Hybrid DOM-Sensitive Change Impact Analysis for JavaScript

Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman

University of British Columbia
Vancouver, BC, Canada
saba,amesbah,karthikp@ece.ubc.ca

Abstract

JavaScript has grown to be among the most popular programming languages. However, performing change impact analysis on JavaScript applications is challenging due to features such as the seamless interplay with the DOM, event-driven and dynamic function calls, and asynchronous client/server communication. We first perform an empirical study of change propagation, the results of which show that the DOM-related and dynamic features of JavaScript need to be taken into consideration in the analysis since they affect change impact propagation. We propose a DOM-sensitive hybrid change impact analysis technique for JavaScript through a combination of static and dynamic analysis. The proposed approach incorporates a novel ranking algorithm for indicating the importance of each entity in the impact set. Our approach is implemented in a tool called TOCHAL. The results of our evaluation reveal that TOCHAL provides a more complete analysis compared to static or dynamic methods. Moreover, through an industrial controlled experiment, we find that TOCHAL helps developers by improving their task completion duration by 78% and accuracy by 223%.

1998 ACM Subject Classification D.2.5 Testing and Debugging; D.2.7 Distribution, Maintenance, and Enhancement

Keywords and phrases Change impact analysis, JavaScript, hybrid analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

To remain useful, a software program must continually change to adapt to the changing environment [13]. Code change impact analysis (CIA) [3] aims at identifying parts of the program that are potentially affected by a change in the code. Impact analysis has been a popular research area [2, 5, 17, 19, 20]. Most of the research, however, is focused on traditional programming languages and ignores code, which is used extensively in modern web applications today.

JavaScript requires a novel approach for effective change impact analysis, because it has a set of unique features that makes it challenging to comprehend [1] and analyze by traditional code analysis techniques [8].

The first feature is the interplay between the JavaScript code and the Document Object Model (DOM) at runtime. The DOM is a standard object model representing HTML at runtime. DOM APIs are used in JavaScript for dynamically accessing, traversing, and updating the content, the structure, and the style of HTML pages. We have observed that the impact of a code change can be propagated through the DOM, even when there may be no visible connections between JavaScript functions and variables in the JavaScript code. The second feature pertains to the highly dynamic [21] and event-driven [25] nature of JavaScript code. For instance, a single fired event can dynamically propagate on the DOM tree [25] and

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1023

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

trigger multiple listeners indirectly. These implicit relations between triggered functions are not directly visible in the JavaScript code. Finally, XMLHttpRequest (XHR) objects used for asynchronous communication in JavaScript can transfer the impact of a change between two parts of the program that are not explicitly connected through the code. For instance, a server response can dynamically generate and execute JavaScript code on the client-side through callbacks.

Traditional impact analysis has been performed either by static analysis or dynamic analysis. However, the aforementioned challenges make it difficult for JavaScript static analysis techniques [8, 12, 22] to carry out impact analysis effectively. None of these techniques can provide support for the DOM-based, dynamic and asynchronous JavaScript features. Further, current dynamic and hybrid analysis techniques [26, 27] ignore the aforementioned challenges, i.e., they overlook the important role of the DOM in their analysis and they do not support the hidden relations that are created through event-handler registration, event propagation, and asynchronous client/server communication.

In this paper, we propose a hybrid analysis method for change impact analysis of JavaScript-based web applications that combines the advantages of both static and dynamic analysis techniques to obtain a more complete impact set. Our analysis is DOM-sensitive and aware of the event-driven, dynamic and asynchronous entities, and their relations in JavaScript. It creates a novel graph-based representation capturing these relations, which is used for detecting the impact set of a given code change. The main contributions of our work are as follows.

- A formalization of factors and challenges involved in change impact analysis for JavaScript.
- An exploratory study to investigate the existence and role of impact paths that require the analysis of DOM-related and event-based features in JavaScript. The results show that these features exist in real-world applications and cannot be ignored in proper change impact analysis for JavaScript.
- A DOM-sensitive event-aware hybrid change impact analysis technique for JavaScript. The approach creates a novel hybrid model that is used for identifying the impact set of a change in a given JavaScript application.
- A set of metrics for ranking the inferred impact set to facilitate the finding and understanding of the desired change impact by developers.
- An implementation of our approach in a tool called Tochal (TOol for CHange impact AnaLysis). Tochal is open source and available for download [24].
- An empirical evaluation of Tochal through a comparison with traditional pure static and dynamic analysis approaches, as well as a controlled experiment to assess the usefulness of Tochal in an industrial setting.

Our results show that event-driven and dynamic interactions between JavaScript code and the DOM are prominent in real applications, can affect change propagation, and thus should be part of a JavaScript impact analysis technique. We also find that a hybrid of both static and analysis techniques is necessary for a more complete analysis. And finally, Tochal can improve the performance of developers in terms of both impact analysis task completion duration (by 78%) and accuracy (by 223%).

2 Impact Transfer in JavaScript

Many unique features of JavaScript applications require special attention during impact analysis. These features include (but are not limited to) DOM interactions, dynamic event-driven execution of functions, and asynchronous communication with the server. The impact

```
function checkPrice() {
  var itemName = extractName($('#item231'));
  var cadPrice = $('#price_ca').innerText;
  $.ajax({
     url : "prices/latest.php",
     type : "POST",
     data : itemName,
     success : eval(getAction() + "Item")
  });
  confirmPrice();
}
function updateItem(xhr) {
  var updatedInfo = getUpdatedPrice(xhr.responseText);
  suggestItem.apply(this, updatedInfo);
}
function suggestItem() {
  if (arguments.length > 2) {
    displaySuggestion(arguments1);
  }
}
function calculateTax() {
  $(".price").each(function(index) {
    $(this).text(addTaxToPrice($(this).text()));
  });
}
$("#price-ca").bind("click", checkPrice);
$("prices").bind("click", calculateTax);
```

Figure 1 Motivating example: JavaScript code

```
<img id='item231' src='img/items/231.png'
          itemName='dress'/>
<fieldset name='prices'>
    <div class='price' id='price-ca'>120</div>
    <div class='price' id='price-us'>110</div>
</fieldset>
```

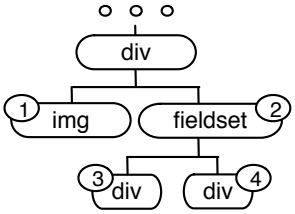

Figure 2 Motivating example: HTML/DOM

can be transferred through these entities, without direct visible relations in JavaScript. Throughout the rest of the paper, we use the term *indirect* impact to refer to change impact transferred through such features. Impact transferred directly through JavaScript code, e.g., through function calls, is referred to as *direct* impact.

▶ **Definition 1** (Relevant Entities). Let f be a JavaScript function, d a DOM element, and x an XHR object. If F, D, and X are sets representing each of those entities, respectively, then the set of all relevant entities is defined as $E : \sum \varepsilon \leftarrow F \cup D \cup X$

Change impact can propagate between these JavaScript entities through a series of *read* and *write* operations.

▶ **Definition 2** (Read/Write Operations). Let ε_1 and ε_2 be arbitrary entities in E. Suppose ε_1 writes to ε_2 at time τ. Then the relation is represented as $\varepsilon_1 W_\tau \varepsilon_2$. If ε_1 is read by ε_2 at time τ, then the relation is represented as $\varepsilon_1 R_\tau \varepsilon_2$.

The semantics of the relations between entities are drawn from actual JavaScript execution mechanisms (e.g., function f reads from a DOM element d). However, for each W/R relation between two entities, there is a conceptual R/W relation between the same two entities in

the opposite direction (e.g., d writes to f). Definition 3 formalizes the notion of impact transmission between two entities.

▶ **Definition 3** (Impact). Let ε_1 *and* $\varepsilon_2 \in E$. If the value and/or the behaviour of ε_2 depends on the value and/or the behaviour of ε_1, then ε_1 is said to have an impact on ε_2, represented as $\varepsilon_1 \rightarrow \varepsilon_2$.

The impact can also be indirectly transferred from entity ε_1 to ε_2, if ε_1 writes to ε_3, which is later read by ε_2. We call such this relation a `WR` pair.

We use a simple motivating example, presented in Figures 1–2, to illustrate the challenges and explain each definition. We use different portions of this example in the following subsections. Note that this is a simple example and these challenges are much more potent in large and complex web applications.

2.1 Impact through the DOM

In modern web applications, the DOM structure [7] evolves dynamically through the execution of JavaScript code, to update the content, structure, and style of the application in a responsive manner. A JavaScript function can write to a DOM element, which in turn can be read by another function and thus impact its behaviour. Such DOM elements can transfer the impact of a change indirectly. Impact transferred through the DOM introduces an important challenge in identifying change impact in web applications. Hence, in this work, we propose DOM-related dependencies as additional means of impact transfer.

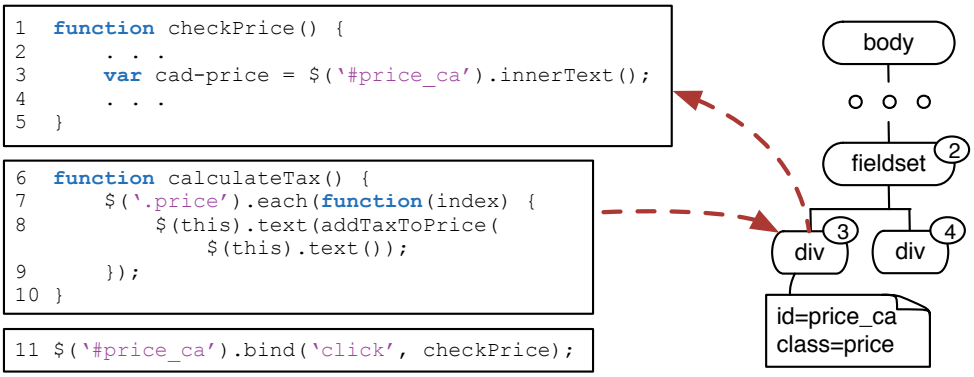

Figure 3 Impact transfer through DOM elements.

▶ **Example 4.** Figure 3 displays a portion of the motivating example that contains a hidden DOM-based dependency. Function `calculateTax()` retrieves all DOM elements having the class attribute `price` (line 7). The function then recalculates the price of each element to include the tax and rewrites the value of the element with the new price (line 8). Later, when the function `checkPrice()` (line 1) is invoked through a user event (registered in line 11), it retrieves the value of a DOM element with id `"price-ca"` (line 3) and uses this value to perform other operations. So far, there is no direct relation between functions `calculateTax()` and `checkPrice()` that shows any dependency between the two code segments. However, looking at the DOM structure shown on the right side of Figure 3, we can see that the element with ID `"price-ca"` is also an instance of the `price` class (element ③ on the DOM tree). This means that the value used by `checkPrice()` may be affected by `calculateTax()`. This is a simple example of dynamic DOM dependency, which needs to be taken into account in impact analysis of JavaScript applications.

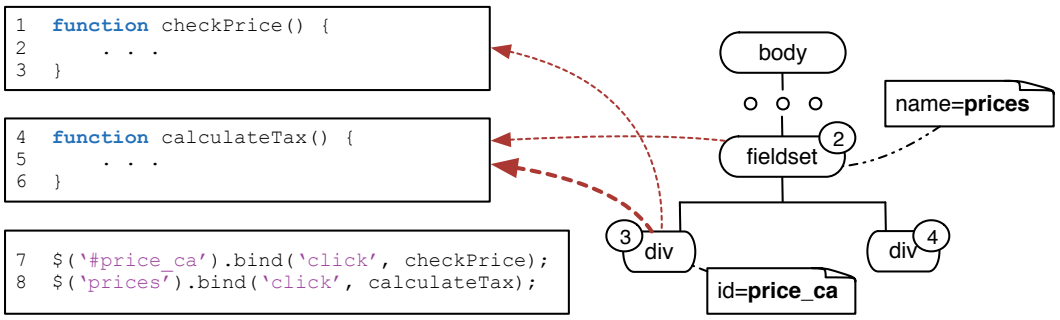

Figure 4 Impact transfer through event propagation.

2.2 Impact through Event Propagation

In web applications, a single event can propagate on the DOM tree and invoke multiple handlers of the same event-type attached to any of the ancestors of the target element [25]. The direction of the event propagation depends on whether the capturing or bubbling mode is enabled. When *capturing* is enabled, the event is first captured by the parent element and then passed to the event handlers of children, with the deepest child element being the last. With *bubbling* enabled, an event first triggers the handler of the target element on which the event was fired, and then it bubbles up and triggers the parents' handlers. The second type of impact dependencies we introduce pertain to the hidden relations between the handlers invoked via propagation of the original event on the target DOM tree. Such invoked handlers can be involved in change impact propagation, i.e., they can affect the control flow of the application and thus need to be considered in impact analysis for JavaScript.

▶ **Example 5.** In a segment of the motivating example shown in Figure 4, `checkPrice()` is attached to the element with id `price-ca` as an event handler (line 7). Therefore, if a user clicks on that element (element ③ on the right side DOM tree), function `checkPrice()` gets invoked. However, `price-ca` is contained within a `fieldset` element with the name `prices` (element ② on the DOM tree), which is similarly bound to an event handler for the `click` event (Figure 4, line 8). Due to event *bubbling*, any click on `price-ca` will bubble up to `prices` and trigger its event handler as well. Hence, `calculateTax()` is invoked through propagation of an event originally targeting `price-ca`. As a result, the execution of `calculateTax()` also depends on the `price-ca` element, in addition to the `prices` element.

2.3 Impact through Asynchronous Callbacks

XMLHttpRequest (XHR) objects help developers enrich user experiences with web applications through asynchronous communication with the server. While increasing the interactivity and responsiveness of applications, XHR usage adds additional complexity to impact analysis. Each XHR consists of three main phases: *open*, *send*, and *response*. A callback function is invoked when the XHR response is received from the server, without a user involvement. A change in opening the request, sending it, or the sent message can impact the response of the server, as well as the behaviour of the application after receiving the response through a callback function. Different components of XHR objects can make the detection of control and data flow relations more troublesome, particularly when these components are not necessarily collocated in the same function or module. This motivates the third type of impact dependencies we introduce in this work.

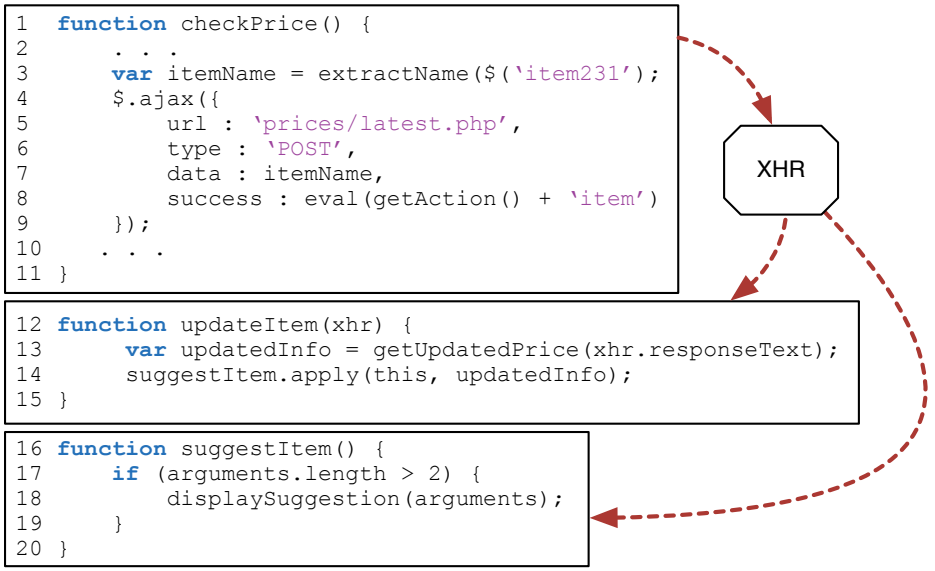

Figure 5 Impact transfer through asynchronous callbacks.

▶ **Example 6.** `checkPrice()` sends an asynchronous request to the server (lines 4–9 in Figure 5). However, the assigned callback function cannot be recognized statically, as the code uses the action chosen by the user dynamically to invoke the appropriate function (line 8, `eval`). Let's assume that in this example the selected action is to "update" the price of an item, and hence the `updateItem()` function is assigned as the callback function of the XHR object. As it can be seen, there are no direct function calls or shared variables between `checkPrice()` and `updateItem()` to enable traditional change impact analysis techniques to derive a dependency relation between the two functions. However, the XHR message along with the data that was sent with it can affect the response that comes back from the server and thus can impact the behaviour of `updateItem()`.

2.4 JavaScript Dynamism

Many traditional impact analysis techniques use static aspects of the code to determine the impact set of a change. Dynamic features of the JavaScript language pose a challenge to static analysis techniques. For instance, almost everything in JavaScript, from fields and methods of objects to their parents, can be created or modified at runtime. Also, JavaScript's dynamic policies for invoking functions can add more complexities. One such policy is function variadicity, which is common in web applications [21]; i.e., in JavaScript, functions can be invoked with more or less arguments compared to the parameters specified in a function's static declaration. In addition to the DOM, event, and XHR challenges, the dynamic features of the JavaScript language need to be addressed in an effective change impact analysis technique.

▶ **Example 7.** In line 14 of Figure 5, `updateItem()` invokes `suggestItem()` through the `apply()` function, which makes it impossible to infer the number of passed arguments statically. Function `suggestItem()`, the callee, takes no arguments according to its declaration (line 16). Yet, the function is invoked with an arbitrary non-zero number of arguments, which can change the execution of the application (Figure 5, line 17). Knowledge of the passed arguments at runtime is crucial for performing precise data-flow analysis, required for impact

analysis.

2.5 Impact Paths

The concept of WR pairs can be generalized to any mechanism that can transfer impact in JavaScript, such as XHR objects, function arguments, and function return values. Consecutive WR pairs involving all JavaScript entities can form general impact paths, as described in Definition 8.

▶ **Definition 8** (Impact Path)**.** An impact path of an entity ($P(\epsilon)$) is a directed acyclic path starting from entity ϵ. The nodes on the path are entities in the system, and the edges are the directed impact relations that connect those entities.

For instance, `updateItem()` → `suggestItem()`, `checkPrice()` → `#price-ca` (DOM element with id=`price-ca`), `checkPrice()` → `#price-ca` → `calculateTax()` → `addTaxToPrice()` are examples of impact paths that exist in the running example (Figure 1).

The length of an impact path is defined as the number of entities in the path. The minimum length for propagation of the impact of a change through DOM elements is 3 ($f\ W_{\tau_1}\ d\ R_{\tau_2}\ g \mid \tau_1 < \tau_2,\ d \in D,\ f, g \in F$).

3 Exploratory Study: DOM-related and Event-based Impacts

We conducted an exploratory study to investigate the role of JavaScript's DOM-related, event-based and dynamic features in code change propagation. Our goal was to understand whether DOM elements, event handlers, and propagated events contribute to forming new impact paths in JavaScript code.

Subject Applications. We selected ten web applications that make extensive use of JavaScript on the client-side for this study. We selected these applications from (1) participants of recent JavaScript programming contests, and (2) trending and popular JavaScript projects on GitHub[1]. They are listed in column 1 of Table 1.

Design. We capture JavaScript-DOM interactions as well as any occurrences of event propagation on the DOM tree. For each DOM access that occurs during the execution, we collect the accessed entity, the JavaScript function that accesses the DOM, and the access type. Access types are directed operations performed on DOM elements by JavaScript functions, where the direction of the access is determined by the direction of the flow of data. For instance, assume function foo creates DOM element e at time τ, which means $foo\ W_\tau\ e$. Then the type of access is `element-creation` and the direction of access is from foo to e. The collected data is analyzed to extract the impact set for each of the JavaScript functions involved in the execution. The DOM elements that are located on at least one impact path of a function are the ones that can contribute to the impact propagation process and are called WR elements for simplicity. The considered impact paths are required to have a length of at least three (e.g., $foo W_{\tau_1} e R_{\tau_2} bar$, for functions foo and bar and DOM element e). Moreover, redundant impact pairs (reads and writes between same entities) do not contribute to the impact paths and are therefore eliminated from the analysis.

[1] `https://github.com/trending/`

Table 1 (A) Results of analyzing JavaScript's DOM-related and dynamic features. (B) Factors in determining impact metrics.

JavaScript		(A) DOM and dynamic features				(B) Factors of impact metrics				
Application	**LOC**	**# DOM elem.**	**% WR DOM elem.**	**# of handlers**	**% prop. handlers**	**Avg. Path Length**	**fan-in elem.**	**fan-out elem.**	**fan-in func.**	**fan-out func.**
same-game	229	62	98	20	45	6.3	1.9	2.9	23.1	15.2
ghostBusters	343	44	61	39	0	4.3	3.3	0.4	2.6	20.0
simple-cart	9238	41	51	14	0	3.9	2.1	1.7	2.7	3.3
mojule	522	47	17	18	33	7.0	1.5	2.1	4.6	3.3
jq-notebook	839	1	100	21	38	4.0	16.0	11.0	0.9	1.3
doctored.js	3534	2	50	47	15	5.3	4.0	7.0	1.0	0.8
jointlondon	2498	34	9	16	0	3.7	0.8	2.2	1.6	0.6
space-mahjon	983	61	10	53	4	4.0	1.8	3.0	1.5	0.9
listo	354	5	20	10	0	4.0	0.1	1.5	21.4	1.9
peggame	1274	17	6	23	0	3.0	0.8	1.3	4.3	2.1
Average	1981	31	**42**	26	**14**	**4.6**	**3.2**	3.3	**6.3**	**4.9**

Results. Each application is manually exercised in different scenarios multiple times and the results are integrated. The results are shown in section (A) of Table 1. The first column of section (A) displays the total number of DOM elements accessed by JavaScript code during execution. The second column of the section shows the ratio of WR DOM elements to the total number of involved DOM elements from the first column. The number of DOM event handlers that were triggered during the execution is shown in column three. Column four represents the percentage of handlers that were invoked through event propagation (capturing or bubbling) to the total number of triggered handlers. The average length of impact paths in each application is depicted in column one of section (B).

The results show that on average, 42% of the DOM elements that were accessed during the execution of these applications, were part of an impact path between two functions. Moreover, about 14% of the executed event handlers were invoked through event propagation mechanisms. The results thus reveal the importance of DOM elements in transferring the impact. Also, the role of propagated event handlers is significant in determining the dynamic behaviour of a JavaScript application. Hence, a CIA technique for JavaScript application should consider the DOM-related and event-based features as media for propagating the impact.

We further analyze the structure of the created dependency graphs to gain more insight into the nature of DOM- and event-based relations within JavaScript applications. Among all structural and semantic aspects of the graphs, the average fan-in and fan-out scores of the functions and DOM elements in the graphs are reported in section (B) of Table 1. These factors are selected due to their correlations with the ratio of WR elements in subject applications. We use this information later in the paper, when we propose a set of metrics for ranking the impact set (section 5).

4 Hybrid Analysis

We propose a hybrid technique, called TOCHAL, which augments static analysis with dynamic analysis to enable a DOM-sensitive and event-aware change impact analysis method for JavaScript applications.

4.1 Static Control-Flow and Partial Data-Flow Analysis

Our approach first identifies JavaScript entities that *can* be analyzed statically. Among entities described in Definition 1, JavaScript functions (F) are the only entities that fit

this criterion. The DOM is created and mutated during execution. This limits the static reasoning about its structure and possible event propagations that would affect change impact. Regarding XHR objects, it is not easy to infer statically what messages are received from the server. Moreover, there is no static information available regarding the order and timing of asynchronous callbacks.

Our static analysis module incorporates *direct* relations between functions into a static call graph (SCG) by analyzing the JavaScript code. In JavaScript, functions are first-class citizens and receive the same treatment as objects; we augment the same static call graph with global variables, which we treat similarly as the functions.

To increase the precision of the static analysis, which in turn improves the quality of the impact set, we perform a pruning algorithm on the extracted dependencies. The pruning is conducted based on a partial data-flow analysis of the call graph. Function invocations are not considered as impact relations unless the two functions have a data dependency through passed arguments or return values, as described in Theorem 9. This does not concern data dependencies through global variables shared between two functions, where separate dependency relations are formed.

▶ **Definition 9** (Function Dependencies)**.** Let $\rho, \delta \in P$. Then impact relations between f and g are defined as:

- $f \rightarrow g$ if f invokes g and the signature of g indicates that it takes parameters.
- $g \rightarrow f$ if f invokes g and the definition of g includes a return value.

4.2 Analyzing the Dynamic Features

To include the dynamic features of the JavaScript language in our impact analysis, our dynamic analysis module intercepts, transforms, and instruments the JavaScript code on-the-fly to collect execution traces. To collect a trace of function executions, the beginning and the end of each JavaScript function are instrumented. Function declarations are modified to collect traces of function invocation and passed arguments. We also trace function terminations and return statements (if they exist in the function). These traces are then used to create a dynamic call graph (DCG) that captures dependencies between function executions at runtime.

The DCG is an under-approximation of the call graph, while the SCG is an over-approximation of the call graph. The DCG contains fewer false positives compared to the SCG, and we augment it to capture DOM-related, event-driven, and asynchronous features of JavaScript, as explained below.

DOM-Sensitive Impact Analysis

A JavaScript function can impact a DOM element and vice versa, which is defined as:

▶ **Definition 10** (Direct Impact between JavaScript and DOM)**.** Consider a DOM element d ∈ D, and a function f ∈ F. Then f can directly impact d and vice versa:

$$\begin{cases} f \rightarrow_\tau d & \text{if } fW_\tau d \\ d \rightarrow_\tau f & \text{if } fR_\tau d \end{cases}$$

Furthermore, the impact can travel from function f to function g, through a DOM element d, under certain conditions as defined in the next definition.

▶ **Definition 11** (Indirect JavaScript Impact through DOM)**.** Consider two functions f, g ∈ F, and a DOM element d ∈ D. f can indirectly impact g through d, if and only if the following

conditions hold:

$$f \rightarrow_d g \text{ if } \begin{cases} fW_{\tau_1}d & \& \\ gR_{\tau_2}d & \& \\ \tau_2 > \tau_1 \end{cases}$$

In other words, function f can potentially impact function g through DOM element d, if f writes to d and g reads from the same element. Such a write-read (`WR`) pair indicates the existence of a potential impact between the two functions, if the read instruction happens after the write. Such `WR` pairs can occur subsequently, involving more elements and functions in the application. The reading function can itself write to a DOM element and augment the propagation path. The same change can then potentially impact all elements and functions that are on such a path.

To analyze how the DOM transfers the change impact (Theorem 11), all *read* and *write* accesses to the DOM need to be monitored dynamically. Each access is made from a JavaScript function to a DOM node, element, or an attribute, and through standard DOM API calls (e.g., `getElementById`, `querySelector`). We modify the prototype of the `Node`, `Document` and `Element` classes to be able to dynamically intercept DOM accesses, while preserving the original behaviour of these classes. This allows us to monitor changes to the structure of the DOM tree, as well as the existence, content, and attributes of DOM elements.

It is worth mentioning that the caller functions are extracted from the dynamic context of the intercepted DOM API calls. As a result, if a function does not exist or is not detected statically (e.g., it is created through an `eval` statement), it will still be captured in the dynamic phase if it interacts with other entities and is part of the execution.

For each function and DOM element involved in an access, we augment the dynamic call graph by adding two nodes (if they do not already exist), and connecting them through an edge representing the type of access that was made from the function to the element.

Event-Based Impact Propagation

TOCHAL also captures all event handlers called directly and indirectly through event propagation on the DOM tree.

Definition 12 summarizes the potential impact transmission between a DOM element and all event handlers that are called through event propagation.

▶ **Definition 12** (Indirect Impact via Propagation)**.** Let d be a DOM element that has an event handler for event e. Consider $prop_e[d]$ to be the set of all JavaScript functions that are submitted as handlers for event e to d or any of its ancestors in the DOM tree, and thus can be triggered by event propagation. Then d can impact all these handlers indirectly: $d \rightarrow prop_e[d]$.

Moreover, the dynamic analysis module yields information on function arguments and return values for all directly and indirectly invoked functions. These variables may differ from what has been declared in static function signatures, due to function variadicity in JavaScript.

XHR Relations

There are three main phases in the lifecycle of each XHR object: open, send, and response. These three phases can be scattered throughout the code. In addition, callbacks from the server-side could invoke other functions on the client-side, and hence it is not trivial to find the

Table 2 Impact transfer through different entities.

Assume $f, g \in F$ (functions), $d \in D$ (DOM) and $x \in X$ (XHR)	
Relation	**Description**
$fW_\tau g$	1. f calls g and passes arguments. 2. g calls f and f returns a value.
$fW_\tau d$	1. f creates element d, adds it to the DOM tree, deletes it, or detaches or relocates it from the DOM. 2. f modifies the content or the attributes of d.
$dR_\tau f$	1. f uses information regarding the content, attributes, or location of d in the DOM. 2. f is bound to d through an event handling mechanism. 3. f is set as an event handler of one of the ancestors of d, that can be triggered via event propagation.
$fW_\tau x$	1. f opens x as a new XHR object. 2. f sends a previously-created XHR object x.
$xR_\tau f$	1. f is set as the callback function of x 2. f sends a previously-created XHR object x.

XHR components statically. Our technique instruments and intercepts each component of an XHR object by wrapping around the XHR object of the browser. The gathered information is then used to augment the dynamic call graph. Similar to the previous features, new nodes representing XHR objects are added to the graph, the involved functions nodes are only added if they were not previously included, and the function nodes are linked to the XHR node based on the type of access they make to the XHR object. The access types are defined by the type of the interaction between a function f and an XHR object x, and determine the direction of the impact relation. For instance, if f creates and opens x, then $f \rightarrow x$, while if f is registered as the callback function of x, then $x \rightarrow f$.

Similar to the static call graph, we enhance the dynamic call graph using inter-procedural data-flow analysis. Arguments and return values of functions are used to trim the call graph where there is no data flow between two functions (Definition 9). However, instead of using the static function code, the dynamic arguments and return values are used to support function variadicity at run time.

4.3 Hybrid Model for Impact Analysis

At this stage, TOCHAL creates a system dependency graph by integrating the obtained static and dynamic call graphs. This graph is used for performing impact analysis on JavaScript applications. The dynamic part of the model contributes to the precision of the analysis, while its static features make it more complete. We take a best-effort approach for fulfilling soundness, following the soundiness manifesto [14]. In order to satisfy the practicality of our approach in terms of precision and scalability, complete soundness is not a concern of our approach. The hybrid model is represented as a directed graph. The vertices are system entities and the edges are the potential impact relations as summarized in Table 2.

Vertices. The vertices in the graph are all entities that are present statically or are created during the execution of an application. The vertices can take one of the following types:

- *JavaScript functions.* JavaScript functions extracted by our static analyzer (Section 4.1) are added as vertices. Functions found dynamically are added as well due to their involvement in the impact propagation, even if they are not connected directly (Section 4.2).

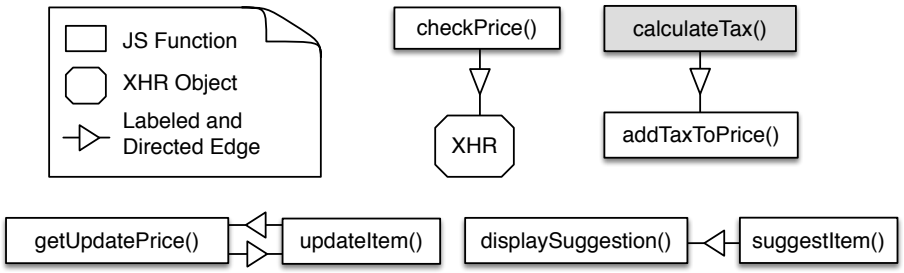

Figure 6 A static call graph, displaying the dependencies extracted from the running example (Figures 1 and 2).

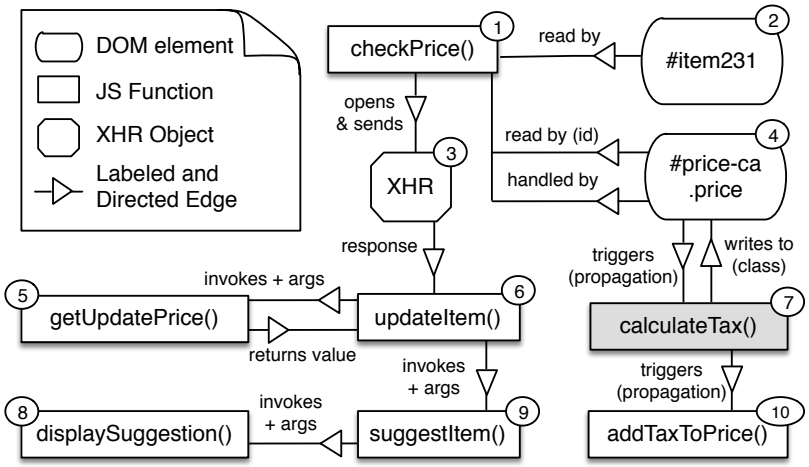

Figure 7 A sample hybrid graph, including the dynamic and DOM-sensitive dependencies extracted from the running example (Figures 1 and 2).

- *DOM elements.* The importance of DOM elements in transferring the impact dynamically was discussed in Section 4.2. Accessed DOM elements (and their contextual information) are captured as vertices in the model.
- *XHR objects.* XHR vertices incorporate information regarding the creation and sending of messages, as well as callback functions and data transmitted dynamically between the server and the browser.

Edges. The edges in the graph are labeled and directed.

- *Direction.* The direction of each edge depicts the flow of the data between two vertices. The edges are categorized into *read* and *write* accesses. An edge is directed from the vertex that writes (offers) the data to the vertex that reads it.
- *Labels.* The edge labels indicate the type of dependency relations that connect the vertices. Different labels are used to connect different vertices, since the valid operations vary for each category of vertices.

▶ **Example 13.** Figure 6 shows a dependency graph for the running example (Figures 1 and 2) obtained through the static analysis module alone. On the other hand, Figure 7 depicts a

Table 3 Impact Metrics

Entity	Metric	Description	CC with % WR DOM elem.
$d \in D$	$f_{in}(d)$	Number of functions f such that $fW_{\tau}d$	0.66
$f \in F$	$f_{in}(f)$	Number of elements d such that $fR_{\tau}d$	0.74
	$f_{out}(f)$	Number of elements d such that $fW_{\tau}d$	0.62
$\epsilon \in D or F$	$\widehat{L}[P]$	Average length of impact paths in the application	0.59
	$D_m(\epsilon)$	Minimum distance of ϵ from the change set	-

simplified hybrid graph utilizing both the static and dynamic analysis modules of TOCHAL. This is hard to do with the static graph. However, using the hybrid graph, one can find the potential impact set of a change in entity ε by tracing the graph forward, starting from ε.

Consider a case where a developer plans to make a change to the `calculateTax()` function (line 21 of Figure 1), and would like to find the potential impact before making the actual change. A DOM-agnostic static change impact analysis method (see Figure 6) would report that only `addTaxToPrice()` would be affected. The source code shows that next to the `addTaxToPrice()` function, DOM elements with class=`price` (lines 22–23 of Figure 1) can also be affected.

However, our hybrid DOM-sensitive analysis reveals that there exist more impact paths. The DOM element with id=`price-ca` is also a member of a class=`price` (box 4 of Figure 7). This element can thus be impacted by `calculateTax()`, and in turn can propagate the impact to `checkPrice()` indirectly (lines 3 & 26 of Figure 1, box 1 of Figure 7). Furthermore, evaluating the response of an XHR object, `checkPrice()` can then transfer the impact to `updateItem()` (box 6), which can propagate the impact to more functions (boxes 5, 8 & 9). To summarize, as our hybrid model shows, changing the `calculateTax()` function can affect six more elements in addition to the two elements that can be detected by statically analyzing the code. Thus, the proper impact set consists of functions `addTaxToPrice()`, `checkPrice()`, `updateItem()`, `getUpdatedPrice()`, `suggestItem()`, `displaySuggestion()`, the DOM element with id=`price-ca`, and the anonymous XHR object.

5 Impact Metrics and Impact Set Ranking

An impact set inclusive of the contributions of both static and dynamic analyses can become large and overwhelm the user. Considering that not all entities in the impact set are equally important, providing a ranking mechanism is essential for helping developers identify relevant impacted entities more efficiently.

We propose a set of *impact metrics* to estimate the importance of each entity in the produced impact set. The impact metrics, outlined in Table 3, are variables derived from the semantic and structural characteristics of the hybrid graph. These metrics can affect the probability of impact propagation, through DOM-related and dynamic mechanisms of JavaScript. Based on the impact metrics, we propose an impact ranking mechanism as outlined in Definition 14. The *impact rank* score of each entity ϵ, referred to as $IR(\epsilon)$, is an estimation of the importance of ϵ in propagating the change, relative to other elements in the impact set.

▶ **Definition 14** (Impact rank). Let ϵ be an entity in the impact set. Then the impact rank

of ϵ is defined as:

$$IR(\epsilon) = \frac{IR_{pr}(\epsilon) * \widehat{L}[P(\varepsilon)] * Fan_w(\epsilon)}{D_m(\epsilon)}$$

where, the value of the impact rank of an element in the impact set depends on four variables. (1) $IR_{pr}(\epsilon)$: the accumulation of impact ranks of immediate precedents of entity ϵ in the hybrid graph that are on an impact path from the change set to ϵ. If the impact is transferred to ϵ from entities with higher ranks, then the importance of ϵ in transferring the impact potentially increases. (2) $D_m(\epsilon)$: the shortest distance of ϵ from the change set in the hybrid graph. The closer ϵ is to the change set, the higher is the probability of being impacted by the change in practice. Hence, a longer distance of ϵ from the change set has a lesser effect on its impact rank. (3) $\widehat{L}[P(\varepsilon)]$: the average length of an impact path starting from entity ϵ. If ϵ can cascade the impact to more elements and deeper levels in the propagation graph, then ϵ is potentially an important entity in the impact set. (4) $Fan_w(\epsilon)$: the weighted fan-in / fan-out score of functions and DOM elements in the hybrid graph is indicative of the number of entities that can impact and be impacted by ϵ directly; hence, we consider it as a determining factor in impact rank determination. $Fan_w(\epsilon)$ is calculated as follows:

$$Fan_w(\epsilon) = \begin{cases} w_1 * f_{in}(\epsilon) & \text{if } \epsilon \in D \\ w_1 * f_{in}(\epsilon) + w_2 * f_{out}(\epsilon) & \text{if } \epsilon \in F \end{cases}$$

$\widehat{L}[P(\varepsilon)]$ and $Fan_w(\epsilon)$ are extracted from the findings of our exploratory study (section 3). During the course of the study, we measured the semantic and topological properties of hybrid graphs of ten web applications. Among the analyzed variables, the length of impact paths, fan-in of DOM element node, and fan-in and fan-out of function nodes (section (B) of Table 1) have a correlation with the number or ratio of DOM elements that can transfer the impact. Hence, these variables can affect the probability of impact propagation through an entity. They thus play a role in these two determining factors influencing the overall impact rank.

6 Tool Implementation: Tochal

We implemented Tochal using JavaScript libraries such as Esprima,[2] Estraverse[3] and Escodegen[4] for parsing and transforming the JavaScript code. We use these libraries to instrument functions in a manner that permits us to collect data about function invocations, function entries and function exits. The collected data also include dynamic values of functions' arguments and return values. External files are attached to the beginning of each document that allow instrumentation and interception of DOM events, XHR objects and timeouts. We use the Mutation Summary library [16] to detect JavaScript code appended to the DOM on the fly. Therefore, we can extend function instrumentation to the JavaScript code that gets created dynamically during application execution. We use WALA [23] to extract the static call graphs of applications.

Tochal provides an interface for developers to utilize our hybrid change impact analysis. The main goal is to facilitate the comprehension of change impact "during" development and debugging activities. Hence, the analysis needs to be available to developers while they make

[2] http://esprima.org
[3] https://github.com/Constellation/estraverse/
[4] https://github.com/Constellation/escodegen/

changes to their application. We have integrated our impact analysis method within Google Chrome's DevTools [10], a popular web development environment. This decision entails a number of benefits, namely (1) the approach is complementary to existing web development platforms and environments; it does not change the functionality they provide, but augments their capabilities. (2) The developer can perform the impact analysis in the same context as the code, and can preserve her mental model of the code. (3) The developer is not required to learn a new tool, or divide her attention between two different tools.

The interface allows the user to select JavaScript functions (including XHR callbacks) or DOM elements as the change set, and then perform the impact analysis. Chrome's DevTools includes a set of panels, each providing a window to a subset of functionalities that the platform provides. Two panels are of more interest for us: "elements" and "sources". The elements panel visualizes and provides inspection mechanisms on the DOM. The sources panel displays all of the JavaScript code that contributes to the application. We add a sidebar to each of these panels, allowing the user to invoke the CIA unit, on a selected entity, at any stage of the development. TOCHAL is publicly available for download [24].

7 Evaluation

We empirically evaluate the effectiveness and usefulness of TOCHAL through the following research questions:

RQ1 How does our hybrid method compare to static/dynamic analysis methods?
RQ2 Does TOCHAL help developers in performing change impact analysis in practice?

We address these questions through two studies, each described in the following two subsections, respectively.

7.1 Study 1: Comparing Static, Dynamic, and Hybrid Analyses

To address RQ1, we conduct a study to evaluate the impact sets extracted using TOCHAL in comparison with those detected by static and dynamic analysis techniques separately. We compare TOCHAL with a state-of-the-art static analysis technique. We also examine the differences in the outcomes of TOCHAL with those of the dynamic analysis unit of TOCHAL. The term *dynamic* analysis encompasses the DOM-sensitive, dynamic, event-driven, and asynchronous analysis performed by TOCHAL. Our first hypothesis is that TOCHAL outperforms static impact analysis due to its support for *dynamic* analysis. We also hypothesize that while dynamic analysis is a significant part of Tochal, it is outperformed by tochal's hybrid analysis.

We decided to compare TOCHAL with static and dynamic approaches, since to the best of our knowledge, TOCHAL is the first change impact analysis technique for JavaScript and there is no similar tool available for JavaScript.

Design and Procedure

The only entities that can be analyzed by both static and *dynamic* analysis methods are JavaScript functions. Hence, to be fair to both static and dynamic analyses, we configure TOCHAL to only deliver functions in the impact sets. TOCHAL's hybrid model and analysis, however, do not differ from what is described in section 4 and use DOM-based, dynamic and asynchronous entities and relations to extract the functions in the impact sets.

Table 4 Results of comparison between static, dynamic and TOCHAL (RQ1) (A) Comparison of impact sets (B) Comparison of functions in system dependency graphs

Application	(A) Impact sets											(B) Functions				
	Tochal			Static			$\frac{\text{Static}}{\text{Tochal}}$	Dynamic			$\frac{\text{Dynamic}}{\text{Tochal}}$	Tochal	$\frac{\text{Static}}{\text{Tochal}}$	$\frac{\text{Dynamic}}{\text{Tochal}}$	Pure Static	Pure Dynamic
	avg	min	max	avg	min	max	%	avg	min	max	%	avg	%	%	%	%
same-game	4.33	2	7	0.67	0	1	15	3.67	2	6	85	16	56	93	6	44
ghostBusters	1.33	0	2	0.33	0	1	25	1.33	0	2	75	10	80	100	0	20
simple-cart	1.67	0	3	0.67	0	1	40	1.67	0	3	100	44	74	91	9	26
mojule	1.00	0	2	0.33	0	1	33	0.67	0	2	67	21	24	90	10	71
jq-notebook	2.67	0	7	0.67	0	2	25	2.33	0	6	87	19	47	100	0	53
doctored.js	1.67	0	4	0.33	0	1	20	1.33	0	3	80	38	67	50	56	33
jointlondon	2.33	0	5	0.67	0	1	29	1.67	0	4	72	36	31	85	15	69
space-mahjon	2.67	1	5	1.00	0	2	37	2.00	0	5	63	27	56	93	7	44
listo	1.33	1	2	0.00	0	0	0	1.33	1	2	100	12	75	58	25	42
peggame	2.67	1	6	1.00	1	1	37	2.00	0	5	75	24	83	75	25	17
Average	**2.07**	0.5	4.3	**0.57**	0.1	1.1	**26**	**1.8**	0.3	3.9	**80**	**25**	**59**	**84**	**15**	**42**

We use the same set of subject applications from our exploratory study of section 3 (Table 4, column 1). For each application, we randomly sample three functions and extract their impact sets using each of the methods. We compare the impact sets to assess the completeness of the outcomes of analysis with each approach. The sample functions are selected from a pool of functions that are recognized by all three analysis methods. In other words, static and *dynamic* analysis alone are not able to detect some functions that are detected by TOCHAL and are involved in its hybrid analysis. If static/*dynamic* analysis is performed on any of the functions it does not recognize, the impact set will be empty. We aim at comparing impact sets at this stage and these functions are unable to provide useful information regarding the analysis. Thus, we do not include such functions in the comparison. However, this indicates the need for an investigation of the functions involved in each type of analysis (in the dependency graphs), as described in the next paragraph.

We measure the number of functions that are included in the dependency graphs of each analysis, contributing to the detection of the impact sets. The average number of functions in each type of analysis denotes the extent of the analysis performed for extracting the impact sets. Moreover, the lower number of recognized functions by an analysis method means that there are more functions for which the method is unable to perform CIA.

Pure Static Analysis. We use the static analysis part of our approach, which is built using WALA [23]. WALA is a leading static analysis tool for JavaScript, used by many other JavaScript analysis techniques [26, 27, 22]. It should be noted that WALA by itself does not support change impact analysis. For the purpose of this evaluation, we extended and directed it towards performing static impact analysis to conduct the comparisons.

Pure Dynamic Analysis. We disable the static module of TOCHAL and only utilize the *dynamic* analysis module. The applications are exercised through their test suites when available and manually within multiple sessions and the results are integrated. Each manual session follows a set of pre-defined scenarios that covers all main use-cases of the application that are accessible to an end-user.

Hybrid Analysis. We use the hybrid model of TOCHAL for performing impact analysis and obtaining the set of functions that are involved in the hybrid analysis.

Results and Discussion

To answer RQ1, we discuss the outcomes of the study, summarized in table 4.

Completeness of Impact Sets. Section (A) of Table 4 depicts the results of the impact set detection using static, *dynamic* and hybrid analysis methods. The first column of this section displays the average, minimum and maximum sizes of the impact sets of the selected JavaScript functions, detected by TOCHAL. The second column displays the average, minimum and maximum impact set size by static analysis. The third column represents the percentage of the ratio of the impact set size of static analysis to that of TOCHAL. Similarly, the fourth and fifth columns, respectively, show the impact set size for *dynamic* analysis, and the ratio of the size of impact sets using *dynamic* analysis compared to TOCHAL. We observe an increase in completeness for all applications in favour of TOCHAL. On average, static and *dynamic* analysis methods detected 0.57 and 1.80 functions in the impact set of each sample function, respectively. The hybrid TOCHAL method, however, extracted an average of 2.07 functions to be potentially impacted by each of the sample functions.

Overall, the impact sets extracted by TOCHAL include 74% more functions on average compared to those detected by static impact analysis. The outcome conforms the findings of our earlier exploratory study, showing the prevalence and importance of DOM-related and dynamic characteristics of JavaScript in impact analysis. TOCHAL takes into account new types of entities in its dependency graph that are more aligned with the nature of JavaScript (DOM elements, XHR objects). It also recognizes new (and mostly hidden) types of relations between these entities, that lead to more complete and more precise dependency graphs and impact sets at the same time. The static analysis still remains useful in TOCHAL since *dynamic* analysis can only cover 80% of the impact sets detected by hybrid analysis and cannot replace it.

It is worth noting the small sizes of static impact sets. Considering the conservativeness of static methods, the dependency graphs in general can include many relations between functions that are not feasible in practice. Therefore, static impact sets in CIA methods for traditional languages are expected to get large and contain infeasible relations. Our results show an *opposite phenomenon* for JavaScript applications. The small sizes of static dependency graphs and the resulting impact sets attest to the difficulties and limitations of static analysis for JavaScript. The findings further confirm that new forms of definitions and usages of functions, objects, DOM elements and asynchronous objects negatively affect analysis of useful dependency graphs. Static analysis confronts more barriers during the analysis JavaScript applications that should be mitigated using an approach that supports these features.

Necessity of Hybrid Analysis. Separate data sets are collected from each type of analysis, to let us distinguish between the statically detected and dynamically invoked functions. Through these datasets, we extract the functions that were invoked dynamically but were not detected statically, and the functions that were extracted before the execution, but did not play a role at runtime. The results are summarized in section (B) of Table 4. The first column of this section contains the total number of functions that were included in our hybrid analysis. The second and third columns represent the percentages of these functions that were covered by static and *dynamic* analysis units, respectively.

The static analysis method only covers about 56% of the functions that are covered during hybrid analysis. As expected, this inadequacy is caused due to the dynamism of JavaScript even in simple function invocations. Moreover, the *dynamic* analyzer includes 86% of the functions detected by the hybrid analysis. This confirms our anticipation of incompleteness of dynamic analysis, due to its reliance on specific executed scenarios of the code. The increase in the covered JavaScript functions by our proposed hybrid analysis leads to a more

complete system dependency graph, which is used to perform the impact analysis. Hence, the proposed hybrid approach can improve the accuracy of JavaScript impact analysis.

Column 4 in section (A) of Table 4 represents the percentage of JavaScript functions that were detected by the static analyzer, but were *not* invoked during any of the executions of the applications. Note that the existence of such functions, that would be missed from pure *dynamic* analysis, is sufficient evidence for the necessity of a hybrid analysis technique. Column 5 of the same section of the table, on the other hand, displays the percentage of functions that were executed during the execution of each application, but were *not* detected by the static analyzer. The results confirm our previous intuition regarding the shortcomings of static JavaScript analysis, even in a well-established type of analysis based merely on function declarations and invocations.

7.2 Study 2: Industrial Controlled Experiment

We conducted a controlled experiment [28] in an *industrial* environment[5] to address the following two questions, derived from RQ2:

RQ2.1 Does TOCHAL increase the CIA task completion *accuracy*?
RQ2.2 Does TOCHAL decrease the CIA task completion *duration*?

Experimental Subjects

We recruited 10 participants, 5 female and 5 male, with ages ranging from 20 to 34. At the time of conducting the experiment, all participants were employed in a large software company in Vancouver. Their skills in web development ranged from medium to professional. All participants volunteered for taking part in our experiment and did not receive monetary compensation.

Experimental Design

The experiment had a "between-subject" design to avoid carryover effects. We formed two independent groups of participants. The experimental group used TOCHAL for performing the tasks; none of the participants were familiar with TOCHAL prior to attending the experimental sessions. Since TOCHAL is the first CIA tool for JavaScript applications, the control group performed the tasks using the development tool they used in their day-to-day web development activities (without TOCHAL). To avoid bias, it was important for the members of the two groups to have similar proficiency levels. We manually assigned the participants to the groups to ensure this was the case based on a pre-questionnaire evaluating their experience and expertise.

Tasks. We designed a set of four tasks that involved finding change impacts during web application maintenance activities. The tasks, outlined in Table 5, require the participants to detect and comprehend the potential impact of a change in a JavaScript function or a DOM element. Moreover, two of the tasks require participants to use their understanding of the change impact to find a bug or an inconsistency that occurs after the change.

All features of TOCHAL were available to the experimental group during the experimental session. We were particularly interested in assessing the usefulness of the ranking mechanism

[5] The experimental material is available at: `http://ece.ubc.ca/~saba/tochal/study-materials.zip`

Table 5 Impact analysis tasks used in the controlled experiment.

Task	Description
T1	Finding the potential impact of a DOM element (the button changing the size of the displayed slideshow images)
T2	Finding the potential impact of a JavaScript function (the function toggling the play/pause state of the slideshow)
T3	Finding a conflict after making a new change (problem in submitting new comments after changing the table containing all comments of a picture). Ranking is disabled.
T4	Finding a bug in JavaScript code (entered email format is not properly checked)

of TOCHAL, which was deployed based on our proposed impact metrics (Section 5). Hence, we designed a smaller study that enabled us to compare the effects of using the ranking. However, this comparison was only meaningful for the experimental group, who used TOCHAL and had access to its ranking feature. Having this in mind, the two debugging tasks, tasks T3 and T4 in Table 5, were designed to have similar levels of difficulty and to require similar amount of effort and expertise. However, we disabled the ranking feature for T3, while leaving it enabled for T4. These two tasks were counterbalanced to avoid order effects.

Dependent and Independent Variables. We measured two continuous variable as our dependent variables. Task completion *duration* was measured in minutes and seconds. Tasks completion *accuracy* was measured in marks based on a fixed and predefined grading rubric, and the marks were converted to percentages for consistency across all tasks.

The independent variable is a nominal variable including two levels. One level represents using TOCHAL, and the other level depicts using a different tool (e.g., Chrome DevTools, Firefox Developer Tools, or Firebug).

Experimental Object

We used Phormer [18], an online photo gallery in PHP, JavaScript, CSS and XHTML, which consisted of around 6,000 lines of code and over 38,000 downloads at the time of conducting the experiment. Throughout the experiment, the participants had to understand and debug parts of the application related to displaying a slideshow of pictures, viewing the pictures, and authoring comments.

Experimental Procedure

The experiment consisted of three main phases.

Pre-Experiment. The participants were given a pre-questionnaire form before attending the experimental session. They were required to provide information regarding their proficiency and experience in web development and software engineering. This information was used for assigning them into one of the two groups prior to the session. The pre-questionnaire also inquired about the tools the participants normally used for performing their every-day web development tasks. The answers to this question were used to determine the proficiency levels of the participants for assigning them into the control and experimental groups. The information was also used to indicate the tool the control group would use for the experiment. It is worth mentioning that all participants of the control group selected Google Chrome as their preferred web development tool. In the experimental session, the participants were introduced to TOCHAL for the first time and were trained to use it. Then they were given a few minutes to familiarize themselves with the tool, and ask us questions if needed.

Tasks. During the experimental session, the participants were asked to perform a set of tasks, as indicated in Table 5. To avoid experimental bias, each task was handed out on a separate sheet of paper to the participant, when the investigator marked the start time of the task. The investigator terminated the measurement of the task duration when the participant returned the paper to the investigator along with her answer. The task accuracy was marked after the session, using a rubric created prior to conducting the experiment.

Post-Experiment. We asked the participants to fill a post-questionnaire form after completing the tasks. The questionnaire contained questions regarding both the helpful features and the shortcomings of the tool they used in the experiment. Moreover, we enquired about the metrics our participants considered to affect the importance of an entity in the impact set.

Results and Discussions

Figures 8 and 9 depict the results of task completion duration and accuracy for both experimental and control groups.

Accuracy (RQ2.1). We ran the Shapiro-Wilk normality test on the accuracy data. The results showed that the data collected for tasks T1, T3 and T4 were not normally distributed and hence, Mann-Whitney U tests were used for analyzing the results of these tasks. The data gathered for task T2 and the total accuracy of the tasks were normally distributed and were analyzed using t-tests. The results of conducting the tests revealed a statistically significant difference for the experimental group using TOCHAL (Mean=84%, STDDev=14%) compared to the control group (Mean=26%, STDDev=11%); (p-value=0.0001). *Overall, participants using* TOCHAL *perform 223% more accurately compared to the control group, across all tasks in the experiment.*

Further, the accuracy for all tasks was higher when the participants used TOCHAL. The improvement was statistically significant for tasks T1, T2 and T4, but not T3. Recall from the tasks table (Table 5), that the ranking mechanism of TOCHAL was disabled for T3. This outcome thus emphasizes the value of ranking the impact set in helping the user find the important impacts more efficiently. Not having access to the impact ranks, participants had to expend more manual effort. Enforcing more analysis burden on the participant increases the variation in the answers based on the individual differences in abilities of the participants. The high variation prevents the statistical significance of the results of T3, in spite of the 60% higher accuracy average for TOCHAL users.

Duration (RQ2.2). We first used the Shapiro-Wilk test on the duration data and confirmed that the data was normally distributed. Therefore, we ran a set of t-tests on all individual tasks, as well as on the total time spent on all of the tasks throughout the experiment. The results show a statistically significant difference in the total duration for the experimental group using TOCHAL (Mean=19:54, STDDev=1:23), compared to the control group using other tools (Mean=35:26, STDDev=8:21); (p-value=0.01). *Overall, participants using* TOCHAL *spent an average of 78% less time on the same set of tasks compared to those using other tools.*

Further, the participants using TOCHAL showed improvements for all of the tasks compared to the control group, with the difference being statistically significant for T1. Considering the higher accuracy scores for other tasks when using TOCHAL, we observe that many participants of the control group terminated the tasks with incomplete and incorrect answers,

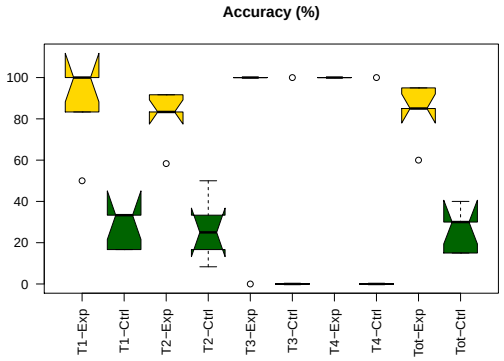

Figure 8 Task completion accuracy per task and in total for the control (ctrl) and experimental (exp) groups (RQ2.1).

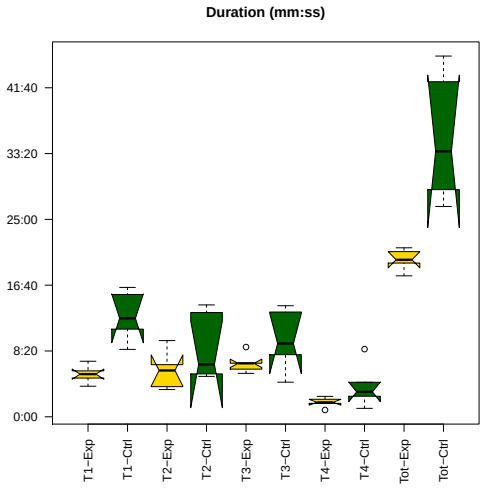

Figure 9 Task completion duration data per task and in total for the control (ctrl) and experimental (exp) groups (RQ2.2).

erroneously assuming that they had completed the task. This caused them to obtain lower accuracy scores, while still spending more time on average.

Ranking. The results of the experimental group were analyzed further to investigate the effects of using our proposed ranking system on the duration and accuracy of understanding and debugging an application after a change. T3 and T4 were both debugging tasks requiring similar levels of expertise. As mentioned earlier, the use of Tochal's ranking feature was disabled for T3, while it was enabled for T4. Performing a t-test on the results revealed a statistically significant difference in task completion duration between participants who used the ranking mechanism (Mean=1:50, STDDev=39), compared to those who did not (Mean=6:34, STDDev=1:16), with p-value<0.05. We used a Mann-Whitney U test to analyze the accuracy results for the ranking mechanism. Although using the ranks led to an average of 20% higher accuracy of the answers, the results were not statistically significant in this case. However, the participants completed T4 about 3.7 times faster than T3. This significant improvement highlights the importance of ranking the impact set, and is an indication of the usefulness of our impact ranking method (section 5).

Participants' Feedback

We gathered qualitative data from both experimental and control groups through a post-questionnaire form. The questionnaire asked participants about the usefulness of the tool used in the study, its strengths, and its shortcomings. Overall, all Tochal users mentioned that they found the tool useful. The participants were particularly pleased with the idea of finding the potential impact of JavaScript functions and DOM elements. Understanding the dynamic behaviour and underlying dependencies were mentioned to be most useful. The users found Tochal to be helpful in solving the problems faster, especially in the presence of its ranking mechanism. The participants in the experimental group were also interested to see more features in Tochal. The feature requests were mostly attributed to improving the user interface. Some participants were also interested in having direct debugging support in

the tool.

Furthermore, we asked our participants about the metrics they thought could determine the importance of an entity in the impact set. We analyzed and categorized the participants' opinions on impact metrics. A few of the final categories were considered in our ranking strategy, as we expected. These metrics included (1) distance of an entity from the change set and (2) number of dependencies of an entity. However, there were many other categories that are not included in our methods, such as: (1) number of invocations of a function, (2) visibility of the impact in the interface, (3) importance of the feature to the customers, (4) "breadth" of usage of the entity: whether it is used by multiple files, or is isolated to one file (5) complexity of the function, and (6) "history" of a function: rate of having faults in the function.

Threats to Validity

The first internal threat is the the population selection threat, and specifically the equivalency of the two groups in terms of their expertise. We addressed this threat by first manually dividing the participants into different proficiency levels, which were extracted from the information gathered in the pre-questionnaire forms. We then distributed the members of each level into control and experimental groups by random sampling. The second threat is the investigator's bias while marking the accuracy of the answers. We mitigated this threat by creating a marking rubric while designing the tasks, and using the same rubric for marking the results later. A similar threat can arise from the bias in measuring the duration of the tasks. We resolved this issue by enforcing a mutual supervision on time measurement by both the investigator and the participant. We assigned a separate sheet of paper to each task, which was handed to the participant in the beginning of the task, and was returned to the investigator after task completion. The fourth threat can be introduced by the choice of the tool that the control group used. This threat was mitigated by allowing the control group to choose the tool of their preference. Finally, the compensatory incentives were not a threat to validity as all of our participants volunteered for the experiment, with no monetary compensation.

An external threat is with regard to the representativeness of our sample of population. We mitigated this threat by recruiting professional web developers from industry as our participants. Another concern is raised regarding the representativeness of tasks used in the experiment. We used general tasks enquiring about understanding the impact of a code change and also detecting potential faults in the code, which are faced by developers in their daily professional activities.

8 Related Work

Static Analysis. Static CIA is performed by analyzing the source code without executing it. A common pattern in many traditional CIA techniques is the usage of dependency-based impact analysis methods [4]. Static impact analysis techniques typically find syntactic dependencies by performing forward slicing on the graph. This type of analysis, however, is based on assumptions made for all possible executions of the software, and hence incurs false positives, which hinders its adoption [11]. The dependency graph can become large and may contain invalid paths. Hence, the resulting impact set can be large and difficult to comprehend.

More recently, static analysis has been applied to analyze JavaScript applications. Sridharan et al. [22] adapt traditional points-to analysis for JavaScript through correlation

tracking of dynamic properties in the code. Jensen et al. [12] statically model the role of the DOM and browser in their analysis. However, they acknowledge gaps and shortcomings in their analysis, which can result in many false-positives. Feldthaus et al. [8] present an approach for constructing approximate JavaScript call graphs. However, their analysis completely ignores dynamic property accesses and interactions with the DOM. Madsen et al. [15] combine pointer analysis with use analysis to investigate the effects of JavaScript libraries and frameworks on the applications' data flow.

These techniques neglect the dynamic DOM interactions as well as event-driven, and asynchronous features of the JavaScript language. Therefore, their analysis can be incomplete for performing change impact analysis for JavaScript applications.

Dynamic Analysis. Existing dynamic methods produce a precise but incomplete analysis. Apiwattanapong et al. [2] propose a dynamic technique in which execute-after relations are used to reduce the overhead caused by the amount of collected dynamic information. Dynamic CIA tools have been applied to various fields of software engineering. For instance, Ramanathan et al. [19] avoid testing unchanged test cases by comparing strings of different traces in their tool. Chianti [20] is a CIA tool for Java that reports the change impact in terms of the subset of the test suite affected by the change. These techniques provide more precision compared to static analysis, especially when integrated with other techniques such as information retrieval [6][9]. However, they do not target JavaScript code and its unique analysis challenges, such as DOM interactions, dynamic function calls involving event propagations, and asynchronous callbacks.

Wei and Ryder's recent JavaScript blended analysis approach [26] and state-sensitive points-to analysis [27] are perhaps the closest to our work. Their work integrates the information gathered during both static and dynamic analyses to perform a points-to analysis of JavaScript applications. However, their methods do not focus on analyzing the change impact, and hence do not incorporate the dependencies that are formed through DOM interactions and asynchronous JavaScript mechanisms. Moreover, they do not take into account the important role of events and event propagation [25] on the DOM tree, which connects JavaScript functions, unlike our analysis which does.

9 Concluding Remarks

The dynamic, asynchronous, and event-based nature of JavaScript and its interactions with the DOM make modern web applications highly interactive and responsive to users. These same features also introduce new types of dependencies into the system, making the prediction and detection of change impact challenging for developers. In this paper, we proposed an automated technique, called Tochal, for performing a hybrid DOM-sensitive change impact analysis for JavaScript. Tochal builds a novel hybrid system dependency graph, by inferring and combing static and dynamic call graphs. Our technique ranks the detected impact set based on the relative importance of the entities in the hybrid graph. Our evaluation shows that the dynamic and DOM-based JavaScript features occur in real applications and can lead to significant means of impact propagation. Furthermore, we find that a hybrid approach leads to a more complete analysis compared with a pure static or dynamic analysis. Finally, our industrial controlled experiment shows that Tochal increases developers' performance, by helping them to perform maintenance tasks faster and more accurately. In future work, we plan to extend the granularity of the analysis to intra-procedural JavaScript dependencies and explore more effective ranking functions.

References

1 Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

2 Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 432–441. ACM, 2005.

3 Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996.

4 Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

5 Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 55–65. IEEE, 2006.

6 Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. Impactminer: A tool for change impact analysis. In *Companion Proceedings of the International Conference on Software Engineering (ICSE)*, pages 540–543. ACM, 2014.

7 Document Object Model (DOM). `http://www.w3.org/DOM/`.

8 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

9 M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 430–440. ACM, 2012.

10 Google. DevTools: The Chrome Developer Tools. https://developer.chrome.com/devtools.

11 Lile Hattori, Dalton Guerrero, Jorge Figueiredo, Joao Brunet, and Jemerson Damásio. On the precision and accuracy of impact analysis techniques. In *Proceedings of the International Conference on Computer and Information Science*, pages 513–518, 2008.

12 Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 59–69. ACM, 2011.

13 Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

14 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

15 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM, 2013.

16 Mutation summary. `https://code.google.com/p/mutation-summary/`.

17 Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 10–19. IEEE, 2009.

18 Phormer PHP photo gallery. `http://p.horm.org/er/`.

19 Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 241–252. IEEE, 2006.

20 Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 432–448. ACM, 2004.

21 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010.

22 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. Springer, 2012.

23 T. J. Watson Libraries for Analysis. WALA. http://wala.sourceforge.net/.

24 Tochal. `https://github.com/saltlab/tochal`.

25 W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000.

26 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013.

27 Shiyi Wei and Barbara G Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26. Springer, 2014.

28 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer, 2000.