FeedLack Detects Missing Feedback in Web Applications

Amy J. Ko and Xing Zhang
The Information School | DUB Group | University of Washington
{ajko, xingz2}@uw.edu

ABSTRACT
While usability methods such as user studies and inspections can reveal a wide range of problems, they do so for only a subset of an application's features and states. We present FeedLack, a tool that explores the full range of web applications' behaviors for one class of usability problems, namely that of missing feedback. It does this by enumerating control flow paths originating from user input, identifying paths that lack output-affecting code. FeedLack was applied to 330 applications; of the 129 that contained input handlers and did not contain syntax errors, 115 were successfully analyzed, resulting in 647 warnings. Of these 36% were missing crucial feedback; 34% were executable and missing feedback, but followed conventions that made feedback inessential; 18% were scenarios that *did* produce feedback; 12% could not be executed. We end with a discussion of the viability of FeedLack as a usability testing tool.

Author Keywords
Feedback, program analysis, static analysis, JavaScript.

ACM Classification Keywords
H.5.2. Information interfaces and presentation.

General Terms
Human Factors, Algorithms

INTRODUCTION
Sometimes computers ignore us. We click save buttons, but often do not know if our documents are saved; we click on links in web pages, but are taken nowhere; we submit forms, but do not know if the site is broken, or simply slow to respond. Software that appears to ignore user input violates a basic principle of effective user interface design: for every user input, software should produce a corresponding output that explains how the system responded to the input.

The importance of this principle is reflected in the methods we use to detect feedback problems. For example, Nielsen's Heuristic Evaluation [16] focuses evaluators on feedback, stating that "The system should continuously inform the user about what it is doing and how it is interpreting the user's input." Inspection techniques such as the Cognitive Walkthrough [4] have evaluators confirm that the result of taking some action results in visible feedback. Task-based usability testing [16] can also reveal missing feedback in prototypes of widely ranging fidelity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2011, May 7–12, 2011, Vancouver, BC, Canada.
Copyright 2011 ACM 978-1-4503-0267-8/11/05....$10.00.

Figure 1. By analyzing paths from input, FeedLack found that function `post()` does not produce feedback upon success.

While these empirical methods are quite effective at detecting feedback issues, they often overlook problems in outside the scope of the tasks selected by evaluators [14]. Moreover, because they require users, these methods operate at a slower pace than other forms of software testing such as unit and regression testing, which run on the order of minutes and hours, not weeks. This disparity in scope and speed means that feedback issues and other usability problems can easily escape notice as code is readied to deploy.

To address this problem, we present *FeedLack*, a tool that automatically detects missing feedback in web applications. It does this by verifying that all paths originating from user input produce some form of output. To illustrate, consider the FeedLack warning in Figure 1. FeedLack has found that when the user submits the form (lines 22 and 23) and its comment is considered valid (line 10), the application provides no feedback about success or failure. It found this by enumerating all of the paths from form submission and reporting the single path that lacked output-affecting code.

Most up-to-date version: 06/25/2021

In the rest of this paper, we describe FeedLack's analysis in detail. We then present an evaluation of FeedLack on a corpus of 330 web applications ranging from small personal web pages to sophisticated applications such as calendars, visualizations, and games. Of the 129 that contained JavaScript input handlers and did not contain syntax errors, 115 were successfully analyzed, resulting in 647 warnings; 36% of these were legitimate, reproducible scenarios that needed feedback, while another 34% were missing feedback, but would likely not be confusing because they followed user interface conventions. We end with a discussion of FeedLack's limitations, its generalizability to other platforms, and the viability of its role in user-centered software engineering processes.

RELATED WORK

Feedback has been a central topic in HCI research and practice for several decades. Ensuring that it is timely and understandable is one of the major heuristics in Nielsen's Heuristic Evaluation [16], it is a foundational concept in Norman's gulf of evaluation [17], and it is much of the basis for the cognitive account of direct manipulation [9].

Empirical methods for detecting missing or problematic feedback come in several forms. One of the most common is usability testing, in which evaluators devise tasks and engage representative users to attempt them. Such testing can reveal feedback issues in user interface prototypes of varying fidelity. There have been several attempts to automate data capture and analysis aspects of usability testing [10], including remote usability testing [6] and logging techniques [13]. For example, recent work by Akers et al. sought to identify usability problems by analyzing logs of undo commands from real use, revealing a number of actionable usability problems [1].

Another approach to detecting missing feedback is using inspection methods. For example, the Cognitive Walkthrough [4] has evaluators ensure that each action is not only visible and apparent, but that the result of user actions produces visible feedback. These techniques are powerful in their ability to assess both the quality and presence of feedback.

The approaches most closely related to FeedLack are automated analyses of the user interface source code. For example, basic HTML validators are capable of finding feedback problems, in that malformed HTML may often not render properly or at all. JSLint (http://jslint.com) also finds common JavaScript defects that may cause silent failures in web browsers. Other validation tools have been developed to assess the accessibility of web sites against government guidelines; for example, the Functional Accessibility Evaluator (http://html.cita.illinois.edu/iitaa.php) checks web sites against several hundred accessibility rules by inspecting the structure and content of HTML. Similarly, Mahajan and Shneiderman explored automated consistency checking tools, evaluating the consistent use of vocabulary, capitalization, type face, and color in user interfaces [15].

Considering program analysis more generally, there is a long history of verification research focusing on software qualities other than usability. For example, a recent system by Artzi et al. [2] executes and analyzes PHP programs to find scenarios that generate malformed HTML, based on a standard HTML validator. Other recent program analysis and testing-based approaches detect scenarios that may lead programs to crash [8], hang [5], leak memory [18], or expose security vulnerabilities [7]. Our work complements these approaches by analyzing a program's feedback.

DESIGN AND IMPLEMENTATION

The goal of FeedLack is to find control flow paths through web applications that begin with some user input but fail to produce any change to the web page's appearance (evaluating the *quality* of that feedback, for example, whether it was timely, visible, or comprehensible, is out of scope). To do this, FeedLack finds all functions that handle user input, explores all paths through these functions, and identifies which of these paths lack output-affecting code.

We divide our discussion of this analysis into ten steps:

1. Identifying and naming functions
2. Generating function control flow graphs (CFGs)
3. Propagating type information
4. Resolving function calls
5. Identifying output-affecting statements
6. Identifying input-handling functions
7. Enumerating paths through input handlers
8. Expanding paths through input handlers
9. Identifying output-lacking paths
10. Clustering output-lacking paths

A major decision underling these steps was whether to use *static* analysis (analyzing code without executing it), *dynamic* analysis (analyzing executions of code), or a combination of the two. Static analyses have the benefit of verifying properties of a program independent of its inputs, considering the full breadth of a program's behaviors. They can also be much less precise, however, because they must make assumptions about what inputs and program states are actually possible or likely. Dynamic analyses avoid such limitations by using real inputs, but in doing so, sacrifice breadth. Some analyses combine static and dynamic information [2]. We decided on a pure static analysis FeedLack, primarily to complement the empirical nature of usability methods. We chose not to use dynamic information to avoid the need for complex testing configurations. For example, by using only static information, FeedLack can analyze the feedback of server transactions (e.g., creating user accounts and changing passwords) without communicating with a real server.

In the rest of this section, we describe the static analyses in the 10 steps above in detail. We use the example in Figure 1 to illustrate these steps.

Step 1: Identifying and Naming Functions

FeedLack requires as input a folder containing all of the JavaScript and HTML files necessary to run the client-side user interface of the web application. It does *not* require server-side code, even when such code is responsible for generating feedback, since client-side scripts are the only scripts capable of *presenting* feedback to users.

The first step in FeedLack's analysis is to find JavaScript code. It looks in three places: (1) JavaScript source files ending in ".js", (2) `<script>` tags in HTML, and (3) attribute values that compile as JavaScript without parsing errors (as in `<div onclick="alert('error')>"`). All code is parsed using the Rhino JavaScript parser (`http://www.mozilla.org/rhino/`), generating a set of abstract syntax trees (ASTs).

From the ASTs, FeedLack identifies nodes representing JavaScript functions. Because the names of functions are particularly important for exploring paths through JavaScript functions, FeedLack makes extensive efforts to find the names by which a function is referred by considering the contexts of the function's declaration and uses. JavaScript allows developers to declare functions in a variety of contexts, including standard declarations (`function open() {...}`), inside object literals: (`{ open: function() {...}}`), as local variables (`var open=function() {...}`), or as arguments (`enable(function() {...})`). In the first three cases, the names can be extracted quite easily; in the last case, no name is extracted. However, because functions can be used as values, they can take on multiple names. For example, the function `open` in the examples above might be assigned to properties as in `element.onclick = open`, enabling a developer to call it as `element.onclick()`. As described in Step 3, FeedLack analyzes the assignments and references to variables, detecting additional names by which functions can be referred in the process.

As part of this naming process, FeedLack also classifies functions as one of four kinds, to help later determine what functions a call might invoke in Step 4. Functions that are declared at the script level are classified as `GLOBAL` and are presumed to be reachable from any function. Functions declared in object literals or assigned to a property (as in `this.open=function() {...}`) are classified as `OBJECT` functions and are presumed to be reachable only by calls on objects. Functions declared inside of functions are classified as `LOCAL` and are presumed to be reachable only from within the function, unless FeedLack finds references to these functions in function calls or return statements (meaning the function can escape the local scope). In these cases, the local function is given the type `CALLBACK`. All other functions are given the type `CALLBACK` and are not considered in determining what functions a call might invoke.

The above classifications exploit well-documented patterns in how developers use JavaScript functions [19], but they do not cover all possible uses. Our hope was that detecting these patterns would be sufficient for detect missing feedback, with the understanding that they would be one source of false positives in FeedLack's warnings.

Step 2: Generating Control Flow Graphs of Functions

The next step in FeedLack's analysis is to convert each function's AST into a control flow graph (CFG), representing the flow of execution through the function. FeedLack uses these throughout its subsequent analyses.

To create a function's CFG, FeedLack starts with the function's AST, which is made of nodes representing tokens in the program. Each type of node is responsible for adding outgoing edges from itself to its child nodes in a way that represents the potential paths through the node. For example, an addition (`+`) node, which has left and right operand nodes, evaluates left to right; therefore, the `+` node adds an outgoing edge from itself to the left operand and then an outgoing edge from the left operand to the right operand. Similarly, an `if` node adds an outgoing edge to its `expression` node, and then two outgoing edges from the expression: one to the `then` path and one to the `else` path. This is illustrated in Figure 2, which shows the two paths through the conditional in `post()` from Figure 1 (omitting literals, which do not affect control flow).

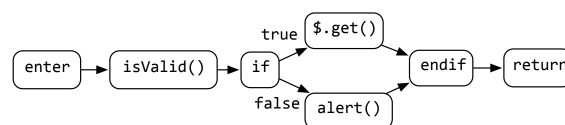

Figure 2. The CFG for `post()` in Figure 1, omitting literals.

FeedLack handles the full range of JavaScript language constructs in the same way. For example, FeedLack accounts for some runtime errors, adding edges for possible divide by zero errors on division nodes and null pointer and undefined runtime errors on object property expressions. By the end of CFG creation, FeedLack has constructed a directed acyclic graph representing the potential paths through a function for all functions in the provided source code (except for outgoing paths from function calls, which are considered later).

One decision in constructing CFGs is how to handle loops, since paths through loops can be infinite in length. FeedLack treats loops as conditional blocks, assuming that loops execute either zero or one times. This simplification was applied because FeedLack's analysis of feedback is conservative: if there's any way to produce output through the loop, then it assumes that way is feasible. This is nevertheless another source of false positives.

Step 3: Propagating Type Information

The next step is to propagate type information through each function. FeedLack needs type information to increase confidence in which functions a call might invoke (Step 4) and to identify code that might affect output (Step 5). Of course, because JavaScript is a dynamic and weakly typed language, there are few guarantees about what functions and properties are valid for any given expression at runtime (for example, even if the expression `element.innerHTML` is known to produce a string, the `innerHTML` property may have been deleted at runtime or `element` may not point to an element).

FeedLack does several things to infer the possible types of variables and properties despite the potential for imprecision. FeedLack infers the types of expression ASTs, propagating type information along data flow edges. For example, to infer the type of the `+` node in `var msg="Hello"+subject`, it inspects the possible types of its two children and determines that it may produce a string. The same type propagation on the assignment in this expression would determine that the type of `msg` is a string. FeedLack also documents all W3C DOM API types, enabling it to determine, for example, that `document.getElementById('home')` returns an `HTMLElement`.

To determine the possible types of variables and object properties, FeedLack explores the paths through the function's CFG using a depth-first search, storing the type information of expressions assigned to variables and properties by name. When this search finds a reference to a variable or property, it propagates the type information previously assigned to the reference. (This process handles local variables, but does not propagate type information for function arguments or function returns; this occurs in Step 4).

To determine the possible types of object expressions (e.g, the `el` in `el.style`), FeedLack gathers the names accessed on each identical object expression and looks for DOM API types and object literal declarations (from Step 1) that contain at least two matching property names. For example, if a function contained the expressions `el.style`, `el.innerHTML`, and `el.onclick`, FeedLack would look for types that have the names `style`, `innerHTML`, and `onclick` and find `HTMLElement`. The type with the most matching names (and in the case of a tie, the most widely used type in the program), is added to the possible types of the object expressions.

Step 4: Resolving Function Calls

After type information is propagated through each function, FeedLack's next step is to resolve all function calls in the program to the functions they might invoke. For each call, it determines the name of the function called and first checks if there are any `LOCAL` functions (as defined in Step 1) in the scope of the call. If there are not, FeedLack checks the calling context to determine whether the call is on an object (e.g., `object.run()`) or not (e.g., `run()`). If the call is on an object and object expression has type information, the function search is limited to the known functions of the expression's possible types. If no functions are found or there is no type information, FeedLack searches *all* `OBJECT` functions for functions with matching names. If the call is not on an object, FeedLack searches all `GLOBAL` functions for matches. If there are no matching names, the failure is noted so that this can be mentioned in FeedLack warning.

After resolving all calls, FeedLack uses the resolved functions to further propagate type information. It propagates the types of arguments sent to functions to each call's resolved function's parameter locals. It also propagates the types of return statements' expressions to the call itself. Lastly, FeedLack repeats Step 3 to further propagate this new type information throughout the program.

FeedLack does not resolve calls to `apply()` and `call()` or calls on arrays (e.g., `object[getFunction()]()`). While this is a source of false positives, prior work has shown that 81% of JavaScript calls only ever invoke one function and that less than 3% have more than two targets [19].

Step 5: Identifying Output-Affecting Statements

With the type information from the previous steps, FeedLack's next step is to search all functions for statements that affect output. FeedLack considers two kinds of statements output. The first are assignments to W3C DOM properties that affect the appearance of page, such as `className` and `id` (which may change the appearance of an element via CSS), `innerHTML`, which explicitly modifies the HTML inside of an element, and a variety of other properties such as `style`, `textContent`, and so on. Some assignments also cause the browser to navigate to a new URL, including assignments to `document.location` and `window.location`.

The second kind of statement considered output includes W3C DOM calls that can affect the appearance of a page. These include functions such as `appendChild`, `setAttribute`, on `HTMLElements` and calls to global functions such as `alert()` and `open()`. In addition to these native calls, FeedLack also accounts for the jQuery and Prototype APIs, recognizing calls such as `$('#home').hide().css('color', 'blue')`.

It should be noted that the statements above do not always affect output. For instance, the statement `el.style.color = 'blue'` only has an effect if the element's color was not already `'blue'`. Similarly, a call to `removeChild()` may fail if the child provided is not found. Because FeedLack is a static analysis, it cannot verify these side effects.

Step 6: Identifying Input Handling Functions

After identifying input, FeedLack's next step finds functions that handle user input. FeedLack considers the full range of input events originating from mice and keyboards, including `click`, `mouse down/up/over/move/enter/out/wheel`, `key down/up/press`, `cut`, `copy`, `paste`, `contextmenu`, `error`, all seven JavaScript drag events, and `href` attributes (which are sometimes used to handle clicks on links). FeedLack ignores events related to focus and element property change events, under the assumption that feedback is not expected for these events since they are not explicitly user invoked.

FeedLack looks for three kinds of input handling code. First, it looks for any tag with input handling attribute values that parse as JavaScript code without errors (as in `onclick="goHome()"`). Each inline script is treated as an input handling function. We include `<input>` tags with a `type` attribute equal to `submit`, `image`, `button`, `checkbox`, or `range` and `<button>` tags with a `type` attribute equal to `submit` or `button` because users expect them to provide some feedback beyond that provided by the control itself. However, we exclude `<input>` tags with type `password` or `radio` as input, assuming their intrinsic feedback is adequate. (We treat checkboxes and radio buttons differently because radio buttons explicitly label each possible mode, whereas the meaning of a checkbox with a static label can be ambiguous).

FeedLack also looks for assignments to object properties that represent input handling functions. For example, the expression `getElementById('home').onclick=goHome;` assigns the function `goHome` to the `onclick` attribute of the element returned by the `getElementById()` function.

FeedLack also looks for event binding calls that represent input handlers, including the W3C and Internet Explorer `addEventListener()` and `attachEvent()` calls as well as jQuery and Prototype event binding APIs (as in `$('#home').click(goHome)` and `$('#home').observe('click', goHome)`, respectively. Functions passed to these calls are also treated as input handling functions.

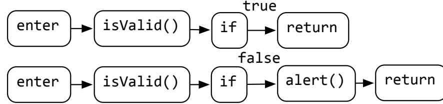

Figure 3. The two paths through `post()` in Figure 1. The first path skips the conditional's `true` block since it lacks output affecting assignments and calls.

Step 7: Enumerating Paths Through Input Handlers

Having identified the program's set of input handlers, FeedLack's next step is to find all paths through each handler. FeedLack uses a depth-first search through each function's CFG, adding each visited node to a list. At each node with multiple outgoing edges, FeedLack duplicates the set of existing paths through the function and then recursively explores each edge. Figure 3 shows the two paths through function `post()`, derived from the CFG in Figure 2. Because FeedLack's analyses are memory intensive, it represents paths as sequences of both nodes and other paths, reusing path leading up to decision points.

To simplify FeedLack's warnings, the above algorithm includes two special cases. First, FeedLack only includes calls, returns, and conditionals, and output-affecting assignments. All other program events such as expressions and non-output affecting assignments are excluded, limiting paths to control flow events. Second, FeedLack only explores *output-affecting* blocks; these are that contain at least one output-affecting call or assignment, where all functions a call might invoke are recursively inspected for output-affecting code. For example, the first path in Figure 3 omits the code within the `true` case of `post()`'s conditional because FeedLack determined that `$.get()` does not affect output.

These special cases have two rationales. First, because conditional blocks can double the number of paths through a function, this simplification mitigates the growth of the number of paths through a function. Second, and perhaps more importantly, this limits FeedLack's warnings to blocks that could possibly affect output, assuming that any block that *cannot* affect output is not one that the developer intended to affect output, and therefore not of interest.

Step 8: Expanding Paths Through Input Handlers

After generating paths through each input handling function, the next step is to replace the calls in these paths that might affect output with all possible paths such calls may result in. This process expands the scope of paths through a single input handling function to the scope of the whole program.

```
function ExpandPaths(function, callstack)
  if callstack contains function, return {}
  push function onto callstack
  let expandedPaths = {}
  let paths = paths through function
  for each path p in paths
  |  add ExpandCalls(p, callstack) to expandedPaths
  pop callstack
```

```
function ExpandCalls(path, callstack)
  let expandedPaths = {[]}
  for each node n in path
  |  append n to all paths in expandedPaths
  |  if n is a call
  |  |  let newPaths = {}
  |  |  let functions = functions call could invoke
  |  |  for each function f in functions
  |  |  |  if f contains > 1 output-affecting statement
  |  |  |  |  let paths = ExpandPaths(f, callstack)
  |  |  |  |  if |paths| x |expandedPaths| < 1 million
  |  |  |  |  |  for each path p in expandedPaths
  |  |  |  |  |  |  for each path q in paths
  |  |  |  |  |  |  |  let r = append q to copy of p
  |  |  |  |  |  |  |  add r to newPaths
  |  |  if newPaths != {}, expandedPaths = newPaths
  return expandedPaths
```

Figure 4. FeedLack uses `ExpandPaths` and `ExpandCalls` to convert the paths through a function into all possible paths from the function through the program, focusing only on calls that can invoke functions containing output-affecting code.

We list the two algorithms that achieve this in Figure 4. `ExpandPaths` iterates through each path through an individual function, converting each individual path into multiple paths with the function `ExpandCalls`. `ExpandCalls` iterates through each node in its given path, resolving calls with the results from Step 4. For each function resolved that contains an output-affecting block (as defined in Step 7), `ExpandCalls` creates new paths to represent all possible paths through all possible functions called. `ExpandCalls` does *not* expand calls to jQuery and Prototype API functions recognized as output-affecting, nor does it attempt to resolve functions passed to `call()` or `apply()`; these latter calls are assumed to affect output to avoid false positives.

An example of the result of `ExpandPaths` appears in Figure 5, showing the paths through the `onclick` handler in Figure 1. These paths show how the two paths through `post()` (in Figure 3) and the two paths through `isValid()` result in four paths through the `onclick` handler. Four identical paths (not shown) are generated for the `onsubmit` handler in Figure 1.

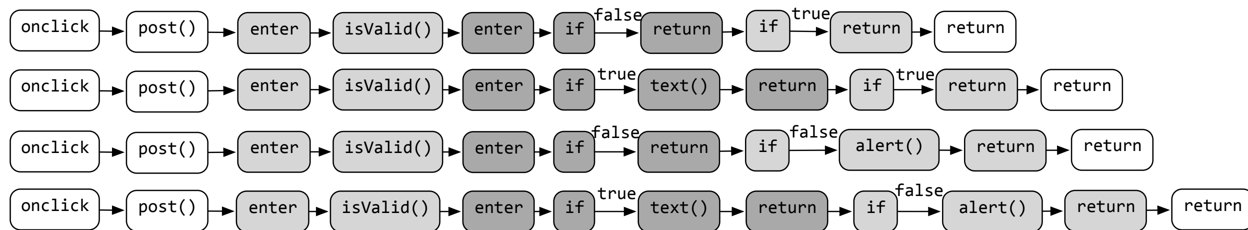

Figure 5. The 4 paths through the `onclick` handler in Figure 1 (and 4 identical paths through `onsubmit`, not shown), with successive calls in darker grey. The 3rd path is infeasible, since the comment cannot be both valid and invalid.

Limiting call expansions to only those functions that could possibly affect output is a critical part of minimizing "path explosion," or the phenomena of path analysis growing exponentially. Also note that `ExpandPaths` and `ExpandCalls` maintain a call stack of functions visited. This allows the algorithm to identify recursive calls, meaning that FeedLack assumes each recursive call occurs once (a similar assumption to that made for loops). Of course, there are cases were neither of these measures are enough to avoid path explosion. Therefore, we empirically derived a limit of one million paths by testing FeedLack on several applications with a Java process allocated 2 GB of RAM.

Finally, it is also important to note that `ExpandPaths` assumes a single thread of execution. JavaScript does allow developers to spawn threads with `setInterval()`, `setTimeout()`, and `AJAX` calls, but FeedLack does not consider the functions they call as output-affecting, requiring the input handling thread to produce feedback itself. This is because they introduce the potential for feedback delays: even if a timeout is supposed to start immediately upon calling, stutters in the network or operating system can cause delays. For example, in addition to AJAX calls producing feedback when they succeed or fail, threads invoking AJAX calls must present feedback while the call is pending.

Step 9: Identifying Output-Lacking Paths

The result of the previous step is a set of paths through each input handing function, some of which contain output-affecting statements, some of which do not. FeedLack groups these paths by the HTML tags and input events from which they originate, eliminating groups of paths that contain at least one handler that always produces output. This accounts for tags that have multiple handlers for similar events, one of which is responsible for output. FeedLack then iterates through the remaining paths, selecting ones that do not contain output-affecting statements. For example, in Figure 5, the only path lacking output-affecting code is the first one; this results in the two paths from the `onclick` and `onsubmit` handlers in Figure 6.

Step 10: Clustering Output-Lacking Paths

There are a number of reasons why presenting output-lacking paths directly to FeedLack users would be unnecessarily complex. For instance, some handlers reuse functions that are responsible for providing feedback (as in the case of `post()` in Figure 1); presenting separate handlers with intersecting paths as distinct would be redundant. Moreover, intersecting paths often share a *critical sequence*. For example, the two paths at the top of Figure 6 hinge upon two particular conditionals. This is an opportunity to highlight these commonalities, rather than require users to notice them.

To identify these commonalities, FeedLack groups output-lacking paths into *path clusters*. It starts with an empty set of path clusters, `Clusters`. Then, for each output-lacking path `P`, FeedLack considers each cluster in `Clusters`, and for each path `C` in each cluster, computes the number of nodes that `P` and `C` have in common. FeedLack remembers the smallest intersection of each cluster, and chooses the cluster with the largest minimum intersection. If there are no clusters,

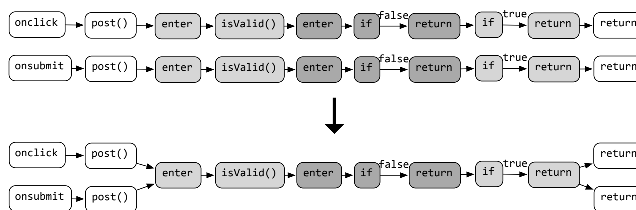

Figure 6. The two output-lacking paths from Figure 5, grouped into a path cluster with two routes to `post()`.

FeedLack creates a new cluster and adds P. The result is a set of path clusters, where all paths in each cluster have at least one node in common. For example, the two paths at the top of Figure 6 are clustered into one single path cluster.

Next, FeedLack identifies the longest sequence of nodes that appears in all paths in a cluster; we call this the *critical sequence*. FeedLack takes the first path in the cluster and numbers each of its nodes from 1 to the number of nodes in the path, also adding each node in the path to a list representing the critical sequence. Then, it iterates through the remaining paths in the cluster, removing all nodes from the intersection list except those also contained in the remaining path. It then takes the final intersection and orders it using the numbers from the first path.

For the final step, FeedLack iterates through all paths in the cluster and identify all paths leading to and from the critical sequence. For example, the path cluster in Figure 6 has to paths leading to the critical sequence and two paths from it. FedLack then presents path clusters as these three parts. Figure 1's warning, for example, lists the two input handlers and the critical sequence; the outgoing paths were omitted since they only included function returns. More complicated paths can have several outgoing paths; for example, FeedLack will often select the conditional of a `switch` statement as a critical sequence and then enumerate the various cases the `switch` might select.

EVALUATION

There are many aspects of FeedLack to evaluate, ranging from the feasibility and legitimacy of its warnings to the understandability of its warnings to developers and usability engineers. In this paper, we focus specifically on FeedLack's *true and false positives* (paths that FeedLack believes are missing output). We do not assess its *false negatives* (paths FeedLack believes provide output but do not), primarily because of the sheer number of negative paths generated by the analysis (there were generally two orders of magnitude more negative paths than positive paths).

Sampling

In sampling web sites, we focused on sites with JavaScript input handlers, avoiding those that used rich internet application frameworks such as Flash or Silverlight. Our sampling approach was stratified and opportunistic and aimed at retrieving at least 300 applications with diverse functionality. One class of applications we chose were highly trafficked sites listed on `http://www.alexa.com`, including sites used for photos, videos, and shopping. Another class of applications included the smaller sites used frequently by the 2nd author, including those of schools, student organizations,

restaurants, churches, and government. We also searched the web for "HTML 5 demo" and "HTML 5 application," resulting in several sites that used the `<canvas>` tag for output. Finally, we sampled applications from projects on Google Code (http://code.google.com/hosting/) with live demos.

To obtain the client-side source for these sites, we thoroughly exercised all interactive elements in the page to ensure that all source code for the page was downloaded and then used Google Chrome's page archiving feature to save the HTML and JavaScript source. For the Google Code projects, we downloaded the latest source for the project. The result of this process was 330 web applications and their source code.

Applying FeedLack to the Sample

Next, we ran FeedLack on these 330 web applications. All 330 applications were analyzed in less than 1 minute on a 2 GHz MacBook Pro with a Java process given 2 GB of RAM.

Of the 330 applications, 89 had syntax errors that FeedLack's JavaScript parser could not overcome (including unsupported unicode characters and missing semicolons). Of the remaining, 112 lacked JavaScript input handlers. Of the remaining 129, there were 14 that caused out of memory exceptions. We found two underlying reasons for these exceptions. In 12 applications, there was a function with anywhere from 26 to 119 sequential output-affecting conditionals, causing FeedLack to generate trillions of paths. In the other two cases, FeedLack ran out of memory while clustering tens of thousands of warnings.

In the remaining 115 applications, FeedLack identified 6,887 input handling sites, 6,362 (92%) of which FeedLack believed successfully produced output on all paths. We did not analyze these handlers for true negatives (paths that FeedLack believed produce output but do not) because of the sheer number of paths that would need to be tested manually.

Of the 115 applications, 33 resulted in no FeedLack warnings, leaving 82 applications with at least one output-lacking path to verify. Table 1 shows descriptive statistics about these applications. The average app had 2 HTML files, multiple JavaScript source files, dozens of input handlers, and several hundred JavaScript functions. To get a sense of the functionality in our sample, we categorized each as one of the 7 categories from http://versiontracker.com. As shown in Figure 7, most were games, productivity apps, design tools, or developer tools, including interactive visualizations, calculators, action games, calendars, educational lessons, graphic design tools, photo management tools, social networking apps, web storefronts, and note taking apps.

	min	mean	max
# HTML files	1	2	54
# JS files	0	5	20
# HTML handlers	0	20	278
# JS handlers	0	10	63
# JS functions	6	623	2,176
# JS statements	67	6,678	25,567

Table 1. Aggregate statistics about file, input handler, function and statement counts in our sample of applications.

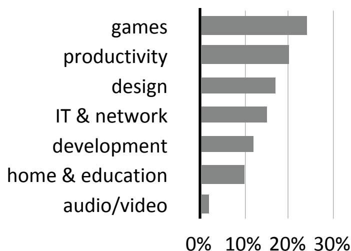

Figure 7. Distribution of application types in our sample.

Applying FeedLack to the remaining 82 applications resulted in 647 output-lacking paths. To evaluate each path, we began by attempting to execute it through manual testing of the live web site (we primarily used Firebug breakpoints, attempting to execute each step the path). If the path was *not* executable, we diagnosed the source of infeasibility in FeedLack's reasoning. If the path *was* feasible, we noted whether the path provided feedback, and if so, diagnosed the cause of the false positive. If it did *not* provide feedback, we described the missing feedback in detail for later analysis. The 1st author then classified each path as one of the following:

- **infeasible** paths, which could not be executed.
- **output-producing** paths, which *did* produce feedback.
- **output-missing** paths, which did *not* produce feedback, but did not lead to confusion about application state.
- **output-deserving** paths, which did *not* produce feedback, causing confusion about application state.

To choose between the last two categories the first author applied widely-used conventions for GUI components to make these decisions. For example, buttons that appeared disabled and did not produce feedback were classified as output-missing; buttons that appeared enabled but did not produce feedback were classified as output-deserving. Similar conventions were applied to other interactions.

Results

Frequencies of warning types appear in Table 2, separated by input event. Of all paths, 12% were infeasible and 18% produced feedback despite FeedLack's warning; 34% did *not* produce feedback but did not appear to need it; and finally, 36% of warned paths lacked feedback and needed it. There was a significant relationship between the kind of input event and warning category ($\chi2(n=647,df=42)=261,p<.001$). For example, `click`, `href`, and `mousedown` events were more likely to be warned and were less likely to be false positives than `href`, `mouseover` or `mousewheel` events.

	infeasible		output-producing		output-missing		output-deserving		TOTAL	
`click`	**24**	12%	**31**	16%	**69**	35%	**76**	38%	**200**	31%
`href`	**2**	4%	**24**	43%	**3**	5%	**27**	48%	**56**	9%
`mousedown`	**5**	9%	**8**	15%	**30**	57%	**10**	19%	**53**	8%
`mousemove`	**2**	4%	**4**	8%	**22**	46%	**20**	42%	**48**	7%
`mouseup`	**0**	0%	**3**	6%	**13**	27%	**33**	67%	**49**	8%
`mouseenter`	**0**	0%	**1**	100%	**0**	0%	**0**	0%	**1**	0%
`mouseover`	**2**	4%	**20**	43%	**18**	39%	**6**	13%	**46**	7%
`mouseout`	**5**	17%	**3**	10%	**13**	45%	**8**	28%	**29**	4%
`mousewheel`	**5**	71%	**2**	29%	**0**	0%	**0**	0%	**7**	1%
`keypress`	**21**	53%	**0**	0%	**8**	20%	**11**	28%	**40**	6%
`keydown`	**3**	7%	**1**	2%	**19**	42%	**22**	49%	**45**	7%
`keyup`	**7**	20%	**0**	0%	**15**	43%	**13**	37%	**35**	5%
`cut/paste`	**0**	0%	**2**	100%	**0**	0%	**0**	0%	**2**	0%
`multiple`	**2**	6%	**15**	42%	**13**	36%	**6**	17%	**36**	6%
TOTAL	**78**	12%	**114**	18%	**223**	34%	**232**	36%	**647**	

Table 2. The frequency of warning categories by input event; `multiple` represents paths invoked by multiple input event types. Percentages represent the proportion of the cell to its row; total percentages are relative to all warned paths.

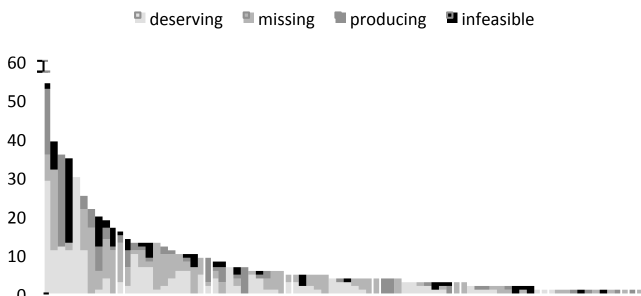

Figure 8. Warned paths per app by category, sorted by decreasing per-app total (excluding apps with no warnings).

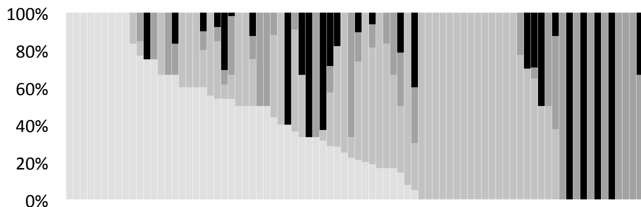

Figure 9. Proportions of warning types per app, sorted by output-deserving warnings (excluding apps with no warnings).

We show the distribution of each of these warning category per application in both absolute counts (Figure 8) and as proportions (Figure 9). Figure 8 shows that the number of warnings in our data ranged anywhere from 1 to 55 and that the number of output-deserving paths was rarely over 10 for an individual application. Figure 9 shows that FeedLack detected at least one output-deserving path for 50 of 115 applications. Therefore, if a team were considering testing a deployed web app with FeedLack, there would be roughly a 43% chance that it would detect at least one problem, if not more (the likelihood of detecting missing feedback on an app in development may be higher, but this was not studied) .

In the rest of this section, we consider each warning category individually. To begin, the 12% of **infeasible** arose from several distinct sources, listed in Table 3. Most of these stemmed from imprecision in FeedLack's call graphs, its lack of data flow analysis, and the impossibility of specific inputs. However, some infeasible warnings revealed unhandled error conditions that were impossible to reach in the current version of the application.The 18% of **output-producing** paths came from several distinct sources (see Table 4). Most came from unresolvable calls, which were assumed to not produce feedback. Many of these could have been due to incomplete archiving of an application's source code.

The 35% of **output-missing** paths were also false positives. As seen in Table 5, they primarily concerned code that was never intended to provide feedback. The most common scenario identified were interactive situations in which users would not expect feedback, such as auto-completing text fields that showed no results when empty. Most of the other handlers tracked mouse clicks for web analytics or time-delayed interactions. Although we considered these negligible, there may be some warnings that others might assess differently. For example, a privacy-sensitive site might actually want to tell users each time their clicks are tracked or explain to users why buttons are disabled.

#	description
26	**Multiple conditions** checked in separate functions that could not be simultaneously true. For example, one function in a calculator had a special case for the `1/x` button; FeedLack reported several paths from non-`1/x` buttons through the `1/x` conditional block.
21	**Infeasible calls**. For example, one application had several calls to a function named `insert()`, but FeedLack mistakenly resolved these calls to functions named insert() that were not reachable at runtime.
11	**Unreachable handlers**, such as abandoned or unfinished code that was never attached to HTML elements. One common source was jQuery expressions that returned empty sets.
10	**Impossible values** in sequences of conditionals that checked for one of from a set of values. For example, one function handled the display of two popup dialogs; if the `id` argument passed by the caller was not one of these two `id` strings, no output would occur, but there were no calls that passed an `id` other than these two strings.
7	**Hidden controls**, where the input that would have led to no feedback was not possible because the control was not visible. For example, in one warning, a cancel button had no effect when the progress dialog containing it was hidden.
3	**Unreachable error cases**, such as exceptions and errors with output-lacking else cases. We were unable to cause these errors.

Table 3. Causes of infeasible warnings.

#	description
54	**Unresolved calls**, where FeedLack could not find matching functions for a call that ultimately produced output. Some of these functions may not have been archived in our sampling.
20	**Undetected multiple handlers** on the same HTML element, at least one of which always produced output. For example, in several cases, an `onclick="return false;"` attribute was added to an HTML element, but a jQuery handler was also added.
12	**Overlooked native output**, such as assigning `window.location.hash` a new value to navigate to a new URL and jQuery extensions.
8	**Timers with imperceptible delay**. Uses of `setInterval()`, `setTimeout()`, and `clearInterval()` with no delay were effectively behaved explicit calls.
8	**Output-affecting state**, where applications modified state that was later used by a timer to affect output. For example, one handler changed the value of a `paused` variable which was inspected in an animation loop to halt feedback.
7	**Inadequate type inference**, causing FeedLack to overlook output (e.g., FeedLack overlooked changes to text area's `value` property when it was the only property referred to on an object expression).

Table 4. Causes of output-producing warnings.

#	description
61	**Negligible modal interaction states**. For example, many popup dialog handlers would hide a popup when clicking on a page body, but would have produced no feedback when the popup was already hidden. These were scenarios where the visual state of the page removed an expectation of feedback.
50	**Web analytics handlers** only intended capture click information.
46	**Event propagation handlers**, coordinating with other handlers to track mouse button states and keyboard event consumption.
32	**Time-delayed behaviors**, such as custom tooltip and link-preloading functionality intended only to appear after a mouse dwell.
14	**Ignored keystrokes**, where nothing in the user interfaces suggested that these keys would provide feedback. These were often unhandled else cases of switch statements that handled a limited set of keys.
14	**Disabled elements**, which had handlers, but provided no feedback when styled to appear disabled or inactive.
5	**Inactive in-progress animations**, such as clicked images that were inactive while animating to full screen, but active before and after.

Table 5. Causes of output-missing warnings.

#	description
41	**Hidden modal behaviors** including buttons and other controls that only produced output when the application was in a particular state. For example, in a chess game, the check mate game over state prevented any further input, but there was no message to indicate that the check mate state had been reached.
36	**Inactive command buttons appearing enabled**, including copy, cancel, load, and other commands. In these cases, the buttons were disabled, but did not appear so.
34	**Ignored keystrokes** in keyboard-driven applications. For example, in one game, the character was controlled by one of seven letter keys; if some key other than these was typed, there was no feedback that the key was not accepted. In other cases, keys that had some conventional behavior had no effect. For example, on a library search page, the enter key failed to submit a query.
32	**Dead links**, similar to those found by web site validators.
31	**Count-limited repeated inputs**, where actions that were invoked repeatedly (e.g., firing missiles in a shooting game) ceased after some number of clicks without explanation.
20	**Silent error conditions**, such as failed checks for particular browsers or keyboard layouts, that provided no feedback on failure.
19	**Missing hover feedback** where hovering or dragging over particular targets would provide no change in output. For example, a calendar application's event resize interaction supported spanning days but did not visualize the days spanned.
10	**Delayed feedback**, including behaviors that took some action, but provided feedback through a `setTimeout()` or `AJAX` call, pausing or lagging the UI for several seconds without intermediate feedback.
9	**Silent state changes**, including controls meant to change state, but when clicked, provided no feedback about the success of the change. One app had a save link that did not indicate success or failure.

Table 6. Causes of output-deserving warnings.

The last 35% of **output-deserving** paths all lacked feedback and violated the conventions of common GUI interactions. As shown in Table 6, the most common warnings involved modal behaviors in which input events only had an effect when the application was in a particular state, but that state was not visible. Other common problems included ignored keystrokes, dead links, silent error conditions, and missing selection feedback on items that appeared selectable.

DISCUSSION

The results of our evaluation show that FeedLack can detect a variety of significant feedback problems while also detecting several places in web application code potentially in need of error handling code. In our discussion, we consider FeedLack's limitations and generalizability in detail.

Prospects for Reducing False Positives

While FeedLack's false positive rates are high, they are comparable to the 50% rates reported for the widely used static analysis tool FindBugs [3]. Nevertheless, there may be ways to eliminate some false positives. For example, many of the sources of false positives were related to inadequate type inference and call graph precision; this could be improved by using more sophisticated type inference analyses (e.g., http://doctorjs.org/). Similarly, there were many kinds of input events with high false positive rates; `mousedown`, `mouseover`, `mouseout`, and `keypress` events, and handlers invoked by multiple input events, were least likely to require feedback. Were these omitted from FeedLack's analyses, most warnings would have been output-deserving. Of course, omitting these warnings would also omit some true positives; this is a tradeoff inherent to any defect detection analysis.

Issues that FeedLack Cannot Detect

First and foremost, FeedLack cannot detect issues with the *quality* of feedback. To be sure, many of the scenarios that FeedLack identified as providing feedback were still confusing. Output was often so far away from the source of input, there was no perceptible change; detecting such problems might require modeling of the location and appearance of HTML elements on screen. Moreover, much of the output produced had a weak conceptual connection to the input that caused it (in one application, clicking a save button caused a mysterious icon to appear, apparently indicating success). Without further research on feedback verifications like FeedLack, analyzing the semantic correspondence between input and output still requires the talent of experienced usability engineers.

FeedLack cannot find all missing feedback in web applications. For example, there are many things that can cause a JavaScript input handler to halt or stall, including references to undefined properties, unresolvable functions, memory errors, uncaught exceptions, infinite loops, slow algorithms and a variety of other runtime issues. While these are outside of FeedLack's scope, there are complementary approaches to detect these problems [2,11,18].

Another feedback issue that FeedLack cannot detect is the *absence* of input handlers on any HTML element that might appear to handle input but does not. For example, most web site's logos navigate to the site's home page, but some site's logos do not have these links. There is no obvious way for a machine to know which elements should have handlers (moreover, verifying that elements have handlers at all is complicated further by the flexibility of runtime binding).

FeedLack also overlooks situations where an application assigns an output-affecting property a value that is equivalent to its old value. For example, there are many cases where an element might be assigned an equivalent `class`, meaning the user would experience no visible change in the web page. More generally, applications might redirect users to the same page they were on already, or web servers might return dynamically-generated but identical web pages, again leading to situations where the application appears not to respond.

Making Sense of FeedLack Warnings

One major aspect of FeedLack we have yet to evaluate is to what extent usability engineers and software developers can actually understand FeedLack's warnings. We were able to comprehend the warnings (even the unfamiliar code of the 82 applications in our sample), but this does not mean that it would be easy for users without significant knowledge of FeedLack's analyses. Given that FeedLack report paths through code and not actions on a concrete user interface, usability engineers may have challenges understanding and triaging these issues. Future work might involve converting FeedLack's warned paths into concrete actions on the web application UI, better enabling testers to assess the warnings.

Generalizing FeedLack to Other Platforms

Few of FeedLack's algorithms are particular to the web; most of the work necessary to adapt FeedLack to other platforms is identifying input-handling and output-affecting statements,

and creating language-specific CFGs. One possible challenge, however, comes from the extent to which user interface event handling and output is *declarative*. It is simple in JavaScript and HTML to detect UI controls and changes to their behavior, because most APIs require users to declare those changes explicitly. In many statically typed imperative languages, however, creating a UI button requires several lines of instantiation, configuration, and event listening code, as do customizations to these controls. Tracking these customizations, especially across procedures and subclasses, could prove difficult, although prior work has had some success on object-oriented UI toolkits [12].

CONCLUSIONS AND FUTURE WORK

We have presented FeedLack, an analysis for automatically detecting missing feedback in web applications. We have demonstrated that FeedLack can detect significant feedback issues in real web applications, as well as presented an analysis of its false positives and limitations. While FeedLack is not a replacement for usability testing or expertise, it may be an effective supplement to empirical approaches to detecting feedback issues, much like HTML validators and other software verification tools.

Our results also suggest several directions for future work. We want to explore the utility of FeedLack alongside other forms of software testing and verification by deploying it into a real web development team. Part of this deployment could involve tracking feedback issues over successive versions of web application UIs, and adding explicit support for suppressing known false positives. There may also be ways to extend FeedLack to support accessibility analyses, checking to see not only whether applications provide feedback, but that the feedback it provides is compatible with screen readers and other accessibility tools.

More generally, we would like to explore the automatic detection of other usability problems beyond feedback, such as issues with graphic design consistency, recognition vs. recall problems, confusing error messages, and support for cancel and undo. We believe that tools that tie usability concerns to code are a key part of integrating the work of usability engineers with the rest of a software team.

ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-0952733. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. Akers, D., Simpson, M., Jeffries, R., & Winograd, T. 2009. Undo and erase events as indicators of usability problems. *ACM Conf. on Human Factors in Computing (CHI)*, 659-668.
2. Artzi, S., Dolby, J., Tip, F., & Pistoia, M. 2010. Practical fault localization for dynamic web applications. *Int'l Conf. on Software Engineering (ICSE)*, 265-274.
3. Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., & Zhou, Y. 2007. Evaluating static analysis defect warnings on production software. *ACM Workshop on Program Analysis For Software Tools and Engr.*, 1-8.
4. Blackmon, M. H., Polson, P. G., Kitajima, M., & Lewis, C. 2002. Cognitive walkthrough for the web. *ACM Conf. on Human Factors in Computing Systems (CHI),* 463-470.
5. Bodden, E. & Havelund, K. 2008. Racer: effective race detection using AspectJ. *Int'l Symposium on Software Testing and Analysis (ISSTA)*, 155-166.
6. Bruun, A., Gull, P., Hofmeister, L., & Stage, J. 2009. Let your users do the testing: a comparison of three remote asynchronous usability testing methods. *ACM Conf. on Human Factors in Computing Systems (CHI)*, 1619-1628.
7. Godefroid, P., Levin, M.Y., & Molnar, D.A. 2008. Automated whitebox fuzz testing. *Network Distributed Security Symposium (NDSS)*.
8. Hovemeyer, D. & Pugh, W. 2004. Finding bugs is easy. *ACM Conf. on Object-Oriented Prog. Systems, Languages, and Applications (OOPSLA)*, 132-136.
9. Hutchins, E. L., Hollan, J. D., & Norman, D. A. 1985. Direct manipulation interfaces. *Human-Computer Interaction 1*(4), December, 311-338.
10. Ivory, M. Y. & Hearst, M. A. 2001. The state of the art in automating usability evaluation of user interfaces. ACM Computing Surveys 33(4), December, 470-516.
11. Ko, A.J. & Wobbrock, J.O. 2010. Cleanroom: Edit-time error detection with the uniqueness heuristic. *IEEE Symposium on Visual Languages and Human-Centric Computing*, to appear.
12. Ko, A.J. & Myers, B.A. 2008 Debugging reinvented: asking and answering why and why not questions about program behavior. *Int'l Conference on Software Engineering (ICSE)*, 301-310.
13. Lecerof, A. & Paterno F. 1998. Automatic support for usability evaluation. *IEEE Trans on Software Engineering (TSE)* 24(10), October, 863– 888.
14. Lindgaard, G. & Chattratichart, J. 2007. Usability testing: what have we overlooked? *ACM Conf. on Human Factors in Computing Systems*, 1415-1424.
15. Mahajan, R. & Shneiderman, B. 1997. Visual and textual consistency checking tools for graphical user interfaces. *IEEE Trans. on Soft. Engr. 23*(11), 722-735.
16. Nielsen, J. & Molich, R. 1990. Heuristic evaluation of user interfaces. *ACM Conf. on Human Factors in Computing Systems,* 249-256.
17. Norman, D.A. 1988. *The design of everyday things*. New York: Doubleday.
18. Novark, G., Berger, E.D., & Zorn, B.G. 2009. Efficiently and precisely locating memory leaks and bloat. *ACM Conf. on Programming Language Design and Implementation (PLDI)*, 397–407.
19. Richards, G., Lebresne, S., Burg, B., & Vitek, J. 2010. An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Notices 45*(6), June, 1-12.