Detection of Feature Interaction in Dynamic Scripting Languages

Omar A. Al Harthi
Queen's University
Kingston, Canada
14oaah@queensu.ca

Manar H. Alalfi
Ryerson University
Toronto, Canada
manar.alalfi@ryerson.ca

Thomas Dean
Queen's University
Kingston, Canada
tom.dean@queensu.ca

ABSTRACT

In Plugin-based systems, merging plugins in one page is usually safe, but sometimes it may fail due to feature interaction. While these plugins work correctly when tested alone, they may fail or show unexpected behavior when one or more are used together. To identify and locate the causes of interference, one must inspect the plugin code and possibly examine assignment traces using proper analysis methods. However, manually testing thousands of plugins is not feasible, especially if many of the plugins lack appropriate documentation. Our goal is to introduce an automated detection framework based on static and dynamic analysis to detect potential feature interaction. In this paper, we will focus primarily on static analysis and plan to cover the dynamic analysis at a later stage.

First, we present the results of an initial study we conducted on an example plugin system, JQuery, and that to classify the types of conflicts found and to presents a list of patterns of objects assignment, then we present our automated static analysis approach designed to capture the patterns of conflicts. Finally, we present an experiment which applies our approach to a dataset of 2081 JQuery plugins and identified around 357K expression patterns. Our approach flagged 255 expressions as cases of duplicate objects names, and 180 as cases of duplicate global names.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; *Software design tradeoffs*; *Automatic programming*.

KEYWORDS

Features interactions, Static Analysis, Plugin Systems

ACM Reference Format:
Omar A. Al Harthi, Manar H. Alalfi, and Thomas Dean. 2019. Detection of Feature Interaction in Dynamic Scripting Languages. In *CASCON '19: 29th Annual International Conference on Computer Science and Software Engineering, Nov 04–05, 2019, Toronto, ON*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

CASCON '19, Nov 04–06, 2019, Toronto, ON

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In software development, features are considered as extra functionality added to the software system. Features interactions have been an issue in telecommunication and software systems, it is unintended behaviour in software systems due to a conflict in the use of one or more resources by different features. The more features included in a system, the more interactions can occur [1]. In plugin-based systems, features are added gradually at different phases in the software life-cycle by separate developers. Usually, plugins are tested in isolation, but when added to a particular system, there is an opportunity for interaction (i.e., behavioural changes) between the features merged within that system. Generally, interactions can cause problems in system development and user expectations. Some interactions, however, are harmless [2].

JavaScript [28], is a popular language for client-side development which supports a variety of libraries (e.g., MooTools, jQuery, and Dojo), templates (e.g., Bootstrap), and frameworks (e.g., Amber, Angular, React). Some of these libraries, in turn, support hundreds of thousands of plugins. Such libraries have their ecosystem and support communities that may use, extend and modify the resources. This complicated, dynamic and interactive environment may suffer from a lack of proper support for declaring modules, classes, and namespaces in JavaScript language and consequently generates several potential conflicts unless enhanced with continuous updates [15]. Developers need a way to realize and avoid harmful interactions and to develop applications in a conflict-aware environment, as such, in this paper we present an automated approach for interference detection for JavaScript plugins to address this challenge.

JQuery is a popular JavaScript library that improves the capabilities of JavaScript by creating extensions to manipulate the properties of objects on the page. Developers rely on jQuery plugins to support their applications' performance, appearance and security. JavaScript web applications use libraries to provide convenient functionalities, and these libraries usually take advantage of the dynamic features of the language [3].

To understand how the problems arise when using jQuery plugins, consider a scenario in which a developer decides to download and use one or more of the published plugins. For example, a developer may download a plugin for an image slider to view images and may also install a date picker plugin to allow users to pick start and end dates. When using both plugins on the same page, they might fail to function correctly due to attempts to manipulate or alter the same resources. The failures may be minor, or they may result in ultimate failure of one or both plugins.

To identify and locate the causes of interference, one must inspect the plugin code and possibly examine assignment traces using proper analysis methods. However, manually testing thousands of plugins

is not feasible, especially if many of the plugins lack appropriate documentation. Our goal is to introduce an automated detection framework based on static and dynamic analysis to detect potential feature interaction. In this paper, we will focus primarily on static analysis and plan to cover the dynamic analysis at a later stage.

The rest of this paper comprises of six sections. Section 2 presents some background to the problem we address. Section 3 shows the results of the initial study on the sources of interaction in JQuery plugins. Section 3.4 gives our approach to finding interactions statically. Section 4 provide the results of our approach. Section 5 give some related work and section 6 concludes the paper.

2 BACKGROUND

Plugin-based systems used increasingly in web applications, both on the server and client side. Problems with the merging of multiple plugins may cause frustration for both users and developers. By tracking some of code exchange and learning forums such as StackoverFlow [21], Stackexchange [22] and Github [23], we find that users complain of problems when merging two plugins into one page. Due to time constraints, a user will start to look for another plugin instead of investigating and fixing the problem. Duplicate names in different source files is a common problem. It seem simple at first glance and to avoid it we can change the conflicting name in one of the plugins. However, changing the name may cause other problems because the name may have other usages, or could be involved in unknown dependencies that are intentionally hidden.

Recent research examines possible interference in web applications and presents approaches to help in detecting them. Eshkavari et al. [4] identify plugin interference in WordPress by instrumenting PHP code. This work motivated us to try something similar for JavaScript and jQuery plugins.

Developers use traditional techniques in programming languages to facilitate code encapsulation and code reuse. For example, constructors such as classes and modules, usually need encapsulation to decrease the amount of code, and code reuse to reduce code repetition. Such concepts are used extensively by JavaScript communities and individual developers, which enrich the code modularity of JavaScript. A developer can do a separate task in different syntax, although it might create some debugging issues, it allows to apply better solutions to emulate such syntaxes [6].

2.1 The Issue with JavaScript

JavaScript is ranked as the most popular programming language on GitHub [20]. The significant number of published JavaScript applications, libraries, frameworks and plugins along with the lack of naming standards makes JavaScript applications more exposed to feature interaction. A plugin can easily overwrite an object of another plugin, or it can modify, delete, or accidentally use the object of another plugin. It is one consequence of the weak typing in the JavaScript language [14].

In JavaScript, everything is an object, and these objects have multiple methods and properties. Defining variables and functions in the global scope make them properties of the 'window' object. Also, all loaded scripts have access to global objects, and they can be redefined or assigned to new names. If a name already exists, it can be overwritten, and this usually leads to a conflict. This type of conflict can be detected via static code analysis if all predefined properties of the window object are known, or we can get a complete list of code patterns that are responsible for objects creation and assignment [28].

The same happens when creating an object, if multiple scripts access and write to the same object, then it is a potential conflict. Since jQuery plugins are an extension of a global object, a different code can access those global objects. Strict naming rules are needed to avoid possible interference. In reality, developers often use common names. Generally speaking, the same name can be detected by static and dynamic analysis. However, the dynamic feature of JavaScript makes such analysis more complicated [5]. Another feature in JavaScript is that one can add properties to the object after class definition, change the type of a property, or change a property from a value to a method (and vice versa). This feature makes JavaScript a very flexible and powerful object-oriented language, but at the same time, it complicates error tracking. In JavaScript, developers can do one thing in several ways, but this has advantages and disadvantages. From one side it enables the developer to work around the restrictions and rules, but also more opportunities for error [7].

A real-world example of a plugin conflict shown in Fig. 1, listing 1 shows code snippets of jquery.sliderJS.js, a plugin that allows sliding the images in the page. While jShowOff is a content rotator plugin. When combined on the same page, both plugins write to the global object named "jQuery.fn.slideIt". By accessing this object and assigning it to different functions or values, the second plugin jShowOff will overwrite the value and as a consequence sliderJS.js will use the wrong function. A similar error when one function is overwritten by another is shown in listings 1.3 and 1.4 of Fig. 1, where two conflicting plugins, Birdman.js and StatusBoard.js, are storing conflicting functions in the same property: jQuery.fn.lettering. Common names like cleandata, remove, resetform, gallery, and slide, are used frequently by developers.

In Fig. 2, plugin 1 and plugin 2 declare functions names in the global scope. In JavaScript, when a function or a variable declared in the global scope, it becomes a property of the object window. If both plugins loaded in one page, then the latest will ultimately override the first one and return the wrong value. Fig. 3, shows a real-world example of conflicting plugins declaring same function name in global scope. And more examples of real-world plugins that are declaring the same global variable names shown in Fig. 4.

3 APPROACH

We conducted an initial study and found that duplicate names are still a problem with jQuery plugins. It remains a problem even when applying techniques like namespaces and the noConflict() function. We tracked users' complaints from different forums on the Internet regarding cases where duplicate names are responsible for conflicts between jQuery plugins and we found many instances. The main complaints are:

- A new plugin is not working as expected, different from when running alone
- An unexpected change in the behavior of a particular function
- Alteration of an element's location, shape, size, color or it disappears.

```
(function($) {
$.fn.slideIt = function(options) {
    var container = $('.slideIt');
    /* code */
    }
}
```

Listing 1: plugin 1: jquery.sliderJS.js

```
function($) {
$.fn.slideIt = function(settings,callback) {
    var config = { direction : 'left'};
    /* code */
    }
}
```

Listing 2: plugin 2: jquery.jshowoff.js

```
(function($){
  jQuery.fn.lettering = function( method ) {
     /* code */
  }
}
```

Listing 3: plugin 1: Birdman.js

```
(function($){
jQuery.fn.lettering = function( method ) {
     /* code */
  }
}
```

Listing 4: plugin 2: StatusBoard.js

Figure 1: A real-world example of two conflicting plugins that may experience unexpected behavior when loading together.

```
$.fn.disable = function disable() {
return this.on('click.disable'
}).addClass('disabled').prop('disabled',true);
/* code */
})(jQuery);
```

Listing 5: plugin 1: mpolun-disable.js

```
 $.fn.extend({
 disable: function () {
 $(this).attr("disabled", "disabled"); /* code */
 }(jQuery);
```

Listing 6: plugin 2: pem-functions.js

Figure 2: Another example from real-world using different patterns $.fn.disable & pattern: $.fn.extend.foo = ...

```
function linkify(string, noFollow){
relNoFollow = "";
if (noFollow) {  /* code */}
```

Listing 7: plugin 1: Opal-plugins.js

```
 var linkify = function() {};
;(function($, linkify) {
relNoFollow = /*code*/; }
```

Listing 8: plugin 2: jquery.twitter.js

Figure 3: A real-world example of conflicting plugins declaring same function name in global scope object name: $.fn.disable and pattern: $.fn.extend.foo =.

```
window.onresize = throttle(function() {
var widthFlagResize = window.innerWidth;
if (flag != widthFlagResize) {
self.init(self.options);/* code */
 }
});
```

Listing 9: plugin 1: cardshow.js

```
jQuery(document).ready(function($) {
    window.onresize = function() {
        $(".modelImage").height($(".modelImage").width() / 3 * 2);
            }
    /* code */
});
```

Listing 10: plugin 2: home.js

```
(function($) {
$.pixelAlignSvgs = function() {
$.each($("svg"), function(){ /*code*/
window.onresize = $.pixelAlignSvgs
};})(jQuery);
```

Listing 11: plugin 1: jquery.pixel-align.js

```
if($(this).length > 0){
window.onresize = function(){
 });} /* code */
 return $this; };
}(jQuery));
```

Listing 12: plugin 2: pycs-layout.jquery.js

Figure 4: Another example of real-world's plugins declaring same global variable name

Based on our initial analysis we created a classification of patterns of conflicts as explained in the following subsection.

3.1 Classification of Conflicts

This section classifies the types of conflicts found in our study and presents a list of patterns of objects assignment. The patterns are collected from different resources, such as jQuery guide manuals, books, and forums. We found some patterns from the source code of jQuery release versions and published plugins. Our tool also helped to identify new patterns. JavaScript conflicts can be grouped into:

- *Global Names Conflicts*: A global object in a browser is a property of the window object. Global variables, constants, and functions are properties of this window object, an example illustrated in Fig. 5.
- *jQuery Object Name Conflicts*: Objects can be mutated, and their properties can be changed, built-in objects in JavaScript like Array, String, etc., can also be extended and their prototype properties can be changed too. A possible case of conflict

```
$ foo = window;$
$ window.foo = foo2 = 4;$
$ var foo = 4;$
```

Figure 5: Different ways of declaring global variables

```
 $.fn.foo = function(){ }          /* Plugin one*/
jQuery.fn.foo = function(){ }      /* Plugin two*/
```

Figure 6: Example of two plugins trying to write to same global jQuery name

```
1. $.fn.name =
2. $.fn.extend({ name:  })
  // different ways of representing jquery objects
  2.1 $.fn.extend({drag(){
  2.2 $.fn.extend({ foo: { bar: x } })
  2.3 $.method_name = ({ [property name1, name2,..nameN}
  2.4 (function(a){  a.fn.name = })(jQuery)
  2.5 eval($.fn.name = )
  // possible ways of accessing & changing jQuery objects and
       properties
3. $.fn.prototype.name =
4. $.fn.__proto__.name =
5. $.fn.prototype.prototype.name =
6. Object.defineProperties(jquery, object )
7. Object.defineProperty(jquery, name, value)
8. Object.assign(target, obj1, objn);
9. Object.freeze()
10. Object.preventExtensions()
11. Object.seal()
12. Object.setPrototypeOf()
```

Figure 7: Different patterns of targeted assignment expressions

is when two plugins are trying to access the same object, and it does not matter what they assign to, they will conflict with each other due to the closure scope concept, an example of such conflict illustrated in Fig. 6:

In JavaScript there are multiple ways of assigning objects, adding or deleting properties of an object. Fig. 7, shows a list of some of the patterns which can be detected statically. The list shows the possible variations in JavaScript. While these patterns are similar to those in conventional object-oriented languages, it can create new properties or change the class (prototype) of an object, or change a value in the prototype which will affect all objects refer to the same prototype. This unintended alteration can lead to incorrect output or unexpected behavior. JavaScript language is very flexible, with exceptional programming skills, a developer can find multiple new ways to do a single task, therefore, building a complete list of patterns is challenging. Instead, we look to provide coverage of the most common patterns.

For instance, pattern 1

$$\$.fn.name = \quad (1)$$

is responsible for accessing properties in the global jQuery object, such pattern presents potential cases of conflict. Patterns 2.1-2.5 are another way of expressing the jQuery object with same function as pattern 1

$$\$.fn.extendname : \quad (2)$$

Patterns 2.1-2.3 in Fig.7 use the extend method, which allows the programmer to add new properties to the parent property. Our approach detects such expressions and extracts all names included in the method argument. Patterns 2.4 and 2.5 allowing passing a parameter as a jQuery object and using of eval() command to pass JavaScript code include assignment expression respectively. While patterns 3-12 are properties of the standard built-in objects in

```
$.undefined :
{"type":"computed","file":"/Users/omar/Documents/deploy/all/jquery
      -i18next.js","start":{"line":115,"column":2}}
$.easing.undefined :
{"type":"computed","file":"/Users/omar/Documents/deploy/all/jquery
      .scrollify.js","start":{"line":207,"column":2}}
```

Figure 8: Investigating "undefined" names from the results helped us to detect more patterns

JavaScript, and the impacts on the performance of changing inheritance are ambiguous, it can reach the code that has access to objects with an altered prototype. Therefore, it is always recommended to create a new object with the desired prototype using Object.create(), instead of setting the prototype of an object [19].

Other built-in properties (i.e., defineProperty, defineProperties, freeze, seal, assign, setPrototypeOf and presevrntExtensionOf) are commonly used and can alter the object and its properties as well. To obtain the best results we detect these expressions and extract the names used within their arguments.

We analyze each pattern through the AST (Abstract Syntax Tree) builder and create an algorithm that emulates their usage in the jQuery files to extract the objects' and properties' names.

3.2 New Patterns

Our tool helps us to detect new patterns, through the observations of the tests results, shown in Fig. 8, and after locating the "undefined" names of the object member expression, we can investigate possible patterns of the object's assignments, once verified, we add it to our analysis algorithm to capture the correct property name.

3.3 Challenges

Some of the published JQuery plugins lack standard documentation which makes the plugin unreliable for testing. The presence of proper documentation helps to collect the required dependencies and build a reliable test environment. This behavior is usually found in a few online resources such as the jQuery website, where the community approves the published plugins. But the jQuery site has less than 3000 plugins, which is small compared to the 30000 jQuery plugins published on Github alone. Our goal is to provide automated detection of conflicts using a combination of static and dynamic analysis. Most of the patterns are static, but we also list some patterns to be covered by the future dynamic analysis. Another difficulty found in our initial study is when a plugin file has more than one version with different names, this leads to spurious duplicate results. Developers may also clone part of a plugin into a new plugin with a very minimal change, leading to duplicate results as well.

3.4 Static Analysis

Our static analysis starts by scanning the pool of candidates, one by one. For each candidate, we parse into an AST using the Acorn parser [24]. We check if it is a JavaScript file or a jQuery plugin file. JQuery plugins are identified by the use of $ as a global variable or the identifier jQuery. We search for occurrences of the patterns given in the list of conflict patterns found in our initial study. When a match is detected, we capture the members of the expressions which includes the object name and the chain of properties. We save the

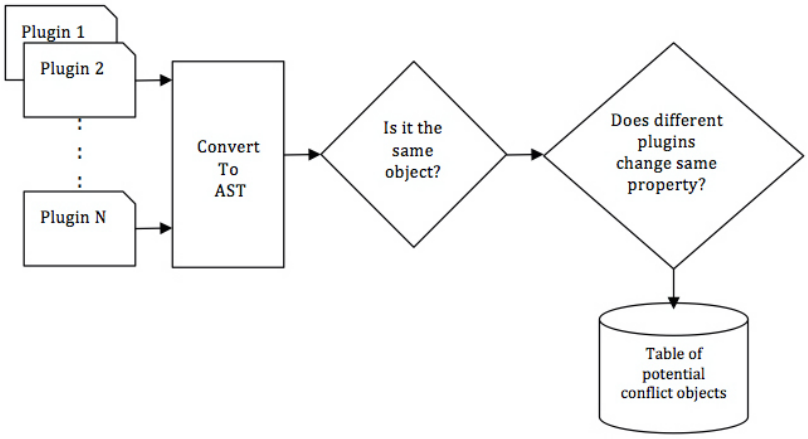

Figure 9: Proposed Static Analysis Approach

```
-----------------------------------------------------
Table 1: Used JavaScript plugins for the evaluation.
-----------------------------------------------------
                 Min     Median      Max       Total
jQuery plugins    1       1040      2081        2088
Lines of code     3       1397     13,837     758,118
Size (bytes)     47      56,080    750,207    30,567,105

Patterns expression analyzed :  357281

Cases of duplicate objects names 255

Cases of duplicate global names 180

most common patterns: 1- $.fn.foo = 702
                      2- $.fn.extend = 124
```

Figure 10: Test Set

types of the value and the name of the files in JSON format keyed by the assigned name.

We use Node.js [29], and the Acorn parser. The Acorn parser package provides a traversal mechanism that we use to examine the nodes looking for the target assignment patterns. We detect 'jQuery' and $ on the left side of an assignment expression and extract the full names of objects and their properties.

We check function body nodes in the AST. Each function body node has a fixed parameter array, containing the function body, type, start and end. Then we track the "type" node, find any assignment expression positioned on the left side of the body code because it is the explicit way of accessing and altering the object. Next we check the first value of the left part of the assignment expression, if it has a type of "object" which has a value of "jQuery" or the dollar sign, then we traverse the chain of this node and capture the values of each property on that chain (full dotted name).

Fig. 9, illustrates the process flow of our static analysis for collisions of global names and jQuery objects names, where we have at least two inputs of JavaScript code(e.g., variables, functions, methods). We compare the names from both modules, if there is a match, and both plugins are trying to write into the same object, then it is considered as a potential cause of conflict.

```
name: $next -
{"type":"auto","file":"/adrianjmiller-jquery.animate.js"}
{"type":"auto","file":"/jquery.inputMachinator.js"}

name: result -
{"type":"auto","file":"/birdman.js"}
{"type":"auto","file":"/jquery-barcode.js"}
{"type":"auto","file":"/jquery-smart-append.js"}
{"type":"auto","file":"/jquery.flatshadow.js"}
{"type":"auto","file":"/jquery.jqpagination.js"}
```

Figure 11: Report1: Results of detected potential conflicts of global variables names

4 RESULTS AND DISCUSSION

We tracked complaints in different web forums [8,9,10], and tested such plugins and found that duplicate names can cause conflicts. We randomly collected 2088 jQuery plugins from the jQuery website and from Github to be checked statically against our initial pattern list. We chose from a broader set of plugins with clear documentation. Fig. 10, shows the test set and the results.

We found 255 cases of potential conflicts. Half of the conflicts found involved one pair of plugins, while the rest of the conflicts involved groups of 3 to 13 plugins which increases the probability of conflict. Through static analysis we analyze the following:

(1) Global names, including: variables declared in global scope, functions declared in global scope, and variable names used in any scope without declaration
(2) The jQuery version
(3) Assignment to jQuery properties (i.e. full concatenated names)
(4) Duplication of jQuery custom properties
(5) Changing prototypes for example $xxx.\_\_proto\_\_.yyy$ and $.xxx.prototype.yyy$
(6) Property names extending the jQuery object. (e.g. $.fn.extend(name: ...))
(7) Detect non-standard assignments expressions such as: someFunctionName($.xx.yy = ...)

Our tool detected 180 instances of duplicate names of global variables that repeated in a minimum of two plugins, where almost half of these cases found repeated in groups of 3 to 24 plugins. The larger groups give a total of 1504 possible conflicts.

```
  $.cleanData :
{"type":"method","file":"/DoctorBot-manipulation.js"}
{"type":"method","file":"/jquery.datalink.js"}
{"type":"method","file":"/jquery.geo.js"
{"type":"method","file":"/jquery.tooltip.js"}

  $.fn.remove :
{"type":"method","file":"/DoctorBot-manipulation.js"}
{"type":"method","file":"/dom.js"}
{"type":"property","file":"/jquery-1.5-hack.js"}
{"type":"method","file":"/jquery.geo.js"}
{"type":"method","file":"/mDOMupdate.js"}
```

Figure 12: Report2: Results of detected potential conflicts of jQuery global names

Out of 2088 plugin files, we found 255 cases of duplicated property names repeated in a pair of at least two plugins, where almost half of these cases are in groups of 3 to 13 plugins, giving a total of 1370 possible case of conflicts.

Figures 11 and 12 show the reports generated from our tool for duplicate global variables and jQuery objects respectively. We can still find simple and common names of global variables like next, config, results. The same with objects and properties, names like cleanData, remove, and reset are found. The results show that developers often neglect basic standards of uniqueness in naming. The output also indicates that these names are used repeatedly in different plugins. Since developers sometimes publish new versions of their plugins, we avoid reading the same file by checking file name and size. We also find identical clones of targeted code repeated in many plugins.

4.1 Validation

We validated the results of our analysis by manually validating a selection of the conflicting plugins. We extended JSfiddle [25] to accept two plugins, and we called it JSfiddle2. We also used Headless Chrome and the NodeJS framework to provide an execution environment. Since there is no standard format for the distribution or use of a JQuery plugin, each test of a plugin requires manual intervention. Thus we randomly selected 20 conflicts reported by our tool (total of 40 pairs of plugins).

To set up our test environment, we first examine the selected plugin's documentation and collect a minimal configuration. Each of the plugins is run individually to verify that it works correctly as a stand-alone in the JSFiddle2 environment. We store the built configuration in a JSON file, allowing us to combine and run them in a single page. The table in Fig. 13, shows the results of the tested plugins. Our results confirm that each of the identified pairs of plugins conflict.

To confirm that the conflict was a result of the plugins and not a result of the JSFiddle2 testing environment, we also tested each of the plugins against a plugin from different conflict pair (i.e., one round-robin step). None of the resulting pairs of plugins conflicted, giving us confidence that the differences in behavior are due to the interaction between the plugins and not the testing environment.

Our testing environment allows to load and execute two plugins at the same time. Each plugin has a separate drop-down menu to upload HTML, CSS and JavaScript components of the plugin. The version of the jQuery library that used for the test is selected from a pull-down menu as well. We test the plugins alone through JsFiddle2, where all required dependencies can be loaded prior to execution.

4.2 False Positives

Through our testing we found some cases of false positive and false negative:

(1) Targeted patterns for dynamic analysis (Fig. 16) are possible cases of false positives.
(2) A clear false positive is when the value of two objects or variables is the same, this can be seen a lot due to the extensive use of code reuse. Our tool could reduce such false positives by eliminating repeated files names, and it can also checks file length to avoid such duplication.
(3) Correct enveloping the method: when a developer expects another plugin to have same global name enveloping the previous function and pass the data to the next function (e.g., sniffing), in this case, the first plugin won't be aware that third party can actually intercept the data, such example could be challenging to resolve because of its complexity, see Fig. 14.
(4) Another false positive example is when the code is not executed at all, it happens that same global names are defined in two different plugins, but one plugin doesn't actually use that name in a way that it changes it's value, in other words, nothing is assigned to that name; therefore it won't create any conflict.
(5) Some wrong coding practices can also lead to false positives, for example, Fig. 15, shows wrong declaration of variables which might turn local variables to global ones.

4.3 False Negatives

We examined false negative cases, and we found other conflicts between jQuery plugins that couldn't be captured through our tool due to a different type of conflicts such as DOM and event handling issues, such conflicts can not be detected statically. For example, there is a conflict between the following two plugins:

(1) TableSorter v2.9.1
(2) bootstrap-popover.js v2.2.1

These plugins are working just fine separately, but when combined on one page, the bootstrap-popover stops working. The plugin tablesorter.js is somehow removing the titles from the DOM before loading bootstrap-popover script [26].

Another case of false negative is with event handling. When merging those two plugins: dragscrollable.js and scrollto.js, each plugin is working correctly on a particular div, but when combining these plugins, it looks to have a kind of interaction between the mouse down and the click events. When holding down the mouse to move around, the drag event occurs correctly, but when releasing the mouse, it scrolled back to the element which held the mouse on to start dragging [27].

4.4 Threat to Validity

While we initially planned to validate our tool by computing precision and recall, we are manually validating results, it is infeasible to compute false positives with the large number of possible conflicts, more than 1500 produced by 2088 plugins. Instead we tested

	Object name	Plugin 1	Plugin 2	Observation (when combined)	Web Consol Error messages
1	$.cleandata	Tooltip.js	geo.js	Tooltip plugin stopped working after the combining of geo plugin	No errors shown in web consol
2	$.fn.drag	jquery.event.drag.js	jquery.gridster.js	both plugins stopped working, if we change the order, plugin 2 will not work	No errors shown in web consol
3	$.fn.insertAtCaret	jquery.mb.emoticons.js	jQuery.chineseIME.js	the input of plugin 2 interfer with input of plugin 1	No errors shown in web consol
4	$.fn.cropper	plugin.js	jquery.cropper.js	plugin 2 stopped working after merging with plugin 1	No errors shown in web consol
5	$.fn.unmask	jquery.mask.js	jquery.unmask.js	plugin 2 stopped working after merging with plugin 1	No errors shown in web consol
6	$.fn.mark	jquery.mark.js	jquitelight.js	plugin 1 stopped working	No errors shown in web consol
7	$.fn.textWidth	utilities.js	Marquee.js	both plugins stopped working	No errors shown in web consol
8	$.fn.once	jquery.sparkle.js	jquery.once.js	both plguins not working as expected	No errors shown in web consol
9	$.fn.scale	scale	quake.slider	plugin 1 stopped working	No errors shown in web consol
10	$.fn.preload	jquery.interactive-3d	jquery.preload	plugin 2 stopped working, plugin 2 should load some images, but it failed to do so after merging with plugin 1	No errors shown in web consol
11	$.fn.applyStyle	jquery.gridrotator	jquery.pagination	plugin 2 stopped working, it suppose to slide the images, if failed to do so after combining with plugin 1	No errors shown in web consol
12	$.fn.getSelector	selectorator	jquery.fcs	plguin 1 shows the content of the div when we click on the div label shown in the output, it dons't after merging with plugin 2	TypeError: $(...).getSelector(...).join is not a function
13	$.fn.modal	Less-Color-master	venobox-modal	plugin 2 stopped working after merging with plugin 1	No errors shown in web consol
14	$.fn.disable	mpolun-disable.js	pem-functions.js	plugin 2 stopped working after merging with plugin 1	No errors shown in web consol
15	$.fn.fullpage	honghu-zepto.fullpage.js	jquery.fullpage.js	plugin 2 stopped working after merging with plugin 1	No errors shown in web consol

Figure 13: Validation of conflicts between real-world plugins

```
$.name1
if ($.name1) var tempName1 = $.name1;
$.name1 = function(){ .... tempName1.call(arguments) }
```

Figure 14: Correct enveloping of the method name

```
var a = xyz,
    b = efg;
    c = hij,
var c is declared as global because the previous statement ended
     with ";".
```

Figure 15: Wrong coding practices may alter the declaration of local variable name to global name

a group of 40 plugins, which we statically detected to have 20 conflicts. While the false negative is more challenging because we need to have proved conflicts and validate them. It might be possible to calculate the false positives and negatives by analysing and detecting the patterns them but some of the cases have single patterns in our test set.

As future work, we plan to use mutational testing approach to generate test cases of plugin conflicts. Mutational operators would be drawn from the patterns of conflicts we identified. Using this approach, we can automatically and quantitatively validate our results by computing precision and recall.

5 RELATED WORK

Recent researches examine possible interactions in web applications and explain how to detect such conflicts, especially if the detection analyzes client and server side code.

Schuster et al. [12] focus on structural interactions and investigate structural feature interaction patterns. They propose an approach to detect design patterns in feature-oriented SPLs (Software Product Line). The debate that the occurrence of such patterns from a feature viewpoint shows how features interact to understand higher-level concepts, they applied an algorithm to find a relation of the design pattern concerning feature interaction. Machado et al. [13] present an Eclipse plugin FeatureJS to enable mapping between domain and implementation level features. With the use of the open-source platform for feature-oriented programming known as FeatureIDE, they provide automated support to SPL development in JavaScript and HTML. The contribution was mainly to expand the abilities of FeatureIDE to enable or disable the selection of a particular feature, and to detect their functionality and the features it contains.

For JavaScript Analysis, Eshkavari et al. [5] detect plugin interference in WordPress by instrumenting PHP code. This work has motivated us to do something similar for JavaScript and jQuery plugins. Patra et al. [14], although this work is similar to our work, it covers JavaScript libraries and focuses only on pairwise library conflicts, while we target jQuery plugins and can include two plugins or more at a time. It investigates the occurrence of global variable names in different JavaScript libraries, while we examine both global variables and jQuery objects names. Rostami et al. [15], introduce an Eclipse plugin to detect class emulations and function constructors in JavaScript programs, which is one pattern of a list of patterns we are covering in our approach. Hanam et al. [16], propose another approach to capture duplicate function names using static analysis. Maras et al. [17], present Firecrow, a plugin added to the browser, this work applies to the stand-alone application, it analyzes live pages loaded into the browser but not files in a folder, it doesn't do an in-depth analysis on JavaScript code as we do. Frad et al. [18], they intercept JavaScript code to find if the code following proper programming syntax. Our approach is quite similar but with a different scope.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we highlight the possible feature interaction problem that can occur when combining two or more jQuery plugins on one page. We identify the conflicts, mainly caused by duplicate names of global variables and jQuery objects. We explain how they might occur, possible detection mechanisms, and validation methods. We

```
1.$.fn[var1][var2]
2. eval('$.fn.extend({ name: ... })')
3. Global names assigned to the window object.
   w = (function(){ return window })(); or
   w = window;
4. Global object passed as argument or variable,
  (function(obj1, obj2){})
```

Figure 16: Examples of patterns to be analyzed dynamically

propose using static analysis as part of a broader framework to detect potential conflicts in jQuery plugins. Our initial results show that static analysis can identify some of the existing conflicts, however, there are also some cases that cannot be detected by static analysis alone and are the subject of future work. Two examples are the use of array syntax with variables to access properties, and code that is stored as a string and parsed at run time. We plan to annotate these cases with a probe function and extract the targeted names during run time.

Fig. 16, shows examples of patterns that can be detected statically but require dynamic analysis to capture and extract the values of the computed objects and properties:

We also plan to apply the same approach for other JavaScript libraries such as Dojo and MooTools, and propose an automatic mitigation approach for duplicate names if applicable. This work will help to understand such conflicts and facilitate proper mitigation in the future.

REFERENCES

1. T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In Proceedings of the Int. Conf. on Software Engineering for Telecommunication Switching Systems (SETSS), pages 59-62. IEEE, 1989.
2. Calder, Muffy, et al. Feature interaction: a critical review and considered forecast. Computer Networks 41.1 (2003).
3. Andreasen, Esben, and Anders Moller. Determinacy in static analysis for jQuery. ACM SIGPLAN Notices. ACM, 2014.
4. Eshkevari, L. Antoniol, G. Cordy, J. R., di Penta, M.Identifying and locating interference issues in Content Management Systems.
5. Maras, Josip, Jan Carlson, and Ivica Crnkovic. Towards automatic client-side feature reuse. Int. Conf. on Web Information Systems Engineering, 2013.
6. A. Osmani, Learning JavaScript Design Patterns - a JavaScript and jQuery Developer's Guide. O'Reilly Media, 2012.
7. David Calhoun, Different Ways of Defining Functions in JavaScript. Jun 24th, 2011.
8. http://davidbcalhoun.com/2011/different-ways-of-defining-functions-in-javascript-this-is-madness, Retrieved: September 2016.
9. https://forum.jquery.com/topic/rename-the-function-of-a-jquery-plugin-to-avoid-duplicate-function
10. https://stackoverflow.com/questions/11898992/conflict-between-two-jquery-plugins-with-same-function-name
11. https://www.techcartnow.com/how-to-use-jquery-plugins-with-conflicting-names-on-the-same-page-in-asp-net-mvc-web-application/
12. Schuster, Sven, Sandro Schulze, and Ina Schaefer. "Structural feature interaction patterns: case studies and guidelines." Proceedings of the Eighth Int. Workshop on Variability Modelling of Software-Intensive Systems. ACM, 2014.
13. Machado, Ivan do Carmo, et al. "Low-level variability support for web-based software product lines." Proceedings of the Eighth Int. Workshop on Variability Modelling of Software-Intensive Systems. ACM, 2014.
14. Patra, J., Dixit, P. N., Pradel, M. (2018). ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries.
15. Rostami, S., Eshkevari, L., Mazinanian, D., Tsantalis, N. Detecting function constructors in JavaScript. In Software Maintenance and Evolution (ICSME),(pp. 488-492). IEEE.
16. Hanam, Q. Brito, F. S. D. M., Mesbah, A. Discovering bug patterns in javascript. In Proceedings of the 2016 24th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (pp. 144-156). ACM.
17. Maras, Josip, Maja Stula, and Jan Carlson. "Firecrow: a tool for web application analysis and reus." Proceedings of the 29th ACM/IEEE int. conference on Automated software engineering. ACM, 2014.
18. Fard, Amin Milani, and Ali Mesbah. "Jsnose: Detecting javascript code smells." Source Code Analysis and Manipulation (SCAM) IEEE 13th Int. Working Conf. on. IEEE, 2013.
19. Standard built-in objects. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global-Objects/Object/ , Retrieved: May 2018.
20. A. La. Language trends on github. [Online]. Available: https://github.com/blog/2047-language-trends-on-github
21. Stack Overflow - Where Developers Learn, Share, and Build Careers. [Online]. Available:https://stackoverflow.com/
22. Stack Exchange: Hot Questions. [Online]. Available:https://stackexchange.com/
23. The world's leading software development platform Âů GitHub. [Online]. Available:https://github.com/
24. GitHub - acornjs/acorn: A small, fast, JavaScript-based JavaScript parser. [Online].Available:https://github.com/acornjs/acorn
25. JSFiddle: Create a new fiddle.[Online].Available: https://jsfiddle.net/
26. stackoverflow - Jquery plugins interfering with each other [Online].Available:https://stackoverflow.com/questions/16345921/jquery-plugins-interfering-with-each-other
27. stackoverflow - Conflict between two jquery plugins (dragscrollable and scrollto) [Online].Available:https://stackoverflow.com/questions/5619420/conflict-between-two-jquery-plugins-dragscrollable-and-scrollto
28. JavaScript, Standard ECMA-262, ECMAScriptÂő 2018 Language Specification [Online].Available:https://www.ecma-international.org/publications/standards/Ecma-262.htm
29. NodsJs, A JavaScript runtime built on Chrome's V8 JavaScript engine. [Online].Available:https://nodejs.org/en/