# Extracting URLs from JavaScript via Program Analysis

Qi Wang[1], Jingyu Zhou[1,2], Yuting Chen[1], Yizhou Zhang[1,3], Jianjun Zhao[1,2]

[1]School of Software, Shanghai Jiao Tong University, China
[2]Department of Computer Science & Engineering, Shanghai Jiao Tong University, China
[3]Department of Computer Science, Cornell University, USA
aywq@sjtu.edu.cn, {zhou-jy, chenyt, zhao-jj}@cs.sjtu.edu.cn, yizhou@cs.cornell.edu

## ABSTRACT

With the extensive use of client-side JavaScript in web applications, web contents are becoming more dynamic than ever before. This poses significant challenges for search engines, because more web URLs are now embedded or hidden inside JavaScript code and most web crawlers are script-agnostic, significantly reducing the coverage of search engines. We present a hybrid approach that combines static analysis with dynamic execution, overcoming the weakness of a purely static or dynamic approach that either lacks accuracy or suffers from huge execution cost. We also propose to integrate program analysis techniques such as statement coverage and program slicing to improve the performance of URL mining.

## Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

## General Terms

Design

## Keywords

Dynamic URL, JavaScript, Program Analysis

## 1. INTRODUCTION

JavaScript is the predominant scripting language used in web browsers, which allows programmers to create rich user interfaces and sophisticated functionality for web applications. A recent study [12] found that all top 100 sites and 89% of top 10,000 sites use JavaScript.

Unfortunately, the ubiquity of JavaScript, along with its dynamic nature, is now posing significant challenges for web search engines, because a large amount of web information, including URLs, is now buried in client-side JavaScript code. Major search engines, e.g. Google [4], only analyze the static parts of the HTML documents, and have difficulty in indexing scripting parts. As a result, many URLs embedded in JavaScript code are missed, which may significantly reduce the coverage and quality of a search engine.

Client-side JavaScript poses many novel challenges for our analysis. For example, JavaScript programs exist in the context of an HTML page and operate in a browser where they interact with HTML elements and access the browser APIs. Program execution is driven by events and accepts asynchronous inputs from the user. JavaScript libraries are widely used but often involve tricky coding and are large in size. It is important to note that JavaScript can generate script code or inject arbitrary HTML expressions into the document dynamically, making its behavior hard to predict.

A straightforward approach is to employ a browser or a JavaScript engine to execute JavaScript programs in HTML documents, instrumenting the engine to output URLs encountered during execution. However, a purely dynamic approach like this does not scale well because it can incur prohibitive execution overhead. Another drawback is that only one control-flow path is executed, possibly missing many URLs in other paths.

Hence, it is natural to enlist the help of static program analysis methods. However, most client-side JavaScript programs use dynamic code generation techniques and often access HTML DOM and browser APIs. Static analysis is not able to tackle these dynamic aspects of JavaScript because some dynamic values can only be determined at run time. Based on these insights, we propose an approach that combines static program analysis and dynamic program execution, leveraging the advantages of both.

The novelty of our idea can be summarized as follows: First, we propose an approach that combines static program analysis and dynamic program execution to find precise URL values in client-side JavaScript code. Second, we integrate techniques such as statement coverage, range analysis, and program slicing to further improve the performance of our approach.

The rest of this paper is organized as follows. Section 2 describes our approach in detail. In Section 3 discusses some design decisions. Section 4 discusses the related work. Finally, Section 5 concludes with future work.

## 2. APPROACH IN DETAIL

### 2.1 Basic Workflow

Our approach combines static analysis and dynamic program execution, leveraging the advantages of both. We use static analysis techniques to reason about control flows of JavaScript programs, and implement a special JavaScript



*ESEC/FSE'13*, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2494583

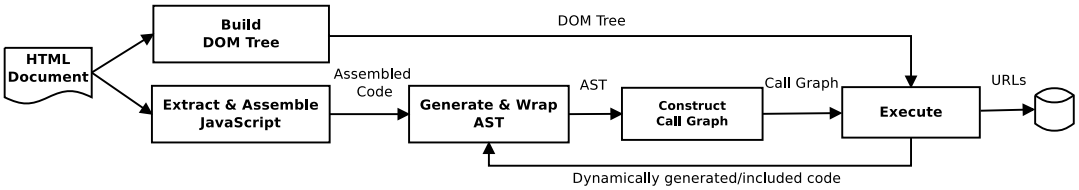


**Figure 1. The workflow of our approach.**

interpreter to execute JavaScript code. Fig. 1 illustrates the workflow of our approach.

For a given web page, we first extract all the code fragments from the HTML document and assemble them into a complete and compilable JavaScript code. Then we take the assembled code as input and parse it into an abstract syntax tree (AST), and then construct a call graph (CG) using the AST. A DOM tree is also constructed for the HTML document. Finally, we use a lightweight JavaScript interpreter to simulate JavaScript executions by traversing the AST. During execution, the interpreter performs different actions when visiting various AST nodes to approximate the dynamic behavior and output the URLs encountered.

**Extracting and Assembling JavaScript.** Client-side JavaScript code fragments exist in different forms and places in an HTML document. For statically included JavaScript code [15], we employ regular expressions to identify and extract them. Dynamically included JavaScript [15] is handled by the JavaScript interpreter in later steps of our approach.

*Sequence of Scripts.* A single HTML document may contain more than one pair of non-overlapping `script` tags. We assemble all JavaScript code fragments together and preserve their order in the original HTML document.

*Wrapping Scripts Defined in HTML Element Attributes.* As discussed in [15], JavaScript code can be embedded in the attribute of an HTML element. In fact, such a usage signifies a new entry point to the whole program, and the execution scope chain that is used to resolve variables includes all DOM objects on the path from the HTML element to the root of the document [7]. We wrap each piece of this kind of code into a separate function and record the path to the HTML element as well as the attribute the code resides in. In this way, scripts embedded in HTML elements can be easily recognized and receive special treatment in execution.

**Modeling the HTML DOM and BOM.** Client-side JavaScript uses Document Object Model (DOM) interfaces to manipulate and interact with the HTML document and uses Browser Object Model (BOM) interfaces to interact with the browser. URLs are often formed via the interaction of these interfaces. However, both the HTML DOM and BOM typically are not provided in JavaScript engines such as Chrome V8 [5] and Rhino [11]. As a result, only using a JavaScript engine to analyze these URLs becomes impossible.

We have implemented the HTML DOM and BOM interfaces in our interpreter to model the host environments. We define a separate class for each type of HTML DOM element and implement properties and methods of each DOM element.

**Execution.** JavaScript has multiple entry points and browsers execute JavaScript in an event-driven fashion. It is important to model the fact that *load handlers* are executed before other kinds of *event handlers* [7]. Every event handler is an entry point of the program, but our analysis cannot predict the exact invocation sequence of event handlers, because it involves user interactions. Fortunately, the execution order of event handlers seldom affects the result of URL mining in our experience and is not crucial for the precision of a static analysis [7]. As a result, our analysis just randomly picks a sequence and executes it.

Algorithm 1 illustrates the execution sequence of our JavaScript interpreter. Given an AST and a CG, the interpreter first executes the load handlers. Then, the interpreter randomly chooses an event handler and executes it. When executing, the interpreter adds dynamically generated event handlers (i.e., via calls to `addEventListener` or assignments to event attributes) to `EH`, which contains unvisited event handlers. Finally, to improve coverage, the interpreter executes all functions in CG that have not been visited.

**Algorithm 1** Pseudo-code of interpreter's execution.

```
Input: the AST ast, the call graph CG

VF = ∅        // visited functions
EH = {statically identified event handlers}

for each load handler l ∈ EH do
    VisitFunction(l)
    EH = EH − l
end for
while EH ≠ ∅ do
    randomly pick a handler l from EH
    VisitFunction(l)
    EH = EH − l
end while
for each f ∈ CG.getAllFunctions() \ VF do
    VisitFunction(f)
end for

function VisitFunction(func)
    VF = VF ∪ {func}
    execute func
    EH = EH∪ {dynamically identified event handlers}
end function
```

*Handling Dynamically Included Code.* The `eval` function is a popular way to dynamically inject JavaScript code — a study shows 59% of the most popular websites used `eval` [12]. During the execution, we capture the actual value that is passed to the `eval` function, and then parse the value to build an AST for the dynamically included JavaScript code. Finally, we insert the root of this syntax tree to the current point of the program AST and start traverse on this injected AST. Other `eval`-like functions such as `window.setTimeout()` and `window.setInterval()` are handled in the same manner.

If the dynamic code injects external JavaScript files during execution, the interpreter will download these files, parse them, and add the resulting AST back.

## 2.2 Further Improvement

To further improve coverage and performance, we integrate several techniques to the execution process of our approach.

We gave a preliminary result of some of these techniques in [15].

**Statement Coverage.** A problem with purely dynamic execution is that they can execute only one path of the program. In order to improve the execution coverage, we implemented a *statement coverage* technique [10]. Specifically, at run-time, before the interpreter returns the results of executing a branch structure (e.g. `if-then-else`), the interpreter is forced to execute other unvisited branches in cloned context scopes.

The code below shows an example that improving statement coverage is beneficial to mining more URLs. The `recommend` variable contains the URL string. Assuming today is Tuesday, the `else` branch of the `if` statement is taken. As a result, `recommend`'s value becomes string `"weekdayRec.asp"` and we find one URL. With statement coverage, another URL can be found on the other branch of the `if` statement.

```
var recommend;
var today = new Date();
if (today == 6 || today == 7)
    recommend = "weekendSpecial.asp";
else
    recommend = "weekdayRec.asp";
document.write("<a href=\"" + recommend + "\">Today</a>");
```

When execution reaches the `if` AST node, the interpreter makes a clone of the current scope, which records variables declared and variables defined together with their values at that point of execution. After executing the `else` branch, the interpreter records the execution result and then forces to execute the `then` branch in the cloned scope. When statements in both branches are visited, the interpreter returns the execution result recorded to the upper AST node.

In addition to `if-then-else`, other control flow structures including `switch-case`, `for` loop, `for-in` loop, `while` loop, `do-while` loop and `try-catch-finally`, as well as conditional expressions, are handled in similar ways.

Note that we do not intend to achieve high path coverage, which is too expensive to use. A relatively high statement coverage is sufficient enough for our URL mining task.

**Range Analysis.** A simple range analysis can be helpful sometimes. Take the code below for an example. Because there is no definition to `hrs` from the condition of the `if` statement to the use of `hrs`, it can be inferred that the variable `hrs` can have integer values ranging from 8 to 23 at the program spot `images[hrs]`. Therefore, by applying range analysis, we can find 14 more URLs in this case.

```
var hrs = new Date().getHours();
if (hrs>=8 && hrs<=23)
    window.open("images/entry" + images[hrs]);
else
    window.open("images/entry" + defaultIndex);
```

Data types of `number`, `boolean` and `string` are also candidates for range analysis.

**Program Slicing.** A great deal of client-side JavaScript code is concerned with presentation details. In finding URLs in client-side JavaScript programs, it is a good idea to concentrate our effort only on program portions where URLs are bound to occur. If we could safely determine, using syntactic criteria, which code is irrelevant to our analysis, we could skip a large volume of it. This would give us the scalability we will need as client-side JavaScript programs grow in size and complexity. Based on this observation, we perform slicing on JavaScript programs to remove URL-irrelevant code, thus reducing the size of code to be executed. In the context of this work, a slicing criterion is a JavaScript statement that contains a variable whose value is a URL string. In [15], we give a taxonomy of URL-relevant program points in a client-side JavaScript program. According to the categorization, we traverse the AST and identify seed statements for slicing.

**Supporting Common JavaScript Libraries.** JavaScript libraries are often large in size and use unusual coding tricks which make the analysis difficult and imprecise. We deal with common JavaScript libraries, such as jQuery, Prototype and YUI , in a different way. Rather than analyzing and executing the code of a common JavaScript library, we identify URL-relevant points in the library APIs and define matching rules for them. If a certain kind of library is identified, the matching rules for the library will be checked when a function call or an assignment occurs. This strategy reduces more than 70% of the JavaScript code to be analyzed and saves half of the network time to download them [15].

Take jQuery as an example. The following gives syntax of two jQuery APIs that indicate the occurrence of URLs or dynamically injected HTMLs.

```
$('#DOM_ELEM_ID').attr("src", url)
$('#DOM_ELEM_ID').html(html)
```

The matching rules for the above jQuery APIs are:

```
$(*).attr({0}="src", url={1})
$(*).html(html={0})
```

In the matching rule syntax, `url` and `html` are keywords indicating the occurrence of a URL or a piece of HTML code. `{0}` represents the value of the first argument and so forth. `{0}="src"` defines a constraint that the value of the first argument must equal to the string value "src".

# 3. DESIGN DECISIONS

We have decided to develop a lightweight JavaScript interpreter rather than instrumenting or modifying a JavaScript engine or a browser engine for several reasons.

First, to support statement coverage and value range analysis, we designed the value of a variable in our interpreter to be a set (i.e., a variable is allowed to have multiple distinct values at a time), instead of strictly being a value. Operations on a variable are performed on each value in the variable's value set. Take the following code as an example.

```
var course, code;
        ......
courseCode = course + code;
location.href = courseCode;
```

Supposing somehow from previous analysis we determine variable `course` can be string value `"CS"` or `"SE"` and variable `code` can be number value `101` or `201`, then the value of `courseCode` is string concatenations between elements of the two sets, which will yield a new value set containing of string values: `{"CS101", "CS201", "SE101", "SE201"}`. However, such design would require substantial changes if it is applied on existing JavaScript or browser engines. Second, implementing a JavaScript interpreter allows us to manipulate the control flow in the execution to implement the statement coverage mechanism as described in Section 2.2. Third, a self-implemented interpreter gives us the flexibility to incorporate program slicing and to support JavaScript libraries via matching rules (Section 2.2).

Nevertheless, instead of building a JavaScript interpreter from scratch, our interpreter is implemented as a wrapper of Rhino [11] and Closure Compiler [6].

## 4. RELATED WORK

String analysis, such as [9, 13], is a particular form of program analysis to infer string values arising at run time. Different with their purpose to detect potential security vulnerabilities using static approach to approximate strings, our analysis needs to compute the concrete value of potential URLs, which relies more on dynamic execution to tackle the dynamic nature of JavaScript.

Redirection spam detection also tries to find URLs. In [16], text-based pattern matching is used to detect changes to `location` object. Later, Chellapilla et al. [1] summarized techniques of JavaScript-based redirection. Thomas et al. [14] used redirection as a feature to detect spam URLs, where instrumented Firefox browser is employed. As we show in [15], text-based approach leads to many false positives, and using a real browser suffers from huge execution cost. Our approach can be used as a lightweight alternative to detect JavaScript redirections.

Several recent work [8, 3] focuses on crawling AJAX contents, where a browser engine is used to execute JavaScript. Our work, instead, focuses on finding all URLs in JavaScript, including those used in AJAX communication. Our work can be extended to record AJAX contents for these crawlers.

To detect drive-by-download attacks, the Wepawet tool [2] instruments the JavaScript interpreter to keep track of all functions. When the execution finishes, Wepawet forces the execution of functions that have not been invoked. Our approach not only executes all functions, but also uses the statement coverage technique to cover all branches [10].

## 5. CONCLUSION AND FUTURE WORK

By combining static program analysis and dynamic program execution, our approach interprets only as much of the JavaScript code as needed to compute the URLs and explores multiple execution paths simultaneously to shorten the execution time needed, achieving high accuracy and improving coverage and performance. This approach can not only be applied to web crawlers to improve their coverage, but also be used for other applications such as URL filtering or redirection spam detection.

There are several remaining challenges. For example, how to deal with compressed and obfuscated JavaScript code and how to make our method reliable across different versions of JavaScript libraries. Another challenge is how to design an efficient slicing algorithm for our purpose.

## 6. ACKNOWLEDGMENTS

This work was supported in part by NSFC 61003012 and 61261160502, and 863 program 2011AA01A202 of China.

## 7. REFERENCES

[1] K. Chellapilla and A. Maykov. A taxonomy of JavaScript redirection spam. In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, pages 81–88, 2007.

[2] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World Wide Web (WWW)*, pages 281–290, 2010.

[3] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. AJAX Crawl: Making AJAX applications searchable. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE)*, pages 78–89, 2009.

[4] Google. Cloaking, sneaky javascript redirects, and doorway pages. `http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=66355`.

[5] Google. Chrome V8 introduction. `https://developers.google.com/v8/intro`, 2012.

[6] Google. Closure Compiler. `http://code.google.com/p/closure-compiler/`, 2012.

[7] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 59–69, 2011.

[8] A. Mesbah, E. Bozdag, and A. v. Deursen. Crawling AJAX by inferring user interface state changes. In *Proceedings of the 2008 Eighth International Conference on Web Engineering (ICWE)*, pages 122–134, 2008.

[9] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 432–441, 2005.

[10] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP)*, pages 231–245, 2007.

[11] Mozilla. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`, 2012.

[12] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *Proceedings of the 25th European conference on Object-oriented programming (ECOOP)*, pages 52–78. Springer, 2011.

[13] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 166–176, 2011.

[14] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*, pages 447–462, 2011.

[15] Q. Wang, J. Zhou, Y. Zhang, and J. Zhao. Extracting URLs from JavaScript via Program Analysis. Technical Report SJTU-CSE-TR-13-01, Center for Software Engineering, SJTU, 2013. `http://stap.sjtu.edu.cn/pdf/2013/ExtractingAnalysis.pdf`.

[16] B. Wu and B. D. Davison. Cloaking and redirection: A preliminary study. In *Proceedings of the First International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, pages 7–16, 2005.