Precise Interface Identification to Improve Testing and Analysis of Web Applications

William G.J. Halfond, Saswat Anand, and Alessandro Orso
Georgia Institute of Technology
Atlanta, GA, USA
{whalfond|saswat|orso}@cc.gatech.edu

ABSTRACT

As web applications become more widespread, sophisticated, and complex, automated quality assurance techniques for such applications have grown in importance. Accurate interface identification is fundamental for many of these techniques, as the components of a web application communicate extensively via implicitly-defined interfaces to generate customized and dynamic content. However, current techniques for identifying web application interfaces can be incomplete or imprecise, which hinders the effectiveness of quality assurance techniques. To address these limitations, we present a new approach for identifying web application interfaces that is based on a specialized form of symbolic execution. In our empirical evaluation, we show that the set of interfaces identified by our approach is more accurate than those identified by other approaches. We also show that this increased accuracy leads to improvements in several important quality assurance techniques for web applications: test-input generation, penetration testing, and invocation verification.

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

General Terms: Algorithms, Experimentation, Reliability, Verification

Keywords: Web application testing, interface identification

1. INTRODUCTION

The importance of automated quality assurance techniques for web applications has grown with these applications' increased complexity and sophistication. As users come to expect integrated content and a personalized web experience, developers respond by building web applications that can generate dynamic and customized content. Behind the scenes, the components of a dynamic web application communicate extensively via their interfaces to generate this dynamic content. This makes accurate interface identification an important part of many quality assurance techniques for web applications, such as test-input generation, penetration testing, and invocation verification.

Interface identification is typically not an issue for traditional software, as interfaces are normally explicitly defined in terms of required parameters and their types (*e.g.*, an application programming interface (API)). In contrast, the interfaces of web applications are implicitly defined and can vary at runtime based on the values of input parameters and the flow of execution. This makes it difficult to completely and precisely identify web application interfaces.

There are several approaches to identifying web application interfaces, but they have drawbacks that can limit the effectiveness of quality assurance techniques. For small web applications, manual inspection is possible. However, for larger and more complex web applications, multiple layers of abstraction and the use of frameworks can obscure the intended function of the components. Several approaches [3, 15, 19] rely on developer-provided interface specifications. Although developer-provided specifications can accurately indicate the intended behavior of an application, they are time consuming to produce and may not be consistent with the implementation. Other approaches [6, 14] interact with the web application at runtime and use dynamic analysis to identify interfaces exposed during the interaction. The main limitation of these approaches is that they cannot provide any guarantees of completeness and may not identify hidden interfaces or interfaces that are not accessed during the observed executions. Lastly, another approach [5] uses static analysis to identify interfaces. However, this approach is imprecise, which can result in false positives or false negatives in analyses that rely on the information it produces.

In previous work [12], we proposed a new static analysis based approach for interface identification. In the evaluation of this approach, we showed that a more accurate interface identification can lead to a significant improvement in test-input generation. The primary drawback of this approach is that, even though it is more accurate than existing techniques, in many cases it can compute an overly conservative approximation of an application's interfaces. In this paper we present a novel technique that addresses the limitations of our previous approach by leveraging a specialized form of symbolic execution. The approach represents certain types of input data to a web application as symbolic values and models interface related operations during symbolic execution. The approach then uses the results of the symbolic execution to identify the interfaces of the application.

In this paper we also present the results of an extensive empirical evaluation that measured the efficiency, precision,

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

and usefulness of our new approach. For efficiency, we found that, although symbolic execution based techniques are notoriously expensive, the analysis time of our approach was comparable to that of other static analysis techniques. Our approach achieved this efficiency by customizing the symbolic execution to capture only constraints related to interface definitions and by taking advantage of the high amount of modularity in web applications to avoid scalability problems. For precision, we found that our technique identified fewer spurious interfaces than other approaches. For usefulness, we found that the information produced by our approach led to improvements in the effectiveness of several quality assurance techniques. Our results show that, for the subject applications: 1) test-input generation achieved higher coverage with fewer test cases, 2) invocation verification had fewer false positives and false negatives, and 3) penetration testing discovered a higher number of vulnerabilities. Overall, the results of our evaluation indicate that our technique is efficient, more precise than existing approaches, and can improve the effectiveness of several quality assurance techniques for web applications.

The rest of this paper is organized as follows: Section 2 provides background information on symbolic execution and defines web application terminology. In Section 3 we introduce a motivating example that we use throughout the paper and discuss some of the limitations of existing approaches. We present our approach in Section 4 and its implementation in Section 5. The evaluation is in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2. BACKGROUND INFORMATION

In this section we provide background information on symbolic execution and define web application terminology that we use in the rest of the paper.

2.1 Symbolic Execution

In symbolic execution, a program executes in a normal way except that, instead of concrete input values, it operates on symbolic inputs that can represent arbitrary values [18]. Therefore, at any point in the execution, variables dependent on the inputs are represented as algebraic expressions over the symbolic input values. In addition, a path condition (PC) expresses constraints on these symbolic inputs that must be satisfied in order for execution to reach a specific point in the program. Every time the execution follows a branch whose predicate involves symbolic values, the PC is suitably updated. A PC is normally represented as a conjunction of constraints, $PC = c_1 \wedge c_2 \wedge \ldots \wedge c_n$, where each c_i is a constraint on one or more symbolic values. To illustrate, we use the following code snippet:

```
function foo(int a, int b)
1.  if (a > 5)
2.    if (b = 2)
3.      //do something
4.  if ( b > 3)
5.    //do something else
6.  return
```

In this example, inputs `a` and `b` would be represented as symbolic values. Lines 1, 2, and 4 contain predicates that would contribute to the PCs. Symbolic execution of method `foo` would result in the following PCs for the six paths from lines 1 to 6:

1. $a > 5 \wedge b = 2 \wedge b \leq 3$
2. $a > 5 \wedge b \neq 2 \wedge b > 3$
3. $a > 5 \wedge b \neq 2 \wedge b \leq 3$
4. $a \leq 5 \wedge b > 3$
5. $a \leq 5 \wedge b \leq 3$
6. $a > 5 \wedge b = 2 \wedge b > 3$

Symbolic execution can be combined with constraint solving and used to determine if a path is feasible and which inputs would cause that path to be followed. For example, a constraint solver could determine that the sixth PC, which corresponds to the path that takes the true branches at lines 1, 2, and 4, is not satisfiable (*i.e.*the path is infeasible).

2.2 Terminology

A *web application* is a software system that is accessed over the web via the HyperText Transport Protocol (HTTP). It is comprised of a set of *web components*, which are the modules that implement the application's functionality, such as logging in, displaying an order, or processing a request. A user typically accesses a web application by using a client application, such as a web browser, that displays web pages produced by the web application, allows users to enter data (*e.g.*, via a web form), and submits data on behalf of the user. The web application processes user input and returns generated content to the client application. To generate content for the end user, the components of a web application communicate by sending a certain type of HTTP request, called an *interface invocation*, to the interfaces of other components. An interface invocation provides arguments in the form of name-value pairs (*e.g.*, `login=username`).

A web component receives invocations via its interfaces. In traditional applications, the interfaces of a module are explicitly defined via an API or other form of specification. In contrast, the interfaces to a web application are implicitly defined. A web application can access different sets of parameters by name along different paths of execution. We call the set of input parameters (IP) accessed by a web application during a particular execution an *accepted interface* of the web application. A component accesses an IP by calling a *parameter function* (PF), which takes as input the name of the IP and returns a string that contains the corresponding value. Each IP is uniquely identified by its name. The general form for accessing the value of an IP is $\text{IP}_{value} = \text{PF}(\text{IP}_{name})$.

Like the accepted interfaces, the domain of each IP is also implicitly defined—PFs return all IP values as strings, so the domain of an IP can only be inferred based on the operations performed on the IP along a particular execution path. Certain types of operations that we call *domain-constraining operations*, implicitly constrain the domain of an IP. Examples of these operations are functions that convert an IP value into a numeric value or comparisons of the IP value against a specific value. We call the set of constraints placed on an accepted interface along a specific execution path an *interface domain constraint* (IDC). An accepted interface may have more than one IDC associated with it, if different domain-constraining operations are performed on its IPs along different paths.

3. EXAMPLE WEB APPLICATION

In this section we introduce an example web application that we will use in Section 4 to illustrate our technique.

```
 1.  String actionValue = getIP("action");
 2.  if (actionValue.equals("checkeligibility")) {
 3.    String nameValue = getIP("name");
 4.    int ageValue = getNumIP("age");
 5.    String stateValue = getIP("state");
 6.    if (!stateValue.equals("GA")) {
 7.      error=true;
 8.      errorMessage = "Bad state";
 9.    }
10.    if (ageValue < 16) {
11.      error=true;
12.      errorMessage += "Too young to drive.";
13.    }
14.    if (error) {
15.      displayErrorMessage(errorMessage);
16.    } else {
17.      saveSession(nameValue, ageValue, stateValue);
         String action = "doQuote";
18.      displayQuotePage(action, nameValue);
19.    }
20.  }
21.  if (actionValue.equals("doQuote")) {
22.    String nameValue = getIP("name");
23.    String carType = getIP("type");
24.    int carYear = getNumIP("year");
25.    if (!nameValue.equals("")) {
26.      Session s = getSession(nameValue);
27.      int age = s.getAge();
28.      String state = s.getState();
29.      if (carType.contains("motorcycle")
             && nameValue.equals("Evel Knievel")) {
30.        displayErrorMessage("Absolutely not.");
31.      } else {
32.        calculateQuote(nameValue, age, state, carType, year);
33.      }
34.    } else {
35.      displayErrorMessage("Session expired.");
36.    }
37.  }
38.  if (!actionValue.equals("checkeligibility")
         && !actionValue.equals("doQuote")) {
39.    String action = "checkeligibility";
40.    displayWelcomePage(action);
41.  }
42.  int getNumIP(String name) {
43.    String value = getIP(name);
44.    int param = Integer.parse(value)
45.    return param;
46.  }
```

Figure 1: Example servlet.

Figure 1 shows an excerpt from the main method of a Java-based web application. The example is implemented as a *servlet*, which is the Java language's implementation of a web component. The example application allows users to obtain a car insurance quote via a company's website.

When a user visits the example servlet, the first action is for the servlet to make a call to PF `getIP()` and retrieve the value of the IP named `action` (line 1). If this is the user's first visit to the servlet, the parameter will be undefined, so the condition at line 38 will be true, which causes the servlet to call method `displayWelcomePage`. This method (not shown) generates an HTML page that contains a web form with user input fields for `name`, `age`, and `state`. The form also contains a hidden state field that maintains the value of `action`, which is set to "checkeligibility." When this form's data is submitted to the servlet, the value of `action` makes the condition at line 2 true, so the IPs named `state`, `age`, and `name` are accessed. If the user lives in the state of Georgia and is over 16, the condition at line 14 is false, which causes the session information to be saved, and a final web form with `action` equal to "doQuote" is returned. This form allows the users to enter additional information regarding their vehicle and submit the information back to the servlet. On receiving the data, `action` is equal to "doQuote," so the condition at line 21 is true, and the IPs named `carYear`, `carType`, and `name` are accessed. If `name` is defined, the servlet retrieves the state information and checks that Evel Knievel is not trying to insure his motorcycle. If the input passes this check, the quote is displayed to the users; otherwise, an error message is returned.

Interface	*IP Names*
1	action
2	action, name, age, state
3	action, name, carType, carYear

Table 1: Interfaces of the example servlet.

In the example servlet, there are three accepted interfaces, which are shown in Table 1. The first accepted interface is accessed on the user's first visit to the servlet. The second is accessed when `action` is equal to "checkeligibility." The third is accessed when the user has passed the eligibility check and `action` is equal to "doQuote."

1. action = "checkeligibility" $\wedge$ type(age) = int $\wedge$ age $\geq$ 16 $\wedge$ state = "GA"
2. action = "checkeligibility" $\wedge$ type(age) = int $\wedge$ age $<$ 16 $\wedge$ state = "GA"
3. action = "checkeligibility" $\wedge$ type(age) = int $\wedge$ age $\geq$ 16 $\wedge$ state $\neq$ "GA"
4. action = "checkeligibility" $\wedge$ type(age) = int $\wedge$ age $<$ 16 $\wedge$ state $\neq$ "GA"

Figure 2: Interface Constraints.

Each of the accepted interfaces has several interface domain constraints. In Figure 2, we enumerate the constraints associated with the second accepted interface. In all four of the interface domain constraints, `action` is equal to "checkeligibility" since this constraint is imposed by the check at line 2. Additionally, the type of `age` is always an integer because it is accessed through function `getNumIP`, which converts the string returned by the PF at line 44 to type `int`. The remainder of each constraint varies according to the branches followed at lines 6 and 10.

In general, identifying the accepted interfaces and interface domain constraints of a servlet is non-trivial. Although the example servlet is straightforward, many servlets have thousands of lines of code, access input parameters through multiple layers of abstraction, and have interfaces that are displayed only after certain conditions have been met by the end user. These characteristics make it difficult to identify interfaces via manual inspection or even via automated techniques.

Interface Identification with Web Crawling.

Web crawling is a popular and common approach to identifying accepted interfaces. In this approach, a program, typically called a web spider, visits and analyzes web pages to discover links to other pages and web-form-related information, such as input boxes, combo boxes, and radio buttons. The spider then follows all the links in the page and repeats this process until there are no new links to visit. All of the information discovered during this process is saved and used to infer interface information about the individual components.

Even our simple example servlet would cause problems for a web spider. In the servlet there are multiple forms that must be filled with specific values in order to trigger the display of subsequent forms. For example, to display the page that contains the elements corresponding to interface 3, the spider would have to guess that `state` must be equal to "GA" and that `age` must be greater than or equal to 16. If the spider does not correctly guess the constraints on the input parameters, then the interface information gathered by the spider would be incomplete and would include only interfaces 1 and 2. If this information were to be used to drive test-input generation, the resulting test suites would be unlikely to cover lines 21–37 of the example servlet. Additionally, it is possible for a component to have interfaces that do not correspond to an HTML form (*e.g.*, for use only in direct inter-component communication), and it would not be possible for a spider to find these interfaces under any circumstances.

Interface Identification with Existing Static Analysis Techniques.

Existing static analysis based techniques also have difficulties in identifying interfaces. A web application modeling technique by Deng, Frankl, and Wang [5] scans the bytecode of a web application and identifies the names of the IPs accessed by the application. This information is useful for the technique's original purpose of modeling web applications; however, for quality assurance purposes, its usefulness is limited because it does not group IPs into logical interfaces. For example, the result of running this technique on the example servlet would be the set containing the name of each accessed IP (i.e., "action", "name", "age", "state", "carType", "carYear"). As we will show in Section 6, our more focused technique performs better for several common quality assurance techniques, such as test input generation and invocation verification.

Our previous work [12] proposed a two-phase static analysis based technique for identifying web application interfaces and domain constraints on those interfaces. Although this technique led to significant improvements versus other techniques, the analysis computes an overly conservative approximation of an application's interfaces. The approximation occurs for two reasons. First, the technique is based on an iterative data-flow analysis, which assumes that all paths are feasible. This leads to the identification of spurious interfaces, such as {"action", "name", "age", "state", "carType", "carYear"}. Although these IPs are accessed along the same path in the example servlet, it is clear that the conditions at lines 2 and 21 are mutually exclusive and this combination of IPs would not be accessed at runtime. Second, the identification of interface constraints is not path-specific. Instead, it is tied to the name of the identified IP. To illustrate, the technique would define the domain of `action` as `action` $\in$ {"checkeligibility", "doQuote"}. Although this is correct, it is a conservative approximation, and for many quality assurance techniques it would be useful to have more precise information, such as knowing that `action` is always equal to "doQuote" in interface 3. Precision is important because it affects the accuracy and efficiency of techniques that leverage the interface information, such as test input generation, invocation verification, and penetration testing. Our evaluation in Section 6 provides evidence of the effect of precision on these types of techniques.

4. OUR APPROACH

The goal of our approach is to automatically and precisely identify a web application's accepted interfaces and interface domain constraints. Our approach works in three main steps. In the *first* step, our technique performs a transformation of the web application so that IPs are represented as symbolic values and domain-constraining operations are modeled by symbolic operations. In the *second* step, our technique symbolically executes the web application and generates a set of PCs for each component. In the *third* step, our technique identifies the accepted interfaces and interface domain constraints of the web application by analyzing the PCs generated during symbolic execution. In the following sections, we discuss each of these steps in more detail.

4.1 Step 1: Symbolic Transformation

The goal of this step is to transform a web application so that its symbolic execution will provide information about accepted interfaces and interface domain constraints. There are two parts to this transformation. The first is to identify points in the application where symbolic values must be introduced to precisely model the application's IPs. The second is to replace domain-constraining operations with special symbolic operators that will appropriately update the PC as the application is symbolically executed.

Introduce symbolic values: A straightforward symbolic execution of a web application would not capture information related to the individual IPs accessed by the application. Since IPs are passed to the application as part of an invocation (see Section 2.2), many symbolic execution techniques would model them as an array of symbolic characters. This could create scalability issues and would not provide us with IP information at the right level of abstraction. To address this issue, our approach models each individual IP as a symbolic value. Our approach modifies PFs so that, when the application accesses an IP, they return a reference to a symbolic string instead of a normal string object. Each symbolic string returned by a PF is uniquely identified by the name of the IP that was passed as an argument to the PF. This name is always known at the time of execution, unless the name itself is also a symbolic string or it is defined externally (*e.g.*, in a resource file). In this case, it is not possible to determine the name of the IP and the approach adds a constraint to the PC that specifies that the IP's name is equal to the value of another IP or is defined externally. If a specific IP is accessed more than once along a path, a reference to the previously returned symbolic string is returned. This is consistent with the normal behavior of the PFs, which return the same value when called with the same IP name.

Replace domain-constraining operations: To accurately capture the constraints placed on IP values, our approach replaces domain-constraining operations with specialized symbolic versions of the operations. These specialized versions handle the symbolic strings that represent the IPs and update the PC to reflect the domain constraints they place on the IPs' values. Although the specific operations vary according to the web application framework, the following is a general list of the types of operations that are replaced: 1) string comparators, 2) functions that convert a string to an integer, float, or double, and 3) arithmetic comparison operations: $>$, $<$, $\neq$, $\geq$, $\leq$, and $=$. In Section 4.2, we formally define the effects of the domain-constraining operations on the symbolic execution.

(PC, state)–Before	Operation	(PC, state)–After
(C, ST)	`s = getIP(name)`	$(C, ST[s \hookrightarrow \overline{s}_{name}])$
$(C, ST[s \hookrightarrow \overline{s}_{name}])$	`v = Integer.parse(s)`	$(C \wedge type(\overline{s}_{name}) = \texttt{int}, ST[s \hookrightarrow \overline{s}_{name}, v \hookrightarrow \overline{v}_{name}])$
$(C, ST[s \hookrightarrow \overline{s}_n, t \hookrightarrow \overline{t}_m])$	`if(s.equals(t)){}`	$(C \wedge \overline{s}_n = \overline{t}_m, ST[s \hookrightarrow \overline{s}_n, t \hookrightarrow \overline{t}_m])$
	`else{}`	$(C \wedge \overline{s}_n \neq \overline{t}_m, ST[s \hookrightarrow \overline{s}_n, t \hookrightarrow \overline{t}_m])$
$(C, ST[v \hookrightarrow \overline{v}_n, w \hookrightarrow \overline{w}_m])$	`if(v` $\otimes$ `w)){}`	$(C \wedge \overline{v}_n \otimes \overline{w}_m, ST[v \hookrightarrow \overline{v}_n, w \hookrightarrow \overline{w}_m])$
	`else{}`	$(C \wedge \neg(\overline{v}_n \otimes \overline{w}_m), ST[v \hookrightarrow \overline{v}_n, w \hookrightarrow \overline{w}_m])$

Figure 3: Path condition and program state before/after execution of specific types of statements.

4.2 Step 2: Generating Path Conditions

In the second step, our approach generates the web application's PCs by symbolically executing the transformed web application. Each PC generated in this step corresponds to a family of paths from the entry to the exit of one of the application's web components. The symbolic execution generates the PCs by collecting constraints on the symbolic values during execution of the component. These constraints are created by the execution of operations on symbolic values.

To explain the details of our symbolic execution, we use the table in Figure 3. This table shows a formal specification of the effect of different program statements on the PC and on the symbolic state of the program. The PC is shown as a conjunction of constraints, and the symbolic state of the program is represented by a valuation function ST that maps each variable in the program to its corresponding value in the state. For example, $ST[x \hookrightarrow v]$ specifies that, in the symbolic state, variable x is mapped to the symbolic value v. In the table, the left-hand column shows the PC and relevant parts of ST before the operation, the middle column shows the operation, and the right-hand column shows the PC and the relevant parts of the ST after the operation.

Accessing the IP: The symbolic execution of a PF (i.e. `s = getIP(name)`) creates a symbolic string $\overline{s}_{name}$ and assigns it to `s`. The access creates a one-to-one mapping in ST between the IP name and the symbolic string. In the example servlet, an IP is accessed at lines 1, 3, 5, 22, 23, and 43. The execution of these lines updates the symbolic state of the program with new symbolic strings. For example, line 1 creates the symbolic string $\overline{s}_{action}$ and maps it to the variable `actionValue`.

Conversion to numeric type: When a statement of type `i = Integer.parse(s)` is executed, and the value of `s` is a symbolic string $\overline{s}_{name}$, the technique updates ST and the PC. The constraint $type(\overline{s}_{name}) = \texttt{int}$ is added to the PC to record the fact that, on the current path, the symbolic string $\overline{s}_{name}$ is converted to an integer value. The approach updates ST by adding a new symbolic integer $\overline{v}_{name}$ that represents the numeric value of the symbolic string. A symbolic string can also be converted to `float` and `double` types. These cases are handled similarly to the case of `int`. A one-to-one mapping between a symbolic string and its corresponding symbolic numeric value is maintained via the name attribute. As a consequence, if a symbolic string is converted to a numeric value multiple times on a path, only one symbolic value is created during the first conversion and then reused for subsequent accesses.

In the example, every IP that is accessed via `getNumIP` is converted to an `int` by the call to `Integer.parse()` at line 44. For example, the call at line 4 modifies the PC by adding the constraint $type(\overline{s}_{age}) = \texttt{int}$ and adds a mapping `ageValue` $\hookrightarrow \overline{v}_{age}$ to ST.

String comparison: When a branch condition uses a symbolic string in a string equality operation ($\overline{s}$`.equals(t)`), our approach attempts to evaluate the condition. If the constraints in the PC are sufficient to evaluate the condition, the approach can determine which branch to follow. Otherwise, the symbolic execution follows both branches. Along the true branch, the approach conjoins the PC with the branch condition; along the false branch, it conjoins the PC with the negation of the branch condition.

To illustrate with an example, consider the comparison of $\overline{s}_{action}$ at line 2. When this comparison is evaluated, $\overline{s}_{action}$ is not a concrete value, and the correct branch to follow cannot be determined. Therefore, the symbolic execution follows both branches and creates two PCs, one with the constraint $\overline{s}_{action}$ = "checkeligibility", and the other with $\overline{s}_{action} \neq$ "checkeligibility". Along one of the paths reaching line 21, $\overline{s}_{action}$ is equal to "checkeligibility", so the constraint solver can evaluate this comparison and determine that $\overline{s}_{action}$ cannot also be equal to "doQuote." Along another path reaching line 21, $\overline{s}_{action}$ is not equal to "checkeligibility". Therefore, $\overline{s}_{action}$ may or may not be equal to "doQuote" and, once again, two PCs are generated, one for each branch.

Arithmetic constraints: If an arithmetic expression of the form $i \otimes j$ is evaluated in a predicate, and one of the operands is a symbolic numeric value, our approach adds the arithmetic constraint to the PC. Operator $\otimes$ can be one of the following arithmetic comparison operators: $>$, $<$, $\neq$, $\geq$, $\leq$, and $=$. In the example servlet, an arithmetic comparison on a symbolic value occurs at line 10. Since the value of $\overline{v}_{age}$ cannot be determined, the result of the evaluation of this statement is two PCs, one with the constraint as true ($\overline{v}_{age} < 16$) and the other one with the constraint as false ($\overline{v}_{age} \geq 16$).

4.3 Step 3: Interface Identification

In the third step, our approach identifies accepted interfaces and IDCs by analyzing the PCs and symbolic states generated in Step 2. To do this, our approach takes advantage of several insights. First, each IP accessed along a path is added to the symbolic state; therefore, each unique collection of names in ST corresponds to the IP names that define an accepted interface of the component. Second, each constraint in the PC represents part of a domain-constraint on the value of a particular IP; therefore, the PC corresponds to an IDC. Third, the IDC corresponds to the accepted interface identified in the symbolic state.

To illustrate with an example, we consider the ST of the family of paths that take the true branch at line 2 of the example servlet in Figure 1. As described in Section 4.2, lines 1, 3, 4 and 5 contain statements that modify the symbolic state. The relevant part of the symbolic state for this family of paths is:

ST[`actionValue` $\hookrightarrow \overline{s}_{action}$, `nameValue` $\hookrightarrow \overline{s}_{name}$, `ageValue` $\hookrightarrow \overline{s}_{age}$, `stateValue` $\hookrightarrow \overline{s}_{state}$]

Based on the first insight, our approach can analyze the ST and determine the unique collection of names {action, name, age, state} that define the accepted interface (interface 2 from Table 1). Based on the second and third insights, we analyze the PCs of these paths to identify the accepted interface's IDCs. Since all paths take the true branch at line 2, all PCs have the constraint $\overline{s}_{action}$ ="checkeligibility". Line 4 adds the constraint `type`($\overline{s}_{age}$) =`int`. Line 6 causes the execution to fork into two paths. The first path adds $\overline{s}_{state}$ ="GA" to one PC, and the second adds the negation, $\overline{s}_{state} \neq$"GA" to the PC of the second path. Line 10 causes another fork in the execution. The first path adds $\overline{s}_{age} < 16$ to the PC and the second adds the negation, $\overline{s}_{age} \geq 16$, to the other PC. The four IDCs identified in this analysis are shown in Figure 2.

5. IMPLEMENTATION

To evaluate our approach, we developed a prototype tool called WAM-SE (Web Application Modeling with Symbolic Execution). WAM-SE is written in Java and implements our technique for web applications written in the Java Enterprise Edition (JEE) framework. The implementation consists of three modules, TRANSFORM, SE ENGINE, and PC ANALYSIS, which correspond to the three steps of our approach. In the rest of this section, we discuss the implementation of the modules in detail.

The **transform** module implements the symbolic transformation described in Section 4.1. The input to this module is the bytecode of the web application and the specification of program entities to be considered symbolic (in this case, symbolic strings). The module transforms the application to introduce symbolic values and replaces domain-constraining operations with their special symbolic counterparts. The output of the module is the transformed web application, which is ready to be symbolically executed in Step 2.

To perform the transformation, we use Stinger, a technique and tool that was previously developed by two of the authors [1]. Stinger identifies points in an application where symbolic values are introduced. It then analyzes the code to determine which operations and types in the code may interact with the symbolic values and transforms them into their symbolic counterparts. A benefit of using Stinger is that it allows our approach to only translate types and operations that should be symbolic and avoid the unnecessary overhead that would be introduced by transforming the entire application.

To specify the program entities to be considered symbolic, we built a customized version of the JEE libraries. This version creates a new symbolic string for an IP when a PF function in the JEE library is symbolically executed. We made two main customizations: 1) the definition and implementation of a symbolic string class for Java, and 2) the rewrite and modification of all PFs so that they return a symbolic representation of each accessed IP. The symbolic string is implemented as an extension to the normal Java `String` class with overridden member functions to account for the different semantics of a symbolic string. Currently, the only string operator modeled by our implementation is string equality, which includes equality between two symbolic strings, two constant strings, or a constant string and a symbolic string. Constraints involving more complex operations, such as matching of regular expressions, are currently ignored by the symbolic execution. Extending the technique to model these types of constraints would increase the precision of the IDCs, but would also increase the cost of the constraint solving. Our examination of subject applications suggests that the increase in code coverage would be minimal with this extension.

The modified PFs, when accessed, create a symbolic string, associate the name of the accessed IP with the symbolic string, and maintain a map of IP names to symbolic values to ensure that the same symbolic value is returned when an IP name is accessed multiple times. Along with these customizations, we also implemented symbolic versions of the numeric conversion functions. No further implementation was necessary to handle arithmetic operations, as symbolic versions of these operations are provided by the underlying symbolic execution engine we use.

Stinger also identifies two types of situations in the code that might cause problems for the symbolic execution: 1) constraints that cannot be handled by the underlying decision procedure, and 2) symbolic values that may flow outside the scope of the symbolically executed code (*e.g.*, to native code). For the first situation, a limitation of the underlying decision procedure that we used was that it could not handle symbolic floating point values. When Stinger detected floating point values and operations that needed to be replaced, we rewrote the code in the applications so that the same operations were expressed in terms of integer values. This occurred only three times in the applications we used for our evaluation and involved rewriting predicates of the form "$value \otimes X.0$" to "$value \otimes X$," where $\otimes$ is any of the arithmetic operators and X is some integer value. There were no other constraints that could not be handled by our decision procedure. Although the subjects did contain more complex conditions involving floating point values, none of these involved the symbolic IP values, so they were not an issue. Almost all of the constraints that did involve symbolic IP values were fairly small (2 – 4 conditions) and typically only involved numeric equality, string equality, or a numeric conversion. For the second potentially problematic situation, we found that although there was extensive use of external libraries in the subject applications, none of the symbolic values flowed into these libraries. This occurred because we symbolically modeled only the IPs and these were never passed to the external libraries.

The **se engine** module implements the symbolic execution described in Section 4.2. The input to this module is the bytecode of the transformed web application, and the output is the set of all PCs and corresponding symbolic states for each component in the application. To implement the symbolic execution, WAM-SE leverages a symbolic execution engine [17] built on top of Java PathFinder (JPF) [22] and the YICES (`http://yices.csl.sri.com/`) constraint solver. JPF is an explicit-state model checker for Java programs that supports all features of Java. JPF explores all program paths systematically, when it reaches the end of the program it backtracks to every non-deterministic branch on the path and explores other paths from that branch. This process continues until every path in the program has been explored. The symbolic execution engine handles recursive data structures, arrays, numeric data, and concurrency. To check satisfiability of path conditions, JPF can use differ-

Subject	*LOC*	*Classes*
Bookstore	19,402	28
Classifieds	10,759	18
Employee Directory	5,529	11
Events	7,164	13

Table 2: Subject applications.

	Total Time (s)			
Subject	W_{df}	W_{se}	DFW	*Spi.*
Bookstore	4,093	1,479	1,138	2,774
Classifieds	1,985	766	377	239
Employee Dir.	741	905	253	101
Events	333	586	231	15

Table 3: Analysis time.

ent decision procedures [2]. If the satisfiability of the path condition cannot be determined, as the problem of checking satisfiability is undecidable in general, JPF assumes that both branches are feasible. This is a safe way to handle the situation, but can reduce the precision of the analysis. In our evaluation, we did not have any constraints that could not be solved by the standard constraint solver used by JPF.

The **pc analysis** module implements the analysis described in Section 4.3. The input to this module is the set of PCs and symbolic states for each component in the application, and the output is the set of IDCs and accepted interfaces. The module iterates over every PC and symbolic state, identifies the accepted interfaces, and associates the constraints on each IP with its corresponding accepted interface.

6. EVALUATION

In our empirical evaluation we assess the efficiency, precision, and usefulness of our approach for interface identification. To do this we compared WAM-SE against several other approaches for interface identification. In the evaluation, we investigated the following three research questions:

RQ1: Efficiency – Is the new approach efficient in terms of its analysis time requirements?

RQ2: Precision – Is the new approach more precise than previous approaches?

RQ3: Usefulness – Does the new approach improve the performance of quality assurance techniques?

6.1 Experimental Subjects

Our experimental subjects consist of four commercial Java-based web applications available from GotoCode (`http://www.gotocode.com/`) that we have used in previous work [11, 12, 13]. Table 2 provides general information about the subject applications. For each application, the table lists the application's size (*LOC*) and number of classes (*Classes*).

6.2 Interface Identification Approaches

In our empirical evaluation, we compare our approach, WAM-SE, against three other techniques for interface identification, WAM-DF, SPIDER, and DFW. The SPIDER technique is based on the OWASP WebScarab Project,[1] which is a widely-used Java-based implementation of a web crawler. We extended the OWASP spider by adding to it the capability of extracting interface related information from the web pages and recording default values, if present, for the interface elements. Additionally, we enabled the spider to perform the most thorough exploration possible by giving it administrator-level access to the subject web applications.

[1] `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`

The other two techniques, WAM-DF and DFW, perform static analysis of a web application. WAM-DF is the prototype implementation of the previous approach developed by two of the authors [12]. This approach is based on a two-phase static analysis algorithm. In the first phase, the approach computes domain information for each IP in a servlet by identifying the use of the IP in domain-constraining operations. In the second phase, the approach uses an iterative data-flow analysis to identify IP names and group them into logical interfaces based on the servlet's control and data flow. DFW [5] is a technique developed by Deng, Frankl, and Wang that scans the code of a web application and identifies all IPs accessed by the application. Although DFW's original purpose was modeling of web applications, it identifies some of the same information as our technique (See Section 3), so we included it in this evaluation. Deng, Frankl, and Wang provided us with the original DFW implementation, and we used their code as a guide to reimplement the technique so that it would work within our analysis framework. We also extended the technique to address an implementation limitation where it could only identify IP names that were defined by constant strings in the same method scope. Both DFW and WAM-DF are implemented in Java, and where possible, we reused code in their implementations (e.g. they share the code to build control-flow graphs and scan the web applications' bytecode.)

We chose to use these three techniques because they are representative of the two main automated approaches to interface identification: dynamic analysis via web crawling (SPIDER) and static analysis of the web application code (WAM-DF and DFW). We did not compare against techniques based on manual specification since the performance of such techniques is dependent on the skill-level of the developers who provide the specification.

6.3 RQ1: Efficiency

In the first study, we investigated the efficiency of the four different approaches for identifying web application interfaces. To do this, we measured the running time of the approaches on each of the subject applications. The evaluation machine had a Pentium D 3.0Ghz processor, 2GB of memory, and a GNU/Linux 2.6 based operating system. Table 3 shows the results of our timing experiment. For each application, the table shows the time in seconds to analyze the web application (*Total Time*).

Overall, the results show that WAM-SE is the third fastest technique for interface identification and significantly faster than WAM-DF. It seems counter-intuitive that WAM-SE is faster than WAM-DF, as symbolic execution is a notoriously expensive analysis; however, there are several factors that contribute to this result. First, the subject applications contain a high number of infeasible paths, which indicates the

Subject	Identified Interfaces W_{se}	W_{df}
Bookstore	70	338 (268)
Classifieds	41	222 (181)
Employee Dir.	18	88 (70)
Events	25	118 (93)

Table 4: Precision of wam-se and wam-df.

possibility that the time to propagate data-flow information along the extra infeasible paths is higher than the cost of the symbolic execution. Second, as compared to typical symbolic execution, our approach only symbolically models the IPs, and there are not many constraints that involve the IPs (2–4 per IP in our subjects). This means that the PCs generated by the symbolic execution tend to be relatively small and can be solved quickly by the constraint solver. Third, WAM-DF is a prototype implementation, whereas Stinger and JPF are more mature and have been optimized for performance. Fourth, as compared to traditional software, web applications do not cause as many scalability problems for symbolic execution. Web applications are highly modular, components can be analyzed independently to identify interfaces, and the size of a typical component is generally no more than several thousand lines of code, which can be handled by most modern symbolic execution implementations. Lastly, our approach models the IPs at the string level, which reduces the total number of constraints that would otherwise be generated by modeling the IPs at the character level.

Overall, we believe that the results indicate that WAM-SE is a practical technique. It is faster than WAM-DF, which is a conservative analysis, only slightly slower than SPIDER, which is a widely used, but inherently incomplete, technique, and slower than DFW, which only performs a lightweight analysis of the code and does not group IPs into interfaces.

6.4 RQ2: Precision

In the second study, we investigated the precision of the WAM-SE approach as compared to the WAM-DF and SPIDER approach. We did not consider the precision of DFW since it does not group the identified IPs into logical interfaces. To perform this investigation, we measured the number of interfaces identified by each approach and compared the relative precision of the different approaches.

Table 4 shows the comparison of WAM-SE and WAM-DF. For each application (*Subject*) we list the number of interfaces identified and, in parenthesis for the WAM-DF approach, the additional amount of identified interfaces as compared to WAM-SE. Since both approaches are conservative in their interface identification (*i.e.*, they identify a superset of an application's interfaces), a lower number of interfaces indicates higher precision. As the results show, the WAM-SE approach was consistently more precise in its identification of interfaces. On average, WAM-DF found almost five times as many interfaces as WAM-SE. Since WAM-DF assumes all paths are feasible, the set of interfaces it discovered was always a superset of the interfaces discovered by WAM-SE. From examining the code, we found that a contributing factor to WAM-DF's lower precision was the high number of infeasible paths. Coding style in the applications also increased this number, as many `if` blocks followed each other sequentially and did not use `else if` constructs that would have made the paths mutually exclusive.

We also compared the precision of WAM-SE and SPIDER. The SPIDER approach discovered a smaller number of interfaces; however, manual inspection of the code revealed that its results were incomplete. Furthermore, we discovered that almost half of the interfaces discovered by this approach did not actually correlate with the accepted interfaces accessed by the code. From our inspection, it appeared that this was due to developer errors in which parameters of the web form were either ignored by the code or not defined in the web forms analyzed by SPIDER. These results demonstrate that not only is the SPIDER approach incomplete, but it can be imprecise as well.

6.5 RQ3: Usefulness

In the third study, we investigated whether the increased precision provided by our approach can actually improve the performance of quality assurance techniques for web applications. To answer this question, we modified three quality assurance techniques so that they could work with the interface information provided by the four different approaches considered. We then ran each technique on the subject web applications and evaluated the performance of the techniques. In the rest of this section, we explain each of the quality assurance techniques in more detail and discuss the performance results for each of them.

Invocation Verification: As explained in Section 2.2, components of a web application communicate by sending invocations to the interfaces of other components. The first quality assurance technique we evaluated detects errors in this type of communication. This technique, which is based on previous work by two of the authors [13], analyzes the accepted interfaces and the interface invocations of a web application in order to find invocations that do not match an accepted interface of the target component.

Approach	Verification Results *Ok*	*Error*
SPIDER	3 (0)	23 (9)
WAM-DF	24 (12)	2 (0)
WAM-SE	12 (0)	14 (0)

Table 5: Invocation Verification for Bookstore.

We ran the invocation verification technique on subject Bookstore using interface information provided by WAM-SE, WAM-DF, and SPIDER. Again in this study, we did not consider DFW since it does not group IPs into logical interfaces. The results of this study are shown in Table 5. For each of the three approaches considered, we listed the number of invocations that were reported as correct (*OK*) and the number that were listed as errors (*Error*). Next to each result, we list the number of erroneously classified invocations in parenthesis. We determined the number of these misclassifications by manually checking the result of each invocation verification. Due to the time requirements of this manual checking, we only used one subject in the study. As the results show, all of the errors reported by the analysis when using the WAM-SE provided information were confirmed as errors. We manually inspected the code and found that: 1) the WAM-DF approach led to the misclassification of 12 invocations as OK (false negatives) because these invocations

matched with spurious interfaces identified by WAM-DF, and 2) the SPIDER approach led to the misclassification of nine invocations as errors (false positives) because it could not completely identify all of the accepted interfaces; the interfaces missed by SPIDER are the ones matching the misclassified invocations.

	Vulnerable Parameters			
Subject	W_{df}	W_{se}	*Spi.*	DFW
Bookstore	11	36	7	5
Classifieds	14	31	4	18
Employee Dir.	11	19	1	4
Events	11	23	4	2
Total	47	109	16	29

Table 6: Penetration Testing.

Penetration Testing: The second technique we evaluated was penetration testing of web applications. In penetration testing, a tester attempts to discover security vulnerabilities in an application by simulating attacks that could be performed by a malicious user. Many penetration testers use automated tools that interact with a web application and create customized attacks based on the responses received from the application. These tools typically employ a combination of heuristics and dynamic analyses in order to create test inputs that are most likely to discover vulnerabilities. For our evaluation, we used one such tool, SQLMAP (`http://sqlmap.sourceforge.net/`), which is a popular automated penetration testing tool that has been downloaded over seven million times. SQLMAP detects vulnerabilities to SQL Injection Attacks (SQLIA), which are attacks that can give a malicious user complete control and access to the database underlying a web application. SQLMAP uses interface information, such as the names of IPs and their domain information, to generate test cases that target an application with SQLIAs. In typical usage, interface information is provided by testers either via manual specification, or by using a web spider to crawl the web application.

For our evaluation, we used a slightly modified version of SQLMAP that we developed in previous work [10] in which we compared the results of SPIDER and WAM-DF based penetration testing. This version of SQLMAP was modified from the original version to handle our interface information format. We ran SQLMAP on the subject applications using the interface information provided by the WAM-SE, WAM-DF, SPIDER, and DFW approaches. Table 6 lists the number of vulnerable IPs discovered by SQLMAP. We manually confirmed that each reported IP could be used to inject an attack. The results show that penetration testing based on information provided by the WAM-SE approach leads to the identification of the highest number of vulnerable IPs. In our investigation of the results, we determined that this improvement was primarily due to the precise domain information provided by WAM-SE. As we explain in the next study, the improved domain information allows more database commands to be run, therefore providing more opportunities for attacks to enter the database. In general, the number of IPs discovered by an approach could also affect the number of vulnerable IPs discovered. However, in our experiment, all of the approaches, except for SPIDER, identified all of the IPs, so this did not affect the results for the DFW and WAM-DF approaches.

Test-Input Generation: The third technique we used to evaluate the usefulness of our approach is test-input generation. For each of the four interface identification approaches, we generated test suites using its interface information and measured the structural coverage achieved on each application by the test suites. Although higher coverage does not necessarily imply better fault detection, coverage is an objective and commonly-used indicator of test suite quality. We measured coverage using three different coverage criteria: basic block, branch, and database command-form. Basic block coverage measures the number of distinct basic blocks of the program that were executed by a test suite. Branch coverage measures the number of distinct branches (*e.g.*, the true or false branches of each `if`) that are traversed during the execution of a test suite. Database command-form coverage [11] measures the number of distinct types of database commands generated by an application. Since most web applications are data-centric, database command-form coverage is useful to determine if the application is exhibiting different behaviors with respect to its underlying database.

To perform the study, we generated four test suites for each subject application. Each test suite was generated using the information from a different interface identification approach. For the WAM-SE approach, we generated the test cases by solving the constraints in each IDC and using the values in the solution as test inputs. In our evaluation, the constraint solver was able to solve all of the IDCs. For the other three approaches, we created the test cases by taking the Cartesian product of the domain values of each IP for each accepted interface. The test case generation for these three approaches differed only in how IPs' domain values were determined. For the WAM-DF approach, the domain information is tracked per IP instead of per path (see example in Section 3). If no values were provided by the approach, we augmented the domain values with either a random string or a numeric value depending on the identified type of the IP. For the SPIDER approach, we generated test cases using default values for the IPs. Because default values are supplied with a web form, they represent a good source of legal values for interface elements. If no default values were discovered for an IP, we used a random alphanumeric string and the empty string as its possible values. For the DFW approach, no domain information is identified, so each IP was assigned the possible values of a random alphanumeric string and the empty string. Our reason for including the empty string as a possible value is that many applications check whether an IP is defined. Including the empty string can therefore help these approaches achieve higher coverage.

After generating the test inputs, we instrumented the code of each web application to measure the coverage of the different criteria. To monitor basic-block and branch coverage, we used COBERTURA (`http://cobertura.sourceforge.net/`), and to measure command-form coverage we used DITTO [11]. Some of the applications required a priming script to be executed before the actual test cases could be run (e.g. a login to create a session ID). For these applications, our testing infrastructure ran the priming script before each test case. Table 7 shows, for each application and approach (W_{df}, W_{se}, *Spi.*, and DFW), the level of basic block (*Block*), branch (*Branch*), and command-form (*Cmd-form*) coverage achieved by the test cases. For the basic block and branch criteria, the number represents a percentage; for the

	Block (%)				Branch (%)				Cmd-form (abs.)			
Subject	W_{df}	W_{se}	*Spi.*	DFW	W_{df}	W_{se}	*Spi.*	DFW	W_{df}	W_{se}	*Spi.*	DFW
Bookstore	84.1	87.3	75.6	68.7	55.2	59.7	42.1	34.8	88	737	63	54
Classifieds	81.6	83.7	76.0	66.3	51.3	54.8	41.7	32.3	96	366	99	19
Employee Dir.	83.0	84.6	76.4	69.3	52.9	56.1	42.4	34.9	30	351	22	16
Events	83.5	84.8	76.8	68.2	55.3	57.2	43.9	34.5	37	186	22	16
Average	83.0	85.1	76.2	68.1	53.7	56.9	42.5	34.1	63	410	52	26

Table 7: Test-input generation and coverage.

	# Size of test suite			
Subject	W_{df}	W_{se}	*Spi.*	DFW
Bookstore	258,565	10,634	68,304	33,279
Classifieds	47,352	3,968	7,238	10,732
Employee Dir.	627,820	3,772	46,099	54,887
Events	36,448	1,735	4,145	5,566
Average	242,546	5,027	31,447	26,116

Table 8: Test suite size.

command-form criterion, the number is in absolute terms because DITTO cannot accurately estimate the total number of test requirements for this criterion.

The results of the study show that test suites generated using the information provided by WAM-SE outperformed the test suites generated by the other approaches. For block coverage, WAM-SE outperformed WAM-DF by 3%, SPIDER by 12%, and DFW by 25%. For branch coverage, WAM-SE outperformed WAM-DF by 6%, SPIDER by 34%, and DFW by 67%. For command-form coverage, WAM-SE outperformed WAM-DF by 651%, SPIDER by 788%, and DFW by 1,577%. Overall, the results indicate that, for the subject applications, test suites generated using the information provided by WAM-SE result in consistently higher coverage across the three coverage criteria.

We manually inspected several of the servlets to investigate why branch and block coverage were similar across WAM-DF and WAM-SE, while database command-form coverage exhibited a higher improvement. We found that many branch conditions in the subjects compare hard-coded strings against the value of the IPs or check to ensure that an IP value is numeric. Both approaches are able to identify these types of constraints and generate testcases that would cause them to be covered. However, WAM-SE has the advantage that it can model the constraints of two IPs being equal to each other. For example, in registration pages, a servlet would only proceed if the user entered the same new password twice. The other approaches were not able to model or detect this type of constraint and were thus unable to get high coverage of this type of servlet. We also found that database queries were built using multiple nested `if` statements. Therefore, even though the branch increase provided by WAM-SE was small, the additional branches that were covered resulted in a significant increase in command-form coverage.

Besides coverage, another important metric in test-input generation is the size of the test suites. Table 8 shows for each application, the number of testcases generated by each approach. As the results show the test suites for WAM-SE were significantly smaller than the test suites generated using the other approaches. On average, the WAM-SE test suites are 2% of the size of the WAM-DF test suites, 16% of the size of the SPIDER test suites, and 19% of the size of the DFW test suites.

Overall, the results show that the test suites generated using information provided by WAM-SE are not only significantly smaller, but they consistently result in higher coverage of the subject web applications than test suites generated by other approaches.

7. RELATED WORK

There are many approaches for interface identification. Several of these rely on developer-provided interface specifications: work by Ricca and Tonella uses developer-provided UML models [19], Jian and Liu use a formal specification [15], and Andrews, Offutt, and Alexander [3] use finite state machines. As compared to our approach, the drawback of these approaches is that they are not automated and are susceptible to developer errors. Another group of approaches uses dynamic analysis and web crawling to identify an application's interfaces. An approach by Elbaum and colleagues [6] uses a series of requests to an application to identify its interfaces and infer constraints on the IPs of the interfaces by analyzing responses to the request. A spider by Huang and colleagues [14] uses sophisticated heuristics to more effectively and thoroughly explore a web application. The main limitation of these approaches is that they cannot discover any interfaces that do not correspond to an HTML web form and cannot guarantee that they will see every possible web form generated by the web application. An approach by Deng, Frankl, and Wang [5] uses static analysis to identify interfaces. However, this approach only identifies IPs in an application and does not group them into interfaces.

There have also been many approaches devoted exclusively to testing web applications. Two recent approaches [4, 23] use concolic execution to generate test cases for web applications. These approaches do not explicitly perform interface identification, and in fact, could leverage the information generated by our approach to further augment their test-generation technique. An approach by Emmi, Majumdar, and Sen [9] applies symbolic execution to solve constraints on SQL queries and then uses this information to generate test cases for database-based web applications. Since web application make extensive use of databases, this technique could be used together with our WAM-SE-based test-input generation to achieve additional coverage on parts of the code that are dependent on responses to SQL queries. A final group of techniques [7, 8, 16, 20, 21] use captured user session data and logs to model a web application and generate realistic test cases. These approaches are not able to provide any guarantees of completeness since they are limited to analyzing only parts of the application interaction they have observed.

8. CONCLUSIONS

Interface identification is an important part of many quality assurance techniques for web applications. There are many proposed approaches for identifying interfaces, but these techniques rely on developer-provided specifications, are not complete, or do not generate precise enough information. In this paper, we presented a new approach for interface identification that is based on a specialized form of symbolic execution. The approach uses symbolic values to precisely model input parameters to the web application and identify their domain constraints. We also discussed an extensive empirical evaluation of our approach. The evaluation shows that the efficiency of our technique is comparable to that of the other techniques, achieves higher precision and improves the performance of quality assurance techniques that leverage interface information. In particular, using interface information from our approach as compared to other approaches led to smaller test suites that had higher coverage, more precise invocation verification analysis results, and more vulnerabilities when used to support penetration testing. Overall, we believe that the results of the evaluation indicate that our approach is a useful and feasible technique for interface identification and can improve the performance of quality assurance techniques for web applications.

Acknowledgements

This work was supported in part by NSF awards CCF-0725202 and CCF-0541080 to Georgia Tech.

9. REFERENCES

[1] S. Anand, A. Orso, and M. J. Harrold. Type-dependence Analysis and Program Transformation for Symbolic Execution. In *Proc. TACAS*, pages 117–133, 2007.

[2] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In *Proc. TACAS*, pages 134–138, 2007.

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

[4] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding Bugs in Dynamic Web Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2008.

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

[6] 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.

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

[8] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging User-Session Data to Support Web Application Testing. *IEEE Transactions On Software Engineering*, 31(3):187–202, March 2005.

[9] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[10] W. G. Halfond, S. R. Choudhary, and A. Orso. Penetration Testing with Improved Input Vector Identification. In *Proceedings of the IEEE International Conference on Software Testing*, April 2009.

[11] W. G. Halfond and A. Orso. Command-Form Coverage for Testing Database Applications. In *The IEEE and ACM International Conference on Automated Software Engineering*, pages 69–78, September 2006.

[12] W. G. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the Joint ESEC/SIGSOFT Symposium on the Foundations of Software Engineering*, September 2007.

[13] W. G. Halfond and A. Orso. Automated Identification of Parameter Mismatches in Web Applications. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 2008.

[14] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proc. of the 12th International World Wide Web Conference (WWW 03)*, pages 148–159, May 2003.

[15] 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, November 2002.

[16] 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.

[17] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. TACAS*, pages 553–568, 2003.

[18] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

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

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

[21] P. Tonella and F. Ricca. Dynamic Model Extraction and Statistical Analysis of Web Applications. In *Proceedings of the Fourth International Workshop on Web Site Evolution*, pages 43–52, October 2002.

[22] W. Visser, K. Havelund, G. Brat, S. J. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.

[23] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic Test Input Generation for Web Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2008.