A Tool for Finding Bugs in Web Applications

D.Ramalingam
M.E., / Assistant Professor
Department of Computer Science and Engineering
Saveetha Engineering College, Chennai, Tamilnadu, India.

Abstract- Web script crashes and malformed dynamically generated WebPages are common errors, and they seriously impact the usability of Web applications. Current tools for webpage validation cannot handle the dynamically generated pages that are ubiquitous on today's Internet. We present a dynamic test generation technique for the domain of dynamic Web applications. The technique utilizes both combined concrete and symbolic execution and explicit-state model checking. The technique generates tests automatically, runs the tests capturing logical constraints on inputs, and minimizes the conditions on the inputs to failing tests so that the resulting bug reports are small and useful in finding and fixing the underlying faults. The earlier version of Apollo implements the technique for the PHP programming language. Apollo generates test inputs for a Web application, monitors the application for crashes, and validates that the output conforms to the HTML specification. In this paper, we propose that we are finding bugs in all kind of web applications such as ASP, JSP and PHP with extended concept of dynamic test generation technique. In addition, we can find bugs for different web applications at different computers at a time by this tool with the help of proxy server. This paper presents Apollo's algorithms and implementation and an experimental evaluation that revealed 673 faults in six PHP Web applications.

Keywords – Watermarking, Haar Wavelet, DWT, PSNR

I. INTRODUCTION

DYNAMIC test generation tools, such as DART, Cute, and EXE, generate tests by executing an application on concrete input values, and then creating additional input values by solving symbolic constraints derived from exercised control-flow paths. To date, such approaches have not been practical in the domain of Web applications, which pose special challenges due to the dynamism of the programming languages, the use of implicit input parameters, their use of persistent state, and their complex patterns of user interaction.

This paper extends dynamic test generation to the domain of web applications that dynamically create web (HTML) pages during execution, which are typically presented to the user in a browser. Apollo applies these techniques in the context of the scripting language PHP, one of the most popular languages for server-side Web programming.

According to the Internet research service, Netcraft,1 PHP powered 21 million domains as of April 2007, including large, well-known webs sites such as Wikipedia and Word Press. In addition to dynamic content, modern Web applications may also generate significant application logic, typically in the form of JavaScript code that is executed on the client side. Our techniques are primarily focused on server-side PHP code, although we do some minimal analysis of client-side code to determine how it invokes additional server code through user-interface mechanisms such as forms.

Our goal is to find two kinds of failures in web applications: execution failures that are manifested as crashes or warnings during program execution, and HTML failures that occur when the application generates malformed HTML. Execution failures may occur, for example, when a web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output contains an error message and execution of the application may be halted, depending on the severity of the failure.

HTML failures occur when output is generated that is not syntactically well-formed HTML (e.g., when an opening tag is not accompanied by a matching closing tag). HTML failures are generally not as important as execution failures because Web browsers are designed to tolerate some degree of malformedness in HTML, but they are undesirable for several reasons. First and most serious is that browsers' attempts to compensate for malformed web pages may lead to crashes and security vulnerabilities. Second, standard HTML renders faster. Third, malformed HTML is less portable across browsers and is vulnerable to breaking or looking strange when displayed by browser

versions on which it is not tested. Fourth, a browser might succeed in displaying only part of a malformed webpage, while silently discarding important information. Fifth, search engines may have trouble indexing malformed pages.

The existing systems that it generates test to the domain of web application for finding bugs by a single computer at a time. The system is also finding bugs only for PHP web applications. This system's goal is to find two kinds of failures in web applications: execution failures that are manifested as crashes or warnings during program execution, and HTML failures that occur when the application generates malformed HTML. Execution failures may occur, for example, when a web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output contains an error message and execution of the application may be halted, depending on the severity of the failure.

II. PROPOSED ALGORITHM

A. Apollo algorithm –

Fig. 1 shows pseudo code for the algorithm, which extends the algorithm in Fig. 2 with explicit-state model checking to handle the complexity of simulating user inputs. The algorithm tracks the state of the environment, and automatically discovers additional configurations based on an analysis of the output for available user options. In particular, the algorithm 1) tracks changes to the state of the environment (i.e., session state, cookies, and the database) and 2) performs an "on-the-fly" analysis of the output produced by the program to determine what user options it contains, with their associated PHP scripts.

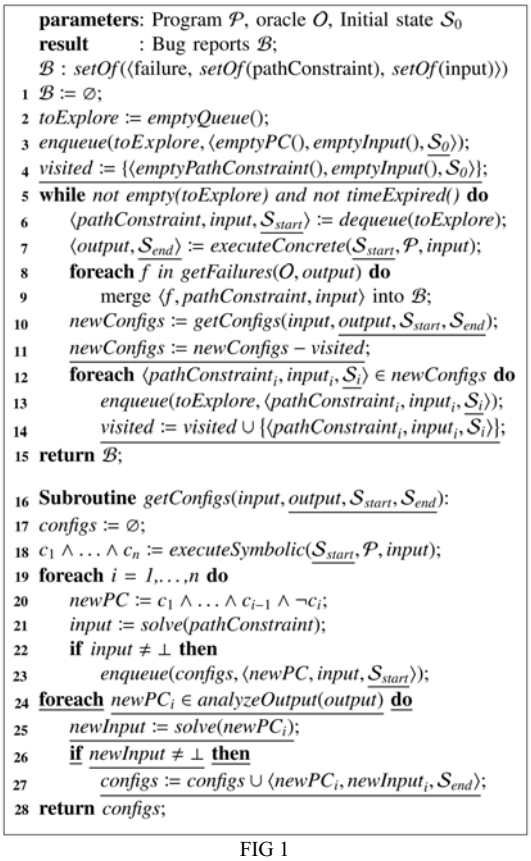
Simplified algorithm that was previously shown in Fig. 2.

1 A configuration contains an explicit state of the environment (before the only state that was used was the initial state S0) in addition to the path constraint and the input (line 3).

2 Before the program is executed, the algorithm (method execute Concrete) will restore the environment to the state given in the configuration (line 7) and will return the new

3 When the getConfigs subroutine is executed to find new configurations, it analyzes the output to find possible transitions from the new environment state (lines 24-27). The analyze Output function extracts parameter names and possible values for each parameter, and represents the extracted information as a path constraint. For simplicity, the algorithm uses only one entry point into the program. However, in practice, there may be several entry points into the program (e.g., it is possible to call different PHP scripts). The analyze Output function discovers these entry points in addition to the path constraints. In practice, each transition is expressed as a pair of a path constraint and an entry point.

4 The algorithm uses a set of configurations that are already in the queue (line 14) and it performs state matching in order to only explore new configurations (line 11).

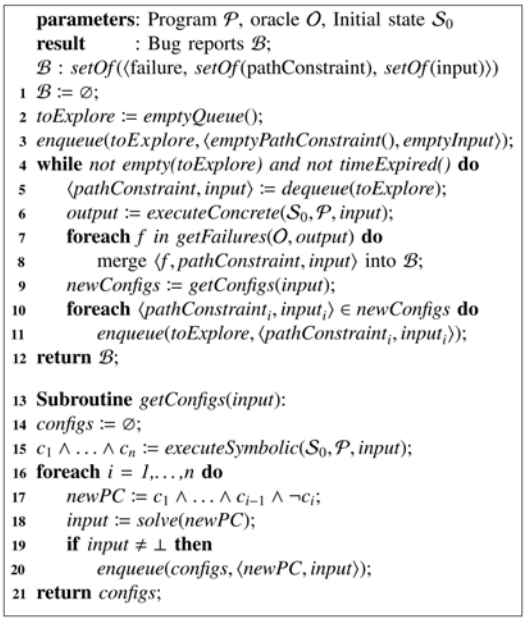


parameters: Program $\mathcal{P}$, oracle O, Initial state $\mathcal{S}_0$
result : Bug reports $\mathcal{B}$;
$\mathcal{B}$: *setOf*(⟨failure, *setOf*(pathConstraint), *setOf*(input)⟩)

1 $\mathcal{B} := \emptyset$;
2 *toExplore* := *emptyQueue*();
3 *enqueue*(*toExplore*, ⟨*emptyPC*(), *emptyInput*(), $\mathcal{S}_0$⟩);
4 *visited* := {⟨*emptyPathConstraint*(), *emptyInput*(), $\mathcal{S}_0$⟩};
5 **while** *not empty(toExplore) and not timeExpired()* **do**
6 ⟨*pathConstraint*, *input*, $\mathcal{S}_{start}$⟩ := *dequeue*(*toExplore*);
7 ⟨*output*, $\mathcal{S}_{end}$⟩ := *executeConcrete*($\mathcal{S}_{start}$, $\mathcal{P}$, *input*);
8 **foreach** *f in getFailures*(*O*, *output*) **do**
9 merge ⟨*f*, *pathConstraint*, *input*⟩ into $\mathcal{B}$;
10 *newConfigs* := *getConfigs*(*input*, *output*, $\mathcal{S}_{start}$, $\mathcal{S}_{end}$);
11 *newConfigs* := *newConfigs* − *visited*;
12 **foreach** ⟨*pathConstraint*$_i$, *input*$_i$, $\mathcal{S}_i$⟩ ∈ *newConfigs* **do**
13 *enqueue*(*toExplore*, ⟨*pathConstraint*$_i$, *input*$_i$, $\mathcal{S}_i$⟩);
14 *visited* := *visited* ∪ {⟨*pathConstraint*$_i$, *input*$_i$, $\mathcal{S}_i$⟩};
15 **return** $\mathcal{B}$;

16 **Subroutine** *getConfigs*(*input*, *output*, $\mathcal{S}_{start}$, $\mathcal{S}_{end}$):
17 *configs* := ∅;
18 $c_1 \wedge \ldots \wedge c_n$:= *executeSymbolic*($\mathcal{S}_{start}$, $\mathcal{P}$, *input*);
19 **foreach** $i = 1,\ldots,n$ **do**
20 *newPC* := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
21 *input* := *solve*(*pathConstraint*);
22 **if** *input* ≠ ⊥ **then**
23 *enqueue*(*configs*, ⟨*newPC*, *input*, $\mathcal{S}_{start}$⟩);
24 **foreach** *newPC*$_i$ ∈ *analyzeOutput*(*output*) **do**
25 *newInput* := *solve*(*newPC*$_i$);
26 **if** *newInput* ≠ ⊥ **then**
27 *configs* := *configs* ∪ ⟨*newPC*$_i$, *newInput*$_i$, $\mathcal{S}_{end}$⟩;
28 **return** *configs*;

FIG 1



parameters: Program $\mathcal{P}$, oracle O, Initial state $\mathcal{S}_0$
result : Bug reports $\mathcal{B}$;
$\mathcal{B}$: *setOf*(⟨failure, *setOf*(pathConstraint), *setOf*(input)⟩)

1 $\mathcal{B} := \emptyset$;
2 *toExplore* := *emptyQueue*();
3 *enqueue*(*toExplore*, ⟨*emptyPathConstraint*(), *emptyInput*⟩);
4 **while** *not empty(toExplore) and not timeExpired()* **do**
5 ⟨*pathConstraint*, *input*⟩ := *dequeue*(*toExplore*);
6 *output* := *executeConcrete*($\mathcal{S}_0$, $\mathcal{P}$, *input*);
7 **foreach** *f in getFailures*(*O*, *output*) **do**
8 merge ⟨*f*, *pathConstraint*, *input*⟩ into $\mathcal{B}$;
9 *newConfigs* := *getConfigs*(*input*);
10 **foreach** ⟨*pathConstraint*$_i$, *input*$_i$⟩ ∈ *newConfigs* **do**
11 *enqueue*(*toExplore*, ⟨*pathConstraint*$_i$, *input*$_i$⟩);
12 **return** $\mathcal{B}$;

13 **Subroutine** *getConfigs*(*input*):
14 *configs* := ∅;
15 $c_1 \wedge \ldots \wedge c_n$:= *executeSymbolic*($\mathcal{S}_0$, $\mathcal{P}$, *input*);
16 **foreach** $i = 1,\ldots,n$ **do**
17 *newPC* := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
18 *input* := *solve*(*newPC*);
19 **if** *input* ≠ ⊥ **then**
20 *enqueue*(*configs*, ⟨*newPC*, *input*⟩);
21 **return** *configs*;

FIG 2

The proposed system is mainly designed to extend the dynamic test generation technique for all domains of web applications (such as JSP, ASP, and PHP). This technique implemented through Apollo algorithm. To implement Apollo algorithm, we are going to develop a tool called Apollo. The tool for which can test multiple web applications at multiple computers at a time by sharing the proxy through server.

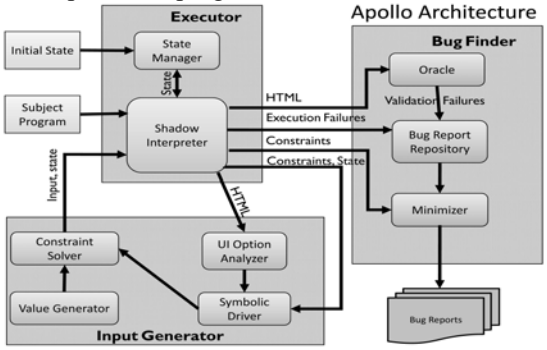
The failure detection algorithm returns bug reports. Each bug report contains a set of path constraints, and a set of inputs exposing the failure. Previous dynamic test generation tools presented the whole input (i.e., many input Parameter; value pairs) to the user without an indication of the subset of the input responsible for the failure. As a postmortem phase, our minimization algorithm attempts to find a shorter path constraint for a given bug report. This eliminates irrelevant constraints, and a solution for a shorter path constraint is often a smaller input. The above problem can completely be eliminated by this exposed system.

SYSTEM ARCHITECTURE

We created a tool called Apollo that implements our technique for PHP. Apollo consists of three major components, Executor, Bug Finder, and Input Generator illustrated in the below Fig. This section first provides a high-level overview of the components and then discusses the pragmatics of the implementation. The inputs to Apollo are the program under test and an initial value for the environment. The environment of a PHP program consists of the database, cookies, and stored session information. The initial environment usually consists of a database populated with some values, and user supplied information about username/password pairs to be used for database authentication.

THE EXECUTOR

It is responsible for executing a PHP script with a given input in a given state. The executor contains two subcomponents: The Shadow Interpreter is a PHP interpreter that we have modified to propagate and record path constraints and positional information associated with output. This positional information is used to determine which failures are likely to be symptoms of the same fault. The State Manager restores the given state of the environment (database, session, and cookies) before the execution and stores the new environment after the execution. The Bug Finder uses an oracle to find HTML failures, stores all bug reports, and finds the minimal conditions on the input parameters for each bug report. The Bug Finder has the following subcomponents: . The Oracle finds HTML failures in the output of the program.



Clearly, this potentially leaves many uses of input unaccounted for. However, our results suggest that this is sufficient to capture the bulk of how PHP code uses inputs in practice. Values derived directly from input are those read from one of the special arrays POST, GET, and REQUEST, which store parameters supplied to the PHP program. For example, executing the statement $x ¼ $ GET½‘‘param1}_ results in associating the value read from the global parameter param1 and bound to parameter x with the symbolic variable param Values maintain their associations through the operations mentioned above; that is, the symbolic variables for the new values receive the same value as the source value had. Importantly, during program execution, the concrete values remain, and the shadow interpreter does not influence execution.

BUG FINDER

The bug finder is in charge of transforming the results of the executed inputs into bug reports. Below is a detailed description of the components of the bug finder. Bug report repository. This repository stores the bug reports found in all executions. Each time a failure is detected, the corresponding bug report (if the same failure was discovered before) is updated with the path constraint and the configuration inducing the failure.

INPUT GENERATOR

UI option analyzer: Many PHP Web applications create interactive HTML pages that contain user-interface elements such as buttons and menus that allow the user interaction needed to execute further parts of the application. In such cases, pressing the button may result in the execution of additional PHP source files. There are two challenges involved in dealing with such interactive applications.

```
<?php
   echo "<h2>WebChess ".$Version." Login"</h2>;
?>
<form method="post" action="mainmenu.php">
<p>
   Nick: <input name="txtNick" type="text" size="15" default="admin"/>
   <br />
   Password: <input name="pwdPassword" type="password" size="15"/>
</p>
<p>
   <input name="login" value="login" type="submit"/>
   <input name="newAccount" value="New Account"
     type="button" onClick="window.open('newuser.php', '_self')"/>
</p>
</form>
```

Constraint solver**:** The interpreter implements a lightweight symbolic execution, in which the only constraints are equality and inequality with constants. Apollo transforms path constraints into integer constraints in a straight forward way, and uses choco15 to solve them. This approach still allows us to handle values of the standard types (integer, string), and is straight forward because the only constraints are equality and inequality. In cases where parameters are unconstrained, Apollo randomly chose values from a predefined list of constants. While limiting to the basic types number and string and only comparisons may seem very restrictive, note that all input comes to PHP as strings; furthermore, in our experience, the bulk of use of input values consists of the kinds of simple operations that are captured by our tracing and the kinds of simple comparisons captured here. Our coverage results suggest this is valid for a significant range of PHP applications.

III. Experiment and Result

We created a tool, Apollo that implements the analysis. We evaluated Apollo on six open-source PHP web applications. Apollo's test generation strategy achieves over 50 percent line coverage. Apollo found a total of 673 faults in these applications: 72 execution problems and 601 cases of malformed HTML. Finally, Apollo also minimizes the size of failure-inducing inputs: The minimized inputs are up to 5:3_ smaller than the unminimized ones.

IV.CONCLUSION

We have presented a technique for finding faults in PHP Web applications that is based on combined concrete and symbolic execution. The work is novel in several respects. First, the technique not only detects runtime errors but also uses an HTML validator as an oracle to determine situations where malformed HTML is created. Second, we address a number of PHP-specific issues, such as the simulation of interactive user input that occurs when user-interface elements on generated HTML pages are activated, resulting in the execution of additional PHP scripts. Third, we perform an automated analysis to minimize the size of failure-inducing inputs.

REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann, "Demand-Driven Compositional Symbolic Execution," Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems, pp. 367-381, 2008.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, "Finding Bugs in Dynamic Web Applications," Proc. Int'l Symp. Software Testing and Analysis, pp. 261-272, 2008.

[3] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically Testing Dynamic Web Sites," Proc. Int'l Conf. World Wide Web, 2002.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation," Proc. 16th USENIX Security Symp., 2007.

[5] C. Cadar, D. Dunbar, and D.R. Engler, "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Proc. USENIX Symp. Operating Systems Design and Implementation, pp. 209-224, 2008.

[6] C. Cadar and D.R. Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself," Proc. Int'l SPIN Workshop Model Checking of Software, pp. 2-23, 2005.

[7] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically Generating Inputs of Death," Proc. Conf. Computer and Comm. Security, pp. 322-335, 2006.

[8] J. Clause and A. Orso, "Penumbra: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting," Proc. Int'l Symp. Software Testing and Analysis, 2009.

[9] H. Cleve and A. Zeller, "Locating Causes of Program Failures," Proc. Int'l Conf. Software Eng., pp. 342-351, 2005.