

Building Call Graphs for Embedded Client-Side Code in Dynamic Web Applications

Hung Viet Nguyen
ECpE Department
Iowa State University, USA

Christian Kästner
School of Computer Science
Carnegie Mellon University, USA

Tien N. Nguyen
ECpE Department
Iowa State University, USA

ABSTRACT

When developing and maintaining a software system, programmers often rely on IDEs to provide editor services such as syntax highlighting, auto-completion, and “jump to declaration”. In dynamic web applications, such tool support is currently limited to either the server-side code or to hand-written or generated client-side code. Our goal is to build a call graph for providing editor services on client-side code while it is still *embedded* as string literals within server-side code. First, we symbolically execute the server-side code to identify all possible client-side code variations. Subsequently, we parse the generated client-side code with all its variations into a *VarDOM* that compactly represents all DOM variations for further analysis. Based on the *VarDOM*, we build conditional call graphs for embedded HTML, CSS, and JS. Our empirical evaluation on real-world web applications show that our analysis achieves 100% precision in identifying call-graph edges. 62% of the edges cross PHP strings, and 17% of them cross files—in both situations, navigation without tool support is tedious and error prone.

Categories and Subject Descriptors

F.3.2 [Semantics of Programming Languages]: Program Analysis

Keywords

Web Code Analysis; Embedded Code; Call Graphs

1. INTRODUCTION

With the pervasiveness of dynamic web applications, supporting developers in implementing and maintaining them becomes increasingly important. Modern integrated development environments (IDEs) should provide editor services such as code navigation, code completion, refactorings, and other code-analysis tools. While existing sophisticated IDEs exist to provide support for programming languages such as Java and C#, support for dynamically-typed languages such as PHP and JavaScript (JS) is challenging. Supporting dynamic web application raises even additional challenges.

One important characteristic of dynamic web applications is that they work in two stages: The first server-side stage (typically written

in PHP, ASP, JSP, etc.) *dynamically generates* a program that is subsequently executed in the second client-side stage (typically consisting of HTML, JS, and CSS). In a sense, web applications can be seen as program generators or multi-stage programs [31, 16]. The server-side code generates the client-side code, often assembling string literals (e.g., HTML templates, JS includes) with custom computations—that is, code elements occurring in the second stage are often represented as string values in the first stage.

Contemporary IDEs support developers in writing and maintaining code either in server-side code or in hand-written or generated client-side code (e.g., providing an editor service “jump to declaration” for PHP or JavaScript). In fact, research on analyses of dynamic programming languages has advanced the ability of IDE support for languages such as PHP and JavaScript significantly (e.g., [21, 29]). However, contemporary IDEs do not provide editor support across stages. That is, within the server-side code, fragments of client-side code are merely string constants, and an IDE would not support navigating in them (e.g., finding a closing HTML tag, navigating to a JS declaration). The staged nature of dynamic web applications prevents a general solution, since client-side code can be generated from arbitrary sources. However, in practice, templates for most client-side code are available in server-side code as string literals within which meaningful tool support is possible.

Our goal is to provide a call graph for client-side code while it is still embedded in server-side PHP code (in the first stage). Such a call graph can be used in IDEs to provide various editor services. Specifically, we want to support “jump to declaration” for JavaScript code embedded in PHP code, “jump to closing tag” for HTML tags embedded in PHP code, and “find CSS attributes” for CSS and HTML embedded in PHP code, even before the client-side code is generated, as illustrated in Figure 1. In practice, navigating within embedded client-side code is often nontrivial. For example, an HTML element opened in one string literal may be closed in another, possibly generated from a different file and processed in a sequence of transformation and concatenation steps. The target of a jump may even differ among different executions of the PHP code.

To build the call graph, we build a tool chain combining symbolic execution, variability-aware parsing, and call-graph analyses. First, we symbolically execute the server-side code to identify how the client-side code will be generated. Symbolic execution approximates all possible executions of the server-side generator, such that the generated client-side programs may contain *symbolic values* and *conditional* parts (i.e., they are generated depending on certain path conditions). Next, we parse the generated output into a conditional DOM (called *VarDOM*) that compactly represents all variations of the generated server-side code. A *VarDOM* is similar to the Document Object Model (DOM) for HTML but has condition nodes to indicate that certain subtrees of the HTML document are not

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FSE'14, November 16–22, 2014, Hong Kong, China

Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...\$15.00.

generated in all executions. Based on the VarDOM, we provide three analyses to build conditional call graphs for HTML, CSS, and JS, describing call-graph edges for navigation among code elements traced back to string literals in the original server-side code. We build our infrastructure on top of our symbolic-execution engine *PHPSync* [42] and our variability-aware parser framework *TypeChef* [33]. For building call graphs for JS, we reencode the problem to reuse the WALA framework [21].

We demonstrate that building call graphs for embedded client-side code is efficient and accurate. In an empirical evaluation on several real-world PHP web applications, our analysis achieves 100% precision in identifying possible jumps within seconds. Although the general problem is undecidable, our results show that in practical cases, we can resolve most client-side relationship within server-side code. In addition, 62% of the jumps cross PHP string fragments, and 17% of them cross files, which suggests that tool support in those cases is valuable.

Our key contributions in this paper include: (1) the VarDOM representation which compactly represents DOM variations generated from the server-side code; (2) a tool infrastructure combining symbolic execution and variability-aware parsing to build the VarDOM; (3) call-graph analyses on the VarDOM for HTML, CSS, and JS tracing nodes to the original string literals in PHP code; and (4) an empirical evaluation to show the analyses’ accuracy and efficiency.

2. CHALLENGES

To illustrate the desired call graphs and involved challenges, we use the running example in Figure 1, adapted from the AddressBook-6.2.12 PHP application (see Section 6). The example shows a typical scenario in which client-side HTML, CSS, and JS code is generated from string literals in server-side PHP code. Note that the directive `<?php...?>` separates PHP code snippets from inline HTML code—string literals that are printed verbatim when the PHP program is executed. Depending on the server-side execution (i.e., depending on the values of PHP variables `$_GET`, `$ajax`, and `$rtl`), different client-side programs are generated.

Editor services in IDEs, such as syntax highlighting, syntax validation, auto-completion, “jump to declarations”, and many others are standard for most languages, including PHP, but missing for embedded client-side code. In particular, we want to build a call graph for nodes embedded in PHP string constants that would allow us to navigate from opening to closing HTML tags, from CSS rules to affected HTML elements, and from JS function calls to their declarations. We illustrate the corresponding call-graph edges for our running example in Figure 1.

In building call graphs for embedded client-side code, we face three key challenges:

1. Embedded client code. The client-side code is dynamically generated from the server-side code and is often embedded in PHP strings and inline HTML. For instance, the HTML `<form>` tag in our running example is concatenated from a string literal, the value of a PHP variable, and another string literal. HTML fragments from string literals can be printed directly with `echo` statements, but can just as well be assigned to PHP variables first, processed, and printed later as done with `$input` in our example. Call-graph edges of embedded code often cross string literals and even PHP files.

2. Conditional client code. Due to the staged nature of generating client-side code from server-side code, a call graph of client-side code within server-side code can only be approximated and the target of a call may differ among different executions of the server-side code. That is, a client-side call may have alternative possible targets, depending on how the server-side code is executed—we say that

(a) index.php

```
1 <?php
2 include ("header.php");
3
4 echo '<form method="'. $_GET['method'] .
5      '" name="searchform">';
6 if ($ajax)
7     $input = '<input ... onkeyup="update()" />';
8 else
9     $input = '<input ... onkeyup="update()" />' .
              '<input type="submit" />';
10 echo $input;
11 echo '</form>';
12 ?>
13
14 <script type="text/javascript">
15 <?php if ($ajax) { ?>
16     function update() { ...
17     }
18 </script>
19 <?php } else { ?>
20     function update() { ...
21     }
22 </script>
23 <?php } ?>
24
25 <?php include("footer.php"); ?>
```

(b) header.php

```
1 <html ... <?php if ($rtl) echo 'dir="rtl"'; ?> ...
2 <style type="text/css">
3 <?php if ($rtl) { ?>
4     #footer {float:left;}
5 <?php } ?> ...
6 </style> ...
7 </body> ...
```

(c) footer.php

```
1 <div id="footer"> ... </div>
2 </body>
3 </html>
```


Embedded string literal: `'...'`; Call-graph edge: 

Figure 1: Example PHP program, including call-graph edges within embedded client-side code in string literals

generated client-side code and edges in a call graph are *conditional*. In Figure 1, the JS call `update` refers to two different declarations depending on whether the AJAX option was selected in the server-side code. Similarly, the HTML `<script>` element is closed in two alternative locations and the footer `<div>` has different CSS styles depending on the server-side execution. Our goal is to identify all possible call-graph edges and document their server-side conditions.

3. Unresolved values. When we consider multiple execution paths in PHP code, certain values are not resolved until the PHP code is executed and may be non-deterministic in general. Values may be computed or retrieved from databases, files, or the user input (e.g., the PHP variable `$_GET['method']` obtains its value from the user’s query string). The analysis cannot statically derive their values, but must be able to work with uncertainty caused by unresolved values.

3. APPROACH OVERVIEW

Our approach builds on the observation that most of the generated client-side code that does not come from databases (75%–96% according to our prior study [3]) is contained in server-side code in the form of PHP string literals (including the inline `<?php...?>` notation) and can be largely traced through the generation process.

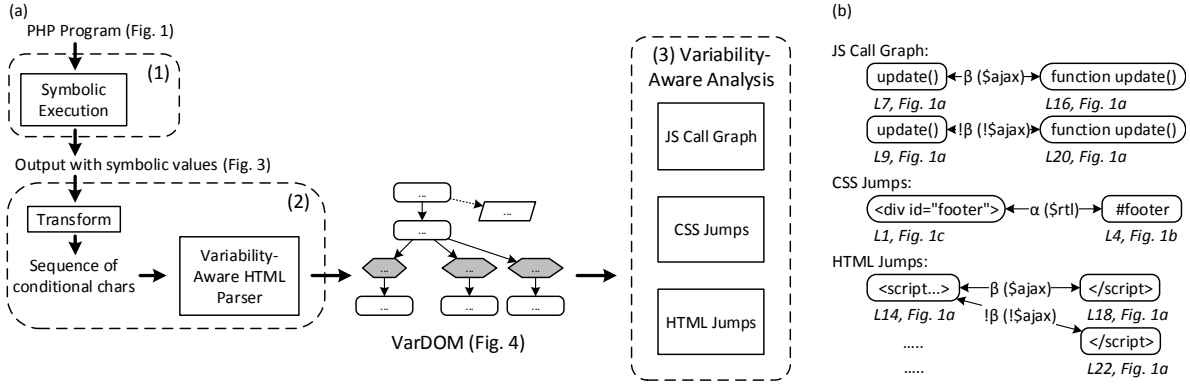


Figure 2: (a) Approach overview and (b) produced call graphs

```

1 <html
2 #if  $\alpha$  //' $rtl'
3 dir="rtl"
4 #endif
5 >
6 <style type="text/css">
7 #if  $\alpha$  //' $rtl'
8 #footer {float:left;}
9 #endif
10 </style>
11 <body>
12 <form method=" $\Phi$ " name="searchform">
13 #if  $\beta$  //' $ajax'
14 <input ... onkeyup="update()" />
15 #else
16 <input ... onkeyup="update()" /><input type="submit" />
17 #endif
18 </form>
19 <script type="text/javascript"> ...

```

Figure 3: An excerpt of the conditional output of the PHP program represented with conditional-compilation directives. Greek letters represent symbolic values in symbolic execution.

To build call graphs, we process the server-side code in three steps—symbolic execution, variability-aware parsing, and analysis—as outlined in Figure 2a.

1. Symbolic Execution. To identify how the client-side code will be generated in all possible executions of the server-side generator, we *symbolically execute* the server-side code. During symbolic execution, we follow all function calls, follow the handling (assigning, reassigning, concatenation, printing) of string literals and variables, explore all conditional branches (if statements), and explore one iteration of each loop in PHP code. We use symbolic values to represent non-determinism and interactions with the environment (e.g., reading from a database or web service, getting the current time, etc.). The result of symbolic execution is the generated client-side code (called D-Model [42]), which may contain symbolic values and in which parts of the output are printed only depending on a (symbolic) path condition. During symbolic execution, we track the origin locations of all string values. For illustration, we represent the output of symbolically executing our running example as a client-side program with `#if` conditional-compilation directives in Figure 3.

2. Parsing into a VarDOM. Next, we parse this output, which contains the (approximated) client-side code of all possible executions, with symbolic values. We ignore symbolic values (typically originating from user input or a database), since it cannot be the source of a

“jump to” editor command (there is nothing in the server-side code that a user could jump from), and it is unlikely to contain a target (see evaluation). However, we preserve the alternatives originating from exploring alternative execution paths in the PHP program. That is, if the output of a client-side code sequence depends on a (symbolic) decision in the PHP code, we preserve the notion that this sequence is conditional (depending on the path constraint).

For parsing, we transform the symbolic output into a stream of conditional characters in which each character has a formula representing the path condition under which it was produced. The token stream contains special SYM tokens to represent symbolic values. To build parse trees from conditional token sequences, we use variability-aware tooling originally developed for software product lines (e.g., parsing and analyzing all configurations of C code with `#ifdef` directives [33]). We parse the generated client-side code into a conditional DOM—called *VarDOM*—that compactly represents all its variations. A VarDOM, the core of our analysis framework, is similar to the Document Object Model (DOM) for HTML, with the addition of *condition nodes* to indicate that certain subtrees of the HTML document may vary depending on some condition. In Figure 4, we illustrate the VarDOM for our running example, with condition nodes highlighted. Note how the two JS function calls `update` and their respective declarations are represented in the same VarDOM with different conditions. Through all parsing steps, we track origin locations to VarDOM nodes. Though beyond our focus, the parser can report identified syntax errors in generated client-side code and the VarDOM can be used for syntax validation, syntax highlighting, and auto completion within PHP string literals in IDEs.

3. Analysis to build call graphs. Finally, we analyze the VarDOM to build call graphs. While traditional analyses for static client code work on a single DOM, our variability-aware analyses work on a VarDOM with conditional parts. Our analyses consider these conditions and build a conditional call graph. Call-graph nodes refer to code elements with corresponding origin locations in string literals of the server-side code. Call-graph edges represent possible jumps between nodes and may have conditions. Specifically, we create call-graph edges between opening and corresponding closing HTML tags, between CSS rules and affected HTML nodes¹, and between JS function calls and corresponding function declarations. For HTML and CSS, we implement our own analyses; for JS, we reencode the problem to reuse the existing call-graph analysis in WALA [5] (Section 5). In Figure 2b, we illustrate the analysis results for the *embedded code* of our example. Notice how the opening HTML tag `<script>` is related to two alternative closing tags with correspond-

¹These are not traditional calls, however, similar in IDE purposes.

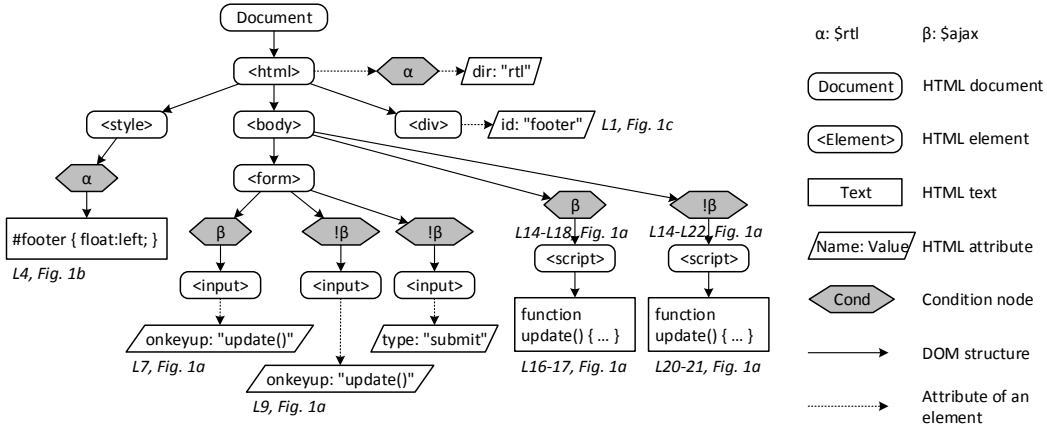


Figure 4: The VarDOM for the PHP program in Figure 1

ing conditions and how the footer element may or may not have CSS annotations in different executions. The conditional call graph is subsequently used for various forms of IDE tool support [1] (e.g., jump to declaration, jump to closing HTML tag, or find CSS rules).

4. THE VARDOM REPRESENTATION

Our core representation is the VarDOM, a compact structural representation the possible client-side programs. In the following, we first describe VarDOM and subsequently explain how we derive it from PHP code with symbolic execution and variability-aware parsing.

4.1 VarDOM Representation

Just as the DOM represents the hierarchical syntactic structure of a web page with nested nodes of four types (HTML elements, attributes, text, and comments), a VarDOM represents the tree structure of a web page with variations. The key difference is that in a VarDOM all elements can be *conditional*, that is, they are part of the web page only given a specific condition called *presence condition*. We model conditional elements with *condition nodes* in the tree, where the condition node holds the presence condition for its subtree, as illustrated in Figure 4. Note how this representation can compactly represent variations within similar pages; in our running example, possible client-side pages differ with regard to input fields depending on whether AJAX is enabled, but all pages share the same `<html>` and `<body>` elements. Conceptually, it is possible to move condition nodes up the VarDOM hierarchy by replicating code fragments yielding a less compact representation (see the choice calculus for a formal treatment [19]).

Presence conditions in condition nodes refer to decisions during the execution of the server-side code. They originate from path conditions during symbolic execution (see below). During analysis, we will attempt to exclude infeasible call-graph edges by checking whether a presence condition is satisfiable. Different formalisms are possible with different accuracy and expense tradeoffs. Conceptually, any formalism supporting conjunction, negation, and a mechanism to detect (some) unsatisfiable formulas is possible; we use propositional logic and SAT solvers, as we will explain.

4.2 Symbolic Execution

To build the VarDOM for embedded client-side code, we need to extract the client-side code from the server-side PHP code. There are several possible strategies from collecting all string literals as they occur in the server-side code [40] to tracing the execution of test cases [46, 53]. We approximate the output of all possible

executions of the server-side code by executing it symbolically, reusing our symbolic-execution engine *PhpSync*, developed in prior work [42]. Symbolic execution assumes all unknown values (user input, I/O, nondeterminism, and so forth) as symbolic. When reaching control-flow decisions, symbolic condition explores all possible paths, keeping track of a path condition and ignoring executions with infeasible path constraint. During symbolic execution, we track all output of the executed PHP code. Output fragments can be concrete, produced from string literals inside the PHP code (possibly after several reassignments, concatenation, and other string processing steps), or symbolic. For each output fragment, we record the path constraint under which it was produced. Additionally, we track the origin location of each string literals such that we can map all output back to the original PHP literals. The output of symbolic execution represents the (usually infinite) number of possible client-side implementations generated from the PHP code as a stream of characters and symbolic values, in which each entry has a path constraint and each character has origin information. To support developers' understanding, we store not only the (symbolic) path condition, but also a textual representation of the corresponding conditions' PHP code as comment, as shown in Figure 3.

Our symbolic-execution engine *PhpSync* is unsound but efficient and effective at approximating all possible outputs of the PHP program. It performs only coarse tracking of symbolic values (e.g., $\alpha + 1$ is tracked as a new symbolic value β instead of tracking expressions exactly; we only track string operations such as concatenation exactly); it explores exactly one iteration of each loop and aborts on recursion; it does not support class and object features in PHP 5 yet; and it uses only conservative approximations to detect some infeasible paths (which is undecidable in general). Since our goal is not detecting bugs in the PHP execution, but to extract call-graph edges within embedded string literals for tool support, these simplifications are acceptable: When analyzing embedded client-side code from string literals, we do not care how often these are printed within a loop; exploring infeasible paths will at most identify additional call-graph edges with infeasible conditions (see next section) which a user can often identify quickly.

4.3 Parsing

For further analysis, we need to transform the character stream into a VarDOM that represents the structures of the client-side implementation. Since the output of symbolic execution contains conditional characters (only printed under given path constraints) and symbolic characters, parsing is challenging. Fortunately, the prob-

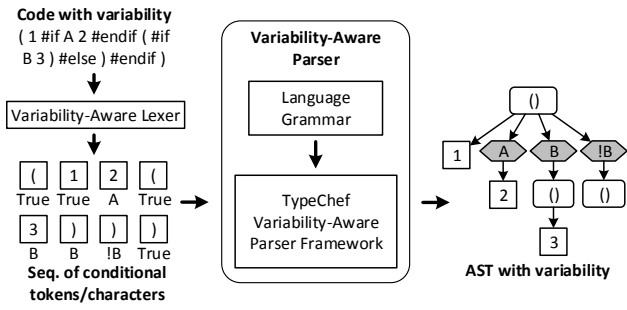


Figure 5: Example of variability-aware parsing with TypeChef

lem resembles closely the challenge of parsing unprocessed C code that still contains `#if` directives, which we solved recently [33]. To illustrate the similarities, we already listed the output of symbolic execution as code with `#if` directives in Figure 3. Next, we describe our variability-aware parser framework *TypeChef* originally developed for C code and subsequently explain how we use it to parse the VarDOM with a lexer, a SAX parser, and a DOM parser.

1. The variability-aware parser framework. The parser framework *TypeChef* provides a library of variability-aware parser combinators that can be used to develop parsers for different languages. All parsers expect a sequence of conditional elements (with a presence condition each) as input and produce a single parse tree with condition nodes (again with presence conditions) as a result. The parser combinators handle variability during the parsing process by forking subparsers when encountering conditional tokens and join subparsers subsequently. Since it exploits sharing among configurations, variability-aware parsing is much faster than a brute-force approach of parsing all variants separately. Details of the *TypeChef* parser framework are described elsewhere [33], and equivalent strategies based on LR parsers have been developed subsequently [23]; here we use *TypeChef* as is to develop new variability-aware parsers.

Parsers written with the *TypeChef* combinators do not make any assumptions about how presence conditions are used in the input, i.e., `#if` directives do not need to align with the underlying code structure. Variability-aware parsing is sound and complete with regard to a brute-force approach that would parse every possible configuration separately with a traditional parser; it reports conditional parser errors if some configurations cannot be parsed and returns an parse tree for the remaining configuration. In case presence conditions do not align with the underlying structure of the host language, the parser parses local sequences multiple times and locally replicates code until alternative conditional parsing structures can be recognized, as visible from the S-expression example with two alternative closing parentheses for the inner S-expression.

As an illustration, Figure 5 shows variability-aware parsing of S-expressions [39] using *TypeChef*. A list of conditional tokens (created from a code fragment with `#if`) is parsed into an S-expression structure with three conditional children, precisely and compactly representing the possible S-expressions in all configurations without replicating shared structures such as the literal `'1'`.

2. From symbolic-execution output to conditional character streams (Lexing). From symbolic execution’s output, the lexer produces a sequence of conditional characters. The condition of each character is derived from the path constraint under which the output was produced during symbolic execution (represented with `#if` directives in Figure 3). We preserve the same formalism for formulas and satisfiability checking used during symbolic execution (propositional formulas and SAT solvers in our implementation). The lexer

produces a special SYM token for symbolic values in the output and propagates origin locations from the PHP code for every individual character. We exemplify the step for our example in Figure 6.

3. SAX parser. To deal with the complexity of HTML, we proceed in two steps common in HTML parsers. The first step recognizes nodes and their attributes in a flat structure (SAX-style parsing) whereas the second step builds a tree from those nodes (DOM parser). The SAX parser takes the stream of conditional characters and produces a list of conditional nodes. Nodes can be opening tags with a name and possibly conditional attributes (e.g., `<div id="i">`), closing tags with a name (e.g., `</div>`), or text fragments containing possibly conditional characters. The parser accepts symbolic tokens in text, as tag names, as attribute names, as attribute values, or as whitespace. As seen in Figure 6, this parsing step produces a very shallow parse tree of a document with a list of conditional tags, texts, and comments, of which start tags can contain conditional attributes, and texts/comments can contain conditional characters. The parser framework propagates origin locations and presence conditions.

4. DOM parser. In the second parsing step, we use the document’s list of conditional nodes from the previous step as conditional token sequence for the subsequent DOM parser. The DOM parser itself is simple, since it recognizes only a tree structure based on matching starting and closing tags. However, the context-sensitive nature of checking well-formedness in HTML requires matching names in opening and closing tags, for which we wrote a simple combinator. Again, the parser framework propagates origin locations and presence conditions. In the VarDOM of our running example, exemplified in Figure 6, the first input field is guarded by condition node β (`'$ajax'`), and the other input fields are guarded by condition nodes $\neg\beta$ (`'!$ajax'`). (Note: we simply propagate attributes inside nodes and variations inside text nodes from the SAX parser.)

5. Reporting parsing errors. During parsing, the two parsers reject ill-formed HTML code (the first parser rejects invalid syntax of tags and attributes and the second parser rejects invalid nesting and missing closing tags). In each case, the parsers report a conditional error message for invalid configurations (with location information tracked from the initial PHP code) and a parse tree for the remaining configurations. For example, the HTML DOM parser would report a missing closing tag for `<div>` in line 2 in the example below in the configurations with `$C` evaluating to true. Although relaxed or error-recovering parsing is conceptually possible, rejecting ill-formed code has the additional benefit of being able to report client-side errors during development while embedded in server-side code. In addition, we could easily check validity and other invariants on top of the VarDOM representation, reporting conditional error messages when structural assumptions are violated.

PHP code:	Output of symb. exec.:
1 if (<code>\$C</code>)	1 <code><form></code>
2 <code>\$div = '<div>';</code>	2 <code>#if α // '\$C'</code>
3 else	3 <code><div></code>
4 <code>\$div = '';</code>	4 <code>#endif</code>
5 echo <code>'<form>'. \$div. '</form>'</code>	5 <code></form></code>

6. Assumptions and Limitations. The output of symbolic execution may contain symbolic values representing a potentially infinite number of possible client-side programs. The symbolic execution engine however explores only a finite number of paths in the server-side code (due to our simplifications with regard to loops and recursion, Section 4.2), which allows us to parse the output into a structure with a finite number of variations expressed through condition nodes, assuming that symbolic values do not affect the output’s structure.

Specifically, we assume that symbolic values produce tag names, attribute names, or attribute values when used in that location and

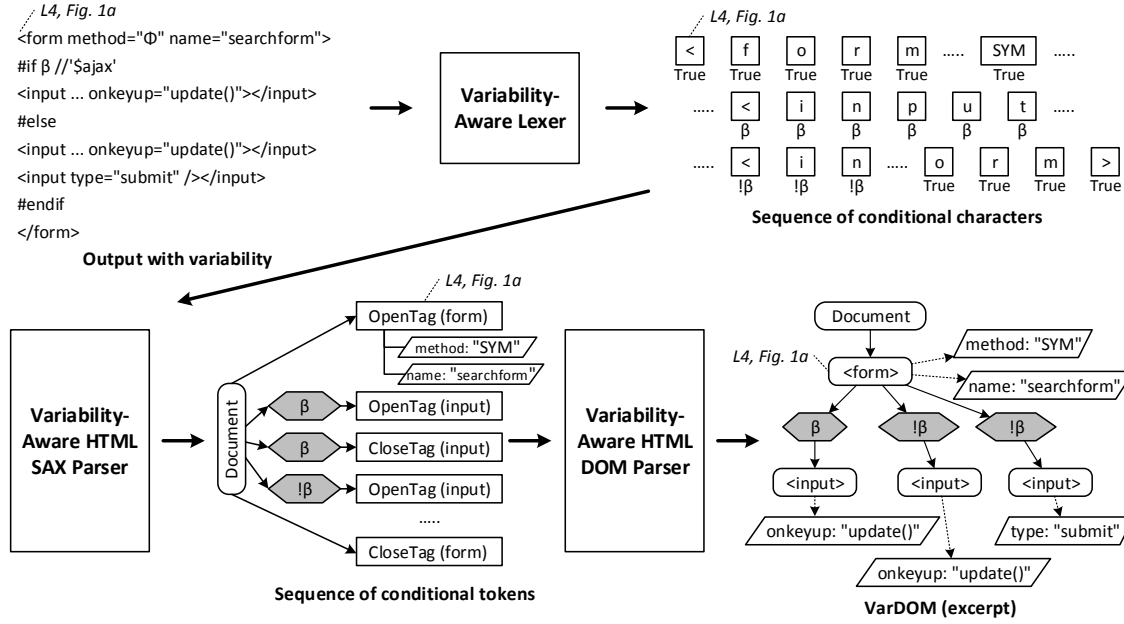


Figure 6: Parsing output with variability to build a VarDOM

produce well-formed HTML fragments otherwise (in which case we can parse it as text or white space). For example, we fundamentally could not parse the output ‘<div> Ψ ’ if we had to assume that Ψ might provide the closing tag. Although it is easy to construct artificial examples that we cannot parse, we have not seen a code fragment in practice in which a symbolic value (e.g., from user input or another source of nondeterminism) affected the structural well-formedness of produced client-side code.

5. BUILDING CALL GRAPHS

The VarDOM is the basis for all subsequent analyses to build conditional call graphs that can be used for tool support. Even though not executable, we interpret relationships among HTML and CSS elements as call-graph edges as well, since they equally provide a foundation for corresponding editor services. In general, a conditional call graph consists of nodes reflecting positions in the server-side code² and edges representing relationships among those nodes (calls, corresponding tags, affected CSS rules, and so forth). Edges in a conditional call graph can have a presence condition, meaning that the node is related to another node only in certain executions of the server-side program. For example, an opening HTML tag can be closed by two different closing tags resulting in two conditional call-graph edges, as illustrated with the script tag in our running example (see Figures 1 and 2b). As usual, call graph edges can be navigated in both directions for different editor services, e.g., “jump to declaration” versus “find usage”. If the server-side code has multiple entry points (e.g., multiple .php files a user can call), we build VarDOMs and analyze them separately for each entry point and subsequently merge all call graphs, in which the nodes point to the same PHP code locations.

We illustrate three analyses to extract conditional call graphs for IDE support in HTML (“jump to opening/closing tag”), CSS (“jump to affected nodes” and “find applicable CSS rules”), and JavaScript (“jump to declaration”), each while embedded in server-side code.

²Technically, a node can refer to multiple positions in the source code if it was concatenated from multiple string literals.

5.1 Supporting HTML Jumps

For an HTML jump, we define a source as an *HTML opening tag* and a target as its corresponding *closing tag* (either the entire tag or the name of the tag). This type of jumps is useful in helping developers understand the structures of HTML tags that would be produced by their PHP program and find closing elements especially if they are generated from different PHP files, such as the body tag in our running example.

Building call-graph edges for HTML jumps is straightforward. We simply traverse the VarDOM and look up the origin locations of the opening and closing tags of each element (as explained in Section 4.3, tokens representing opening and closing tags produced by the SAX parser are used in the DOM parser). We create a call-graph edge between those locations with a presence condition reflecting the element’s presence condition in the VarDOM (i.e., a conjunction of all condition nodes between the element and the VarDOM’s root). When an element has alternative opening or closing tags, it occurs repeatedly in the VarDOM under alternative presence conditions (e.g., the two <script> elements in Figures 2b and 4).

5.2 Supporting CSS Jumps

CSS code consists of CSS rules to define styles for HTML elements. We create call-graph edges between CSS rules and all HTML elements selected by them, to support navigation among those elements within server-side code, similar to the debugging facilities that many browsers provide for generated client-side code.

Unlike our HTML-jump analysis in which opening and closing tags are directly available in the VarDOM, we need to extract CSS rules from text fragments in the VarDOM—typically from <style> tags and from included (potentially generated) files. Notice that each CSS fragment is a sequence of conditional characters, in which we preserved the presence condition of the originating HTML element, and which may contain symbolic values. To analyze the CSS code with its variations, we wrote another variability-aware parser with TypeChef to recognize CSS as a list of conditional rules (ignoring symbolic values as white space), illustrated in Figure 7. The parser framework propagates origin locations and presence conditions.

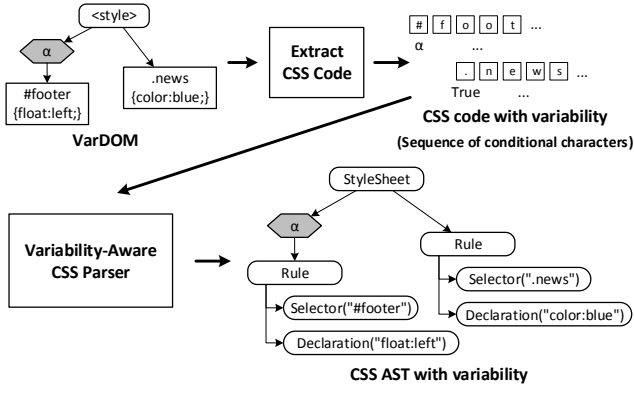


Figure 7: Parsing CSS code with variability

After parsing, we only need to match the selector of each CSS rule against the VarDOM nodes. We create a call-graph edge for every match in the VarDOM, each with a presence condition that conjuncts the presence condition of the CSS rule with the presence condition of the matched HTML element. Edges with infeasible presence conditions can be filtered as far as the used formalism supports it. We reimplemented the matching algorithm for the most common selectors (class selectors, id selectors, element selectors, and nested selectors), implementing remaining selectors is technically straightforward following the specifications of CSS [2]. We list the CSS-related call-graph edge for our running example in Figure 2b.

5.3 Conditional JS Call Graph

To build a conditional call graph for JS, in general, we can develop a variability-aware analysis from scratch in line with our solution for CSS. However, due to the complexity of building call graphs already for non-embedded JS code without conditional parts [21], we want to reuse existing infrastructures for building JS call graph. Our key idea is to reencode JS code with generation-time variability as JS code with runtime variability, in which the presence conditions of JS code are encoded as the conditions of regular JS if statements enclosing those code elements. After transforming the code into regular JS code (still tracking origin locations), we reuse WALA, an existing state-of-the-art tool for building a JS call graph [5]. Through consistent origin tracking, we can translate the identified call-graph nodes and edges back to their location in the VarDOM and hence also back to the original string literals in PHP code. We derive presence conditions for call-graph edges from the presence conditions of the involved VarDOM nodes, again filtering infeasible edges. Our goal is to provide editor support, such as “jump to declaration”, for JS code embedded in server-side code for every call-graph edge that WALA can identify on the generated JS code.

1. Parsing JS code. As for CSS, we first extract all JS code fragments from text elements the VarDOM. Specifically, we collect the content of HTML `<script>` tags, linked JS files, and all event handlers, such as `onload` and `onclick`. Again, we build a variability-aware JS parser on top of the TypeChef framework that accepts a sequence of conditional characters and produces a JS parse tree with conditional nodes (ignoring symbolic values). We follow the JS grammar specification [4], but ignore the context-sensitive semicolon-inserting feature in our prototype. To simplify subsequent steps, we push up conditional nodes to the level of statements, that is, variations inside statements are expanded to two alternative statements.

2. Reencoding variability. After parsing, we reencode generation-time variations (condition nodes with presence conditions in the

AST) as runtime variations with if statements. This strategy, named *configuration lifting* or *variability encoding*, has been used in model checking and deductive verification of product lines to reuse analysis techniques that are oblivious to generation-time variations but can handle runtime variations [45, 51, 7, 6]. We reencode variability with the following two key transformation rules:

R1: $\text{Condition}(cond, \text{Statement}(stmt)) \rightarrow \text{IfStatement}(\text{String}(cond), \text{Statement}(stmt))$

R2: $\text{Condition}(cond, \text{FunctionDeclaration}(name, params, body)) \rightarrow \text{IfStatement}(\text{String}(cond), \text{ExpressionStatement}(\text{AssignmentExpression}(\text{Identifier}(name), "=", \text{FunctionExpression}(params, body))))$

Rule R1 reencodes a statement *stmt* under presence condition *cond* (if *cond* \neq true) as a JS if statement with a condition representing *cond* and *stmt* in the then branch (e.g., lines 12–14 of Figure 8). Rule R2 for a function declaration similarly reencodes function declaration as an equivalent assignment of a function expression inside an if statement (e.g., lines 13 and 18 of Figure 8).

Conceptually, it is possible to prove that a reencoding maintains the execution semantics of all configurations, e.g., by showing that the execution semantics of executing the program is not affected by the reencoding in any configuration (where a configuration would either remove unnecessary code at generation time or initialize the configuration parameters equivalently to be interpreted at runtime). For our purpose, a strict notion of correctness is not necessary, since we perform an unsound call-graph analysis subsequently. It is sufficient to encode the program in a way that the analysis tool can identify the correct call-graph edges; hence we checked correctness of our encoding through testing only.

3. Reencoding HTML code. JS code can interact with the HTML DOM during execution (check existence of an element, access its properties, and so forth), which the used analysis framework WALA takes into account to some degree. WALA takes as input an HTML document and turns it into pure JS code which generates the document using JS instruction `document.createElement`. WALA’s call graph building algorithm for JS is then applied on this transformed JS code. Therefore, we reencode the entire HTML code by removing all condition nodes from the VarDOM. The resulting HTML document contains all possible alternative nodes in which all JS code is properly reencoded. This reencoding of HTML is a crude approximation, but sufficient for WALA’s analysis in our experience. For instance, we reencode our running example as in Figure 8 and analyze it with WALA.

4. Call-graph generation with WALA. To create the actual call graph, we run WALA for JS [5] on the reencoded HTML/JS code. We take WALA’s result as is and track nodes back to their origin in the VarDOM and in the PHP code. We create presence conditions for call-graph edges from the conjunction of the presence condition of the two involved VarDOM nodes. In our running example, WALA in fact ignores the conditions of the created if statements and creates a total of four call-graph edges from each update call to each function declaration. However, since both update calls and declarations depend on the same symbolic path condition in our example (i.e., we know that expression `$ajax` has the same value β in both server-side if statements), two call-graph edges receive infeasible presence condition $\beta \wedge \neg\beta$ and can be discarded. We show the JS call graph for our running example in Figure 2b.

6. EMPIRICAL EVALUATION

Our approach to build call graphs has a number of sources of potential inaccuracies. While variability-aware parsing and call-graph building for HTML and CSS are conceptually sound, symbolic

```

1 <html dir='rtl">
2   <style type='text/css">
3     #footer {float: left}
4   </style>
5 <body>
6   <form method="Φ" name="searchform">
7     <input ... onkeyup="if ('β') update()"/>
8     <input ... onkeyup="if ('!β') update()"/>
9     <input type='submit' />
10  </form>
11  <script type='text/javascript">
12    if ('β') {
13      update = function () {...}
14    }
15  </script>
16  <script>
17    if ('!β') {
18      update = function () {...}
19    }
20  </script>
21  <div id='footer"> ... </div>
22 </body>
23 </html>

```

Figure 8: Reencoding variability

execution may not produce output for all string literals, parsing has limitations regarding symbolic values, and JS analysis uses potentially inaccurate reencodings, as we explained. In an evaluation with real-world PHP applications, we investigate the *practicality* and *accuracy* of our approach.

In addition, IDE support for navigation is especially useful if call-graph edges are nontrivial (e.g., developers need to search across files). Thus, we investigate the *complexity* of the created call graphs to characterize the *usefulness* of IDE support based on our tooling.

Experiment Setup. We collected five PHP web applications (Table 1) from sourceforge.net with various sizes and without heavy use of object-oriented constructs (also used in related work [42, 46]). For each system, we selected a main file, which might also recursively include other files, and ran our analysis on that file. When encountering HTML syntax errors during parsing (most often due to missing closing tags), we manually fixed them in the PHP code and report results after applying all fixes. All subject systems have multiple entry points (PHP files that can be called by a user), the call graphs of which can be merged, but due to the manual effort of fixing syntax errors, we considered only a single main file per system. The main entry file, the number of symbolically executed PHP statements, the size of the generated symbolic output, the size of JS code, and the number of fixed errors are listed in Table 1.

6.1 Practicality and Accuracy

All call-graph computations completed within a few seconds (< 8 seconds on average, < 12 seconds in the worst case). Since this performance is acceptable for executing analyses in the background of an IDE, we did not perform additional rigorous performance measurements. Currently, we need to fix all HTML syntax errors before we can build call graphs, which is a positive side effect, but also a laborious endeavor. An error-recovering parser [17] or automated repair tools [46] could help with this step. Overall, our tool is easy to set up for a new project by pointing it to the main file(s).

Precision. In Table 2, we list the number of call-graph edges detected in all subject systems. We manually checked the created call graphs for correctness. In small call graphs (< 100 edges), we inspected all call-graph edges; in large call graphs, we randomly sampled 50 edges each. We consider a call-graph edge as correct

when it connects two nodes tracked to PHP string literals, and the nodes are actually in a call relationship. All 604 checked edges were correct, yielding a *precision* of **100%**.

Recall. In the absence of ground truth, *recall* is more challenging to measure. We approach recall with three proxy metrics—(1) **coverage**, (2) **symbolic discipline**, and (3) **reencoding losses**—addressing the three sources of inaccuracies in our approach.

1. First, we can cover only literals that are output as part of an execution of the PHP code (string literals in dead code are never printed and we cannot build call graphs for them). In addition, due to limitations of our symbolic-execution engine (see Section 4.2), we may not cover every possible execution path or may not be able to track some string literals to their output. To characterize the potential loss of call-graph nodes, we measure **coverage** as the ratio between the number of output characters that are covered by symbolic execution and the total number of all output characters in the project. As a heuristic to ignore string literals that are not output, such as array index ‘method’ in `$_GET[‘method’]`, we only consider literals that contain the representative HTML tag-opening character ‘<’. When executing the single main file, we cover 6% to 32% of all characters. When symbolically executing all entry points of the PHP code, we cover on average **84%** of all characters, (see the last column in Table 1). This shows that symbolic execution can achieve high coverage in our subject systems.

2. Second, we manually inspected all occurrences of symbolic values in the output of symbolic execution. We did not find a single case where a symbolic value was relevant for the structure of the VarDOM. That is, a symbolic value may add substructure but in no case was it required to provide the closing tag for a concrete opening tag or similar structural parts. This confirms our assumptions in Section 4.3 and makes inaccuracies due to symbolic values unlikely in our subject systems.

3. Third, while HTML and CSS analyses are sound with regard to a brute-force approach, our reencoding for JS analysis may lead to documents in which the used JS-call-graph tool WALA cannot find all call-graph edges. Note that we do not want to check the quality of WALA’s call graphs but only to what degree our reencoding prevents discovering call-graph edges that WALA could discover without reencoding. Thus, we generated random configurations from the symbolic output (i.e., different selections for the #if decisions) and executed WALA on the generated code (without variability and without reencoding), comparing the resulting call-graph edges with the ones identified when analyzing the entire configuration space with reencoding. We did not find a single case where reencoding lost call-graph edges compared to analyzing individual configurations.

Overall, the results and performance are practical and promising. In our subject systems, our tool yields a perfect precision. Investigating the sources of inaccuracies that lead to reduced recall shows that symbolic execution can cover 84% of string literals and that symbolic values during parsing and reencoding of JS are unproblematic. More details are in Table 1.

6.2 Complexity

Besides accuracy, we investigated the potential benefit of such support. Using the created call graphs, we measure several characteristics to show the complexity of the underlying problem, suggesting the effort required when developers do not have IDE support.

First, we investigated the locality of call-graph edges. Call-graph edges are often nonlocal. 46% of all HTML call-graph edges on average and up to 100% of JS call-graph edges connect nodes in different string literals (Table 2). That is, the connected elements are written in different parts of the server-side code, with some PHP code between them. While many call-graph edges for HTML

Table 1: Subject systems and coverage

Subject System	Version	Files	LOC	Main Entry	Exec. Output Size			Re-encoded JS HTML			Coverage	
					Stmts	Chars	Conds	Stmts	Re-enc	Errors	Main Entry	All Entries
AddressBook (AB)	6.2.12	100	18,874	index.php	1,546	15,480	195	223	28	12	32%	94%
SchoolMate (SM)	1.5.4	63	8,183	index.php	2,386	30,318	52	188	46	155	24%	92%
TimeClock (TC)	1.04	69	23,403	timeclock.php	1,311	15,157	99	128	36	22	11%	63%
UPB	2.2.7	394	104,613	admin_forums.php	5,175	35,587	711	876	0	30	6%	79%
WebChess (WC)	1.0.0	39	8,589	index.php	93	3,409	1	84	0	0	6%	97%

Table 2: Complexity of call graphs

Sys.	HTML Jumps			CSS Jumps		JS Jumps		Strings on echo
	Total	xStr	xFiles	Total	xFiles	Total	xStr	
AB	345	68%	2.3%	157	100%	7	85.7%	90%
SM	610	19%	6.2%	127	100%	33	100%	81%
TC	269	49%	4.5%	74	100%	2	0%	99%
UPB	386	49%	4.4%	136	100%	75	0%	44%
WC	40	63%	0%	20	100%	4	75%	97%
Tot/Avg	1,650	46%	3.5%	514	100%	121	52%	79%

xStr/xFiles: Number of call-graph edges that cross strings/files.
Avg: Geometric mean for non-zero relative numbers (percentages).

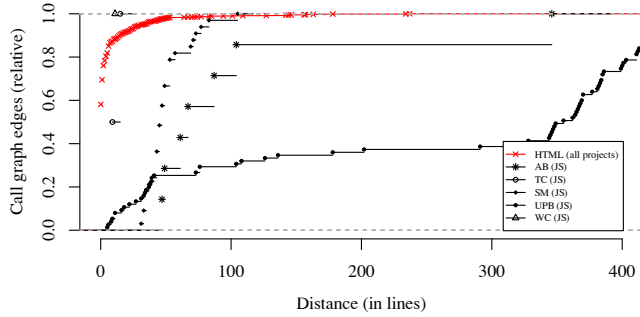


Figure 9: Cumulative distribution of distance of HTML and JS jumps (within the same file), depicting the percentage of jumps shorter than a given distance.

are relatively local, with source and target in the same line or only few lines apart, about 8.3% are more than 20 lines apart and 3.5% are even in different PHP files. JS call-graph edges often span a larger distance, an average of 196 lines, but were mostly within the same PHP file. For CSS, even all call-graph edges connect nodes in different files since all CSS rules are written in separate files in our subject systems. Tool support is especially valuable for long jumps and jumps across files. Details are listed in Table 2 and Figure 9.

To characterize how difficult it is to track string literals through PHP code, when they are (re)assigned, concatenated, or part of other computations, we also tracked how many string literals are printed immediately or appear as inline HTML code in a PHP file. Again, we track only string literals containing the character '<'. We found that on average 21% of all such string literals are assigned to variables before they are eventually printed at a different location (Table 2). Our analysis can follow these string literals and create correct call graphs, while navigation without IDE support might not be straightforward.

Second, conditional jumping where one HTML tag is closed by alternative closing tags depending on the server-side execution is a

challenge for tools and humans. Our solution with variability-aware parsing can correctly handle those cases and create corresponding conditional call-graph edges. We found two cases where the source of an HTML jump has multiple targets in AddressBook and TimeClock each, both similar to the <script> tag in our running example. Here, a developer may accidentally finish manual search before finding all relevant closing tags. In addition, we found 21 cases in SchoolMate in which call-graph edges among nodes with the same name are disambiguated by their respective presence conditions, similar to the JS update call in our running example. Our results show that the produced client-side structure mostly aligns with the server-side execution, so that these cases are relatively rare in practice. They also demonstrate that our more powerful infrastructure can provide accurate results in common as well as difficult cases.

Finally, a common approach for navigation in embedded client-side code is to use a global text search, especially for nonlocal jumps. A naive global text search for closing HTML tags, such as table, a, and form yields hundreds of results in dozens of files even in the smaller PHP projects. A global text search is only more promising for rare tags, such as html and body and uncommon JS variables and function names. For CSS, global text search is almost useless. A developer would not perform a *global* search in most cases and in many cases the corresponding jump target is only few lines away (see Figure 9), but nesting of HTML tags, common jumps across string literals, and occasional jumps over many lines of code and even across files (see Figure 9 and Table 2) show that a local search is also not a universal strategy. The relatively rare but possible case that a jump has alternative targets depending on the server-side execution (as the <script> tag in our running example), emphasizes that an incomplete local search may actually miss important targets potentially leading to inconsistencies. Overall, we conclude that text search can be an effective alternative in many common local cases but that a call graph can support navigation in many nontrivial cases quantified throughout our evaluation.

6.3 Threats to Validity

Regarding external validity, we selected only a small sample of medium-sized subject systems and investigated only a single main file per project, due to the main bottle neck of manually fixing HTML syntax errors. While we cannot generalize over arbitrary PHP systems, all systems are real-world open-source applications developed by others and the results are consistent over all systems.

Regarding internal and construct validity, we used various proxy metrics to carefully characterize possible recall and usefulness measures. Since we do not have ground truth about what call graphs to expect, we decided to break down the evaluation into the three sources of inaccuracies. Implementation defects may also reduce precision and recall, but our tests and manual investigations did not reveal any issues. Instead of performing a user study in which the navigation benefit may be buried in noise or over-exaggerated with artificial tasks or material, we decided to characterize usefulness by

quantifying nontrivial call-graph edges in which developers could likely benefit from a tool. We expect a strong correlation with actual improvements in practice, but a usability study is still required.

7. RELATED WORK

State-of-the-art IDEs do not provide call-graph-based editor support such as “jumps to declaration” for embedded client code within server-side web application. However, there exist analysis approaches for embedded client code to support other software engineering tasks.

Analyzing generated client-side HTML code. The Eclipse plugins PHPQuickFix and PHPRepair [46] detect and fix errors in server-side PHP applications leading to ill-formed generated HTML. PHPQuickFix examines *constant prints*, i.e., the PHP statements that print directly string literals and repairs HTML ill-formed errors. Whereas we symbolically execute PHP code to track how string literals are processed, it analyzes each string literal separately and can only identify local issues. In contrast, PHPRepair follows a dynamic approach in which a given test suite is used to generate client-side code with different server-side executions, while tracing the origin of output strings. In contrast to their dynamic strategy, we symbolically execute PHP code to approximate all possible executions and subsequently use a variability-aware parser to analyze them all. We additionally detect well-formedness issues.

In Wang *et al.* [53], a user generates the client code by executing the (instrumented) server-side code with a specific input. Changes in that generated code can then be mapped to their origins in the PHP code using the recorded run-time mappings and static impact analysis. We similarly track origin locations, but we symbolically execute the PHP code. We previously used our symbolic execution engine *PhpSync* for similar purposes [41, 42]. First, we used our static origin tracking to propagate changes in the client code (output of symbolic execution) back to the PHP code [42]. Subsequently, we designed DRC to analyze both PHP code and generated client-side code to detect cross-language and cross-stage dangling references [41]. In contrast to this work, DRC extracts program entities via heuristics: it just matches path constraints of references/declarations without building the ASTs and DOM for embedded code in different configurations. Apollo [8, 9], a more general fault localization tool for PHP code, rates the PHP echo/print statements higher in suspiciousness by running an instrumented interpreter on test cases.

Minamide proposed a string analyzer [40] that takes a PHP program and a regular expression describing the input, and validates approximate HTML output via context-free grammar analysis. Wang *et al.* [52] compute the approximated output of PHP code and identify the constant strings visible from a browser for translation. Both do not aim to analyze multiple variants of embedded code in JS or CSS. Several string taint-analysis techniques were built for PHP web programs and software-security problems [35, 54, 55, 56]. They can benefit from our VarDOM to analyze the embedded client code.

Outside the web context, there has been significant research on analyzing generators and guaranteeing invariants for the generated code [15, 26, 27, 43]. Staged programming languages such as MetaML integrate generators inside the language and can guarantee well-typedness of a program across all stages [31, 49]. They can give precise guarantees, but are only applicable with restricted, well-designed meta languages, not for arbitrary PHP/JS computations.

String embedding of DSLs is another form of two-stage computations [22], in which a string, such as an SQL command, is constructed in the host language and subsequently executed by a DSL interpreter. Here, the DSL code appears as string literals in a host language without IDE support, just as HTML code appears as strings in PHP. While other DSL-implementation approaches (e.g., pure embedded [28], extensible languages [18], external DSLs [34])

can potentially provide support for navigating DSL code, our infrastructure is applicable for simple string embedding as well.

Symbolic execution. Symbolic execution was initially proposed as a program testing approach [36]. More recently, many forms of dynamic symbolic execution have been proposed to work around undecidability problems by combining symbolic execution with testing to guide the execution to parts of a program [13, 14, 24, 48]. We use a simple form of symbolic execution to explore many possible executions in a web application to produce possible outputs for further analysis, not with the goal of finding bugs in PHP code.

JavaScript call graphs. There is a significant amount of work on constructing call graphs for JS code [20, 21, 25, 29, 30, 38, 47]. Accuracy and performance are challenges due to the dynamic nature of JS code and its interactions with the DOM and the browser. These tools are aimed at improving IDE services, but they all target plain client-side code, not code embedded in server-side programs.

Variability-aware parsing and analysis. The challenge of parsing all configurations of C code without preprocessing it first lead to significant research on parsing, initially with restrictions or heuristics [11, 44] and later in a sound and complete fashion supporting arbitrary use of conditional compilation with our parser-combinator framework TypeChef [33] and later with a modified LR parser [23]. In this work, we used these techniques developed for C and built four new parsers (HTML SAX, HTML DOM, CSS, and JS).

An abstract syntax tree with variations (either optional or choice nodes [19]) is a common abstraction for further variability-aware analysis over large configuration spaces [10, 12, 32, 37, 50], typically targeted at analyzing all configurations of a software product line without resorting to a brute-force approach of analyzing each configuration separately. Variability-aware analysis can be sound and complete with regard to analyzing all configurations separately. A common strategy to avoid reimplementing variability-aware versions of existing analyses is to reencode the build-time variability as runtime variability and use existing analysis mechanisms that can handle runtime variations, such as model checking [7, 45, 51]. We build a new variability-aware analysis to derive call-graph edges for CSS and build variability encoding for JS to reuse WALA, following common strategies. For an overview of this field, see a survey [50].

In this work, we interpret the output of symbolic execution (representing potentially infinitely many executions) as conditional input by ignoring symbolic values in the output and taking path conditions as configuration options. Although this is an unsound approximation, we have shown that this is sufficient for our purposes and allows us to build on the experience available with variability-aware tooling.

8. CONCLUSION

Due to the staged nature of dynamic web applications, supporting analysis on the client code while it is still embedded in the server code is a challenging problem. This paper proposed an analysis framework for embedded client-side code. We introduced the VarDOM, a representation that compactly models all possible DOM variations of the generated client code. To build a VarDOM, we applied symbolic execution and variability-aware parsing techniques on a given PHP program. We then implemented our own analysis for HTML/ CSS, and reencoded the problem to reuse WALA to build call graphs for JS. Empirical evaluation on real-world systems showed that our tool can achieve high accuracy and efficiency.

9. ACKNOWLEDGMENTS

This project is funded in part by US National Science Foundation grants: CCF-1018600, CNS-1223828, CCF-1318808, CCF-1349153, and CCF-1320578.

10. REFERENCES

- [1] Building call graphs for embedded client-side code in dynamic web applications. http://home.engineering.iastate.edu/~hungnv/Research/Varis/?page=video_demo.
- [2] CSS selectors. <http://www.w3.org/TR/CSS21/selector.html>.
- [3] Detection of embedded code smells in dynamic web applications. <http://home.engineering.iastate.edu/~hungnv/Research/WebScent/>.
- [4] ECMAScript language specification - ECMA-262 edition 5.1. <http://www.ecma-international.org/ecma-262/5.1/>.
- [5] WALA tools in JavaScript. http://wala.sourceforge.net/wiki/index.php/Main_Page#WALA_Tools_in_JavaScript.
- [6] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.
- [7] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491. IEEE Computer Society, 2013.
- [8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 265–274. ACM Press, 2010.
- [9] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng.*, 36(4):474–494, 2010.
- [10] L. Aversano, M. D. Penta, and I. D. Baxter. Handling preprocessor-conditioned declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 83–92. IEEE CS, 2002.
- [11] I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. IEEE Computer Society, 2001.
- [12] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL^{LIFT}: Statically analyzing software product lines in minutes instead of years. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM Press, 2013.
- [13] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 122–131. IEEE Computer Society, 2013.
- [14] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.
- [15] W. Choi, B. Aktumur, K. Yi, and M. Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 81–92. ACM Press, 2011.
- [16] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press / Addison-Wesley, 2000.
- [17] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Trans. Program. Lang. Syst.*, 34(4):15:1–15:50, 2012.
- [18] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 391–406. ACM Press, 2011.
- [19] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.
- [20] A. Feldthaus, T. Millstein, A. Möller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 119–138. ACM Press, 2011.
- [21] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 752–761. IEEE Press, 2013.
- [22] M. Fowler. *Domain-Specific Languages*. Pearson Education, 2010.
- [23] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM Press, 2012.
- [24] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM Press, 2005.
- [25] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.
- [26] S. S. Huang and Y. Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 33(2):6:1–44, 2011.
- [27] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, volume 3676, pages 309–326. Springer-Verlag, 2005.
- [28] P. Hudak. Modular domain specific languages and tools. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 134–142. IEEE Computer Society, 1998.
- [29] S. H. Jensen, M. Madsen, and A. Möller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 2011.
- [30] S. H. Jensen, A. Möller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the International Static Analysis Symposium (SAS)*. Springer-Verlag, 2009.
- [31] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 86–96. ACM Press, 1986.
- [32] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3):14:1–14:39, 2012.
- [33] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM Press, 2011.
- [34] L. C. Kats and E. Visser. The Spoofox language workbench: Rules for declarative specification of languages and IDEs. In *Proc. Int'l Conf. Object-Oriented Programming, Systems,*

- Languages and Applications (OOPSLA)*, pages 444–463. ACM Press, 2010.
- [35] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 199–209. IEEE Computer Society, 2009.
 - [36] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
 - [37] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM Press, 2013.
 - [38] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM Press, 2013.
 - [39] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, 1960.
 - [40] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 432–441. ACM Press, 2005.
 - [41] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Dangling references in multi-configuration and dynamic PHP-based Web applications. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, 2013.
 - [42] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 13–22. IEEE CS, 2011.
 - [43] F. Nielson and H. R. Nielson. *Two-level Functional Languages*. Cambridge University Press, 1992.
 - [44] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125. Springer-Verlag, 2009.
 - [45] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE Computer Society, 2008.
 - [46] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 277–287. IEEE Press, 2012.
 - [47] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 488–498. ACM Press, 2013.
 - [48] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 263–272. ACM Press, 2005.
 - [49] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 203–217. ACM Press, 1997.
 - [50] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014. to appear.
 - [51] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM Press, 2012.
 - [52] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings in web applications. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 87–96. ACM Press, 2010.
 - [53] X. Wang, L. Zhang, T. Xie, Y. Xiong, and H. Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 16:1–16:11. ACM Press, 2012.
 - [54] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 171–180. ACM Press, 2008.
 - [55] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. of the 15th Conf. on USENIX Security Symp. (USENIX-SS)*. USENIX Association, 2006.
 - [56] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 251–260. ACM Press, 2011.