Improving Test Case Generation for Web Applications Using Automated Interface Discovery

William G.J. Halfond and Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfond, orso}@cc.gatech.edu

ABSTRACT

With the growing complexity of web applications, identifying web interfaces that can be used for testing such applications has become increasingly challenging. Many techniques that work effectively when applied to simple web applications are insufficient when used on modern, dynamic web applications, and may ultimately result in inadequate testing of the applications' functionality. To address this issue, we present a technique for automatically discovering web application interfaces based on a novel static analysis algorithm. We also report the results of an empirical evaluation in which we compare our technique against a traditional approach. The results of the comparison show that our technique can (1) discover a higher number of interfaces and (2) help generate test inputs that achieve higher coverage.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging;

General Terms: Algorithms, Experimentation, Reliability, Verification

Keywords: Web application testing, interface extraction

1. INTRODUCTION

Web applications play an increasingly important role in our day-to-day activities, such as banking and shopping. The growing popularity of web applications has been driven by the introduction of new web technologies that allow for a higher level of integration among services and provide a richer user experience. Whereas web applications used to be a collection of mostly static content and involved minimal user interaction through simple forms, modern web applications are increasingly customizable, generate content dynamically, combine information gathered from a range of sources, and interact extensively with the user.

This growing complexity has made testing these web applications more challenging because their interfaces tend to be difficult to identify. In testing, a general assumption is that the interfaces of the software being tested are known (*e.g.,* the API for a library or the options and parameters for a command-line utility). For simple web applications that consist of static HTML pages only, identifying the application interface is generally straightforward—it is enough to parse each web page within the application and identify the entities in the page that represent input elements, such as forms' input fields. However, the complexity of modern web applications precludes such a straightforward solution. Modern web applications often have interfaces that are completely hidden, in that they do not correspond to any input elements exposed on a static or dynamically-generated page.

Incomplete information about web application interfaces can limit the effectiveness of testing because it may prevent testers from exercising parts of an application that are accessible only through unidentified interfaces. This can result in unexpected behaviors in the field or even in the deployment of vulnerable applications that could be exploited by attackers. Unfortunately, current techniques for testing web applications either require manual specification of an application's interfaces (*e.g.,* [1, 11, 14]) or use automated approaches that are purely dynamic and, consequently, inherently incomplete (*e.g.,* [4, 5, 10, 12]).

To address the limitations of existing approaches, we present a new technique for automatically discovering the interfaces of a web application. The core of our approach is a novel static analysis algorithm that can analyze web applications that are highly dynamic and have multiple, possibly hidden, interfaces. Our algorithms works in two phases. In the first phase, it identifies domain information (*i.e.,* information about type and possible values) for the parameters that may be part of a web application's set of interfaces. In the second phase, the algorithm identifies how these parameters are grouped in the different interfaces exposed by the web application. The final output of the algorithm is a set of interfaces, where each interface consists of a set of named parameters and domain information for those parameters. This set of interfaces can then be used to support web application testing and verification.

In this paper, we also present a set of studies that evaluate our approach. To perform the studies, we developed a prototype tool, WAM, that implements our technique and compared it to a widely-used approach for interface extraction based on web crawling. The results of the comparison show that, for the web applications considered, (1) our approach identifies a considerably higher number of web interfaces, and (2) the additional accuracy in the interface identification provided by our approach can significantly improve the effectiveness of testing (measured in terms of the structural coverage achieved on the web applications under test). These results are promising, in that they show that our technique has the potential to improve the effectiveness of web application testing and, consequently, the quality of web applications. The contributions of this paper are:

- A novel algorithm for accurately extracting web application interfaces.
- An implementation of the algorithm in a prototype tool.
- Two empirical studies that show the effectiveness of our approach and the usefulness of the information it produces.

2. BACKGROUND

A *web application* is a software system that users can access over the Internet. Web applications are used extensively to provide a number of online services, such as banking and shopping. Typical web applications take user input and generate dynamic pages, customize their response for the user, and interact with other systems such as databases and enterprise servers. Users typically access a web application using a client application, such as a web browser, that allows them to enter input data (*e.g.,* into a form) and submit such data to the web application.

At an abstract level, web applications interact with client applications through logical interfaces that consist of a collection of named *Input Parameters (IP)*, each of which is associated with a domain. More precisely, we define a logical interface for a web application as follows:

$interface = IP*$
$IP = name, domain\_information$
$name =< string >$
$domain\_information = domain\_type, relevant\_value*$
$domain\_type = ANY|NUMERIC$
$relevant\_value =< string > | < number >$

Each interface consists of zero or more IPs; each IP consists of a name and some domain information; domain information consists of a domain type, which is either ANY or NUMERIC, and zero or more relevant values, which can be string or numeric literals. IPs can carry data that includes user input (*e.g.,* web form data fields), cookie session data, and state information for the web application. In order to access a web application, a client issues an HTTP request that contains a set of $\langle name, value\rangle$ pairs corresponding to an interface, and submits it to the web application.

A typical, but unsound, approach for identifying web application interfaces is to assume that the interfaces to a web application can be accurately identified through examination of the HTML pages generated by the application. The approach is based on parsing each HTML page generated by the application to identify its web input forms and assuming that (1) each identified form corresponds to an interface, and (2) each input element in the form (*e.g.,* text box, radio button, or hidden field) corresponds to an IP in that interface. Such an approach is incomplete and inaccurate, in that it can produce a set of interfaces that do not correspond to the actual interfaces of the web application.

In general, this mismatch can exist for several reasons. First, developers are not required to make all interfaces accessible via HTML web forms. Second, even if an interface is made available via a web form, it might be difficult to interact with the application so as to ensure that all possible web forms that can be generated are actually generated. Third, even when a given web form that correspond to an interface can be generated and analyzed, the web application may use a subset (or even a superset) of the parameters exposed as input elements in the web form. (We provide examples of these situations when we discuss our empirical results, in Sections 5.2 and 5.3.) Such discrepancies between visible and hidden interfaces occur frequently and could occur because of a coding error or could be purposely created by a developer (*e.g.,* to provide additional functionality that is intended to be accessed only by certain clients or to account for future extensions).

Since there is generally no formal definition of a web application's interfaces, and examining exposed web forms can produce inaccurate and incomplete results, analyzing a web application's source code is the only way to conservatively identify the application's interfaces; the source code contains information on the *actual* interfaces of a web application, which are the interfaces accessed along specific paths of execution in the application. However, examining the code of a web application and extracting these implicit interfaces is a fairly complex task because of the way IPs are accessed within the application and because different paths in the application can correspond to different interfaces. (*i.e.,* interfaces cannot be extracted by a simple local analysis).

A web application accesses IPs that compose an interface by invoking a specific library function *PF* (*Parameter Function*), which parses the HTTP request containing the IP and returns the values of the requested IPs. More precisely, given a web request *wr*, and an IP with name "foo," the value *val* of the IP can be retrieved as $val = wr.PF("foo")$. Note that retrieving the value of an IP that is not defined in *wr* or not retrieving the value of an IP that is defined in *wr* does not typically result in an error in the web applications.

Typically, the domain of an IP is also implicitly defined in the code and can be inferred by analyzing the operations performed on the IP value after it is returned by PF. For example, the fact that an IP value is cast to an integer along a path implies that the domain of the IP is expected to be an integer on that path. Similarly, the fact that an IP is compared against a specific value implies that such value is relevant in the IP's domain. We refer to IPs for which such relevant values can be identified as *state parameters*, as they are commonly used by web applications to maintain state across web requests.

Web applications can be implemented using a wide range of languages, such as C, Java, or Perl. In Java-based web applications, which we target in our evaluation, the basic component is called a servlet. A *servlet* is a Java program that runs on a web server and provides a *root method* that is invoked when a web request from a client application to the servlet is received. A web application is generally comprised of many servlets, each of which implements part of the application functionality.

3. MOTIVATING EXAMPLE

Before discussing our technique, we introduce an example servlet that is partially shown in Figure 1. This servlet implements the different steps of a generic user-registration process. In the servlet, state parameter `formAction` is used to identify the current phase of the registration process. Assume that, when a user accesses the registration servlet for the first time, the value of state parameter `formAction` is not defined. The servlet would then execute the body of the `else` statement at line 20 and generate an initial registration page with a form where users can choose a login and pin. This form would invoke the same servlet with the login and pin chosen by the user and with parameter `formAction` set to "chooseLogin." On this second invocation, the servlet, based on the value of `formAction`, would execute the body of the `if` statement at line 6 and retrieve the values for the IPs named "login" and "pin." To retrieve the value of the "login" IP, the server calls the PF directly with string "login" as a parameter. To get the value of the IP named "pin," it calls method `getNumParam`, which in turns calls the PF and casts the retrieved IP value to an integer. The servlet then registers the new login and generates a second page with a form that collects additional personal information. This form would also invoke the same servlet with the user-provided input and with `formAction` now set to "personalInfo." When the user submits this form, the servlet would check the state parameter and execute the body of the `else` statement at line 11, which collects and validates the information provided by the user. It does this by calling method `getNumParam`, which ensures that the value of the IP named "zip" is an integer and matches a pre-defined zip code. One of the interfaces for our example servlet is the following:

interface = {'`formAction`', '`login`', '`pin`'}, where:
`formAction`'s domain type is *ANY*, with relevant values 'chooseLogin' and 'personalInfo',
`login`'s domain type is *ANY*, with no relevant values,
`pin`'s domain type is *NUMERIC* with no relevant values.

Knowing this interface, we could generate several test inputs for the servlet by creating a URL consisting of the location of the servlet followed by $\langle name, value\rangle$ pairs for each IP in the interface. For example, assuming that the servlet is accessible at `http://my.dom/myservlet.jsp`, a possible invocation of the servlet through the above interface would be `http://my.dom/myservlet.jsp?formAction=chooseLogin&login=foo&pin=1234`.

Although this is a simple servlet, it has a number of interesting characteristics: (1) it uses a state parameter (`formAction`) to drive the execution along three different paths, each of which accesses a different set of IPs; (2) its different parts can be executed only by specifying the correct values for `formAction` or by entering the right data in the various forms it generates; (3) it accepts two IPs, parameter "zip" and cookie field "sessionID," that can only be numeric; and (4) it requires parameter "zip" to have a specific value to successfully complete the registration process. These are exactly the kind of characteristics that make the identification and testing of a servlet's interfaces difficult for conventional approaches.

4. INTERFACE DISCOVERY ALGORITHM

The goal of our approach is to automatically identify the different interfaces that a web application exposes. The approach is based on a two-phase static analysis algorithm that computes the set of interfaces for each servlet in a web application. In the first phase, our analysis computes domain information (*i.e.,* types and possibles values) for each IP in the servlet. In the second phase, our analysis identifies IP names and groups them into logical interfaces based on the servlet's control and data flow. In the following sections, we explain the two phases of the algorithm in detail and illustrate how they work using our example servlet.

```
    ...
 1. int approvedZip = 30318;
 2. String actionChooseLogin = "chooseLogin";
 3. String actionPersonalInfo = "personalInfo";
 4. String ID = "sessionID";

 5. String formAction = request.getParameter("formAction");
 6. if (formAction.equals(actionChooseLogin)) {
 7.   String requestedLogin = request.getParameter("login");
 8.   int pin = getNumParam(request, "pin");
 9.   registerLogin(requestedLogin, pin);
10.   ... // generate second registration page
11. } else if (formAction.equals(actionPersonalInfo)) {
12.   int id = Integer.parseInt(getCookie(request, ID));
13.   String name = request.getParameter("name");
14.   int zip = getNumParam(request, "zip");
15.   if (zip == approvedZip) {
16.     finishRegistration(id, name);
17.   } else {
18.     error("You do not live in " + approvedZip);
19.   }
20. } else { ... } // generate initial registration page
    ...

21. int getNumParam(ServletRequest request, String paramName) {
22.   String paramValue = request.getParameter(paramName);
23.   int param = Integer.parseInt(paramValue)
24.   return param;
25. }

26. String getCookie(HttpServletRequest request, String name) {
27.   return request.getAttribute(name);
28. }
    ...
```

Figure 1: Example servlet.

4.1 Phase 1: Discover Domain Information

In Phase 1 our technique analyzes each IP in each servlet and computes its domain information. The input of the analysis is the set of servlets in the web application, and its output is an *Interprocedural Control Flow Graph (ICFG)* of the web application annotated with domain information. The algorithm annotates two kinds of nodes: nodes that contain a call to a PF and nodes that represent a call to a method that invokes, directly or indirectly, a PF. An annotation for a node n contains three elements: (1) the location of the PF call that retrieved the original IP value v, (2) the domain type of v, and (3) possible values for v.

The general approach that we use to compute annotations is to start from a PF call, identify the variable that is defined by the return value of the PF call (*i.e.,* the variable that stores the IP value), identify the operations performed on that variable along chains of definitions and uses, and use this information to compute domain information. More specifically, if a variable that contains an IP value is converted to any numeric value, we can infer that the domain type of the corresponding IP value is NUMERIC. Also, if a variable that contains an IP value is compared against a specific value *val*, we can deduce that *val* is a special value for that IP and add *val* to the IP's set of relevant values.

Algorithm 1 shows our Phase 1 algorithms, GetDomainInfo and GDI. Without loss of generality, in the rest of the discussion we assume that (1) there are no global variables in the applications under analysis, (2) each node in the ICFG contains only one definition, and (3) the variable defined with the value returned by a function (in particular, a PF) is used in only one statement. Note that we make these assumptions only to simplify the presentation of the algorithms, and they do not limit the applicability of the approach in any way; any program could be automatically transformed so that it satisfies all three assumptions.

GetDomainInfo first builds an ICFG for the servlets in the web application and computes data-dependence information. We assume that data-dependence information is available through function $DUchain(v, n)$, which returns the set of nodes that (1) are reached by the definition of variable v at node n and (2) use variable v.

After computing ICFG and data-dependence information, GetDomainInfo identifies and processes each PF call.[1] For each PF call, it first identifies $PFnode$, the ICFG node that contains the PF call considered (line 4) and $PFvar$, the variable defined with the return value of the PF call (line 5). Then, GetDomainInfo creates a new annotation and initializes its IP node to $PFnode$ (lines 6–7), its type to ANY (line 8), and its values to the empty set (line 9), and associates the newly-created annotation with $PFnode$. Finally, it invokes the GDI algorithm, providing as input the node that uses the definition of $PFvar$ at $PFnode$, $PFvar$, $PFnode$, and an empty set. When all PF calls have been processed, GetDomainInfo returns the resulting annotated ICFG.

GDI is a recursive algorithm that computes domain information for an IP. GDI takes four parameters as input: *node* is the node to be analyzed; $IPvar$ is the variable that stores the current IP value or a value derived from the current IP value and that is used at *node*; *root_node* is the node to be annotated with the discovered domain information; *visited_nodes* is the set of nodes encountered in the path being traversed (used to ensure that cyclic data dependencies are explored only once in a path). To understand the algorithm, it is important to note that, by construction, the statement represented by *node* is always a use of variable $IPvar$, which stores the current IP value or a value derived from it. The output of GDI is the set of annotations added to the ICFG while processing its nodes.

If *node* is not in the *visited_nodes* set (line 1), GDI performs one of three different operations based on the type of statement represented by *node*. If *node* is an exit node (*i.e.,* it corresponds to a return statement), GDI first identifies all return sites for the method that contains *node* (line 3). Then, for each return site, *retsite*, it identifies the variable defined at *retsite*, *retvar* (which contains the IP value after the call returns) (line 5), copies *root_node*'s annotation (line 6), and associates the annotation with node *retsite* (line 7).[2] Finally, it invokes GDI recursively, passing as inputs the node that uses the definition of *retvar* at *retsite*, *retvar*, *retsite*, and the current set of visited nodes plus *node* (line 8).

If *node* represents a statement that performs a comparison of $IPvar$ against a specific value (line 11), GDI adds that value to the set of values in the annotation associated with *root_node* (lines 12–13). Currently, our implementation handles comparisons with string and numeric constants. It also handles a subset of cases where $IPvar$ is compared against a variable; to do this, it tries to identify the value of the variable at the point of the comparison by following definition-use chains to the variable's initialization. (This

[1]The specific set of PFs considered depends on the language in which the web application is written.

[2]Annotations are copied to ensure that domain information is computed in a context-sensitive fashion, which improves the precision of Phase 2, as explained in Section 4.2.

Algorithm 1 – Phase1

```
/* GetDomainInfo */
Input: $servlets$: set of servlets in the web application
Output: $ICFG$: ICFG for the servlets, annotated with domain information
begin
 1: $ICFG \leftarrow$ ICFG for the web application
 2: compute data-dependence information for the web application
 3: for each $PF call in the servlets$ do
 4:    $PFnode \leftarrow$ ICFG's node representing $PF$
 5:    $PFvar \leftarrow$ lhs of the PF call statement
 6:    $newannot \leftarrow$ new annotation
 7:    $newannot.IPnode \leftarrow PFnode$
 8:    $newannot.type \leftarrow$ ANY
 9:    $newannot.values \leftarrow \{\}$
10:    associate $newannot$ with $PFnode$
11:    GDI($DUchain(PFvar, node), PFvar, PFnode, \{\}$)
12: end for
13: return $ICFG$ /* returns annotated ICFG */
end
/* GDI */
Input: $node$: current node to examine
       $IPvar$: variable storing the IP value and used at $node$
       $root\_node$: node to be annotated
       $visited\_nodes$: nodes visited along current path
begin
 1: if $node \notin visited\_nodes$ then
 2:    if $node$ is an exit node then
 3:       $returnsites \leftarrow$ possible return sites for $node$'s method
 4:       for each $retsite \in returnsites$ do
 5:          $retvar \leftarrow$ variable defined at $retsite$
 6:          $newannot \leftarrow root\_node$'s annotation
 7:          associate $newannot$ with node $retsite$
 8:          GDI($DUchain(retvar, retsite), retvar, retsite,$
                $visited\_nodes \cup \{node\}$)
 9:       end for
10:    else
11:       if $node$ represents a comparison with a constant then
12:          $compval \leftarrow$ value used in the comparison
13:          addValueToAnnotation($root\_node, compval$)
14:       else if $node$ is a type cast/conversion of $IPvar$ then
15:          $casttype \leftarrow$ target type of the cast operation
16:          setDomainTypeInAnnotation($root\_node, casttype$)
17:       end if
18:       if $node$ contains a definition of a variable then
19:          $var \leftarrow$ variable defined at $node$
20:          for each $n \in DUchain(var, node)$ do
21:             GDI($n, var, root\_node, visited\_nodes \cup \{node\}$)
22:          end for
23:       end if
24:    end if
25: end if
end
```

type of value resolution is similar to the one we use to identify IP names in Phase 2, as described in Section 4.2.)

Otherwise, if *node* represents a type cast statement performed on $IPvar$, GDI updates the domain type of the annotation associated with *root_node* (lines 15–16).

At this point, GDI checks whether the statement represented by *node* defines a variable (line 18). If so, it first identifies variable *var* defined at *node* (line 19). Then, for each node n that uses the definition of variable *var* at node *node* (line 20), GDI invokes itself recursively, providing as input n, *var*, the current *root_node*, and the current set of visited nodes plus *node* (line 21).

To illustrate Phase 1 with a concrete example, we show the application of algorithm GDI to the PF at line 22 of our example servlet in Figure 1, whose partial ICFG is represented in Figure 2. The first call to GDI, performed by the GetDomainInfo algorithm, is "$GDI(23, paramValue, 22, \{\})$" node 23 has not been visited before, so GDI proceeds to check whether the node is an exit node, which is not the case. GDI then checks whether the node corresponds to a pred-

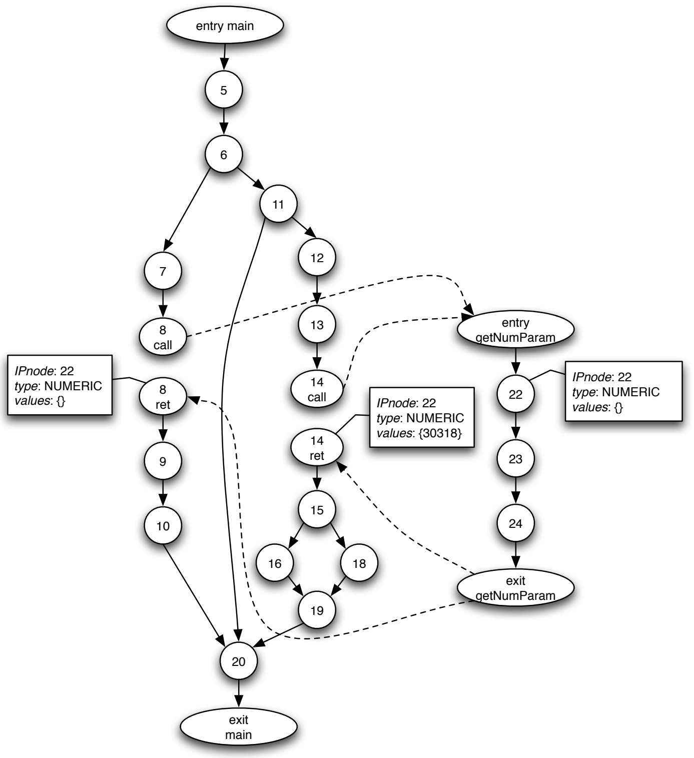

Figure 2: Annotated control flow graphs for two of the methods in the code of Figure 1.

icate statement, which is also not the case. Finally, the algorithm checks whether the node represents a type cast operation. Because the statement at node 23 is indeed a type cast (it parses an integer in a string), it annotates the *root_node*, node 22, with domain type NUMERIC. node 23 also contains a definition of variable *param*, so GDI computes $DUchain(param, 23)$, which returns a set containing only node 24, and invokes itself recursively as

"$GDI(24, param, 22, \{23\})$"

Because node 24 is an exit node, GDI identifies the corresponding return sites: nodes $8ret$ and $14ret$. At this point, GDI associates the current annotation of *root_node* with nodes $8ret$ and $14ret$. Then GDI calls itself recursively twice, once for each return site, as

"$GDI(9, pin, 8ret, \{23, 24\})$" and "$GDI(15, zip, 14ret, \{23, 24\})$"

The first of these two invocations of GDI would identify node 9 as a definition of the formal parameter of *registerLogin* corresponding to the actual parameter *pin* and continue by exploring uses of that formal parameter in *registerLogin* (not shown here). The second invocation of GDI would identify node 15 as a predicate node that performs a comparison with value 30318 (by identifying the initialization of variable *approvedZip*) and add that value to the set of values in the annotation for the current *root_node* (node 14ret). At this point, the algorithm would terminate because node 15 does not contain a definition and, thus, there are no further data dependencies to follow.

The final result after GDI processes this PF would be the three annotations attached to nodes $8ret$, $14ret$, and 22 shown in Figure 2.

4.2 Phase 2: Compute Interfaces

In its second phase our analysis identifies IP names, groups them into logical interfaces, and associates the domain information computed in the first phase with the identified IP names. The input of the analysis is the annotated ICFG produced by GetDomainInfo, and its output is the set of interfaces exposed by each servlet. Intuitively, Phase 2 works by grouping sets of IPs that are used along a path and, thus, could be part of the same interface. To avoid an exponential explosion of the computational cost, the analysis processes one method at a time, computes summary information for the method, and then uses the summary information when analyzing methods that invoke methods already analyzed. Within each method (or set of strongly connected methods), the analysis computes summary information using a worklist-based forward iterative algorithm. Methods involved in recursion are handled by treating them as one "super-method" and assigning the same summary to each of them. The analysis terminates when each servlet's root method has been processed. At that point, the summary of a servlet's root method represents all of the interfaces exposed by the servlet.

Algorithm 2 shows our Phase 2 algorithms, ExtractInterfaces and SummarizeMethod. ExtractInterfaces first builds a *Call Graph (CG)* for the web application servlets. Then, it identifies the sets of strongly connected components in the CG, SCC, and creates a set, CC, which consists of SCC plus singleton sets, one for each node that does not belong to any set in SCC. In this way, every element of CC is either a set of strongly connected nodes in the CG or a single node.

In the presentation, we assume that the ICFG is globally available, together with the individual *Control Flow Graphs (CFGs)*. We also assume that the following functions are available: $target(n)$, which returns the method called at a call site n; $succ(n)$, which returns all successors of n in n's CFG; and $pred(n)$, which returns all predecessors of n in n's CFG. If n is a call (resp., return) site, $succ(n)$ (resp., $pred(n)$) returns the corresponding return (resp., call) site. For example, $pred(14ret)$, for the ICFG in Figure 2, would return $14call$.

ExtractInterfaces iterates over the sets in CC in reverse topological order to ensure that a method is processed before any method that calls it. When all methods have been processed, the algorithm simply returns the interfaces computed for each servlet's root method.

SummarizeMethod is the core algorithm of Phase 2. It takes as input *methodset*, a set of one or more methods, computes the interfaces exposed by these methods, and attaches these interfaces to the methods as summaries. The output of the algorithm is the set of summaries associated to the methods in *methodset*.

SummarizeMethod first initializes the data structures used in the rest of the algorithm. In particular, set N is initialized with all of the nodes in all of the methods in *methodset* (line 1) and *Gen* sets for a node n are initialized in one of four ways.

If n represents a PF call, a new IP entity that corresponds to the parameter accessed at node n is created, initialized, and used to initialize the *Gen* set for n (lines 6–14). If Phase 1's annotation for node n contains domain information (*i.e.,* type or values), SummarizeMethod associates such information with the IP entity (lines 9–13). All successors of n are then added to the worklist (line 10).

If n is a callsite and the target of the call is a summarized method with summary s, n's *Gen* set is initialized with the value returned by function *map* invoked on n and s (line 17). The *map* function takes a method m's summary and a callsite invoking m and modifies the summary by replacing each IP name that is not concrete (*i.e.,* each IP name that corresponds to a variable or to a formal parameter—see below) with the corresponding actual parameter at the callsite. Then, for each IP entity in each interface contained in n's *Gen* set, SummarizeMethod checks whether the annotations created by Phase 1 for n's return site apply to any of the IP entity, that is, whether they refer to the same IP node (line 19–21). If so, it updates the domain information for the IP entity using the domain information in the relevant annotations (line 22). After performing this operation, SummarizeMethod adds n's successors to the worklist. If n is a method entry point, its *Gen* set is initialized to a set containing an empty set (line 28) and n's successors are added to the worklist.

Finally, if n is not a callsite, SummarizeMethod initializes n's *Gen* set to the empty set (line 31).

The *Out* set for n is then initialized with the value of n's *Gen* set (line 33).

Note that, for this algorithm, the *Out* set for a node n represents the set of interfaces exposed by n (where each interface is a set of IP elements, as described in Section 2), and the set of interfaces for a method consists of the *Out* set of its exit node.

After initializing sets and data structure, SummarizeMethod enters its iterative part, where it keeps processing nodes until the worklist is empty (lines 35–52). For each node n in the worklist, SummarizeMethod computes the value of n's In and *Out* sets as follows. The In set is computed straightforwardly as the union of the *Out* sets of n's predecessors. SummarizeMethod computes n's *Out* set as the product of n's In and *Gen* sets; for each interface (*i.e.,* set of IP elements) i in In (line 39) and each interface g in *Gen* (line 40), SummarizeMethod generates an interface that is the union of i and g and adds it to *Out* (line 41).

If the *Out* set for n changed since the previous iteration over n (line 44), SummarizeMethod updates n's *Out* set (line 45) and updates the worklist as follows. If n is a callsite and its target method m is one of the methods in the input set (*i.e.,* m is in the same strongly connected component of the CG as the current method), SummarizeMethod adds m's entry node to the worklist (line 47).[3] Otherwise, SummarizeMethod simply adds n's successors to the worklist (line 49). Note that, if n is a callsite but its target method m is not in the input set, m's return site would be added to the worklist.

When the worklist is empty, SummarizeMethod performs the following operations for each method m in the input set. First, it associates the set of interfaces in the *Out* set of m's exit node to m as its summary (lines 54–55). Then, it considers all IPs in all such interfaces whose name is not a concrete value and tries to resolve them by calling function *resolve*. If successful, the resolution of a reference provides the actual names of the IPs in an interface. For space considerations, we do not show function *resolve* and only provide an intuitive description of it.

[3]By doing so, SummarizeMethod treats nodes in a set of strongly connected methods as a single super-method, as described above.

Algorithm 2 – Phase 2

```
/* ExtractInterfaces */
Input: $ICFG$: annotated ICFG produced by GetDomainInfo
Output: $interfaces[]$: interfaces exposed by each of the servlets
begin
 1: $CG \leftarrow$ call graph for the web application
 2: $SCC \leftarrow$ set of strongly connected components in CG
 3: $SINGLETONS \leftarrow$ set of singleton sets, one for each node in CG
    that is not part of a strongly connected component
 4: $CC \leftarrow SCC \cup SINGLETONS$
 5: for each $mset \in CC$, in reverse topological order do
 6:     SummarizeMethod($mset$)
 7: end for
 8: return interfaces of each servlet's root method
end
/* SummarizeMethod */
Input: $methodset \subset CG\ nodes$: singleton set or set of strongly connected methods in the call graph
begin
 1: $N \leftarrow \bigcup_{m \in methodset} nodes\ in\ m's\ CFG$
 2: $worklist \leftarrow \{\}$
 3: for each $n \in N$ do
 4:     In[$n$]$\leftarrow \{\}$
 5:     if $n$ corresponds to a PF call then
 6:         $newIP \leftarrow$ new $IP$
 7:         $newIP.node \leftarrow n$
 8:         $newIP.name \leftarrow$ parameter of the PF call
 9:         if $n$'s annotation has domain information $dominfo$ then
10:             $newIP.domaininfo \leftarrow dominfo$
11:         else
12:             $newIP.domaininfo \leftarrow null$
13:         end if
14:         Gen[$n$] $\leftarrow \{\{newIP\}\}$
15:         add nodes in $succ(n)$ to $worklist$
16:     else if $n$ is a callsite $AND$ target($n$) has summary $s$ then
17:         Gen[$n$] $\leftarrow$ map($n, s$)
18:         for each $interface \in$ Gen[$n$] do
19:             for each $IP \in interface$ do
20:                 $annot \leftarrow$ annotation associated with $n$'s return site
21:                 if $IP.node == annot.IPnode$ AND
                      $annot$ has domain information $dominfo$ then
22:                     $IP.domaininfo \leftarrow dominfo$
23:                 end if
24:             end for
25:         end for
26:         add nodes in $succ(n)$ to $worklist$
27:     else if $n$ is a method entry point then
28:         Gen[$n$] $\leftarrow \{\{\}\}$
29:         add nodes in $succ(n)$ to $worklist$
30:     else
31:         Gen[$n$]$\leftarrow \emptyset$
32:     end if
33:     Out[$n$] $\leftarrow$ Gen[$n$]
34: end for
35: while $|worklist| \neq 0$ do
36:     $n \leftarrow$ first element in $worklist$
37:     In[$n$]$\leftarrow \bigcup_{p \in pred(n)} Out[p]$
38:     Out$'$[$n$] $\leftarrow \{\}$
39:     for each $i \in$ In[$n$] do
40:         for each $g \in$ Gen[n] do
41:             Out$'$[$n$] $\leftarrow$ Out$'$[$n$] $\cup \{i \cup g\}$
42:         end for
43:     end for
44:     if Out$'$[$n$] $\neq$ Out[n] then
45:         Out[$n$] $\leftarrow$ Out$'$[$n$]
46:         if $n$ is a callsite AND target($n$) $\in methodset$ then
47:             add target($n$)'s entry node to $worklist$
48:         else
49:             add nodes in succ($n$) to $worklist$
50:         end if
51:     end if
52: end while
53: for each $m \in methodset$ do
54:     $summary \leftarrow$ Out[$m$'s exit node]
55:     associate $summary$ to method $m$
56:     for each $interface \in summary$ do
57:         for each $IP \in interface$ such that $IP.name$ is not a
                concrete value do
58:             $IP.name \leftarrow$ resolve($IP$)
59:         end for
60:     end for
61: end for
end
```

Function *resolve* takes as input a string variable and attempts to find one or more statements in the current method where the variable is initialized. To do this, it follows backward definition-use chains starting from the variable and without crossing the current method's boundaries until it reaches a definition involving (1) a string constant, (2) an expression, or (3) a method parameter. In the first case, *resolve* terminates successfully and returns the identified string constant. In the second case, it tries to compute a conservative approximation of the values of the string expression using the Java string analysis developed by Christensen, Møller, and Schwartzbach [2]. If the analysis terminates, *resolve* returns the resulting set of strings. Finally, in the third case, *resolve* returns the formal parameter identified. In this latter case, (1) the formal parameter will be mapped to the corresponding actual parameter when the summary information is used in one of the current method's callers, and (2) *resolve* will be invoked on the actual parameter when computing the summary information for that caller.

We illustrate Phase 2 using our example servlet. In the interest of clarity, we focus only on excerpts that illustrate some of the subtleties of the algorithm and ignore annotations and calls to *resolve*. The first two methods processed by the algorithm are `getNumParam` and `getCookie`. The processing of `getNumParam` begins at node 22, which contains a PF call, so its *Gen* set consists of a newly created IP entity for that PF, whose name is "paramName" (the parameter of the PF call). Traversing nodes 23 and 24 does not add any additional information. After reaching node 24, SummarizeMethod creates a summary with a single interface in it, consisting of the newly created IP. The processing of method `getCookie` is almost identical and creates a similar summary. In the main method, processing starts at node 5, which is a PF call. Therefore, also in this case, its *Gen* set contains a newly created IP entity for that PF, with name "formAction."

SummarizeMethod then proceeds to node 6, at which point either branch can be taken; we assume that SummarizeMethod follows the branch to node 7 first. Node 7 is also a PF call and results in a *Gen* set with another newly created IP entity, whose name is "login." Node 8 is a call to method `getNumParam`, whose summary we computed above. The *map* function, applied to this summary, replaces formal parameter `paramName` with actual parameter `pin`. Nodes 9 and 10 do not add any information, so the *Out* set of node 10 contains the set of three IPs whose names are "formAction," "login," and "pin."

Along the branch starting with node 11 and ending at node 19, the analysis generates an *Out* set (for node 19) consisting of four IPs, with names "formAction," `sessionID`, "name," and "zip." At node 20, SummarizeMethod computes the union of the *Out* sets of nodes 10, 11, and 19, which results in a set containing three possible interfaces: {{"formAction", "login", "pin"}, {"formAction"}, {"formAction", "sessionID", "name", "zip"}}.

5. EMPIRICAL EVALUATION

In our empirical evaluation, we assess the usefulness of our approach in supporting web application testing. To do this, we compare WAM's performance with the performance of an approach based on web crawling (SPIDER, hereafter). Web crawlers, also known as web spiders, are widely used tools that explore web sites by visiting web pages, identifying links in the visited pages, and visiting the linked pages recursively. Web spiders provide an ideal baseline for our evaluation because they are the tools most commonly used for extracting web application interfaces, and many approaches to web application testing rely on the information produced by web spiders. In the evaluation, we investigate the following two research questions:

Table 1: Subject programs for our empirical studies.

Subject	*LOC*	*Servlets*
Bookstore	19,402	28
Checkers	5,415	33
Classifieds	10,702	19
Employee Directory	5,529	11
Events	7,164	13
Office Talk	4,670	38
Portal	16,089	28

RQ1: Does WAM discover a higher number of interfaces than SPIDER?

RQ2: Does testing effectiveness improve when using interface information generated by WAM instead of interface information generated by SPIDER?

In the following sections, we describe our experiment setup, present the studies we performed to address RQ1 and RQ2, and discuss the results of these studies.

5.1 Experiment Setup

5.1.1 Experimental Subjects

Our experimental subjects consist of seven Java-based web applications: five commercial applications available from GotoCode (`http://www.gotocode.com/`) and two student-developed projects. This set of subjects was used in previous work by us and also other researchers [7–9]. Table 1 provides general information about the subject applications. For each application, the table lists the application's size (*LOC*) and number of servlets (*Servlets*).

5.1.2 Tools Used in the Evaluation

The spider we use in our study is based on the web crawler available as part of the widely-used OWASP WebScarab Project [13]. This is a state-of-the-art implementation that is representative of most spider-based approaches. We extended the OWASP spider by adding to it the capability of extracting from web pages (1) form-related information, including information about `<form>` and `<input>` elements, and (2) default values specified for those elements. Additionally, we gave the spider administrator-level access to the web applications used in the study by providing it with login information for the various applications. In this way, we enabled SPIDER to perform a more thorough exploration.

For the evaluation we developed a prototype tool, WAM, that implements our approach for web applications developed using the JEE framework (`http://java.sun.com/javaee/`). As input WAM takes the set of Java classes in a web application. For each servlet in the application, WAM analyzes its bytecode and outputs a list of the servlet's interfaces. To generate callgraphs, CFGs, and ICFGs, WAM uses the SOOT program anal-

Table 2: Number of interfaces identified.

Subject	# *identified interfaces* SPIDER	WAM	*Improvement*
Bookstore	21	24	3 (14%)
Checkers	1	29	28 (2,800%)
Classifieds	11	17	6 (55%)
Employee Dir.	6	9	3 (50%)
Events	9	11	2 (22%)
Office Talk	5	31	26 (520%)
Portal	20	25	5 (25%)

ysis framework (`http://www.sable.mcgill.ca/soot/`). To compute data-dependency information, WAM leverages INDUS (`http://indus.projects.cis.ksu.edu/`), a data analysis library built on top of SOOT. Lastly, the *resolve* function that we use in Phases 1 and 2 uses the Java String Analysis (JSA) library [2] to compute a conservative approximation of the different values a string can assume at a given point in a program.

5.2 Study 1

To address RQ1, we ran SPIDER and WAM on our subject applications, counted the number of interfaces extracted by the two tools, and compared these numbers. Table 2 shows our results. For each application, the table reports the number of interfaces identified by SPIDER (SPIDER), the number of interfaces identified by WAM (WAM), and the absolute and relative (in parentheses) increase in the number of interfaces identified by WAM with respect to SPIDER.

As the table shows, WAM always identifies a higher number of interfaces than SPIDER, and in some cases the increase in the number of interfaces identified is dramatic.

We manually inspected several of the servlets that showed a high difference in the number of interfaces discovered by SPIDER and WAM to check the correctness of our results and improve our understanding of the reasons for WAM's better performance. We found that the difference in the results is generally due to a combination of two factors.

First, many servlets interact with users through a series of forms that must be completed in sequence and by providing a suitable set of values. One example of this situation is a registration servlet, similar to the servlet example in Figure 1, that we found in one of our subjects. The servlet required users to select a login and then enter their password twice before proceeding to the next form, which prompted them for additional information. Unless a spider could provide appropriate information for the various forms, it would only be able to discover the initial default form. While a spider can be trained to solve specific instances of this problem, the problem of generating suitable values for a form is, in general, unsolvable without specific knowledge of the application and its underlying database.

Second, many servlets contain hidden interfaces. These interfaces would not appear in any web page and, thus, could not be detected by a web spider. We found many occurrences of this type of hidden interfaces that were used as a mechanism for inter-servlet communication, where one servlet would invoke another servlet and pass parameters to it for processing. Although designed for inter-servlet use, these interfaces are accessible externally because each servlet in a web application can be invoked directly. Therefore, they should be identified and suitably tested.

Additional findings. During manual inspection of the results, we found some additional evidence of the usefulness of our approach that, although not directly related to the results of Study 1, is worth reporting. Specifically, we encountered many situations where web forms contained input elements that were never actually retrieved by the servlet (*i.e.,* their corresponding input values were simply ignored during execution of the servlet). Without additional information, it is difficult to asses whether these situations correspond to errors in the application. However, they do represent cases in which our technique would identify a more accurate set of interfaces and provide testers or developers with more relevant information; since SPIDER reports all input elements it identifies, regardless of whether they are actually used by the application, as being part of the application's interface, this could result in wasted testing effort that targets the unused input elements.

5.3 Study 2

Our goal in investigating RQ2 is to assess whether identifying more interfaces actually results in a more thorough testing of web applications. In other words, we want to confirm that the additional interfaces discovered by our technique actually provide value for the tester. To address this question, we compared the coverage achieved by a test-input generation technique when relying on the interfaces identified by WAM and on the ones generated by SPIDER. Although higher coverage does not necessarily imply better fault detection, coverage is an objective and commonly-used indicator of the quality of a test suite.

We measured and compared coverage for four criteria: basic-block, branch, command-form, and input-parameter coverage. Basic-block coverage and branch coverage measure the percentage of basic-blocks and branches exercised, respectively. Command-form coverage [9] measures the number of distinct types of database commands and queries (*i.e.,* commands and queries that differ in their structure) generated by an application that interacts with an underlying database. Since most web applications are data-centric, this measure is useful in determining whether a test suite has caused a servlet to exhibit different behavior with regards to its interactions with the underlying database. Lastly, input-parameter coverage measures the percentage of an application's unique IPs that are accessed by a servlet during its execution, which is an indicator of how thoroughly the possible input channels have been exercised.

To exercise the subject applications, we generated two test suites for each application: one based on the interfaces identified by WAM and one based on the interfaces identified by SPIDER. To avoid introducing bias into the test suite generation, we used the same process to create both test suites. For each interface identified by the two approaches, we iterated over each input parameter *param* in the interface and created a set of test cases that varied the value of *param* by setting it to a "normal" value (*e.g.,* a random string or number), the empty string, or an erroneous value (*e.g.,* a numeric parameter when a non-numeric value is expected). The empty string and erroneous values were used to try to exercise error checking code in the application. The other parameters in the interface were set to legal values. In the case of SPIDER-discovered interfaces, we used the default legal values supplied by the crawled HTML pages. For WAM-discovered interfaces, we used the set of relevant values discovered by WAM. In this way, we were able to generate test cases in the exact same way for both WAM and SPIDER, and differences in the resulting test sets were due only to differences in the interfaces identified by the two approaches.

After generating the sets of test inputs, we deployed versions of the subject applications instrumented to monitor the different types of coverage. We used the INSECTJ framework [16] to handle instrumentation and monitoring for basic block, branch, and input-parameter coverage, and the DITTO tool [9] to measure command-form coverage. We then ran all of the test inputs against the instrumented applications and computed the amount of coverage achieved using each criterion. The results of this study are presented in Table 3. For each application, we show the amount of basic block (*Block*), branch (*Branch*), command-form (*Cmd-form*), and input-parameter (*Param*) coverage achieved. For each coverage criterion, we also show the coverage achieved by the SPIDER-based approach (S), the coverage achieved by the WAM-based approach (W), and the percentage improvement achieved when using our approach ($\pm$). Note that, in the table, command-form coverage is reported only in absolute terms because we do not currently have a way of computing the exact number of test requirements for this criterion.

Table 3: Test criteria coverage achieved using the spider-based and the wam-based test generation approaches.

	Block (%)			Branch (%)			Cmd-form (abs.)			Param (%)		
Subject	*S*	*W*	±	*S*	*W*	±	*S*	*W*	±	*S*	*W*	±
Bookstore	62	70	**13**	50	63	**26**	59	102	**73**	97	100	**4**
Checkers	17	31	**86**	5	10	**100**	7	14	**100**	29	63	**120**
Classifieds	58	70	**20**	44	61	**39**	35	77	**120**	96	98	**2**
Employee Dir.	69	78	**13**	52	64	**22**	27	51	**89**	98	100	**2**
Events	58	71	**22**	46	61	**34**	27	64	**137**	97	100	**3**
Office Talk	25	37	**49**	12	24	**100**	16	26	**63**	27	69	**150**
Portal	65	71	**10**	55	64	**17**	88	156	**77**	99	100	**1**
Average	51	61	**30**	38	50	**48**	37	70	**94**	78	90	**40**

The results reported in the table show that test input generation based on the interfaces discovered by WAM resulted in consistently higher coverage across the four testing criteria considered. There is an average increase of 30% for basic block coverage, 48% for branch coverage, 94% for command-form coverage, and 40% for input-parameter coverage. These results provide evidence that, for the subjects considered, the discovery of more interfaces, along with the domain information extracted by our analysis, allows testers to create test inputs that will more thoroughly exercise the different parts of a web application.

In an inspection of several servlets, we found that it was common that large sections of code were either executed or skipped based on the values of state parameters. The ability to set these state parameters to meaningful values, provided by our approach, allows for exercising these sections of code. Conversely, spiders cannot, in general, discover legal values for these state parameters without human intervention or some form of heuristics-based guessing. As the results confirm, this limits the ability of spider-based approaches to thoroughly exercise web applications. We also found that the way many web applications handle parameters of the incorrect type (*e.g.,* string instead of numeric) often involves exiting the servlet with an error message or an exception. In these cases, the domain information provided by our analysis can help generate test inputs that satisfy type constraints and can execute larger parts of the servlets.

For two of the applications, Officetalk and Checkers, the average increase in coverage achieved by the WAM-based approach is considerably higher than for the other applications. Through manual inspection of the code, we found that the servlets in the other five applications read all input parameters as soon as invoked and perform an extensive and systematic error checking on all of them. Therefore, even if a test input causes an error during input validation, it would still execute a considerable percentage of the servlets' code (albeit not its main logic). Conversely, the servlets in Officetalk and Checkers check state parameters first and, based on their values, direct the execution along different paths that access and check different input parameters as needed. As a result, identifying the right relevant values for the state parameters, as WAM does, leads to test inputs that achieve a considerably higher coverage.

A final observation is that command-form coverage exhibited a much higher increase than other criteria. Intuitively, the different types of queries generated by a web application represent different types of behaviors that an application can perform. Such a high increase correlates to our intuition that accurately identifying interfaces and domain information can result in a larger and more varied set of behaviors exhibited by the servlets under test.

Threats to validity. As with all empirical evaluations, there are threats to the validity of our results. The threats that we identified include the quality of SPIDER's implementation, subject selection, and test input generation strategies. First, a flawed implementation of SPIDER could result in an inadequate exploration of the web applications and, thus, in missing interfaces that a spider could indeed identify. To mitigate this issue, we used existing spider infrastructure built by the OWASP group and simply extended its functionality. This infrastructure is actively supported and implements state-of-the-art practices and strategies for web crawling. Second, if the selection of experimental subjects is not representative, our results may not generalize. To mitigate this issue, we used seven different applications that were gathered from different sources and have been used previously in related work [7–9]. Finally, the use of an inadequate test input generation strategy may negatively affect the coverage results achieved using one or both of the considered approaches. To mitigate this threat, we applied standard test-generation techniques and used the same test generation strategy for both approaches. The only difference is that we leveraged the specific information provided by WAM and SPIDER when creating the set of test inputs for the two approaches.

6. RELATED WORK

Many techniques for testing web applications have been proposed in the literature. Most of the spider-based approaches have been developed in the commercial sector and are based on traditional web crawling, with the notable exception of WAVES [10]. WAVES is a sophisticated web spider that uses heuristics to address many of the shortcomings of conventional crawlers. However, these heuristics have to be customized for a particular web application and require extensive developer intervention.

Many approaches generate test inputs for a web application based on a model of the application. An early technique by Ricca and Tonella [14] is based on UML models. In this approach, developers model the links and interface elements of each page in the web application, and these models are used to guide test input generation and estimate coverage of the web application. Jia and Liu [11] propose a similar technique that rely on a formal specification instead of a UML model. These techniques tend to be limited in their ability to capture the dynamic and state-based behavior of modern web applications. Later work addresses this issue by modeling web applications as finite-state machines [1]. The primary drawback of these techniques, as compared to our approach, is that developers must accurately and manually define the web application model, whereas our approach is completely automated. Moreover, our approach is also useful for discovering interfaces that the developer may have inadvertently built into the application.

Subsequent work based on modeling investigates automated ways of extracting interfaces. The approach by Elbaum and colleagues [4] builds a model of a web-application interface by submitting a large number of requests to the application, observing the results of the requests, and dynamically inferring constraints on the domain of the input parameters. Their approach is similar to ours in terms of the type of information it tries to extract, but the underlying technique is significantly different because it is based on purely-dynamic analysis. Therefore, a drawback of their technique is that it is inherently incomplete and relies on the often unsafe assumption that web applications are stateless and deterministic. However, an advantage of their technique is that, unlike ours, it can be used when the source code of a web application is unavailable.

Deng, Frankl, and Wang [3] propose a technique based on static analysis for modeling web applications. Their analysis builds a model of the web application by scanning its code to identify links and names of input parameters. Our technique makes several improvements over their approach. In particular, the static analysis that we use in Phase 2 is context- and flow-sensitive, which allows us to be more precise and to capture distinct interfaces that correspond to different paths of execution. Also, in addition to identifying distinct interfaces, our analysis can associate domain information with the elements of the discovered interfaces. As we noted in Section 5.3, the ability to identify domain information, in terms of both type and relevant values of state parameters, can result in a much more thorough testing of a web application.

Another family of approaches is based on capturing user-session data and using this information to guide test case generation. An early approach by Kallepalli and Tian [12] mines server usage logs to build a statistical model of a web application usage that can be used to guide testing and measure the reliability of a web application. A subsequent approach by Sant, Souter, and Greenwald [15] focuses on how these statistical models could be used to generate better test cases. Another approach by Elbaum and colleagues [5,6] captures user-session data and uses the captured data directly as test inputs. Their empirical evaluation of the approach shows that the technique can be as effective as model-based techniques in terms of exposing faults, while being able to generate test cases at a lower cost. Sprenkle and colleagues [17, 18] propose an automated tool that can support the approach proposed in [5, 6] and generate additional test cases based on the captured user-session data. Compared to our approach, these techniques are limited by the quality of the user-session data collected. A nice feature of these approaches, however, is that the values used as test inputs already correspond to legal values. A combination of the two approaches may lead to interesting results.

7. CONCLUSION

In this paper we presented a novel, fully automated static analysis technique for discovering web application interfaces and supporting web application testing. Most existing techniques either require developers to manually specify the interfaces to an application or, if automated, are often inadequate when applied to modern, complex web applications. We evaluated our approach by comparing it to a traditional approach based on web crawling. Our technique was able to discover a higher number of interfaces for all seven web applications considered. The evaluation also showed that using the discovered interfaces and domain information to guide test input generation led to a significantly higher coverage than test input generation based on the results of conventional web-crawling techniques.

In future work, we plan to combine our approach with symbolic execution, which could allow us to associate path conditions and more precise domain information to specific interfaces. We also plan to improve test input generation for web applications by combining our interface discovery method with user-session capture techniques.

Acknowledgments

This work was supported by NSF award CCF-0541080 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-2-0214. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the US Air Force.

8. REFERENCES

[1] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing Web Applications by Modeling with FSMs. In *Software Systems and Modeling*, pages 326–345, Jul. 2005.

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings 10th International Static Analysis Symposium*, pages 1–18, Jun. 2003.

[3] Y. Deng, P. Frankl, and J. Wang. Testing Web Database Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[4] S. Elbaum, K.-R. Chilakamarri, M. F. II, and G. Rothermel. Web application characterization through directed requests. In *International Workshop on Dynamic Analysis*, pages 49–56, May 2006.

[5] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing with User Session Data. In *International Conference on Software Engineering*, pages 49–59, Nov. 2003.

[6] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user-session data to support web application testing. *IEEE Trans. on Software Engineering*, 31(3):187–202, Mar. 2005.

[7] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, May 2004.

[8] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering*, pages 174–183, Nov. 2005.

[9] W. G. Halfond and A. Orso. Command-form Coverage for Testing Database Applications. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering*, pages 69–78, Sep. 2006.

[10] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 12th International World Wide Web Conference* , pages 148–159, May 2003.

[11] X. Jia and H. Liu. Rigorous and Automatic Testing of Web Applications. In *6th IASTED International Conference on Software Engineering and Applications*, pages 280–285, Nov. 2002.

[12] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, 2001.

[13] Open Web Application Security Project (OWASP). OWASP WebScarab Project. `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`, Mar. 2007.

[14] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *International Conference on Software Engineering*, pages 25–34, May 2001.

[15] J. Sant, A. Souter, and L. Greenwald. An Exploration of Statistical Models for Automated Test Case Generation. In *Proceedings of the Third International Workshop on Dynamic Analysis*, pages 1–7, May 2005.

[16] A. Seesing and A. Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proceedings of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, pages 49–53, Oct. 2005.

[17] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated Replay and Failure Detection for Web Applications. In *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 253 – 262, Nov. 2005.

[18] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. A Case Study of Automatically Creating Test Suites from Web Application Field Data. In *Workshop on Testing, Analysis, and Verification of Web Services and Applications*, pages 1–9, Jul. 2006.