Improving penetration testing through static and dynamic analysis

William G. J. Halfond[1, *, †], Shauvik Roy Choudhary[2] and Alessandro Orso[2]

[1]*Computer Science Department, University of Southern California, 941 Bloom Walk,*
Los Angeles, CA 90089, U.S.A.
[2]*College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332, U.S.A.*

SUMMARY

Penetration testing is widely used to help ensure the security of web applications. Using penetration testing, testers discover vulnerabilities by simulating attacks on a target web application. To do this efficiently, testers rely on automated techniques that gather input vector information about the target web application and analyze the application's responses to determine whether an attack was successful. Techniques for performing these steps are often incomplete, which can leave parts of the web application untested and vulnerabilities undiscovered. This paper proposes a new approach to penetration testing that addresses the limitations of current techniques. The approach incorporates two recently developed analysis techniques to improve input vector identification and detect when attacks have been successful against a web application. This paper compares the proposed approach against two popular penetration testing tools for a suite of web applications with known and unknown vulnerabilities. The evaluation results show that the proposed approach performs a more thorough penetration testing and leads to the discovery of more vulnerabilities than both the tools. Copyright © 2011 John Wiley & Sons, Ltd.

1. INTRODUCTION

Web applications are widely used to provide functionality that allows companies to build and maintain relationships with their customers. The information stored by web applications is often confidential and, if obtained by malicious attackers, its exposure could result in substantial losses for both consumers and companies. Recognizing the rising cost of successful attacks, software engineers have worked to improve their processes to minimize the introduction of vulnerabilities. In spite of these improvements, vulnerabilities continue to occur because of the complexity of the web applications and their deployment configurations. The continued prevalence of vulnerabilities has increased the importance of techniques that can identify vulnerabilities in deployed web applications.

One such technique, penetration testing, identifies vulnerabilities in web applications by simulating attacks by a malicious user. Although penetration testing cannot guarantee that all vulnerabilities will be identified in an application, it is popular among developers for several reasons: (i) it generally has a low rate of false vulnerability reports since it discovers vulnerabilities by exploiting them; (ii) it tests applications in context, which allows for the discovery of vulnerabilities that arise

---

*Correspondence to: William G. J. Halfond, Computer Science Department, University of Southern California, 941 Bloom Walk, Los Angeles, CA 90089, U.S.A.
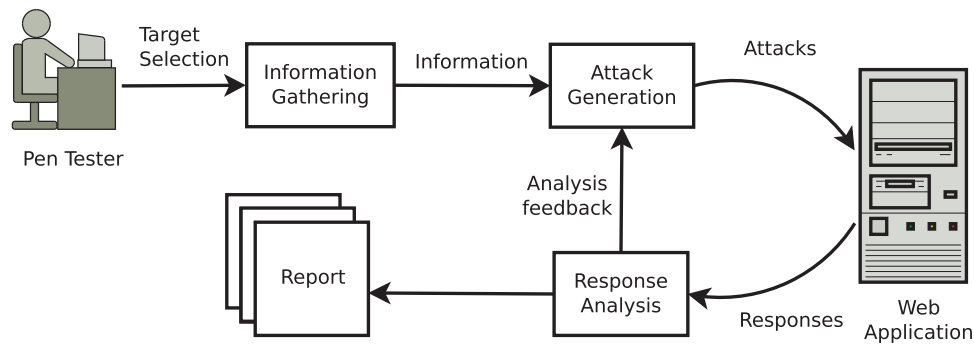†E-mail: halfond@usc.edu

Figure 1. The penetration testing process.

due to the actual deployment environment of the web application; and (iii) it provides concrete inputs for each vulnerability report that can guide the developers in correcting the code.

The widespread usage of penetration testing has led many government agencies and trade groups, such as the Communications and Electronic Security Group in the U.K., OWASP[‡], and OSSTMM[§], to accredit penetration testers and establish standardized 'best practices' for penetration testing. Although individual penetration testers perform a wide variety of tasks, the general process can be divided into three phases: information gathering, attack generation, and response analysis. Figure 1 shows a high-level overview of these three phases. In the first phase, *information gathering*, penetration testers select a target web application and obtain information about it using various techniques, such as automated scanning, web crawling, and social engineering. The results of this phase allow penetration testers to perform the second phase, *attack generation*, which is the development of attacks on the target application. Often this phase can be automated by customizing well-known attacks or by using automated attack scripts. Once the attacks have been executed, penetration testers perform *response analysis*—they analyze the application's responses to determine whether the attacks were successful and prepare a final report about the discovered vulnerabilities.

During information gathering, the identification of an application's *input vectors* (IVs)—points in an application where an attack may be introduced, such as user-input fields and cookie fields—is of particular importance. Better information about an application's IVs generally leads to more thorough penetration testing of the application. Currently, it is common for penetration testers to use automated web crawlers to identify the IVs of a web application. A web crawler visits the HTML pages generated by a web application and analyzes each page to identify potential IVs. The main limitation of this approach is that it is incomplete because web crawlers are typically unable to visit all of the pages of a web application or must provide certain values to the web application in order to cause additional HTML pages to be shown. Although penetration testers can make use of the information discovered by web crawlers, the incompleteness of such information results in a potentially large number of vulnerable IVs remaining undiscovered. Another challenging aspect of penetration testing is determining whether an attack is successful. This task is complex because a successful attack often produces no observable behavior (i.e. it may produce a side effect that is not readily visible in the HTML page produced by the web application) and requires manual, time-consuming analysis to be identified. Existing approaches to automated response analysis tend to suffer from imprecision because they are based on simple heuristics. The authors' experience with such automated analyses indicates that they work well for simple applications, but are fairly ineffective when used on real applications.

In this paper the authors propose a new approach to penetration testing that improves both the information gathering and the response analysis phases. One of the key insights of this approach

---

[‡]http://www.owasp.org/.
[§]http://www.osstmm.org/.

is that many of the limitations of the previous approaches can be addressed by assuming that penetration testers have access to the source code or executable of the web application. This assumption is realistic in the context of in-house penetration testing and is consistent with the best practices defined by both OWASP‡ and OSSTMM§, which assume that potential adversaries have access to one or more versions of an application's source code. The proposed penetration testing approach leverages several newly developed analyses that make use of the web application source code. To improve the information gathering phase, the approach builds on a static analysis technique for discovering IVs that was developed by two of the authors in previous work [1]. The proposed approach also improves the response analysis phase by incorporating the use of precise dynamic analyses to determine when an attack has been successful. The dynamic analyses allow the approach to perform fully automated detection of successful attacks.

The authors implemented a prototype tool, SDAPT (Static and Dynamic Analysis based Penetration Testing), to conduct an extensive empirical evaluation of the proposed approach. In this evaluation, the authors used SDAPT to perform penetration testing on nine web applications, and SDAPT's performance was compared with that of two state-of-the-art penetration testing tools. The empirical results show that the approach was able to (i) exercise the subject applications more thoroughly and (ii) discover a considerably higher number of vulnerabilities than the traditional penetration testing approaches.

The contributions of this paper are

- An approach for penetration testing based on improved input vector identification and automated response analysis.
- An implementation of the approach in a prototype tool that targets SQL Injection and Cross Site Scripting vulnerabilities.
- Four empirical studies that assess the practicality, thoroughness, and effectiveness of the approach on nine web applications.

2. MOTIVATING EXAMPLE

In this section the paper introduces a motivating example to illustrate some of the limitations of traditional penetration testing. The paper also uses this example in Section 3 to illustrate the approach.

Figures 2 and 3 show two servlets that are part of a Java-based web application. (A *servlet* is the basic implementation unit in the Java Enterprise Edition (JEE) framework for developing web applications.) Both servlets contain vulnerabilities to well-known web application attacks. Servlet Register.jsp (Figure 2) contains several vulnerabilities to SQL injection attacks (SQLIA), which allow an attacker to execute commands on the database underlying the web application. Servlet DisplayUsers.jsp (Figure 3) contains several vulnerabilities to Cross Site Scripting (XSS) attacks, which allow an attacker to insert malicious scripts and tags that will then execute in the victim's web browser. For each servlet, this paper explains its functionality, the vulnerability it contains, and the reason that traditional information gathering techniques would not lead to a discovery of the vulnerability.

Servlet Register.jsp allows a user to register a new login and password by filling out a series of web forms. Execution begins in function `service`, which is the standard entry method for all servlets. At line 2, the servlet accesses an input parameter named `userAction`. In general, parameters are passed from an end-user to a servlet as name-value pairs. To access these parameters, a servlet calls a *parameter function*, passes to it the name of the desired parameter, and receives its corresponding value. The initial visit of a user triggers an invocation of the servlet without any parameters, therefore the conditions at lines 3 and 14 are false and the execution continues at line 21. Function `displayCreateLoginForm` generates a web form that is sent to the user browser and that provides text input fields for the user to enter their desired login name and password. In this form, the function also sets a hidden input field called `userAction` to the value 'createLogin'.

```
1. public void service(HttpServletRequest req) {
2.    String action = req.getParameter("userAction");
3.    if (action.equals("createLogin")) {
4.       String password = req.getParameter("password");
5.       String loginName = req.getParameter("login");
6.       if (isAlphaNumeric(password)) {
7.          Connection conn = new Connection("mysql://localDB");
8.          Statement stmt = conn.getStatement();
9.          stmt.execute("insert into UserTable "
                          + "(login, password) values ('"
                          + loginName + "', '"
                          + password + "')");
10.         displayAddressForm();
11.      } else {
12.         displayErrorPage("Bad password.");
13.      }
14.   } else if (action.equals("provideAddress")) {
15.      String loginName = req.getParameter("login");
16.      String address = req.getParameter("address");
17.      Connection conn = new Connection("mysql://localDB");
18.      Statement stmt = conn.getStatement();
19.      stmt.execute("update UserTable set"
                       + " address = '" + address + "'"
                       + " where loginName = '"
                       + loginName + "'");
20.   } else{
21.      displayCreateLoginForm();
22.   }
23. }
```

Figure 2. Example servlet: Register.jsp.

```
1. public void service(HttpServletRequest req) {
2.    out.println("<html><body>");
3.    out.println("<h1>List of Users</h1>");
4.    Connection conn = new Connection("mysql://localDB");
5.    Statement stmt = conn.getStatement();
6.    ResultSet rs = stmt.executeQuery("select * from UserTable" );
7.    while (rs.next()) {
8.       String name = rs.getString("loginName");
9.       String address = rs.getString("address");
10.      out.println("Name: " + name + "<br>");
11.      out.println("Address: " + address + "<br>");
12.      out.println("<hr>");
13.   }
14.   out.println("</body></html>");
15. }
```

Figure 3. Example servlet: DisplayUsers.jsp.

When the user submits the web form, the browser bundles the three input fields (`login`, `password`, and `userAction`) as name-value pairs and sends them to the servlet. On this second execution of the servlet, the condition at line 3 is true, and the servlet retrieves `password` and `login` (lines 4–5). If `password` is not alphanumeric, an error message is returned to the end user (line 12). Otherwise, the servlet generates a query to the database that creates the user account (lines 7–9). The servlet then calls function `displayAddressForm`, which generates another web form that allows the user to enter their address (line 10). This function also sets two hidden input fields in the form: `userAction` is set to 'provideAddress' and `login` is set to the user-chosen login. When the user submits this form, the condition at line 14 is true, so the

servlet retrieves `address` and `login` (lines 15–16) and updates the entry in the database with the supplied information (lines 17–19). At this point, the registration is done.

Lines 9 and 19 of Register.jsp are vulnerable to SQLIAs. To exploit the SQLIA vulnerability at line 9, an attacker could enter the string `''name', 'secret'); drop table UserTable -- '` as their chosen login. This causes the following SQL query to be built and executed on the database: `'insert into UserTable (login, password) values ('name', 'secret'); drop table UserTable -- ','')'`. Besides creating an account with `login` as 'name' and `password` as 'secret', this query would execute a drop command that would delete all of the user information in the user table. (Note that '`--`' is the SQL comment operator, so the extra parenthesis and quotes after the query would be ignored.) Line 19 could be exploited in a similar manner by injecting the SQLIA into either `login` or `address`, which are read at lines 15 and 16. In general, Register.jsp is vulnerable to a wide range of SQLIAs [2].

Servlet DisplayUsers.jsp, which is shown in Figure 3, allows a registered user to see a listing of other users who have registered via the web application. When a user visits the servlet, the servlet starts by outputting opening HTML tags (lines 2 and 3). Then the servlet creates and executes a database query, which retrieves all user entries from the database (lines 4–6). Each database record is accessed, and the user's login name and address are formatted using HTML tags (lines 7–13). Finally, the servlet outputs the closing HTML tags (line 14), and the generated HTML page is displayed in the user's browser.

Lines 10 and 11 of DisplayUsers.jsp are vulnerable to XSS attacks. To exploit these vulnerabilities, the attacker could enter `<SCRIPT SRC=http://hackerworld.org/xss.js> </SCRIPT>` as their chosen login or address (lines 5 and 16) in the web forms displayed by Register.jsp, then complete the rest of the registration as normal. When a user visits DisplayUsers.jsp, each user's login and address is displayed in an HTML page. When the attacker's login or address is displayed, the `<SCRIPT>` tag will be executed by the user's browser, and the script could perform various malicious activities, such as access the user's cookies, steal session information, or exploit a vulnerability in the browser.

For both servlets, it would be unlikely that automated penetration testing based on traditional information gathering techniques would lead to the discovery of the described vulnerabilities. When a web crawler visits Register.jsp, it would be able to discover the web form generated by `displayCreateLoginForm` because this is the default page created by the servlet. From this page, the web crawler would then be able to identify the names of three IVs for this application: `userAction`, `login`, and `password`. At this point, most web crawlers would generate random values for the identified IVs in an attempt to access subsequent pages that may contain additional information. Although the alphanumeric constraint in this example could be easily guessed by a sophisticated web crawler or via manual intervention, real-world web applications typically contain more complex constraints that are not as easy to guess or for which there are no visible clues present in the generated HTML pages. The presence of these unguessable constraints means that there will be pages containing useful information that are never accessed by the web crawler. Sections 4.3 and 4.5 show that this limitation negatively affects the quality of penetration testing based on web crawling for real-world web applications. In the example, this has several consequences for the quality of the penetration testing: (i) the tester will not be able to inject an SQLIA into line 9 of Register.jsp; (ii) the additional input vector information provided by the form that is displayed by the `displayAddressForm` will not be seen by the web crawler; and (iii) the tester will not be able to seed the `<SCRIPT>` tags (at either lines 5 or 16 of Register.jsp) that then trigger the XSS attacks in DisplayUsers.jsp.

3. APPROACH AND IMPLEMENTATION

The goal of the proposed approach is to improve penetration testing of web applications by focusing on two areas where the current techniques are limited: identifying the IVs of a web application and detecting the outcome of an attempted attack. To do this, the paper proposes a new approach

to penetration testing that leverages two recently developed analysis techniques. The first is a static analysis technique for identifying potential IVs, and the second is a dynamic analysis to automate response analysis. In the information gathering phase, the approach leverages the static analysis technique to analyze the code of the application and identify IV names, groupings (i.e. which sets of IVs are accessed together by a servlet), and domain information (i.e. IVs' relevant values and type constraints). In the attack generation phase, the approach targets the identified IVs and uses the domain and grouping information to generate the relevant values for the penetration test cases. Finally, in the response analysis phase, the approach uses the dynamic analysis technique to assess in an automated way whether an attack was successful.

In the remainder of this section, the paper explains the details of each of the three phases of the proposed penetration testing approach. Where applicable, the paper illustrates the advantages of the proposed approach using the example from Section 2. The paper also presents, in Section 3.4, the details of the approach's implementation.

3.1. Information gathering

During the information gathering phase, testers analyze the target application to identify information that may be useful for generating attacks. In particular, testers are interested in gathering information about the application's IVs—their names, groupings, and domain information. Although the current best practices recommend that penetration testers assume that attackers have access to one or more versions of the source code of the application, information gathering for penetration testing is still performed in a mostly black-box fashion. In particular, one of the most popular and widely used techniques for information gathering, web crawling, is a purely black-box technique and does not assume any access to the source code. As discussed in Section 2, this generally leads to incomplete information gathering.

In contrast to web crawling, the proposed information gathering technique is complete. The technique performs a conservative static analysis of the web application's code to identify IV-related information that can be used to generate test inputs for the web application. To perform the analysis, the approach leverages a technique, WAM, which was developed in the previous work by two of the authors [1]. WAM analyzes the code of a web application and computes a conservative approximation of all of the IVs in the application, together with their groupings and domain information. The conservative nature of the analysis may lead to the identification of spurious IV-related information along infeasible paths in the code. However, for the proposed approach, this only leads to the generation of additional test inputs and does not reduce the ability of the approach to find vulnerabilities. There are three basic phases to the WAM analysis: (i) identify the names of the IVs, (ii) compute domain information for each IV, and (iii) group IVs into distinct interfaces. The algorithms that implement each phase are not reproduced here, but are contained in the WAM paper [1]; the following sections summarize each of the phases in more detail.

3.1.1. Identify IV names.
In its first phase, WAM identifies the names of the IVs of the web application. To do this, WAM analyzes the string argument of each call to a parameter function and attempts to determine its concrete value (i.e., the name of the IV accessed through that call). In most cases, it is sufficient for the analysis to follow the definition and use chains of the argument backwards through the control-flow of the application until an assignment to a concrete value is found. In some cases, a definition is provided either by a formal parameter to the method containing the parameter function call or by a string expression. In the first case, a placeholder is used, and the actual name of the IV is identified when a call site to the parameter function's enclosing method provides an argument. In the second case, a string analysis is used to approximate the value of the string expression. In both the general and special cases, the computation of the IV names is a conservative over-approximation of the possible string values of the variables that define the string name.

The string analysis used in this phase is based on the approach proposed by Christensen *et al.* [3]. In most cases, the runtime of this approach is linear with respect to the number of nodes in the web application's inter-procedural control-flow graph. However, in the worst case the runtime cost

can be doubly exponential. Although this worst case performance is high, the results in Section 4 indicate that, in practice, the use of this approach does not add a prohibitive time cost to the WAM analysis.

To illustrate the identification of IV names, consider the parameter function calls at lines 2, 4, 5, 15, and 16 of Register.jsp (Figure 2). For each of these calls, identifying the concrete value of the arguments is straightforward since they are string constants that are defined within the same method scope. For instance, the name of the IV accessed at line 2 is `userAction`, and that of the IV accessed at line 16 is `address`.

3.1.2. Compute domain information. In its second phase, WAM computes domain information for all of the IVs in the application. The general intuition behind this phase is that the domain of each IV is implied by the actions performed on its value. For example, a call to the function `Integer.parseInt(value)`, where `value` contains the value of an IV, implies that the IV is expected to be of type integer. To compute the domain information, WAM identifies all uses of the value of each IV in certain domain constraining operations, such as typecasts and value equality comparisons, and uses these to infer the domain of the IV. (Note that, at this time, WAM does not account for more complex domain constraints, such as regular expressions or relational operators.) The WAM analysis identifies usages of the IV value by first identifying the initial definition of the value of the IV, which is the return value of a call to a parameter function, and then following definition-use chains that involve that value.

To illustrate with an example, consider the IV accessed at line 2 of the example servlet in Figure 2, whose value is assigned to variable `action`. By analyzing the definition-use chains that involve variable `action`, WAM identifies that `action` is used at lines 3 and 14. These uses imply that 'createLogin' and 'provideAddress' are relevant values in the domain of `action`. Similarly, for the IV accessed at line 4, WAM can infer from the use at line 6 that the domain of the IV should be alphanumeric.

The WAM analysis computes a conservative over-approximation of the domain of each IV. The reason for this over-approximation is that the analysis assumes that all paths originating from the return value of the parameter function are feasible. The over-approximation can result in inefficiencies since additional domain information leads to the generation of additional test inputs, some of which correspond to domain constraints along infeasible paths. To reduce this inefficiency, the approach incorporates heuristics that ignore domain information from certain operations used for error checking inputs, such as comparing against empty strings or null values. In practice, this is a safe heuristic since domain information from these operations typically leads to test inputs that cause properly checked errors and are unlikely to result in a successful attack.

3.1.3. Groups IVs into interfaces. In its third phase, WAM groups the identified IV names and domain information into interfaces. To do this, WAM identifies sets of IVs that are accessed along the same path of execution and groups them, along with the domain information identified in the second phase, into interfaces. The WAM analysis is based on iterative data-flow analysis techniques. The use of these techniques results in a conservative over-approximation of the interfaces of a web application. That is, even IVs accessed along an infeasible path are identified as an interface. This happens because the iterative data-flow analysis does not attempt to identify infeasible paths. However, the over-approximation only results in the creation of additional test inputs and does not reduce the effectiveness of the technique for finding vulnerabilities.

Table I shows the interface groupings produced by running WAM on the example web application. The first column (*Path*) is simply a reference number that is used in the subsequent explanation. The second column (*Branches*) shows the branches that define the path of execution. The third column (*IV Names*) shows the set of names of the IVs accessed along that path, and the fourth column (*Domain Information*) shows the identified domain information. Note that, as explained in Section 3.1.2, the domain information is not path-sensitive and represents an over-approximation.

There is a row in Table I for every path through Register.jsp (Figure 2) that accesses an IV. As can be seen in the table, there are four such paths. Paths 1 and 2 both take the true branch at

Table I. Interface information for the example web application.

Path	Branches	IV Names	Domain Information
1	3T, 6F	userAction, password, login	userAction ∈ {createLogin, provideAddress, *} ∧ isAlphaNumeric(password)
2	3T, 6T	userAction, password, login	userAction ∈ {createLogin, provideAddress, *} ∧ isAlphaNumeric(password)
3	3F, 14T	userAction, login, address	userAction ∈ {createLogin, provideAddress, *}
4	3F, 14F	userAction	userAction ∈ {createLogin, provideAddress, *}

line 3 (3T). The first path then takes the false branch at line 6 (6F), and the second path takes the true branch at line 6 (6T). Both of these paths access IVs at lines 2, 4, and 5. Since taking the true or false branches at line 6 does not change the set of accessed IV names, the interface accessed along both of these paths is the same. Paths 3 and 4 take the false branch at line 3 (3F). Path 3 then takes the true branch at line 14 (14T), which means that the IVs accessed on the third path are the ones at lines 2, 15, and 16. Finally, Path 4 takes the false branch at line 14 (14F), so the only IV it accesses is at line 2. In summary, for the four paths that access IVs, there are three distinct sets of IV names, which represent the interfaces of the example web application.

3.2. Attack generation

During the attack generation phase, the gathered information is used to create attacks on the target application. The typical approach in penetration testing is to target each IV using a set of attack heuristics while supplying realistic and 'harmless' input values for the other IVs that must be defined as part of a complete request. The identification of suitable realistic input values for these IVs is a crucial part of this process. Currently there are several approaches in use to determine these values: ask the developers, use values supplied as defaults in the web pages examined during web crawling, or generate random strings. Each of these approaches has its strengths and weaknesses. Asking the developers for each IV is generally accurate, but time-consuming; collecting defaults values in the web pages is inexpensive, but these may not be present or may not correspond to relevant values; and random strings are inexpensive to generate, but give no assurance of covering the relevant values. In particular, the last two approaches are unlikely to provide values that will satisfy the application's constraints and lead to the identification of vulnerabilities. An example of this was discussed in Section 2, and the evaluation in Section 4 shows that this occurs frequently for information gathering based on web crawling.

The proposed approach addresses this problem by using the domain and grouping information identified by the WAM analysis to provide relevant values for all IVs that are not being injected with potential attacks. Note that the approach does not create new attack heuristics; these are supplied by penetration testers based on their own expertise and known attacks. Instead, WAM's domain information provides a way to generate more realistic and relevant values for the non-targeted IVs in the penetration test cases. The approach generates these realistic values by supplying values for the non-targeted IVs that satisfy the constraints discovered by the WAM analysis. Additional types of information, such as session state, could also be used to generate more realistic values for non-targeted IVs. Although the proposed approach does not include heuristics for doing this, the development and inclusion of such techniques would most likely serve to further increase the effectiveness of the penetration testing.

To illustrate with an example, consider one of the IV groupings identified by the information gathering phase for Register.jsp (Figure 2), {login, password, userAction}. During attack generation for SQLIAs, the testers target each of these IVs with attacks based on some heuristics. When the first IV, login, is targeted, both the proposed approach and the traditional approaches generate an attack string and use it as the value for login. The difference between the proposed approach and the other approaches is how the values for the remaining IVs are determined. The proposed approach leverages the domain information discovered by WAM, which would result in using an alphanumeric value for password, and setting userAction first to 'createLogin' and

then to 'provideAddress'. The use of this domain information allows the penetration test cases to pass the checks at lines 3 and 6, and thus successfully exploit the vulnerability at line 9. In contrast, a traditional approach would most likely be able to determine that 'createLogin' was a relevant value for `userAction`, but would then have to guess the constraints on `password`.

3.3. Response analysis

The goal of response analysis is to determine whether an attempted attack has been successful. The typical approach to do this is to analyze the response (i.e., HTML output) of the web application for clues as to whether the attack succeeded. For example, a certain type of error message or string in the response could be a side effect of a successful attack. Because manual checking of web pages is extremely time-consuming and error-prone, testers typically use automated heuristic-based tools to check whether an attack was successful. For example, to detect whether an SQLIA was successful, some tools search the web pages in the response for exceptions thrown by the database. Unfortunately, the success of these approaches is often highly application specific, and it is difficult to identify automated heuristics that are broadly applicable. In fact, the results in Section 4.4 show that the current attempts to do so can be highly ineffective. In the proposed approach, automated response analysis is performed by adapting two existing attack-detection techniques, one for SQLIA and the other for XSS attacks. The adapted techniques work by adding an indication of successful attacks to the responses of the web application. This indicator can be easily recognized by the penetration testing tool. The paper uses SQL injection and XSS as the attack types because many web applications contain vulnerabilities to these types of attacks, and heuristics for discovering the vulnerabilities are conducive to automated penetration testing.

3.3.1. SQL injection response analysis. For detecting SQLIAs, the primary challenge is that a successful attack rarely has an observable side effect. Generally, a successful attack simply results in the execution of an unintended SQL command on the database. In most cases, this does not influence the content of the HTML pages generated by the web application, and therefore is not easily observable. To address this issue, the proposed approach leverages WASP, a technique developed in the previous work by two of the authors [4, 5]. WASP uses a combination of positive tainting and syntax-aware evaluation to accurately detect SQLIAs. Positive tainting marks and tracks all of the trusted strings in an application that may be used to build a database command—in practice, all hard-coded strings in the application. Syntax-aware evaluation parses a query right before it is issued to the database and checks that only trusted strings are used to form the parts of a database command that correspond to SQL keywords and operators; if a database command violates this policy, it is prevented from executing on the database.

The authors extended WASP for use in penetration testing. The primary extension was to implement functionality that added a special HTTP header to the application's response when it detected an attack. The header informs the response analysis whether an attempted attack was successful. The response analysis can then correlate this information with the information provided by the attack generator to identify each vulnerable IV along with the attack that was able to reveal the vulnerability.

To illustrate this part of the approach, consider the example SQLIA that targets line 9 of the servlet in Figure 2. Before the servlet executes, WASP marks all of the trusted strings in the servlet, that is, the hard-coded strings used to build the database queries at lines 9 and 19. (The other hard-coded strings are also marked as trusted, but are not used to build the database queries.) At runtime, WASP tracks the trust markings on the strings. When the servlet attempts to execute a database query, WASP parses and checks the string that contains the query to be executed. In this case, the check would reveal that keyword `drop` was generated using a string that was not trusted. This causes WASP to block the attack and return the special HTTP header that flags a detected attack.

3.3.2. Cross-site scripting response analysis. The detection of successful XSS attacks is more straightforward than that for SQLIAs. The reason for this is that, by definition, XSS attacks produce

an observable side effect in the generated HTML, namely the injected HTML content. For XSS the complication is that a vulnerable IV and the point where the attack appears may be on different pages, as is the case in the motivating example. This makes it difficult to identify the corresponding IV through which the successful XSS was injected.

To address this issue, the proposed approach leverages a commonly used technique for detecting when an XSS attack has been successful. This technique uses seeded HTML `<SCRIPT>` tags as part of the attack payload. Each `<SCRIPT>` tag contains a source attribute that the approach sets to specifically encoded values. If these seeded `<SCRIPT>` tags appear on any pages in the web application during penetration testing, the approach detects their presence and uses the encoded values to correlate the successful attack with the vulnerable IV. For example, when performing penetration testing, the attack payload would carry a tag of the following form: `<SCRIPT SRC="X-Y-Z.js"></SCRIPT>`, where X is the number of the component where the payload was introduced, Y is the number of the IV used to inject the payload, and Z is the number of the test case that performed the XSS attack. During penetration testing, the response analysis parses each page visited to determine if it contains one of the seeded tags. If a seeded tag is found, the response analysis parses the source attribute to determine the corresponding vulnerable IV.

To illustrate, consider the XSS vulnerability discussed in Section 2, which is at line 5 of the servlet in Figure 2. When performing penetration testing, part of the attack payload would carry the following tag `<SCRIPT SRC="12-2-123.js"></SCRIPT>`. When DisplayUsers.jsp (Figure 3) displays the list of users, the response analysis parses the HTML to determine if it contains one of the seeded `<SCRIPT>` tags. If a seeded tag is found, the response analysis parses the source attribute to determine the corresponding vulnerable IV. In the case of this example, it would identify that the second IV of the twelfth component was injected by test case number 123. This information is then correlated with an indexed table of components and IVs to determine the name of the vulnerable IV.

3.4. Implementation

The proposed approach is implemented as a prototype tool, SDAPT. The high-level architecture of SDAPT is shown in Figure 4. The SDAPT tool is written in Java, works on Java-based web applications, and performs automated penetration testing for discovering SQLIA and XSS vulnerabilities. The SDAPT implementation is fully automated and does not require developer intervention except to provide a primer script that must be executed to access a web application (e.g., a login script) and a set of attack heuristics. SDAPT inputs the code of a web application (i.e., a set of servlets in bytecode format) and produces a report with a list of the successful attacks and the corresponding vulnerable IVs.
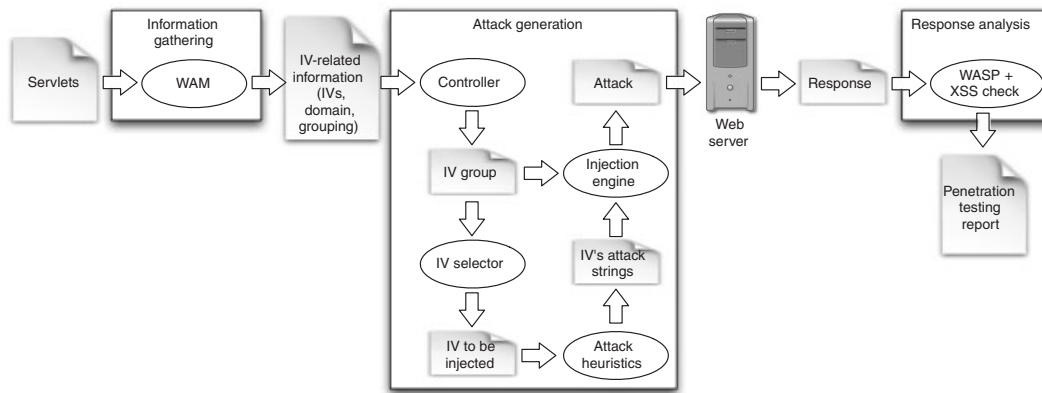
Figure 4. High-level architecture of the SDAPT tool.

The *information gathering* module analyzes the servlets' code and outputs information about the IVs of each servlet. For this module, SDAPT uses the adapted implementation of the WAM analysis tool described in Section 3.1.

The *attack generation* module consists of several submodules. The *controller* inputs the IV-related information and passes the IV groups, one at a time, to the *IV selector*. The *IV selector*, in turn, iterates over each of the IVs in a group and, for each selected IV, passes it to the *attack heuristics* module, which generates possible attack strings for the IV. The *injection engine* generates penetration test cases by combining these attack strings for the selected IV and legitimate values for the remaining IVs in the current IV group. To generate legitimate values, the engine leverages the IVs' domain information. The generated attacks are then attempted against the target web application. In the implementation, the *controller* and *IV selector* were built from scratch, whereas the *attack heuristics* and *injection engine* modules were built on top of the code base of SQLMAP[¶] and WAPITI[‖]. The implementation uses these two penetration testing tools for several reasons: (i) WAPITI and SQLMAP are widely used, popular, and actively maintained penetration testing tools for discovering XSS (WAPITI) and SQLIA (SQLMAP) vulnerabilities; (2) the architecture of both tools is highly modular, which made it easier to integrate them into SDAPT; (3) both tools contain heuristics for performing many different types of SQL injection and XSS attacks; and (4) both tools can interact with a wide range of applications that communicate using different HTTP request methods.

The *response analysis* module receives the HTML responses generated by the target web application and analyzes them to determine whether the attack was successful. It then correlates the results of this analysis with the vulnerable IV. After all of the responses have been analyzed, the output of this module is a report that lists all of the vulnerable IVs along with the test inputs that were able to reveal the vulnerability. For detecting successful SQLIAs, the implementation uses a modified implementation of WASP [4, 5]. Other similar techniques, such as Amnesia [6, 7], CSSE [8], and web application hardening [8], could be used as well. However, WASP has several practical advantages over these techniques: (i) CSSE and web application hardening are not implemented for JEE web applications and (ii) WASP scales better than Amnesia. For detecting successful XSS attacks, the implementation uses the response analysis in WAPITI with code that tracks the specially marked XSS injection tags and correlates their presence in a web page with the IV that introduced the tag (see Section 3.3).

4. EMPIRICAL EVALUATION

The goal of the empirical evaluation is to assess the usefulness of the proposed penetration testing approach (implemented in SDAPT) as compared to traditional penetration testing. To do this, the evaluation measured the proposed approach's *practicality* in terms of time and resources needed to perform the information gathering and attack generation phases, *thoroughness* in terms of number of IVs and components tested, and *effectiveness* in terms of number of vulnerabilities discovered. The evaluation addressed the following research questions:

RQ1: Is SDAPT practical in terms of its time and resource requirements?

RQ2: Does SDAPT result in more thorough testing of a web application than a traditional approach?

RQ3: Is SDAPT's response analysis more accurate than a heuristic-based approach?

RQ4: Does SDAPT's information gathering lead to the discovery of more vulnerabilities than a traditional approach?

As instances of a traditional approach, the evaluation used improved versions of SQLMAP[¶] and WAPITI[‖], the penetration testing tools discussed in Section 3.4. The improvements were

---

[¶]`http://sqlmap.sourceforge.net/`.
[‖]`http://wapiti.sourceforge.net/`.

Table II. Subject web applications.

Subject	LOC	Classes	Servlets
Bookstore	19402	28	27
Checkers	5415	59	32
Classifieds	10702	18	18
Daffodil	18706	119	70
Employee Directory	5529	11	9
Events	7164	13	12
Filelister	8671	41	10
OfficeTalk	4670	63	39
Portal	16089	28	27

implemented by the authors. The improved tools, SQLMAP++ and WAPITI++, were extended in two ways. First, the authors integrated a web crawler into each tool to perform information gathering. Web crawling is one of the most widely used techniques for gathering information about a web application, and is thus a good representative of the current approaches. The web crawler is based on the OWASP WebScarab[‡] project and was modified so that it collected IVs and any default values for these IVs in the web pages visited. (The default values were used as possible values for the IVs during attack generation.) Second, the authors integrated the improved response analysis (see Sections 3.3 and 3.4) into both tools.

4.1. Experiment subjects

The evaluation used nine Java-based web applications as experiment subjects. These applications are listed in Table II. Five of these applications (Bookstore, Classifieds, Employee Directory, Events, and Portal) are commercial open-source products available from GotoCode[**]. Two of the subjects, Checkers and OfficeTalk, are student-developed projects that have been used in previous studies [4, 9]. Filelister and Daffodil are open-source projects available from SourceForge[††].

The GotoCode and student-developed subjects contain a wide range of security vulnerabilities. The remaining two applications contain specific and known vulnerabilities that have been reported in the Open Source Vulnerability Database[‡‡]. Table II lists, for each subject application, its number of lines of code (*LOC*), classes (*Classes*), and the subset of those classes that are servlets (*Servlets*).

4.2. RQ1: Practicality

To evaluate the practicality of the proposed approach, the evaluation compared the analysis time of the three tools (SQLMAP++, WAPITI++, and SDAPT) and the number of test cases they generated during penetration testing. The results of this study are shown in Table III. Note that since both WAPITI++ and SQLMAP++ use the same analysis information, their numbers are listed together as TRADITIONAL. For the analysis time of SQLMAP++ and WAPITI++, the evaluation measured the time needed to crawl and analyze each application's web pages. For the analysis time of SDAPT, the evaluation measured the time to statically analyze each application. These numbers are shown in the column labeled '*Analysis time*'. For the number of test cases, the evaluation counted the number of IV and domain information groupings given to the attack generation module of each approach. This number is shown in the column labeled '*Number of test cases*'.

The results in Table III show that both analysis time and number of test cases are higher for SDAPT than for SQLMAP++ and WAPITI++. The analysis time of SQLMAP++ and WAPITI++ ranges from 5 to 79 s, with an average of about 24 s. The analysis time of SDAPT ranges from

Table III. Practicality results.

Subject	Analysis time (s)		Number of test cases	
	TRADITIONAL	SDAPT	TRADITIONAL	SDAPT
Bookstore	40	2322	802	14711
Checkers	5	146	5	492
Classifieds	79	1797	544	8557
Daffodil	13	1271	442	20698
Employee Directory	15	449	223	3237
Events	11	853	106	3746
Filelister	6	862	45	4465
OfficeTalk	5	477	18	208
Portal	45	726	393	9266

2 to almost 39 min, with an average of about 16 min. Despite being higher than SQLMAP++ and WAPITI++'s analysis time, SDAPT's analysis time is still clearly practical, and since the running of the tests is fully automated, this time represents only machine (and not human) time that must be dedicated to the penetration testing.

In terms of the number of test cases generated, SDAPT consistently generated at least an order of magnitude more test cases than SQLMAP++ and WAPITI++. This result is to be expected, given SDAPT's more complete identification of IV-related information; richer IV information is likely to result in more test cases being generated. Although a higher number of test cases results in more testing time, the maximum testing time for any of the subjects considered was below 10 h, which would not prevent the test cases from being run overnight. Moreover, as the results for RQ2 and RQ4 show, the additional test cases always result in a more thorough penetration testing and in the discovery of more vulnerabilities.

4.3. RQ2: Thoroughness

In general, penetration testers are interested in assessing the thoroughness of their testing. Although thoroughness is not necessarily correlated with finding more vulnerabilities, a higher amount of thoroughness indicates that more of the application has been exercised and provides some objective measure of the degree of completeness of the tests. To evaluate the thoroughness of the proposed approach, the evaluation measured the number of IVs and components tested by SDAPT, SQLMAP++, and WAPITI++. To determine the number of IVs tested, the evaluation analyzed the attacks generated by the three tools and counted the number of unique IV names targeted for each servlet of each subject application. Similarly, to determine the number of components tested, the evaluation counted the number of unique components targeted in each application. Table IV shows the results of this analysis. Here again, since both WAPITI++ and SQLMAP++ use the same information, the paper lists their results together as TRADITIONAL. For each subject and each tool, the table lists the number of unique IVs (*Number of IVs*) and the number of unique components (*Number of Components*) exercised during penetration testing.

As the results in the table show, SDAPT resulted in a consistently higher number of tested IVs and components than SQLMAP++ and WAPITI++. On average, SDAPT tested 111 IVs and 20 components per application, compared to 56 IVs and 8 components per application tested by SQLMAP++ and WAPITI++.

To better understand the reasons for SDAPT's performance, the authors manually inspected the code of several servlets in the subject applications. The findings of this inspection are that SQLMAP++ and WAPITI++ tested less components in part because many of the web pages in the web applications are not linked to each other. Therefore, the crawler was not able to reach all of the pages in an application, and the attack generation based on the information collected by the crawler never targeted the unreachable pages. These unreachable pages partly explain why SDAPT was also able to test a higher number of IVs, but the inspection also revealed two other reasons. The first reason is that several components require the crawler to provide specific IV

Table IV. Evaluation of thoroughness.

Subject	Number of IVs		Number of components	
	TRADITIONAL	SDAPT	TRADITIONAL	SDAPT
Bookstore	104	189	15	27
Checkers	5	69	2	20
Classifieds	61	118	10	18
Daffodil	107	165	7	39
Employee Directory	36	66	6	9
Events	44	79	8	12
Filelister	12	46	1	9
OfficeTalk	16	58	5	20
Portal	123	211	20	27
Average	56	111	8	20

values in order to display subsequent web forms (similar to the example in Figure 2). Because the crawler was not able to guess these values, it could not reach the subsequent web forms and missed their IV information. The second reason is that several components have IVs that do not have a corresponding web form—they are 'hidden' IVs. These IVs may have been developer errors or IVs intended only for use by other components without going through a web form.

Overall, the higher number of tested IVs and components provides evidence that the proposed penetration testing approach can result in more thorough penetration testing of web applications.

4.4. *RQ3*: *Response analysis usefulness*

The evaluation considered the usefulness of the response analysis technique for SQLIAs independently of the effects of the improved information gathering approach. To do this, the authors implemented a version of SQLMAP++ that did not include the response analysis technique and used the standard heuristic-based response analysis provided by SQLMAP. This version is called SQLMAP++$_{NORA}$. The evaluation then measured the number of vulnerabilities discovered by this version when run on all subject applications and compared it against the version with the response analysis. The results of this study are shown in Table V under the column titled SQLMAP++$_{NORA}$. The full version, with the response analysis, is labeled SQLMAP++. The evaluation did not consider a corresponding version of WAPITI++ because the modifications to the response analysis were minor and additional analysis was not needed for detecting successful XSS attacks.

As the results in the table show, SQLMAP++$_{NORA}$ was unable to identify any vulnerabilities in the subject applications. In contrast, using the improved response analysis, SQLMAP++ discovered a total of 36 vulnerabilities. This result indicates that, although SQLMAP is able to generate inputs that cause SQLIAs, its response analysis is ineffective for these applications. The manual inspection of the generated attacks and responses revealed that, in most cases, successful attacks did not have any observable effects on the HTML response from the attacked servlet. That is, in spite of executing malicious SQL commands on the database, there was no significant change in the HTML page returned by the web application after the attack succeeded. In the few cases where there was a change in the HTML response, the change was subtle enough so that the heuristic-based analysis was not able to determine whether it was a normal variation in output or the result of a successful attack. For example, one case involved a page that listed results extracted from a table. A successful attack caused the page to list a specific set of results, but nothing about these results clearly indicated the effect of an attack (i.e., the same results might have been generated by a successful query). Therefore, a heuristic that simply checks differences between HTML responses would be unable to determine if the variation in the listed results was due to an attack. Manual inspection of the results might have been able to recognize the attack, but manually checking each attempted attack is, in general, impractical and error-prone.

Overall, these results indicate that for attacks that do not have an easily observable side effect, such as SQLIA, improved response analysis is necessary for penetration testing to succeed. The

Table V. Number of identified vulnerabilities.

Subject	Cross site scripting		SQL injection		
	WAPITI++	SDAPT	SQLMAP++$_{NORA}$	SQLMAP++	SDAPT
Bookstore	19	63	0	7	11
Checkers	0	1	0	0	2
Classifieds	10	36	0	4	14
Daffodil	1	3	0	6	11
Employee Directory	6	24	0	1	11
Events	10	27	0	4	11
Filelister	0	0	0	1	1
OfficeTalk	1	1	0	2	12
Portal	20	42	0	11	17
Total	67	197	0	36	90

results also show that the proposed techniques for response analysis are effective and an important part of the approach for penetration testing.

4.5. RQ4: *Information gathering effectiveness*

To determine the effectiveness of the proposed technique for information gathering, the evaluation measured the number of vulnerabilities discovered by SQLMAP++, WAPITI++, and SDAPT. The evaluation ran all three tools against each of the subject applications and counted the number of vulnerabilities discovered by each. For this study, all three tools included the improved response analysis. They differed only in the techniques used for information gathering. Table V shows the results of this study, which indicate that SDAPT was able to discover more vulnerabilities to SQL injection and XSS attacks than either SQLMAP++ or WAPITI++.

For SQL injection, SDAPT discovered a total of 90 vulnerable IVs, as compared to 36 found by SQLMAP++; and for XSS attacks, SDAPT identified a total of 197 vulnerable IVs, whereas WAPITI++ found only 66. In addition to discovering more vulnerabilities, SDAPT also had a very low false positive rate. The authors manually inspected each reported vulnerability in order to determine whether it was a real vulnerability or a false positive. The results of this inspection indicated that SQLMAP++ reported three false positives, WAPITI++ reported no false positives, and SDAPT reported two false positives. (These were not included in the vulnerability totals in Table V.) For WAPITI++, the observable side effect of XSS meant that attacks were detected with high precision and, as in the evaluation, with no false positives. For SDAPT and SQLMAP++, the false positives were caused by limitations in the implementation of WASP. In these cases, the subject web applications stored fragments of the query string in a hidden field of a web page, and then used these fragments to build a query string. Since these fragments were not marked as trusted by WASP, their use in the final query string caused WASP to detect an attack. Further engineering to allow WASP to mark other trusted sources of query strings would eliminate these false positives. However, it is worth noting that even though the specific test inputs used in these test cases were benign and led to a false positive, this programming practice did actually lead to a vulnerability that was identified by other test inputs that targeted the hidden field directly.

In this study it was not possible to calculate a false negative rate for the subjects, as there was no complete listing of all of their vulnerabilities. However, both tools were able to identify all of the previously known vulnerabilities to SQLIAs. For Filelister, there was one and for Daffodil, there were two. In addition to identifying these known vulnerabilities, SQLMAP++ discovered another four vulnerabilities and SDAPT discovered another nine that were previously *unknown*.

Overall, the results of this study show that the proposed approach to information gathering—static analysis of the web application's source code—can lead to more effective penetration testing. For both SQLIA and XSS, SDAPT was able to identify a higher number of vulnerabilities than either SQLMAP++ or WAPITI++.

4.6. Threats to validity

In this section, the paper discusses the possible threats to validity of the empirical evaluation and explains how the evaluation addressed each threat.

Construct validity: Construct validity is straightforward in this empirical evaluation, as the evaluation uses typical metrics for measuring the thoroughness, effectiveness, and practicality of the approach. For thoroughness, the number of IVs and components tested is a common measure for both penetration and regular testing. For effectiveness of penetration testing, the number of vulnerabilities is by far the most commonly accepted metric. Similarly, for practicality, analysis time and resources used are generally accepted metrics.

Internal validity: For internal validity the evaluation must ensure that variances in the dependent variable (measures of thoroughness, effectiveness, and practicality) can be attributed to variances in the independent variable (the information gathering and response analysis). To reduce the threats to the internal validity of the studies, the authors maximized the amount of code reuse wherever possible in the implementation of SDAPT, SQLMAP++, and WAPITI++. In particular, SDAPT uses the same attack heuristics that are contained in the original SQLMAP and WAPITI tools. Also, SDAPT, SQLMAP++, and WAPITI++ use the same implementation of the response analysis for their respective attacks. The tools differ only in the information gathering, use of the gathered information, and response analysis (for RQ3). The evaluation also deployed the subject applications with the same configuration when testing them with the three tools.

External validity: The primary concern for external validity in this evaluation is whether the results can generalize more broadly to penetration testing for other types of vulnerabilities and to web applications beyond the subjects considered. As with all studies, a higher number of subjects would increase the external validity of the results. However, the authors believe that the current set is reasonably representative since the subjects come from several different sources, including commercial open-source, auto-generated code, and student-developed projects. The applications also vary in size and type. For other types of vulnerabilities, it is likely that the approach would also be successful because SQLMAP and WAPITI have an architecture and approach similar to many other penetration testing tools. However, for vulnerability types whose detection cannot be automated as easily as for SQLIA and XSS, it is possible that there would not be as significant a benefit from using the proposed approach. Even if this were to be the case, vulnerability to SQLIA and XSS is very common in web applications, and improvement in this area is a useful contribution in and of itself.

5. RELATED WORK

A technique by Miller *et al.*, called *fuzzing*, was an early influential work that led to the development of many subsequent penetration testing techniques [10]. In their work, Miller *et al.* submitted byte streams of random data to common UNIX utilities to determine whether they could crash them. This technique was later adopted and expanded by many testers to discover bugs and security vulnerabilities [11]. Although the concepts and principles behind penetration testing have been known for quite some time, it was not until recently that penetration testing began to receive significant attention [12]. Geer and Harthorne provided an early definition of the goals and techniques of penetration testers [13]. Subsequent work has motivated the need for penetration testing and proposed ways to incorporate the technique into software engineering processes [14, 15].

McAllister *et al.* propose a hybrid approach to penetration testing that leverages usage-based information [16]. The authors attempt to improve penetration testing by improving the underlying information gathering technique. In this case, they use collected user sessions to provide more detailed information about interfaces and legal values. Their evaluation shows that this approach improves over typical web crawling-based approaches. However, the technique is limited by the quality of the initial set of user session information.

There has also been a large amount of research work in static analysis techniques to detect vulnerabilities to SQL injection and Cross-Site Scripting attacks. These approaches typically model

vulnerabilities as information flows that allow untrusted data to perform sensitive operations at certain points in the applications. Techniques such as PQL [17] and information flow analysis [18] allow developers to model the different possible vulnerabilities using an information-flow description language. These techniques analyze a web application and identify information flows that might cause an application to be vulnerable. However, because these techniques cannot precisely model all input validation routines and information-flow operations, they often have a high rate of false positives.

Two recent approaches by Wassermann and Su address the issue of imprecision by combining string analysis and information-flow analysis to more precisely identify vulnerabilities in code [19, 20]. Their evaluation shows that this is a very effective approach for discovering vulnerabilities to SQL injection and XSS attacks. Wassermann and Su also have another approach that is based on using concolic execution to detect vulnerabilities to SQLIAs [21]. This approach is similar to the proposed approach, but does not directly address the problem of information gathering and uses a different underlying mechanism for detecting successful attacks [22]. Kieżun *et al.* also employ concolic execution to drive the identification of SQL injection and XSS vulnerabilities [23]. As compared to traditional information-flow based approaches, this approach has a low false positive rate since vulnerabilities are discovered while executing the application. One drawback of using these approaches is that the question of what to model in the symbolic environment can directly affect the number of vulnerabilities discovered. Many vulnerabilities in web applications exist because of subtle configuration issues or environment settings. For example, to make concolic execution approaches efficient, often only a subset of the environment is modeled. If vulnerabilities depend on aspects that are not modeled, it is very likely that these vulnerabilities will not be detected. One advantage of traditional penetration testing compared to these approaches is that it tests web applications in context, that is, in the environment in which they are deployed.

There has also been work on improving information gathering techniques for web applications. Most of this work has focused on improving web crawling techniques. Early web crawlers were very simplistic—they followed static hyper references encoded as links (e.g., <a> tags) in web pages. As web applications became more dynamic, they included web forms and client-side scripts that could also link to web pages. One of the early techniques from the research community to handle these additional features was VeriWeb [24]. This technique filled in and submitted web forms with developer-provided values and simulated execution of client-side scripts. Nonetheless, automating interaction with the web application via the web forms remained a problematic area. Subsequent work by Huang *et al.* [25] introduced the use of sophisticated heuristics that guided the web crawler's interaction with the web application. These heuristics required initial setup and configuration by the developer, but then allowed the crawler to more autonomously interact with the web application. A recent web crawler-based approach by Elbaum *et al.* focused on automatically discovering rules for generating correct requests to web applications [26]. Their approach submits requests to a web application and uses the responses to infer constraints on the interfaces exposed by the web application. There has also been a substantial amount of web crawling approaches developed by commercial and open-source projects. OWASP's WebScarab web crawler‡, for instance, is a well-known and widely used example of one such project.

The introduction of new client-side technologies in web applications has further complicated web crawling. Technologies such as JavaScript and Adobe Flash are widely used to implement functionality in web applications written in the AJAX framework, which is becoming increasingly popular. Researchers have proposed web crawling techniques to address these new technologies. A recent example of this work is CRAWLJAX [27], which builds 'state-flow' graphs of the client-side of an AJAX based web application. This information is used to build a more complete model of the elements of a web application that would be missed by traditional crawling techniques. The resulting graph can then be used to detect errors in the web application [28].

Overall, web crawling techniques are very popular because they are easy to use and provide precise information about a web application. The primary drawback of these techniques is that they cannot provide any guarantee of completeness with respect to the information they collect since they may not visit every page generated by the web application. However, one advantage of

these techniques, as compared to the proposed information gathering approach, is that they can be used in situations where the source code is unavailable or cannot be statically analyzed due to resource constraints.

In addition to web crawling, other techniques have been proposed for information gathering. These techniques target the problem of interface identification, which is similar to the problem of input vector identification. A group of techniques use developer-provided specifications, which explicitly identify interfaces of the components of a web application [29–33]. The primary drawback of the manual specification-based techniques is that they rely on developers to completely and accurately specify a web application's properties. Although developers may be capable of doing this for small web applications, the size and complexity of modern web applications makes this task challenging, time consuming, and error-prone. In penetration testing, vulnerabilities are often found in IVs that are forgotten or overlooked by developers, so that using these techniques might not be as helpful as using techniques based on the actual implementation of a web application.

Another group of techniques is based around the use of session data and user logs [34–37]. The data collected by these approaches can be analyzed in order to identify useful information about the IVs of the web application. An advantage of this approach is that the data is generally inexpensive to obtain and analyze and, unlike the proposed approach, can be used without source code access. However, similar to web crawling, the primary drawback for this approach is incompleteness; it is only possible to perform penetration testing on parts of the web application that are represented in the gathered data.

A technique closely related to the proposed information gathering technique is the web application modeling technique developed by Deng *et al.* [38]. Their technique scans the code of an application and identifies the names of all IVs. However, unlike the proposed technique, it does not group IVs based on paths of execution or determine possible relevant values for each IV, which reduces the effectiveness of the technique for penetration testing. Another more recent technique by two of the authors makes use of symbolic execution to analyze a web application and identify IV related information [39]. A benefit of this approach is that the analysis takes into account the feasibility of paths and generates more precise information, but the use of symbolic execution can make this approach more difficult to apply to a broad range of applications. Case studies performed using that approach indicate that the increased precision can significantly reduce the number of test cases generated without reducing the effectiveness of the penetration testing. However, the implementation of this approach is not yet readily applicable to all of the subject web applications.

6. CONCLUSION

Penetration testing is a widely used technique to help improve the security of web applications. Two important steps of penetration testing are identifying the IVs of a web application and determining whether an attempted attack was successful. This paper proposed a new approach to penetration testing that improves both of these steps. The approach incorporates a conservative static analysis of the web application that identifies IVs directly from the application's code. The approach improves the response analysis by leveraging automated dynamic analyses that accurately detect when an attack has succeeded. The authors implemented the approach in a prototype tool, SDAPT, and compared its performance against two state-of-the-art penetration testing tools that used web crawling to identify an application's IVs. The empirical evaluation included nine web applications, and the results show that SDAPT was able to test the targeted web applications more thoroughly and discover more vulnerabilities than either of the other two tools, while still being practical in terms of required analysis time. These results indicate that the proposed approach is useful, effective, and can outperform existing alternative approaches to penetration testing.

REFERENCES

1. Halfond WGJ, Orso A. Improving test case generation for web applications using automated interface discovery. *Proceedings of the Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 2007.
2. Halfond WGJ, Viegas J, Orso A. A classification of SQL-injection attacks and countermeasures. *Proceedings of the International Symposium on Secure Software Engineering*, Washington, DC, U.S.A., March 2006.
3. Christensen AS, Møller A, Schwartzbach MI. Precise analysis of string expressions. *Proceedings of the International Static Analysis Symposium*, San Diego, CA, U.S.A., June 2003; 1–18.
4. Halfond WGJ, Orso A, Manolios P. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. *Proceedings of the Symposium on the Foundations of Software Engineering* (*FSE 2006*), Portland, OR, U.S.A., November 2006; 175–185.
5. Halfond WGJ, Orso A, Manolios P. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *Transactions on Software Engineering* 2008; **34**(1):65–81.
6. Halfond WGJ, Orso A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. *Proceedings of the International Conference on Automated Software Engineering*, Long Beach, CA, U.S.A., November 2005; 174–183.
7. Halfond WGJ, Orso A. Combining static analysis and runtime monitoring to counter SQL-injection attacks. *Proceedings of the International Workshop on Dynamic Analysis*, St Louis, MO, U.S.A., 2005; 22–28.
8. Pietraszek T, Berghe CV. Defending against injection attacks through context-sensitive string evaluation. *Proceedings of Recent Advances in Intrusion Detection*, Seattle, WA, U.S.A., September 2005.
9. Gould C, Su Z, Devanbu P. Static checking of dynamically generated queries in database applications. *Proceedings of the International Conference on Software Engineering*, Edinburgh, Scotland, May 2004; 645–654.
10. Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM* 1990; **33**(12):32–44.
11. Sutton M, Greene A, Amini P. *Fuzzing*: *Brute Force Vulnerability Discovery*. Addison-Wesley: Reading, MA, 2007.
12. Thompson HH. Application penetration testing. *Symposium Security and Privacy* 2005; **3**(1):66–69.
13. Geer D, Harthorne J. Penetration testing: A duet. *Proceedings of the Computer Security Applications Conference*, Las Vegas, NV, U.S.A., December 2002; 185–195.
14. Arkin B, Stender S. McGraw G. Software penetration testing. *IEEE Security and Privacy* 2005; **3**(1):84–87.
15. Bishop M. About penetration testing. *IEEE Security and Privacy* 2007; **5**(6):84–87.
16. McAllister S, Kirda E, Kruegel C. Leveraging user interactions for in-depth testing of web applications. *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*. Springer-Verlag: Berlin, Heidelberg, 2008; 191–210.
17. Martin M, Livshits B, Lam MS. Finding application errors and security flaws using PQL: A program query language. *Proceeding of the Conference on Object Oriented Programming Systems Languages and Applications*, San Diego, CA, U.S.A., October 2005; 365–383.
18. Huang YW, Yu F, Hang C, Tsai CH, Lee DT, Kuo SY. Securing web application code by static analysis and runtime protection. *Proceedings of the International World Wide Web Conference*, New York, NY, U.S.A., May 2004; 40–52.
19. Wassermann G, Su Z. Sound and precise analysis of web applications for injection vulnerabilities. *Proceedings of the Conference on Programming Language Design and Implementation*. ACM: New York, NY, U.S.A., 2007; 32–41.
20. Wassermann G, Su Z. Static detection of cross-site scripting vulnerabilities. *Proceedings of the International Conference on Software Engineering*. ACM: New York, NY, U.S.A., 2008; 171–180.
21. Wassermann G, Yu D, Chander A, Dhurjati D, Inamura H, Su Z. Dynamic test input generation for web applications. *Proceedings of the International Symposium on Software Testing and Analysis*, Seattle, WA, U.S.A., July 2008.
22. Su Z, Wassermann G. The essence of command injection attacks in web applications. *Proceedings of the Symposium on Principles of Programming Languages*, Charleston, SC, U.S.A., January 2006; 372–382.
23. Kiezun A, Guo PJ, Jayaraman K, Ernst MD. Automatic creation of SQL injection and cross-site scripting attacks. *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society: Washington, DC, 2009; 199–209.
24. Benedikt M, Freire J, Godefroid P. VeriWeb: Automatically testing dynamic web sites. *Proceedings of the International World Wide Web Conference*, Honolulu, HI, U.S.A., May 2002.
25. Huang YW, Huang SK, Lin TP, Tsai CH. Web application security assessment by fault injection and behavior monitoring. *Proceedings of the International World Wide Web Conference*, Budapest, Hungary, May 2003; 148–159.
26. Elbaum S, Chilakamarri KR, Fisher M II, Rothermel G. Web application characterization through directed requests. *Proceedings of the International Workshop on Dynamic Analysis*, Shanghai, China, May 2006; 49–56.
27. Mesbah A, Bozdag E, van Deursen A. Crawling Ajax by inferring user interface state changes. In *Proceedings of the International Conference on Web Engineering*, Schwabe D, Curbera F, Dantzig P (eds.). IEEE Computer Society: Washington, DC, 2008; 122–134.
28. Mesbah A, van Deursen A. Invariant-based automatic testing of Ajax user interfaces. *Proceedings of the 31st International Conference on Software Engineering* (*ICSE09*) (Research Papers). IEEE Computer Society: Washington, DC, 2009; 210–220.

29. Andrews AA, Offutt J, Alexander RT. Testing web applications by modeling with FSMs. *Software Systems and Modeling* 2005; **4**(3):326–345.
30. Betin-Can A, Bultan T. Verifiable web services with hierarchical interfaces. *Proceedings of the International Conference on Web Services*, Orlando, FL, U.S.A., July 2005; 85–94.
31. Bultan T. Modeling interactions of web software. *Proceedings of the International Workshop on Automated Specification and Verification of Web Systems*, Cyprus, November 2006.
32. Jia X, Liu H. Rigorous and automatic testing of web applications. *Proceedings of the International Conference on Software Engineering and Applications*, Cambridge, MA, U.S.A., November 2002; 280–285.
33. Ricca F, Tonella P. Analysis and testing of web applications. *Proceedings of the International Conference on Software Engineering*, Toronto, ON, Canada, May 2001; 25–34.
34. Elbaum S, Karre S, Rothermel G. Improving web application testing with user session data. *Proceedings of the International Conference on Software Engineering*, Portland, OR, U.S.A., November 2003; 49–59.
35. Elbaum S, Rothermel G, Karre S, Fisher II M. Leveraging user-session data to support web application testing. *Transactions on Software Engineering* 2005; **31**(3):187–202.
36. Hao J, Mendes E. Usage-based statistical testing of web applications. *Proceedings of the International Conference on Web Engineering*. ACM: New York, NY, U.S.A., 2006; 17–24.
37. Kallepalli C, Tian J. Measuring and modeling usage and reliability for statistical web testing. *Transactions on Software Engineering* 2001; **27**(11):1023–1036.
38. Deng Y, Frankl P, Wang J. Testing web database applications. *SIGSOFT Software Engineering Notes* 2004; **29**(5):1–10.
39. Halfond WGJ, Anand S, Orso A. Precise interface identification to improve testing and analysis of web applications. *Proceedings of the International Symposium on Software Testing and Analysis*, Chicago, IL, U.S.A., 2009.