the object unification, each object is identified with a set of properties. However, some of them may be negative. The negative properties refer to properties that should not exist according to the documentations of APIs. The negative properties may be induced by: 1) the over-approximation of our static analysis, 2) developers extension for their own purpose, and 3) wrong usage of APIs.

The property identifier is used to identify positive properties. We believe that most used properties are most likely to be positive. We firstly calculate the percentages of properties for each object, and summarize the percentages from big to small, and identify the properties as positives until the summation reaches a threshold.

D. The Parameter Miner

The parameter miner identifies API parameter types based on the API invoking graph. We identify frequently used parameters as positive. And for parameters identified as objects, we identify their properties based on their frequencies.

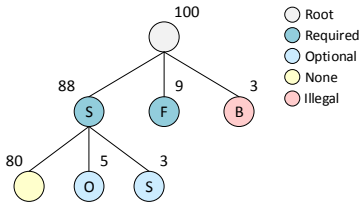


Fig. 3 Parameter Type Tree

This work is accomplished via 2 sub components: a parameter identifier that identifies positive parameters for each API, a property identifier that identifies positive properties for each object parameter.

1) The Parameter Identifier

Up to this step, we get all possible parameters for each API. However, some negative parameters may be involved, due to: 1) the over-approximation of our static analysis, 2) wrongly inferred types, and 3) misuses of APIs.

Before identifying positive parameters, firstly we identify parameters for each functionality of each API. Based on the API invoking graph, we merge the function call nodes from the same function type node and to the same type node, and introduce new parameter type nodes as their child nodes to present the missing parameters during the merging process, as shown in Fig. 2 (f). Then we identify positive child parameter nodes for each function call node.

Similarly, we believe that most used parameters are most likely to be positive. However, we find that some optional parameters are really seldom used. To identify these seldom used optional parameters, we build a parameter type tree, each node is a possible type of a parameter, each path is a kind of potential parameter sequence, and the depth of node presents the order of the parameter. If a node has some child nodes, but the summarization of children's occurrences is less than its own occurrence, then we add an especial node *none* to fill up the gap.

Based on the parameter type tree, we use a statistical method to identify positive parameters. For a node, we summarize the percentage of its child nodes from big to small, when the summarization reaches a threshold, child nodes involved in the summarization are identified as positive. Nodes having a *none* sibling, or child nodes of optional nodes are identified as *optional*. As shown in Fig. 3, there are 5 kinds of parameters, i.e. *(S)* occurs 80 times, *(F)* occurs 10 times, *(S, O)* occurs 5 times, *(B)* occurs 3 times, and *(S, S)* occurs 3 times. If the threshold is set to 0.97, then *(B)* will be identified as negative, while *(S, S)* with the same occurrence will not.

2) The Property Identifier

The identification of parameter properties faces the same problem as that of return value properties. Firstly, we summarize all properties collected from objects whether they are constant or inferred by the type inferer. Secondly, we use a statistical method to identify positive properties, which identifies frequently used properties.

IV. EVALUATION

To the best of our knowledge, our work is the first unsupervised API type specification miner for JavaScript. To demonstrate the precision and potential usefulness of our work, we conducted a series of experiments on the homepages of Alexa top 1000 web sites and a case study on jQuery. We try to answer the following research questions:

RQ1. Precision: How precise is the four key methods used in our approach? How to choose weight/thresholds for them?

RQ2. Effectiveness and potential applicability: What is the overall effect of our approach? Can our approach be used in detecting errors in source code and API documentations?

A. Experimental Subjects and Setup

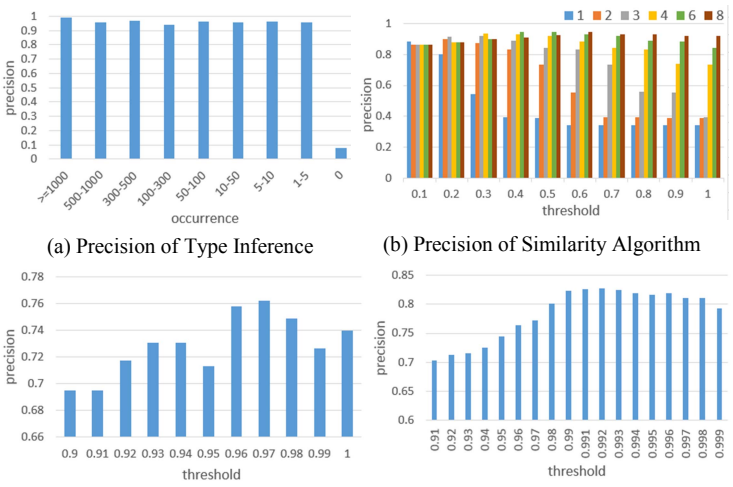
We chose the homepages of Alexa top 1000 web sites as our experimental subjects. We downloaded the source code of these 1000 web pages using a web page spider, which downloaded the homepage html files and JavaScript code files defined statically in the html files. Then we extracted the JavaScript code defined in each html file, and saved them into a new created standalone JavaScript file. Totally, we collected 3,753 JavaScript code files, including 733,923 lines of JavaScript code, with a total size of 149MB.

Then we run our static analyzer upon these JavaScript code files on a computer with Intel Core i7 CPU 3.40 GHz processer, 8GB of memory. We set the running time for analyzing one code file up to 10 minutes, the code files that cost more than 10 minutes were regarded as potential libraries, and were ignored. It cost about 10 hours to analyze all these code files.

Finally, we run the type miner and parameter miner upon the output results of analyzer. We selected the most used 24 API families as subjects, including 2 libraries (i.e. *jQuery, AngularJS* [16]), 8 native APIs (i.e. *RegExp, parseInt, Math, Date, parseFloat, escape, unescape, and Number*), and 14 DOM APIs/variables (i.e. *document, navigator, location, screen, console, localStorage, encodeURIComponent, encodeURIComponent, setTimeout, clearTimeout, setInterval, clearInterval, encodeURI,* and *alert*). The internal and final results are checked manually against the corresponding documentations or specifications.

We chose the aggregation of documentations for all versions that ever existed [17] [18] for libraries, because they may still in use. We chose the most compatible specification [19] for native APIs, i.e. ECMA262v5.1 [20], as our baseline. For DOM APIs, we used the living standard [3], which is an aggregation of existing DOM specifications. Additionally, for plugin APIs, we chose their documentations for active versions, which were recognized via a search engine [21].

The manual work was done by the first author, who has more than ten years JavaScript development experience. To minimize the likelihood for labeling errors, we made decisions using 3 methods: 1) determining based on the author's experience; 2) consulting the corresponding documentations/specifications; 3) executing the APIs locally on Chrome 61.0, Firefox 55.0, and IE 11.0 respectively. Each API was checked by at least 2 methods. This was a painstaking and time-consuming task. Fortunately we found many API calls and property accesses existed many times, and could be checked using the same checking result, such as *$().css()* and *$().find().css()*, which greatly speeded up the process. The manual work cost us about 20 days.



(a) Precision of Type Inference (b) Precision of Similarity Algorithm

(c) Precision of Parameter Identification (d) Precision of Property Identification

Fig. 4 Experimental Results

B. RQ1. Precision

There are 4 key methods used in our approach. We conducted four experiments to choose proper weight/threshold, and measure the precision of them.

1) Precision of Type Inference

We outputted all the inferred types of return values of APIs, built-in variables, and their properties, and manually checked the correctness of them. We didn't check the inferred types of parameters of callback functions, because in webpages, callback functions are always defined as event handlers, and are defined with no parameters.

The collected records were labeled into 3 categories: correct, incorrect, and omitted, where omitted referred to records which were not APIs, such as *document.frmMain* points to a FORM element named *frmMain*. Totally, we collected 4,088 built-in variables and their properties, and 12,574 API calls and their properties, within which, 8684 were identified as correct, 3599 were incorrect, and 4376 were omitted. We divided correct and incorrect records into 9 groups based on the number of accesses on them, and calculated the precision for each group. The result shows that our approach gains quite high precision for values used at least once, from 94.2% to 98.8%, with a total precision 95.8%; and for values never used, which were referred as *undefined*, our approach only gains 7.9% precision. This illustrated that, in all API calls with on usage information, 7.9% of them truly have no return values, such as *document.write()*, while other 92.1% should have return values, because these API calls existed at the end of call paths, we collected no usage information for them, such as *$().on().on()*. For these 92.1% APIs, their return values would be retrieved by object unifier. The full result is shown in Fig. 4 (a).

2) Precision of Similarity Algorithm

We measured the precision of similarity algorithm under various settings of weight *w* and the threshold. Firstly, we computed the similarity of each object pair under various

TABLE II CASE STUDY

Item	Ide	Doc	Cor	Pre	Rec
Property	116	141	105/87	90.5%	61.7%
Return Value	102	98	95	93.1%	96.9%
Parameter	150	330	141	94%	42.7%

Ide: number of identified items; Doc: number of items defined in documentation; Cor: number of correct items; Pre: precision; Rec: recall

settings of weight *w*, ranging from 1 to 10; then we determined whether each object pair should be unified manually; finally we selected a series of threshold, ranging from 0.1 to 1, and calculated precision of each pair of *w* and threshold, and the *w* and threshold pair resulting in the highest precision was selected.

Because our approach calculated the similarity for each inferred object pair, which resulted in up to 138,605 pieces of pairs. To facilitate the process, we randomly sampled 1,059 of them using a confidence interval of 3 and a confidence level of 95%. The final results is shown in Fig. 4 (b), which shows that, when the weight *w* is set to 6, and the threshold is set to 0.4, we can get the highest precision, 94.2%.

3) Precision of Parameter Identification

We measured the precision of parameter identification under various settings of threshold. Because the chosen of threshold will not affect the determination of APIs identified receiving only 1 kind of parameters, we decided to check APIs receiving more than 1 kinds of parameters. We manually checked the correctness of each kind of parameters against corresponding documentations and specifications, and calculated the precision under various settings of threshold, ranging from 0.90 to 1.00. The threshold resulting in the highest precision was selected.

Totally, our approach identified 81 APIs receiving more than 1 kinds of parameters, with total 223 kinds of parameters. The precision under various threshold is shown in Fig. 4 (c), from which we can conclude that when the threshold is set to 0.97, we can get the highest precision, 76.2%.

4) Precision of Property Identification

We measured the precision of property identification under various settings of threshold. Firstly, we determined their correctness by comparing them against corresponding documentations and specifications, then we calculated the precision under various settings of threshold, ranging from 0.91 to 0.999. The threshold resulting in the highest precision was selected.

Our approach identified 10,029 properties totally. To facilitate the process, we randomly sampled 965 of them using a confidence interval of 3 and a confidence level of 95%. The final result is shown in Fig. 4 (d). From the result, we can conclude that when the threshold is set to 0.992, our approach can get the highest precision, 82.7%.

C. RQ2. Effectiveness and Potential Applicability

To illustrate the overall effectiveness of our approach, we conducted a case study on the most popular JavaScript library, jQuery. We measured the precision and recall of our approach in identifying properties, return values, and parameters, and manually checked some of the unmatched JavaScript code to ensure that whether they were real errors in code or API documentation.

We chose jQuery library for the following reasons: 1) jQuery is the most popular JavaScript library, which is used in 83% web sites [22]; 2) jQuery is well documented; 3) jQuery APIs are very complex, many APIs allow more than 1 types of parameters, some of them even return more than 1 type of values.

The case study started with the jQuery core API, *$()*, we checked its return value and properties against the jQuery documentation. Here, the jQuery documentation we used is not specified for a single version, but the aggregation of documentations for all versions that ever existed. That's because websites in real-world use vary versions of jQuery, APIs deprecated in the latest version may be still in use in real-world.

1) Accuracy

We measured the accuracy of our approach from 3 aspects: property identification, return value identification, and parameter identification.

For property identification, we checked the names and their type names against the documentation, and identified a property is correct if its name and type are both correct. Our approach identified 116 properties for jQuery object, against 141 properties defined in jQuery documentation. Out of these 116 properties, 105 were correct, including 18 plugin properties, only 87 were defined in documentation. Furthermore, 86 out of 87 properties were APIs, the only property that was not an API is *length*, which is a Number. So the precision and recall of our approach in identifying properties for jQuery object are 90.5% and 61.7%, as shown in Table II.

For return value identification, we checked the type names of identified return values against the documentation. Note that many jQuery APIs supply both *getter* and *setter* functionalities, such as *css*, *html*, *attr*, etc., when they are used as a *getter* function, they returns the expected values, and when they are used as a *setter* function, they returns a jQuery object. Our approach identified 102 return value types for the correct 86 APIs, against 98 types defined in jQuery documentation, and 95 of them were correct. So the precision and recall in identifying return value types for jQuery object are 93.1% and 96.9%.

For parameter identification, we checked each parameter type composition against the documentation. jQuery APIs always allow users to apply optional parameters to specify the features of functionality. For example, the *hide* API, applying no parameter will hide the selected elements using default settings, or applying only a String (e.g. "*slow*") or a Number (e.g. 800, means 800*ms*) will hide the selected elements in an animation at the given speed. We counted every possible parameter type composition as an individual parameter input, for example, the *hide* API allows 12 kinds of parameters. Our approach identified 150 parameter type compositions for the correct 86 APIs, against 330 defined in jQuery documentation, and 141 of them were correct. So the precision and recall in identifying parameters for jQuery object are 94% and 42.7%.

In summary, our approach gains high precisions, but not that high recall in identifying properties and parameters on jQuery, because many APIs and parameters are seldom used.

2) Detected Errors in Source Code

We examined the collected jQuery calls (i.e. *$()* and *jQuery()*) inconsistent with the API documentation manually, and found 2 API misuses in real-world web sites. One is the wrong usage of jQuery's functionality that binds a function to be executed when the DOM has finished loading. The correct usage is to apply the function to the jQuery API, rather than its return value. Some web sites wrongly apply the return values, such as in the Alexa rank 792th web site*, *www.cnbc.com*, the *$()* API is used in List 2, the inner function contains no *return* statement, so our approach infers the outer *$* function receives an *undefined* as its parameter, which is determined as an error. In this case, the inner

```
$(function() {
    var payload = $('#articalPayload');
    ……
    // no return statement
}());
```

List 2. Error Code Found in www.cnbc.com

function will be executed immediately, rather than when the DOM has finished loading. Because the error code is collected from real-world web sites, it could not have brought obvious functionality loss, i.e. it was *trivial*. However, this type of errors may cause *critical* impacts. Because this error causes the inner function to execute before the DOM has finished loaded, which may cause the inner function to access an undefined element, then an error will be raised and the web page may become unaccessable.

The same error exists in the Alexa rank 810 web site as well.

3) Incompleteness of Documentation

During the examination, we did find an inconsistent case that the code was absolutely correct, while the parameters were absent in the API documentation. That is, applying two String parameters to retrieve some DOM elements using *$()* function, i.e., *$(String, String)*, we tested this parameters on various versions of jQuery, and they did work. However, in the jQuery documentation, it wrote:

$(selector [, context])

where *selector* is a String containing a selector expression, and *context* is a DOM element, document, or jQuery (object) to use as context. Here, the type *String* for *context* is missing, and should be added to the description of *context*, just as another jQuery API *index()* does. We have posted this issue to the jQuery forum, but no response has returned yet.

D. Threats to Validity

Our evaluation requires a great deal of manual work, which is tedious and error-prone. We leveraged several methods to try to get a correct result, including consulting the online documentations, searching from a search engine, and executing APIs locally. We may make some incorrect decisions on some APIs, but we have done our best.

Our case study performed on jQuery only. However, there are a huge number of polymorphism existing in jQuery APIs, and many jQuery APIs implement more than 1 functionalities, and output more than 1 types of return values, jQuery is complex enough to illustrate features of JavaScript APIs. Furthermore, we chose the values for weight *w* and 3 thresholds based on vary types of APIs, including native APIs, DOM APIs, and library APIs. In future work, we will evaluate our approach on more libraries besides jQuery.

* This is the ranking of 2016-6-16, and the source code was downloaded on 2016-6-20

V. Discussion

JavaScript is a highly dynamic language, which allows developers to define APIs with arbitrary parameters and return values. However, APIs ought to be designed for one or more specific purposes, which should be revealed by their parameters. We assume that different parameter type compositions trigger corresponding functionalities, or say, APIs decide perform which functionality based on the types of received parameters. This assumption holds for most of APIs, for which our approach works well. However, there are some APIs contrary to this assumption, they determine their functionalities based on more fine-grained information, such as prototypes or values of some parameters. Our approach cannot distinguish such API calls, we will handle them in the future work.

Another kind of APIs that our approach cannot handle are those which are designed to receive any types, or have no explicit defined return values. For example, the native API *isNaN* is designed to check whether the applied parameter is a Number, so any types of values are acceptable; another example is *eval*, which is designed to evaluate JavaScript code, its return value depends on evaluated JavaScript code. For such APIs, our approach is not able to generalize the types of parameters or return values.

Some developers may overwrite standard APIs, which brings great challenges for static analysis. Our approach is not able to handle such situations. However, because the mining process is performed on a large number of source code, we believe that a small amount of overwrites don't affect the result of mining process.

VI. Related Work

JavaScript Type Inference. Raychev et al. [23] trained a probability model from a set of JavaScript programs that have already been annotated with types for function parameters and "predict" types for function parameters in new programs. Madsen et al. [24] leverage a use analysis to record the usage of an object-like variable, and link the variable to an object defined in library stubs or in the same code file based on their shared properties. Their work require predefined background knowledge, either defined in annotations or in stubs or the same code file. While our approach requires only 7 basic types, then types and properties (for objects and functions) can be inferred.

API Specification Mining. Le et al. [25][26][27] leveraged rule-based method to mine API specifications in form of a finite state automation from execution traces. Pandita et al [28] applied natural language processing on method descriptions to infer API specifications from documentations. Their work focused on implicit dependencies among API calls, while our work focuses on implicit type constraints of APIs.

API Usage Pattern Mining. Zhong et al. [4] proposed MAPO, which extracted API call sequences from open source repositories, and mined the frequently occurred sub sequences as usage patterns. Wang et al. [30] proposed UP-Miner, a variation of MAPO, which was able to mine succinct and high-

coverage patterns. These work aimed to the strongly-typed language, Java. Nguyen el al. [31] proposed an approach to mine usage patterns for weakly-typed language, JavaScript. They represented the API related dependencies using a graph, and mined the frequently occurred sub graph as usage patterns. These work focus mining coexisting API calls, while our approach focuses on mining coexisting parameters.

JavaScript Type Checking. Bae et al. [10] proposed a static method to identify JavaScript API misuses with the assist of formal API specifications. In contrast, our approach detects API misuses by identifying outliers out of usages. Pardel et al. [29] proposed a dynamic method to detect type inconsistent associate to the same variables or properties for a single application, and report them as potential errors after a series of false positive pruning. While our approach is static and requires a large amount of source code.

VII. CONCLUSION

Despite JavaScript is a weakly-typed language, the usages of JavaScript APIs always follow implicit type rules. We propose an approach that mines type specifications for JavaScript APIs. We first extract usage information of constants and APIs, and infer types of API return values based a probability model built on the constant usage information, then we unify the identified objects, and identify parameters and properties. The evaluation on Alexa top 1000 websites and the case study on jQuery show that our approach mines type specifications with high precision and reasonable recall, the case study also shows that our approach can be used in error detection and documentation verification.

ACKNOWLEDGMENTS

This work was supported in part by National Key Research and Development Plan (2016YFB1000803) and National Natural Science Foundation (61672506, 61472407) of China.

REFERENCES

[1] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, B. Hardekopf, "JSAI: a static analysis platform for JavaScript", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 121-132.

[2] "jQuery". [Online]. Available: http://www.jquery.com.

[3] "DOM Living Standard". [Online]. Available: https://dom.spec.whatwg.org.

[4] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, "MAPO: mining and recommending API usage patterns", in *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009, pp. 318-343.

[5] "JSDoc". [Online]. Available: http://usejsdoc.org/.

[6] "KnockoutJS". [Online]. Avaiable: http://knockoutjs.com/documentation/introduction.html.

[7] "Aptana". [Online]. Avaiable: http://www.aptana.com/.

[8] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, W. Hu, "Bing developer assistant: improving developer productivity by recommending sample code", in *Proceedings of the 24nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 956-961.

[9] H. Jiang, J. Zhang, Z. Ren, T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for APIs", in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 38-48.

[10] S. Bae, H. Cho, I. Lim, S. Ryu, "SAFEWAPI: web API misuse detector for web applications", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 507-517.

[11] E. Wittern, A. T. T. Ying, Y. Zheng, J. Dolby, J. A. Laredo, "Statically checking web API requests in JavaScript", in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 244-254.

[12] "Alexa", [Online]. Available: http://www.alexa.com/topsites.

[13] M. Pradel, K. Sen, "The good, the bad, and the ugly: an empirical study of implicit type conversions in JavaScript", in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*, 2015, pp. 519-541.

[14] C. Strachey, "Fundamental concepts in programming languages", in *Higher-Order and Symbolic Computation*, Volume:13, Issue: 1-2, pp. 11-49, Apr. 2000.

[15] P. Cousot, R. Cousot, "Abstract interpretation: past, present and future", in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2014, pp. 1-10.

[16] "AngularJS". [Online] . Available: https://www.angularjs.org.

[17] "AngularJS: API Reference". [Online]. Available: https://docs.angularjs.org.

[18] "jQuery API documentation". [Online]. Available: http://api.jquery.com.

[19] "ECMAScript 5 compatibility table". [Online]. Available: http://kangax.github.io/compat-table/es5/.

[20] "Standard ECMA-262 5.1 Edition". [Online]. Available: http://www.ecma-international.org/ecma-262/5.1/.

[21] "Baidu". [Online]. Available: https://ww.baidu.com.

[22] "Builtwith". [Online]. Available: https://trends.builtwith.com/javascript/javascript-library

[23] V. Raychev, M.Vechev, A.Krause, "Predicting program properties from 'big code'", in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015, pp. 111-124.

[24] M. Madsen, B. Livshits, M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries", in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 499-509.

[25] T. B. Le, X. D. Le, D. Lo, I. Beschastnikh, "Synergizing specification miners through model fissions and fusions", in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 115-125.

[26] T. B. Le, D. Lo, "Beyond support and confidence: exploring interestingess measures for rule-based speicifcation mining", in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 331-340.

[27] D. Fahland, D. Lo, S. Maoz, "Mining branching-time scenarios", in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 443-453.

[28] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, A. Paradkar, "Inferring method specifications from natural language API descriptions", in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 815-825.

[29] M. Pradel, P. Schuh, K. Sen, "TypeDevil:dynamic type inconsistency analysis for JavaScript", in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 314-324.

[30] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, D. Zhang, "Mining succinct and high-coverage API usage patterns from source code", in *10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 319-328.

[31] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, T. N. Nguyen, "Mining interprocedural, data-oriented usage patterns in JavaScript web applications", in *Proceedings of the 36th International Conference on Software Engineering* (ICSE), 2014, pp. 791-802.