Circe: A Grammar-Based Oracle for Testing Cross-Site Scripting in Web Applications

Andrea Avancini and Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
{anavancini,ceccato}@fbk.eu

***Abstract*—Security is a crucial concern, especially for those applications, like web-based programs, that are constantly exposed to potentially malicious environments. Security testing aims at verifying the presence of security related defects. Security tests consist of two major parts, input values to run the application and the decision if the actual output matches the expected output, the latter is known as the "oracle". In this paper, we present a process to build a security oracle for testing Cross-site scripting vulnerabilities in web applications. In the *learning* phase, we analyze web pages generated in safe conditions to learn a model of their syntactic structure. Then, in the *testing* phase, the model is used to classify new test cases either as "safe tests" or as "successful attacks". This approach has been implemented in a tool, called Circe, and empirically assessed in classifying security test cases for two real world open source web applications.**

I. Introduction

Software bugs are in general detrimental for software quality and reliability. They require even further and urgent attention when involving security aspects, because vulnerabilities could be exploited by attackers who intend to steal sensitive data or to spread malware (e.g., computer viruses). According to available surveys [5], one of the most prominent class of vulnerabilities that affects web applications is Cross-site scripting (XSS for short). These attacks consist in injecting malicious fragments of JavaScript or HTML code into the web application under attack, exploiting inadequate validation of untrusted input data.

Security testing is the branch of software testing devoted to stress programs with respect to security features, in order to identify vulnerabilities. While the generation of security test cases has been addressed by many approaches (e.g., [21], [12], [13], [9], [8], [11]), a security oracle to judge if the system passes security test cases is an overlooked problem. It has been initially defined as a manual task [21] and later by more automatic processes [12], [13], [9], [8], [11] but the oracle has always been very specific to the approach adopted for generating test cases. Such oracles are closely tied to the corresponding test case generation algorithm, and they can be hardly used out of their specific context. In [12], for example, the oracle consists in checking if a web page contains the same JavaScript fragment that has been used to generate the corresponding test case.

We state that the construction of a security oracle is a problem in itself. A reusable security oracle should not depend too much on how test cases have been written (either manually or automatically), but just on the specific class of vulnerability (e.g., XSS) to tackle.

In this paper we address the problem of constructing a security oracle for XSS vulnerabilities of web applications with a novel approach called Circe[1]. Our solution is based on the assumption that a successful code injection attack can be recognized as a structural change in the web page, i.e. the malicious content that has been injected. Circe constructs a model of the structural properties of those HTML pages that a web application generates when running on safe conditions. A security test case is classified as a successful attack only if it makes the web page violate this model.

The paper is organized as follows. In Section II we cover the background on XSS vulnerabilities, while in Section III we present the high level process to construct the oracle. Then, in Sections IV, V and VI, the three steps of the oracle construction are presented in detail. Our oracle is validated empirically in Section VII. The comparison with the state of the art (Section VIII) and the conclusions (Section IX) close the paper.

II. Cross-Site Scripting Vulnerabilities

Cross-site scripting vulnerabilities (XSS hereafter) are caused by improper validation of input data (e.g., coming from the user). Input data may contain HTML of JavaScript fragments that, if printed on a web page, may alter the final content such that malicious code is injected.

A. Running Example

Figure 1 shows an example of PHP page that contains a reflected XSS vulnerability, i.e. there exists an execution flow where the input value *param* is not adequately validated before being printed by a *sink* statement (*echo* statement in PHP) in the output page (line 15). Any code contained in the input value that is not properly validated is then added to the resulting page.

The page accepts three inputs, *param*, *cardinality* and *op*. It adopts a quite common PHP pattern, performing different actions depending on the values of inputs. In case *op* is set a table is shown, while a menu is displayed otherwise. The number of rows in the table and the number of links in the

[1] In the Greek mythology, Circe is a goddess of magic mentioned in Homer's Odyssey. She was able to predict and influence the future.

```
    <html>
    <body>
    <?php
 1  $p = $_GET['param'];
 2  $n = $_GET['cardinality'];
 3  $op = $_GET['op'];
 4  if ( $n < 1 )          //input validation
 5     die;
 6  if ( strpos($p,'<script') !== false)
 7     $p=htmlspecialchars($p);
 8  if (isset($op)) {          //print table
 9     echo '<table border=1>';
10     for ($i=0; $i<$n; $i++) {
11        echo '<tr><td>first cell </td>' .
               '<td>second cell</td>' .
               '<td>third cell</td></tr>';
       }
12     echo "</table>";
    }
    else {                          //print menu
13     for ($i=0; $i<$n; $i++) {
14        echo '<a href=first.php>link #' .
          $i . '</a>';
          }
    }
15  echo $p;                    //vulnerability
    ?>
    </body>
    </html>
```

Fig. 1. Running example of a XSS vulnerability on PHP code.

menu depend on the value of *cardinality*. Input *param* is just printed.

By looking at the running example in detail, we can see that, on lines 1–3, input values are read (represented in PHP as the special associative array *$_GET*) from the incoming HTML request and assigned to local variables *$p*, *$n* and *$op* respectively.

On lines 4–7, input values are then validated. In case *$n* contains a value smaller than 1 or a string that does not represent a number, the execution aborts (`die` statement at line 5). On line 7, the value of variable *$p* is validated, removing potentially dangerous characters by using the PHP function `htmlspecialchars`. This function changes special HTML characters (e.g. "<", ">" and "**"**") into their encoded form ("`<`", "`>`", and "`&quot;`"), safe when printed in a web page. Validation, however, is done only when condition on line 6 holds, but that condition is not adequate to cover all the possible dangerous cases. For example, harmful code containing a different tag (e.g. <a>) or with a space between < and `script` would skip the sanitization.

Depending on the value of variable *$op*, either a table (lines 8–12) or a menu (lines 13–14) is shown. In both the cases, the size of the result depends on the value of *$n*, because of the `for` loops at lines 10 and 13. Eventually, variable *$p* is printed at line 15, possibly causing a security threat because of the inadequate validation at lines 6–7.

Although quite simple, the code in Figure 1 contains the key ingredients of a typical XSS vulnerability. An attack, in fact, is required to satisfy the following constraints to exploit a vulnerability:

1) Input satisfies control flow conditions: in order to avoid the premature termination of the execution, *cardinality* must be a number bigger than 0;
2) Validation is skipped: potentially malicious code is removed from string *param* at line 7, so the attack must drive the execution to a different control flow;
3) Input contains attack code: variable *param* is printed in the web page. This input value is the attack vector where to add malicious code in order to get it printed into the final page.

A vulnerability, however, can be much more tangled than this example, as it can involve intricate control flows and/or a complex data flow between input values and sinks.

III. Circe: the Security Oracle

The goal of an XSS attack is to inject JavaScript or HTML code fragments into a web page. Thus, consequences of an injection should be recognizable as structural changes in the parse tree of the page under attack, when compared with the parse tree of the same page running under normal conditions.

Web applications, however, are highly dynamic and the parse tree of the same page may vary a lot, even without code injection. For instance, with respect to the running example (see Figure 1), the same PHP script under harmless conditions can display different results (number of table rows) and can take different alternative actions (showing a table or a menu).

The security oracle should distinguish between those variations that are safe because due to the dynamic behavior of the application and those variations caused by code injection due to successful attacks. To achieve this objective, the security oracle should be equipped with a model of parse trees of HTML pages on *safe* cases, so as to classify as attacks all those outputs that do not satisfy the model. Our security oracle classifies the output of test cases in form of parse trees of the web page under analysis. We construct a model of the parse trees on *safe* cases according to the steps shown in Figure 2. The *oracle construction* (Figure 2 right hand side) requires a preliminary *preparation* phase (Figure 2 left hand side) to start with a minimal set of test cases.

Preparation steps:

1) **Adequacy criterion definition:** a criterion to evaluate the adequacy of test cases should be defined to assess the completeness of the set of test cases to be generated;
2) **Test case generation:** safe (i.e. without injecting any code) input values are generated, either manually or automatically, such that they satisfy the adequacy criterion.

When a minimal set of test cases is available, it is used to start building the model of safe executions.

Oracle construction steps:

1) **Test case perturbation:** the initial test suite is usually minimalist, so it represents a too limited view of the system to permit the construction of the model of safe executions. In order to augment variety among the tests, initially available test cases are mutated into a more complete set of *training test cases*;

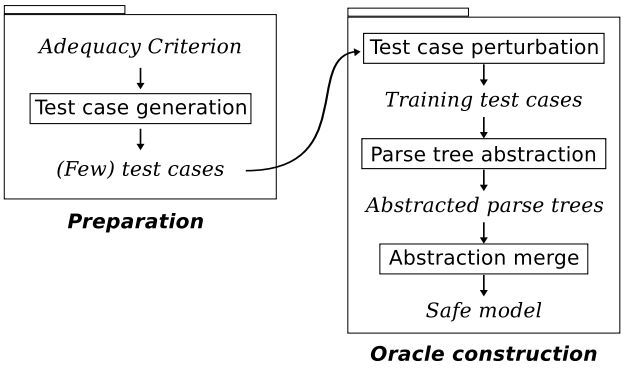

Fig. 2. Overview of the process to construct the security oracle.

2) **Parse tree abstraction:** HTML pages collected when running the training test cases are too specific to particular input values. In order to infer a generic safe model, parse trees are processed to remove all those details that are not relevant to detect a code injection attack; and
3) **Abstraction merge:** all the abstractions are combined into a common structure, which is abstract enough to reasonably represent the structure of the page in *safe* conditions. This combination is called the *safe model*.

Eventually, a new test case that would not satisfy the safe model would be classified as code injection, because it would represent a successful XSS attack.

A. Adequacy Criterion

We decided to adopt an adequacy criterion which is based on those candidate vulnerabilities reported by static taint analysis [10] when analyzing the application under test. Static taint analysis is a technique that tracks the tainted/untainted status of variables throughout the application's control flow. A vulnerability is reported whenever a possibly tainted variable is used in a sink statement (e.g. print). In case of XSS [22], tainted values are those values that come from untrusted sources (data base and user input) and sinks are all the print statements that append a value to the resulting web page. Tainted status is propagated on assignments and tainted variables become untainted upon sanitization (e.g., function `htmlspecialchars` in PHP). Taint analysis is formulated as a flow analysis [18] problem, where the information propagated in the control flow graph is the set of variables holding tainted values.

Taint analysis does not provide executable test cases, but just the *data slice* that gives raise to the vulnerability. The data slice consists of the chain of those assignments that make a tainted value flow into a sink statement, skipping validation. The data slice for the vulnerability depicted in Figure 1, for example, is composed by lines {1, 15}, because variable $p is assigned the input *param* at line 1, and it is later printed at line 15. This list of assignments does not identify a path in the control flow, but a (possibly infinite) set of paths. In fact, it does not specify how many iterations to take in loops (lines 10 and 13) or which alternative action to take, i.e. whether to display the table (lines 8–12) or the menu (lines 13–14).

Identification of candidate vulnerabilities and data slices is done by using Pixy [10], a publicly available tool for taint analysis of PHP code.

We also identified all those control statements that hold a control dependence on the statements in the data slice, because they drive the code execution into (or away from) a vulnerable path. The branches to traverse in order to make a tainted value reach the vulnerable statements are called *target branches*, because they are those branches that a security test case must take to execute the vulnerable data slice. In the running example of Figure 1, the target branches are {4-6, 6-8}. The former (4-6) is required to avoid termination, while the latter (6-8) to skip the validation of variable *$p* at line 7 that would break the data slice and block the propagation of the tainted value towards the sink.

Each candidate vulnerability defines a set of branches to take in order to execute the vulnerability itself, that we call *vulnerability slice*. Vulnerability slices (with their target branches) are used to evaluate the adequacy of the security tests. A vulnerability slice is covered by a test case if, when executing the test cases, all the target branches of the corresponding slice are taken. A test suite is considered adequate when each vulnerability slice is covered by at least one test case.

B. Test Case Generation

To generate test cases that satisfy the adequacy criterion, we rely on a tool that we developed for previous work [1], [2], that integrates heuristics (Genetic Algorithms) and analytic solutions (SMT solver) to produce input values for test cases, with the objective that they have to cover all the candidate vulnerability slices detected by static analysis. Many test case generation approaches, however, exist in literature [21], [12], [13], [9], [8], [11] and any of them is, in principle, applicable to this context. Alternatively, input values can be also manually written, for example by translating the application's use cases into test cases.

In our experiment, test cases are modeled as sets of valid input names and values for the page under analysis. For example, the test case {*param="param", cardinality="2", op="X"*} is the representation of the consistent URL *http://host/page.php?param=param&cardinality=2&op=X* for the application under test.

Since the focus of this paper is on the security oracle, we will not provide further details on the input values generation part, and we will use the available tool as a black box. More details about this phase can be found in previous work [1], [2].

IV. Test Case Perturbation

A test case generation strategy is usually minimalist and it probably gives few (or just one) test cases for each vulnerability slice. Multiple distinct test cases, however, are needed to build a model with an appropriate level of generality. To increase the number and the diversity of the training test cases, initial test cases that have been automatically generated are subject to perturbation (1) by using mutation operators and (2)

by concrete symbolic execution. Perturbation continues until a time-out expires, or until at least 300 distinct training tests are generated for each vulnerability slice.

A. Mutation Operators

Test cases are mutated by applying the mutation operators listed in the following.

Change input value: The value of an input variable is randomly changed. One input variable is chosen with uniform probability and its value is changed in two alternative ways. Either (1) one character of the current value is randomly selected and substituted with a random character or (2) a random string is concatenated to the current value.

Remove input variable: An input variable is randomly chosen from the available in the test case, and it is removed.

Insert new input variable: A new input variable is added to the test case. The variable name is randomly selected among the input variable names referenced by the page under tests and its value and type are generated randomly.

All the constant strings that appear in the page source code are collected and stored in a pool. When a new random string is required, such string is either chosen from the string pool (probability 1/2) or randomly generated (probability 1/2). In the latter case, the following algorithm is used. A character is randomly selected from a set that contains alphanumeric characters and special HTML/JavaScript characters, i.e. from [a-zA-Z0-9], and [<>?&+-*/=\()[ ]"'] . After the first character, a second one is added with probability 1/2, so the probability of having a string of length 2 is 1/2. In case of the addition of a second character, a third one is added with probability 1/2, so the probability of a string of 3 characters is $1/2^2$ = 1/4. More characters are added with a probability that decays exponentially. In general, the probability of generating a string of length *n* is $1/2^{n-1}$.

B. Concrete Symbolic Execution

Mutation operators achieve a limited variation of input values (local perturbation), such as the number of table rows displayed by the running example of Figure 1. Concrete symbolic execution [14] is used to achieve a broader perturbation of test cases. Path conditions are collected along the execution of the program on a test case, in terms of conditions on symbolic input values. Among the collected conditions, one is negated and, together with the remaining conditions, is passed to an SMT solver. The SMT solver elaborates a new set of input values, they are a new test case which is different than the one that originated the current set of conditions. So, the execution of the new test case will be similar to the previous one, but a different branch will be taken according to the negated path condition.

1) Symbolic Values: Path conditions are collected in terms of constraints on inputs, so the symbolic values of program variables are traced in terms of their relation with input values. This is done by instrumenting the code. After each assignment, a dynamic map is updated with the new symbolic value of the assigned variables, in this way:

- The symbolic value of an input is the name of the input variable;
- The symbolic value of a program variable can be found in the dynamic map; and
- The symbolic value of an expression is a string representing the quotation of the expression, where variables are replaced with their symbolic values. In case of operators not supported by the solver (e.g., non-linear arithmetic), concrete values are used.

2) Symbolic Path Constraints: Symbolic constraints are collected at decision points. Conditions are collected on decision points as symbolic values of the condition expressions. When conditions involve operators which are not supported by the solver, we resort to concrete values. Conditions are collected together with a reference of the corresponding traversed branch.

An execution of the running example on input values $\{cardinality{=}0\}$ would take the branch 4-5 and then exit. This would collect just one path constraint:

$$4-5 : GETcardinality < 1$$

3) Constraint Selection and Change: Before passing the path constraints to the solver, one of them is selected and negated, so as to make the solver elaborate new input values that force the execution follow a different path.

In our example, just one path constraint is collected and then negated before being passed to the solver in the form:

$$GETcardinality >= 1$$

Since this formula is satisfiable, a new set of input values is returned:

$$\{cardinality = 1\}$$

These new input data make the execution take the branch 4-6 instead of 4-5, thus executing a different part of the application.

Test case perturbation requires to implement mutation operators and concrete symbolic execution. They are integrated with the existing tool for the automatic generation of input values for security test cases [2]. The TXL language [6] has been used to instrument the code for propagating symbolic values and symbolic constraints during the execution of the PHP code. TXL supports the definition of grammar-based rules to perform source-to-source code transformation. To solve symbolic constraints, we adopted Yices [7], an SMT solver that supports linear arithmetic and operations on bit-vectors. We also used TXL to implement an ad-hoc PHP-to-Yices encoding of PHP path conditions, in order to make them comprehensible for the solver. We resorted to bit-vectors to represent strings and operations on them.

V. Parse Tree Abstraction

Training test cases obtained after perturbation are executed on the instrumented web application to record what branches are traversed at run-time. For those tests that still cover a

vulnerability slice, we collect the generated HTML code and the corresponding parse tree. Each parse tree is then abstracted by applying the subsequent abstraction rules. TXL has been used to implement these transformation rules:

Remove text and formatting: Text and comments are removed, so just HTML tags and scripts remain. We also remove all the formatting tags (e.g., <tt>, <i>, <b>, <big>, <small>, <br> and <hr>) that do not specify JavaScript code to be executed on user events (*onclick*, *ondblclick*, *onmousedown*, *nmousemove* and similar).

Compact list: A sequence composed by the same tag is replaced by a repetition pattern of the given tag. The repetition pattern is represented by the special attribute *pattern="+"*. The replacement tag contains the union of the attributes from the original tags.

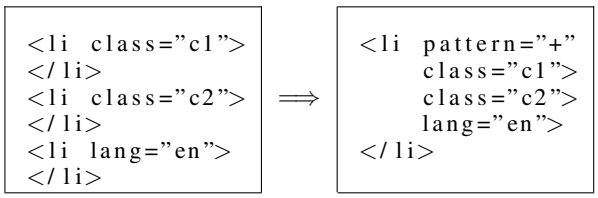

Fig. 3. Compact list pattern.

Compact list with same children: A sequence of the same tag that contains the same nested tags is replaced by a repetition pattern of the given tag. The repetition pattern is represented by the special attribute *pattern="+"*. The attributes of the replacement tag are the union of the attributes from the original tags. The replacement tag will contain the same child tags as the original tags.

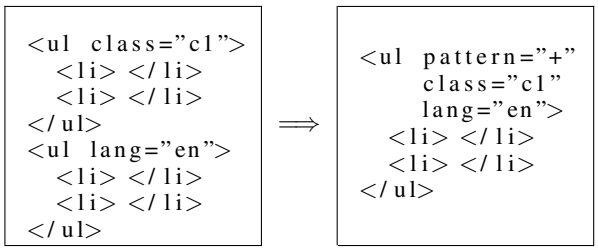

Fig. 4. Compact list with same children pattern.

With respect to the running example of Figure 1, a test case consisting of the input values {*param="param", cardinality="2", op="X"*} would produce the HTML code in Figure 5(a), i.e. a table with two rows. The first abstraction rule that applies is *remove text and formatting*, which removes the text contained in the table cells, originating the intermediate result of Figure 5(b). Then, sequences of <td> tags are transformed in a repetition pattern by the rule *compact list* (Figure 5(c)). Eventually, the rule *compact list with same children* changes the table rows <tr>, returning the final abstraction that is shown in Figure 5(d).

This representation is more abstract than the initial test output, because it does not model table content and size, which are too specific to the particular test case that has generated them. The abstraction just models the presence of the table with all its attributes (e.g., *border=1*).

VI. Abstraction Merge

When parse tree abstractions are available for all the training test cases, they need to be combined into a common representation that models all the safe executions for the same vulnerability slice.

Combination is done pairwise and it starts by merging two parse tree abstractions. The resulting abstraction is then combined with a third tree abstraction. The combination process continues combining one abstraction after the other, until the last one is merged into the final model.

The trees to merge are visited in parallel using a breadth-first strategy. Starting from the root, all the children nodes are fully processed, before continuing with the grand-children nodes. Tags and attributes are merged with the following rules:

Merge equal tags: during the parallel visit of trees *A* and *B*, the same tag <x> is found at the same position in the the two trees, as shown in Figure 6. Tag <x> is then copied into the result. The attributes of the result are the union of attributes of <x> from *A* and *B*. The parallel visit continues with the child tags of <x> in *A* and *B*.

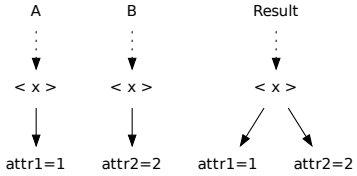

Fig. 6. Merge equal tags rule.

Merge different tags: during the parallel visit, two different tags are found, <x> in *A* and <y> in *B*, according to the example shown in Figure 7. The pattern <x | y> is created in the resulting tree. The attributes of the result are the union of attributes of <x> and <y>. The parallel visit continues on the child tags in <x> and <y>.

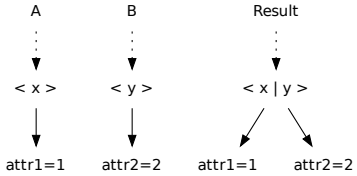

Fig. 7. Merge different tags rule.

Compute difference on child tags: When done with a parent tag, the merge continues on the child tags. The difference between the two lists of children is computed by using the *longest common subsequence algorithm* [4]. The result is the edit difference between the two lists, that consists in those tags that have to be *added* or *removed* from the first list to obtain the second one. Common tags are copied in the result (taking the union of the attributes) using the *merge equal tags* rule, while different tags are transformed by the *merge different tags* rule.

In the example shown in Figure 8, the edit difference between the children of <x> in *A* and <x> in *B* consists in removing <z> and adding <k> and <w>. As <z> and <w>

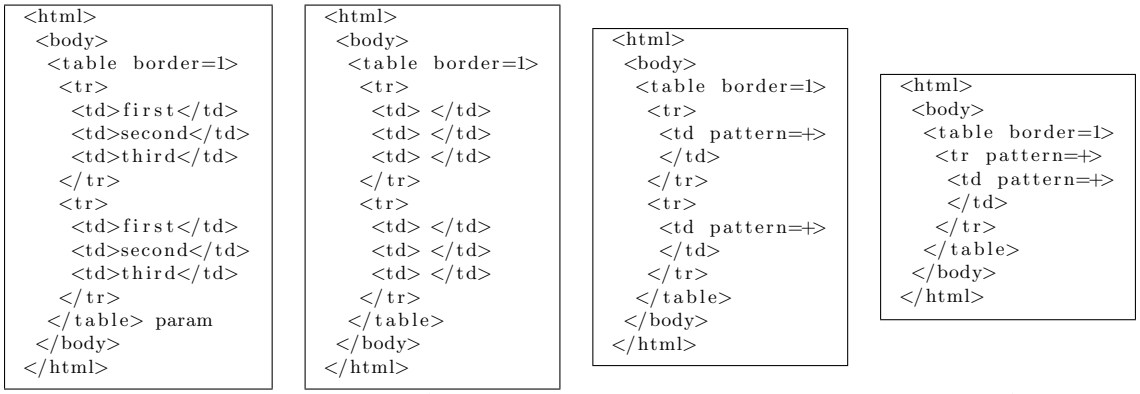

Fig. 5. Abstraction transformation on the running example: (a) HTML output, (b) remove text, (c) compact list, (d) compact list with same children.

appear in the same position (i.e., just after the common <y>) they are replaced by an alternative pattern. Tag <k>, instead, misses a corresponding tag in *A*, so it is copied in the result as an optional tag, i.e. a tag with the optional pattern attribute *pattern="?"*.

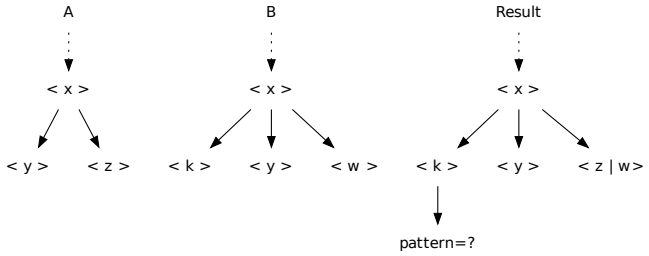

Fig. 8. Compute difference on child tags rule.

Merge pattern attributes: Tags may contain attributes that specify a pattern, which possible values are:

- **?** for optional tags (zero or one tags);
- **+** for a list of one or more tags; and
- ***** for a list of zero or more tags.

When merging tags that contain pattern attributes, patterns are composed applying the composition rule on Table I.

Merge	?	+	*
?	?	*	*
+	*	+	*
*****	*	*	*

TABLE I
RULE FOR MERGING PATTERN ATTRIBUTES OF TAGS.

Figure 9 shows the combination of two abstractions (a) and (b), corresponding to the execution of the running example (Figure 1) on the two alternative functionalities, i.e. displaying a table or a menu respectively. Both the trees start with the same <html> root tag. The common tag is then copied into the safe model (c) by the *merge equal tags* rule, and their children nodes are visited by the *compute difference on child tags* rule. No difference is found, so *merge equal tags* applies and child tag <body> is copied into the result. Then, some differences are found on the child nodes of <body> (the third level of the trees), so *merge different tags* applies and <table> and <a> are merged into the brand new tag <table | a>. The attributes of the new tag are the union of the attributes of the original tags. Then, computation continues on the child tags, but while (a) has one child tag (i.e. <tr>), (b) has none. *Compute difference on child tags* rule merges the second empty list with the first list of children, so the <tr> tag is added to the result with the optional pattern "?". Since this tag already had the list pattern "+", it must be composed with the new pattern "?" as specified by the *merge pattern attributes* rule, resulting in the new pattern "*".

Figure 9(c) represents the safe model of the running example for the vulnerability slice consisting of the target branches {4-6, 6-8}. It models both the alternative behaviors (i.e., table and menu) in all their safe variants (i.e., number of rows and menu entries).

A. Oracle Decision Procedure

Once the safe model is available, it can be used to assess whether a new test case is a successful attack. The decision procedure consists in:

1) Running the candidate attack on the web application under test and collecting the HTML output page;
2) Calculating the parse tree abstraction on the output of the candidate attack;
3) Merging the abstraction with the current safe model, obtaining the *evaluation model*; and
4) Comparing the safe model and the evaluation model.

In case the safe model results to be equal to the evaluation model, it means that the oracle is unable to detect any structural difference between the candidate attack execution and any safe execution, so the test case is classified as pass (the attack is non-successful). A difference between the safe model and the evaluation model, instead, means that the candidate attack has caused structural changes with respect to safe executions. In this case, the oracle detects a code injection and it classifies the test as non-pass (the attack is successful).

VII. EMPIRICAL ASSESSMENT

The proposed approach has been implemented in Circe, and assessed on some case studies. First of all, the safe model for the system under test is learned using the *training test cases* as described in the previous sections. Then, the oracle is

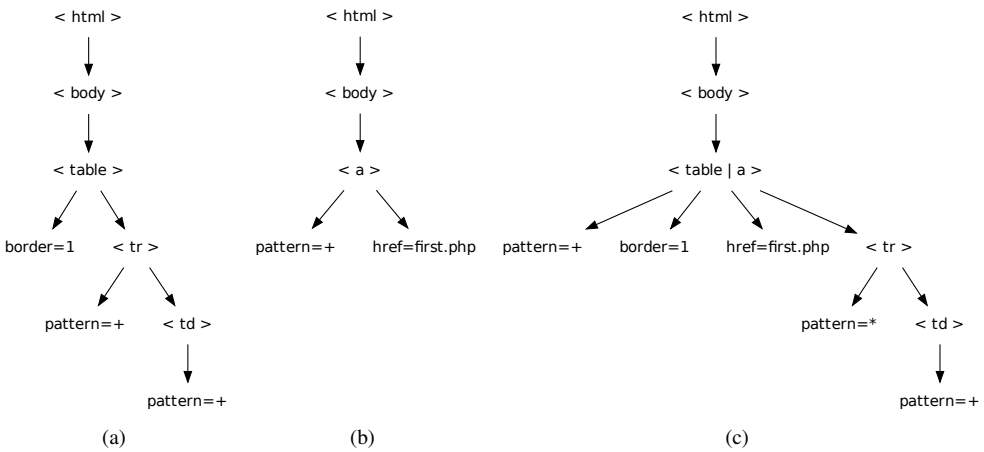

Fig. 9. Example of combination of two parse tree reductions (a) and (b) into the safe model (c).

evaluated, by studying its performance in classifying a (new) set of test cases, the *assessment test cases*, containing both safe tests and successful attacks.

A. Case Studies and Oracle Construction

For the empirical validation, we used two case study applications written in PHP. An important requirement for the case studies is that they contain real vulnerabilities, so the oracle can be evaluated in terms of correct classification of attacks ad safe tests.

The first case study we selected is Yapig version 0.95b, an open-source application that implements an image gallery management system. It consists of 9,113 lines of code and 53 source files, with 160 user-defined functions and 2,638 branches. The second case study is PhpPlanner version 0.4, a calendar/agenda that uses mySQL as back-end, which consists of 1,953 lines of code in 19 source files, 23 functions and 136 branches.

Static analysis reported a total of 41 candidate XSS vulnerability slices, 25 in Yapig and 16 in PhpPlanner respectively. The preliminary test case generation step built test cases for 22 vulnerability slices out of 41.

Test case perturbation expanded this initial set of test cases into a larger set, consisting in 300 distinct test cases for each vulnerability slice. Then, all these training test cases have been abstracted and merged into the corresponding safe model, the knowledge base of the security oracle.

B. Assessment Test Cases

In order to measure the capability of the security oracle to correctly classify the output of test cases, we needed new test cases, different from the training tests already used to build the model. We call these new tests the *assessment test cases*. Assessment test suite will contain both safe test cases and successful attacks, in equal number.

To generate attacks, we could resort to available techniques for attack generation, such as [13], [9], [8], [11]. However, since we need attacks to be general (i.e., not just limited to inject "<script> tags) and we need safe tests and attacks to be balanced, we adopted an ad-hoc generation strategy, described in the following. However, the objective of the assessment is to evaluate the oracle and not to evaluate the generation strategy.

Code injection exploits are generated by using attack strings from a library of 50 known malicious fragments of HTML and JavaScript. They are code fragments used by Surribas [20] in a tool for penetration testing and black-box fuzzing. Attack strings represent different attack strategies, such as displaying new windows with content controlled by the attacker, extracting data from the current page or altering the content of the current page.

Test cases are turned into attacks by appending the strings taken from the attack library to input values. Injection is performed with the following injection operators:

- The attack string is concatenated to input value;
- The attack string is inserted into an input value, at a random position; and
- An input value is substituted with the attack string.

In order to build also a control group of safe test cases that differs from the group used for training, a similar injection procedure is repeated using a list of safe inputs. This list is composed by 15 strings randomly taken from the English vocabulary and 15 integer numbers selected in a random way.

These attacks and safe tests are the candidate *assessment test cases*. They are run on the applications under test, to check if they still cover the vulnerability slices. Those tests that do not cover any vulnerability slice are removed from the assessment test suite. After filtering, our candidate assessment test suite consists of 26,170 tests.

To ensure a fair empirical validation, the test suite needs to be balanced and, for each vulnerability slice, it should contain the same number of safe tests and successful attacks. However, among all the vulnerability slices detected by static analysis, there are few for which the generation procedure described above (based on string concatenation) fails to build either successful attacks or safe tests. Thus, these slices are discarded from the experiment. After the balancing filtering, the final assessment test suite consists of 12,814 test cases.

The assessment test suite is eventually executed and output pages are saved. Pages are then manually inspected, to

verify if injection has been correctly performed. To speed up manual verification, we grouped identical output pages into equivalence classes, and we checked only one representative for each class (sometimes, altered input values did not cause changes in the final web pages).

Then, we used the security oracle to classify the output of assessment test cases and we compared the oracle classification with the manual classification.

C. Empirical Results

Experimental results on the two cases study applications are reported in Table II, left-hand side for Yapig and right-hand side for PhpPlanner. The Table reports a row for each vulnerability slice (vulnerability id in the first column, missing ids are related to those slices for which no assessment tests have been generated) considered in this experiment. The Table shows how many assessment test cases has been generated in the second column. Third and fourth columns summarize the oracle performance in terms of precision and recall. Eventually, the performance of the oracle in classifying test cases is reported in details, *True Positives* (TP in the Tables) are the correctly identified attacks, *False Positives* (FP) are the safe tests detected as successful attacks. *True Negatives* (TN) are the safe tests correctly identified and *False Negatives* (FN) are the successful attacks identified as safe tests.

While many assessment test cases can be generated for some slices (e.g., more than 1,000 tests for vulnerability slices 4, 6 in Yapig, more than 700 tests for majority of the candidate vulnerability slices in PhpPlanner), only few tests are built for other slices (8 tests for vulnerability slice 8). This is due to the presence of complex conditions that control the execution of statements in some vulnerability slices. The generation procedure, in fact, consumes a large portion of the time budget attempting to satisfy these conditions, resulting in few assessment tests created.

Analyzing the obtained results on Yapig (Table II left-hand side), we notice that precision is around 50% or higher, with peaks of 86%-88%, for all the candidate vulnerability slices, while it reaches 100% in one case. As the oracle reported no false negatives, the recall is always 100%, meaning that no real attacks are missed by the classifier. This implies that Circe has shown good performances in detecting attacks, with a fair number of false positives.

Manual inspection revealed that safe tests classified as attacks (false positives) are often due to malformed HTML in the output pages. Since malformed pages do not satisfy the safe model, they are classified as attacks. False positives in vulnerability slice 5, for example, are caused by a programming defect that makes the same web page contain two root <html> tags. This is a structural difference with respect to the safe model, that Circe classifies as an attack. Though we manually classified these cases as false positives because they are clearly not code injections, they still expose programming defects that require a fix. A more exhaustive training that would have also included malformed outputs in the training test cases would have correctly classified these cases. It is questionable, however, if a security oracle should model such faulty behavior as safe execution.

Manual inspection revealed no false negatives, meaning that no attack was erroneously classified as safe execution by Circe. In particular, Circe has shown to be effective in recognizing different attack patterns for the same page. In case of vulnerability slice 13, for example, attack patterns based both on <script> and on <img> tags have been correctly classified as successful attacks. This means that the oracle is general enough to be independent from the kind of HTML fragments that are injected. As in the case of vulnerability 5, the only false positive on vulnerability 13 was due to the presence of two root <html> tags in the same page.

On PhpPlanner (Table II right-hand side), precision was 100% only in one case (vulnerability slice 22). In all the other cases, precision was around 50% or higher, while the recall was always 100%.

Circe scored true positives for all the slices (from 15 to 22). By manually inspecting the application source code involved in these vulnerability slices, we noticed a recurring vulnerable pattern involving the special PHP variable *$_SERVER[PHP_SELF]*, which is used to generate dynamic forms. This special variable implements a sort of introspection, as it contains the complete URL used by the incoming HTTP request, which indicates the PHP page that is currently under execution. As such, this string also contains all the input values passed (by GET) to the current page. The application under test uses the content of this variable, without proper validation, to construct the value of parameter *action* in a form. This causes any malicious code fragment present in the incoming request to appear in the dynamic form. This alteration of the regular structure of the resulting HTML page is correctly recognized as an attack by the oracle.

The same pattern, however, (except for vulnerability 22 which is slightly different), has been recognized as an attack by the oracle even on safe tests. In fact, when the introspection variable is printed, the *action* parameter of the form contains the novel values that, even if safe, are still not matching those values observed during the learning phase.

Also for PhpPlanner, manual inspection revealed no false negatives.

D. Discussion

Considering the purpose of a security oracle, recall can be considered more important than precision. False alarms due to non-perfect precision would require additional effort by developers, who are supposed to manually analyze them. Non-perfect recall, however, would represent a more dangerous case of actual attacks that are overlooked, because classified as safe executions. In this sense, Circe represents a satisfactory trade off, as recall is always 100% and precision is still relatively high.

A possible strategy to increase precision might be dedicated to improve the training phase and adopt a larger set of training test cases. A wider training test suite would make the safe model more general, improving the performance in terms

Id	Tests	Precision	Recall	TP	FP	TN	FN
4	2,420	50%	100%	1,210	1,202	8	0
5	980	50%	100%	490	488	2	0
6	2,366	50%	100%	1,183	1,182	1	0
7	20	100%	100%	10	0	10	0
8	8	50%	100%	4	4	0	0
12	38	86%	100%	19	3	16	0
13	14	88%	100%	7	1	6	0

(a) Yapig

Id	Tests	Precision	Recall	TP	FP	TN	FN
15	960	50%	100%	480	471	9	0
16	374	91%	100%	187	18	169	0
17	900	51%	100%	450	433	17	0
18	920	50%	100%	460	454	6	0
19	1,108	51%	100%	554	543	11	0
20	960	51%	100%	480	460	20	0
21	1,020	51%	100%	510	494	16	0
22	726	100%	100%	363	0	363	0

(b) PhpPlanner

TABLE II
EMPIRICAL DATA ON THE CLASSIFICATION PERFORMANCE OF CIRCE (A: YAPIG, B: PHPPLANNER).

of precision. On the other hand, a too abstract safe model, however, would potentially cause too many false negatives and decrease the performance in terms of recall. Conversely, insufficient training guarantees perfect recall but at cost of low precision, because the resulting safe model would be too specific to the few training tests and any small deviation from them would be classified as an attack, even if perfectly safe. Then, an appropriate training is fundamental to reach an adequate compromise between precision and recall. In our case, appropriate training is guaranteed by the coverage criterion, that is based on the result of static analysis. Alternative criteria could be considered, such as branch coverage, def-use chain coverage, and others.

Another key point is the correct abstraction function adopted to build the safe model which is used by Circe. We decided to drop non-essential tags (such as formatting tags), but to keep all attributes and attribute values, and to represent recurring structure using patterns. This approach is effective in all those cases in which attribute values are defined statically and never change. However, it could be less effective when attributes are defined dynamically, using values that depend on the application state. To address the problem, in a future extension of the safe model, the characterization used for dynamic elements should be abstract enough to avoid false positives, while sufficiently generic not to miss any attack.

The adopted coverage criterion based on vulnerabilities represents a limitation of our experimental validation. In fact, those real vulnerabilities that are not reported by static analysis are not considered in the assessment. Nonetheless, a safe model should be able to completely describe the safe execution of a web application. All the pages that deviate from this model would be classified as potentially attacked, even in the presence of unknown vulnerabilities.

VIII. RELATED WORK

A fundamental part of security testing is deciding about successful attacks, i.e. when a test case is able to inject malicious code. Initially, checking code injection was a manual task delegated to programmers. For instance, in the work by Tappenden et al. [21] security testing is approached with an agile methodology using HTTP-unit, while verification of test outcomes is left as manual task.

Other approaches provide a higher level of automation. In [12], a library of documented attacks is used to generate valid inputs for a web application. A symbolic data base is implemented to propagate tainted status of input values through the data base to the final attack sinks. First stage of the oracle adopts dynamic taint analysis to verify if tainted data are used in a sink, while second stage performs a comparison of safe pages with pages generated by candidate attacks. This check consists in verifying if pages differ with respect to "script-inducing constructs", i.e. new scripts or different *href* attributes.

In other works [13], [9], [8], [11], the oracle consists in checking if a response page contains the same <*script*> tag passed as input. McAllister et al. [13] adopt a black-box approach to detect XSS vulnerabilities. Data collected during the interaction with real users are subject to fuzzing, so as to increase test coverage. The oracle for XSS attacks checks if the script passed as input is also present in the output page. We adopt a similar approach to artificially multiply the limited amount of test cases initially available, in order to make the safe model more general.

In Huang et. al. [9] data-entry-points are identified in web applications and attack patterns are used on them. The oracle consists in checking that input data containing the "<script>" substrings are sanitized before using them in output construction.

The paper by Halfond et al. [8] presents a complete approach to identify XSS and SQLI vulnerabilities in Java web applications. (1) Input vectors are identified as parameters read by the page under analysis, together with their expected type and domain. Flow analysis is resorted to group input vectors and domains into input interfaces. Then, (2) attack patterns are used to generate many attacks for these interfaces. Eventually, (3) page execution is monitored and HTTP response is inspected to verify if attacks are successful, i.e. the executed SQL statement is dangerous or new HTML scripts have been injected. Also in this case, the oracle consists in detecting if the response page contains a script tag injected by input data.

In [11], a scanner identifies input fields on forms which are later used to mount attacks. The oracle consists in detecting whether a script injected in a form appears in the response page. As commented by the authors, this approach suffers false negatives.

Limiting the check to injected script tags guarantees high precision, but recall may be low, because attacks based on other tags may not be detected by these oracles. While [13], [9], [8], [11] verify the that the injected "<script>" tag is present in the output page, we adopt a more general approach,

in fact we detect a larger class of injections, possibly involving any HTML tag. We build a model of a safe execution that is general enough to capture any form of code injection that does not satisfy the model itself.

A different kind of security oracle is adopted in other works on mutation testing [15], [16], [17]. Several mutation operators are defined to expose SQL-injection vulnerabilities on JSP applications [15], format string bugs on C [16] and XSS vulnerabilities on PHP code [17]. While these works rely on mutation to evaluate test case adequacy as the ability to kill mutants, we take advantage of mutation to perturb existing test cases and increase coverage.

A related work by Sprenkle et al. [19] compared 22 oracles in terms of precision and recall in revealing seeded and real faults on web applications. Even if related, their objective was quite different. They were interested in general purpose bugs, and not in security problems. In fact, their oracles ignore the *"programming elements"* such as scripts, that are important to consider when revealing injections attacks.

The work presented in this paper is built on top of previous papers [1], [2], where we presented our approach for generating input values for the test cases, but the concept of the security oracle was missing. In this work, we assume that input values are available and we focus on evaluating response pages, i.e. on classifying them as safe executions or successful attacks. This paper extends a preliminary workshop paper [3] that sketched the initial intuition of the security oracle, by refining the concept of the security oracle and by conducting the experimental assessment.

IX. Conclusion

In this paper we presented Circe, an approach for building a security oracle for one of the most prominent class of code injection vulnerabilities, i.e. Cross-site scripting. The oracle is based on the *safe model*, a model of safe executions of the application under test. The oracle classifies any execution that violates this model as an attack. Circe complements existing work on input value generation and represents a valuable support for the maintenance activity devoted to fix security bugs, by automating the decision on whether the system under analysis passes security tests.

The proposed security oracle has been assessed on two open source PHP applications, with good performances in terms of precision and recall. As future work, we intend to investigate the validity of Circe on case studies that are more and more dynamic, to verify if the oracle is still able to accurately distinguish between safe executions and attacks. Moreover, we intend to study how to extend Circe to cope with Ajax 2.0 web applications.

References

[1] Avancini, A., Ceccato, M.: Towards security testing with taint analysis and genetic algorithms. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, pp. 65–71. ACM (2010)

[2] Avancini, A., Ceccato, M.: Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities. In: Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on, pp. 85–94. IEEE (2011)

[3] Avancini, A., Ceccato, M.: Grammar based oracle for security testing of web applications. In: Proceedings of 7th International Workshop on Automation of Software Test (2012. To appear)

[4] Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on, pp. 39 –48 (2000). DOI 10.1109/SPIRE.2000.878178

[5] Christey, S., Martin, R.A.: Vulnerability type distributions in cve. Tech. rep., The MITRE Corporation (2006). URL http://cwe.mitre.org/documents/vuln-trends/index.html

[6] Cordy, J.: The TXL source transformation language. Science of Computer Programming **61**(3), 190–210 (2006)

[7] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for dpll(t). In: Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 4144, pp. 81–94. Springer Berlin/Heidelberg (2006)

[8] Halfond, W.G.J., Choudhary, S.R., Orso, A.: Improving penetration testing through static and dynamic analysis. Software Testing, Verification and Reliability **21**(3), 195–214 (2011). DOI 10.1002/stvr.450

[9] Huang, Y.W., Tsai, C.H., Lee, D., Kuo, S.Y.: Non-detrimental web application security scanning. In: Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, pp. 219 – 230 (2004). DOI 10.1109/ISSRE.2004.25

[10] Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 258–263. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/SP.2006.29

[11] Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: Secubat: a web vulnerability scanner. In: Proceedings of the 15th international conference on World Wide Web, WWW '06, pp. 247–256. ACM, New York, NY, USA (2006)

[12] Kieyzun, A., Guo, P., Jayaraman, K., Ernst, M.: Automatic creation of sql injection and cross-site scripting attacks. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pp. 199 –209 (2009). DOI 10.1109/ICSE.2009.5070521

[13] McAllister, S., Kirda, E., Kruegel, C.: Leveraging user interactions for in-depth testing of web applications. In: R. Lippmann, E. Kirda, A. Trachtenberg (eds.) Recent Advances in Intrusion Detection, *Lecture Notes in Computer Science*, vol. 5230, pp. 191–210. Springer Berlin / Heidelberg (2008)

[14] Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: Proceedings of the 10th European software engineering conference, pp. 263–272. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1081706.1081750

[15] Shahriar, H., Zulkernine, M.: Music: Mutation-based sql injection vulnerability checking. In: Quality Software, 2008. QSIC '08. The Eighth International Conference on, pp. 77 –86 (2008). DOI 10.1109/QSIC.2008.33

[16] Shahriar, H., Zulkernine, M.: Mutation-based testing of format string bugs. In: High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE, pp. 229 –238 (2008). DOI 10.1109/HASE.2008.8

[17] Shahriar, H., Zulkernine, M.: Mutec: Mutation-based testing of cross site scripting. In: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems, IWSESS '09, pp. 47–53. IEEE Computer Society, Washington, DC, USA (2009)

[18] Sharir, M., Pnueli, A.: Program Flow Analysis: Theory and Applications, chap. Two approaches to interprocedural data flow analysis, pp. 189–233. Prentice Hall (1981)

[19] Sprenkle, S., Pollock, L., Esquivel, H., Hazelwood, B., Ecott, S.: Automated oracle comparators for testingweb applications. In: Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, pp. 117–126 (2007). DOI 10.1109/ISSRE.2007.26

[20] Surribas, N.: Wapiti, web application vulnerability scanner/security auditor (2006-2010). URL http://www.ict-romulus.eu/web/wapiti

[21] Tappenden, A., Beatty, P., Miller, J., Geras, A., Smith, M.: Agile security testing of web-based systems via httpunit. In: Agile Conference, 2005. Proceedings, pp. 29 – 38 (2005). DOI 10.1109/ADC.2005.11

[22] Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 171–180. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1368088.1368112