Towards a Security Oracle Based on Tree Kernel Methods

Andrea Avancini and **Mariano Ceccato**[1]

Abstract.
The objective of software testing is to stress a program to reveal programming defects. Goal of security testing is, more specifically, to reveal defects that could lead to security problems. Security testing, however, has been mostly interested in the automatic generation of test cases that "try" to reveal a vulnerability, rather than assessing if test cases actually "managed" to reveal vulnerabilities.

In this paper, we cope with the latter problem. We investigated on the feasibility of using tree kernel methods to implement a classifier able to evaluate if a test case revealed a vulnerability, i.e. a security oracle for injection attacks. We compared seven different variants of tree kernel methods in terms of their effectiveness in detecting attacks.

1 Introduction

Among the programming defects that threat the reliability of web applications, those that concern security aspects are probably the most critical. In fact, vulnerabilities could be exploited by attackers to block the correct execution of a business service (denial of service) or to steal sensitive data, such as credit card numbers or medical records.

According to statistics on open source projects [2], one of the most severe class of vulnerabilities is Cross-site Scripting (XSS for short). An XSS vulnerability is exploited by input values that contain malicious HTML or JavaScript code. As result of the attack, the vulnerable page will contain the injected code and its content and/or behavior will be controlled by the attacker.

Security testing is a process intended to spot and verify security vulnerabilities, by showing an instance of input data that exposes the problem. A developer requested to fix a security defect could take advantage of a security test case to understand the problem (vulnerabilities often involve complex mechanics) and to elaborate a patch. Eventually, a security test can be resorted to assess if the maintenance task has been resolutive.

There is a number of approaches for security testing of web applications [15, 9, 10, 7, 6, 8], which are mainly focused on the test case generation phase. The problem of verifying if a test case actually exploits a vulnerability has given a marginal importance. In fact, checking if a test case has been able to exploit a vulnerability is either addressed by manual filtering [15] or in a way that is customized for a specific test case generation strategy. For example, in [9], verifying if a test case is a successful attack relies on the knowledge about how the test case has been constructed, i.e. if the output page contains the same JavaScript fragment that has been used to generate the test case itself.

In the present paper we address the problem of developing a security oracle, a classifier able to detect when a vulnerability is exploited by a test case, i.e. verifying if a test case is an instance of a successful attack. Our oracle is meant to be independent from the approach deployed to generate test cases, so that it can be reused in many different contexts.

We propose to construct the oracle resorting to tree kernel methods. The classifier is trained on a set of test cases containing both safe executions and successful attacks. In fact, it is quite common for a software project to document past defects (including vulnerabilities) that have already been fixed. The oracle is then deployed when generating new security test cases, intended to identify new vulnerability problems.

2 Web Application Vulnerabilities

Cross-site Scripting vulnerabilities are caused by improper or missing validation of input data (e.g., coming from the user). Input data may contain HTML fragments that, if appended to a web page, could alter its final content such that malicious code is injected.

Figure 1 shows an example of dynamic web page that contains a vulnerability. The code between "`<?PHP`" and "`?>`" is interpreted by the web server as PHP[2] code and executed when processing the web page. On incoming HTML requests, the web server executes the PHP code in the page, which processes input values and generates a textual output that represents the dynamic part of the requested page. On PHP termination, the web server sends the resulting output back to the requester web browser as an HTML response.

The example contains a reflected XSS vulnerability, i.e. there exists an execution flow along which the input value *param* is not adequately validated before being printed (*echo* statement in PHP) in the output page (line 15). Any code contained in this input value, if not properly validated, could be added to the current page and eventually executed.

The page accepts three parameters, *param*, *cardinality* and *op*, and adopts a quite common pattern, performing different actions according to the value of one of the parameters. In case *op* is set, the page will show a table, otherwise it will display a menu. The number of rows in the table and the number of links in the menu depend on the value of *cardinality*. Parameter *param* is just printed.

On lines 1–3, input values are read from the incoming HTML request (represented in PHP as the special associative array *$_GET*) and assigned to local variables *$p*, *$n* and *$op* respectively.

On lines 4–7, input values are validated. In case *$n* contains a value smaller than 1 or a string that does not represent a number,

[1] Fondazione Bruno Kessler, Trento, Italy, email: avancini, ceccato@fbk.eu

[2] Even if the example is in PHP, the approach is general and can be applied on web applications implemented with server-side language.

```
    <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>
```

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

the execution aborts (`die` statement at line 5). At line 7, the value of variable *$p* is validated. Validation, however, is done only when condition on line 6 holds, which is not sufficient 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` could skip the sanitization.

Depending on the value of variable *$op*, either a table (lines 8–12) or a menu (lines 13–14) is shown. Eventually, variable *$p* is printed at line 15 possibly causing a security threat, because of inadequate validation at lines 6–7.

An example of successful attack is represented by an HTML request containing the parameter *param* set to the subsequent JavaScript code:

```
<a href="" onclick="this.href=
'www.evil.com?data='%2Bdocument.cookie"> click
here</a>
```

When such value is appended on the response page, it alters the HTML structure (%2B is decoded as "+"), and a brand new link "*click here*" (i.e., <a> tag) is injected, pointing to an external web site controlled by the attacker (i.e., *www.evil.com*). In case such link is triggered by the legitimate user, his/her cookie is encoded as a HTML request parameter and sent to the attacker-controlled web site. With the stolen cookie, the attacker can pretend to impersonate the legitimate user.

The automatic generation of input values to test a vulnerable page can be addressed in quite a cheap way. After input generation, however, output needs to be validated, i.e. a *security oracle* is required to check whether code injection took place. In the subsequent sections we present our approach to use kernel methods to implement a security oracle, i.e. to classify executions of dynamic web pages as safe executions or as successful code injections. In the latter case, a vulnerability is detected.

3 Security Oracle

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

Web applications, however, are highly dynamic and their structure or content may vary a lot, even without code injection. For instance, on the running example of 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).

A web page can be represented by the parse tree of the corresponding HTML code. Thus, injection of malicious code corresponds to a change in the parse tree with respect to the intended structure. Figure 2 shows the parse trees of three HTML outputs of the running example. Figure 2(a) and (b) are the parse trees of safe executions that contain, respectively, a table with three lines and a menu with tree links. Figure 2(c), instead, represents the parse tree of the page under attack, a menu with two intended links and, in addition, a malicious link.

By looking at Figure 2, we can observe that the intrinsic variability of safe executions (e.g., between (a) and (b)) can be wider than the variability due to code injection (e.g., between (b) and (c)). So, a similarity metric may not be adequate to detect successful attacks.

The security oracle, then, should distinguish between those variations that are safe due to the dynamic behavior of the application and those variations caused by code injection due to successful attacks. The classifier must be trained with instances from both the classes. Under these assumptions, the security oracle problem can be formulated as a binary classification problem, that can be addressed by relying on kernel methods. In particular, we deal with parse trees, so kernel methods that fit better this problem definition are *tree kernels* [3].

We construct the security oracle according to the subsequent steps:

1. **Test case generation**: test cases are automatically generated for the web page under analysis. For this purpose, any test case generation approach is applicable in principle. We reused, however, a tool we developed in a previous work [1] that combines heuristics (genetic algorithm) and analytic solutions (sat solvers).
2. **Attack generation**: some test cases are turned into candidate attacks by adding selected attack strings to input values, taken from a library of malicious fragments of HTML and JavaScript. This library has been taken from a publicly available tool [14] for penetration testing and black-box fuzzing.
3. **Manual filtering**: test cases and candidate attacks are run on the web application under analysis. Results are manually classified as safe executions or successful injection attacks. The output pages are then parsed by using Txl [5] and the resulting HTML parse trees are stored.
4. **Training**: parse trees of successful attacks and safe executions are used respectively as *positive* and *negative* examples for the learning phase of the oracle.
5. **Classification**: the oracle is ready. To evaluate a new test case, the test must be executed on the application under analysis and the HTML output must be parsed. Eventually, the oracle relies on the kernel to classify the HTML parse tree either as safe execution or as successful attack.

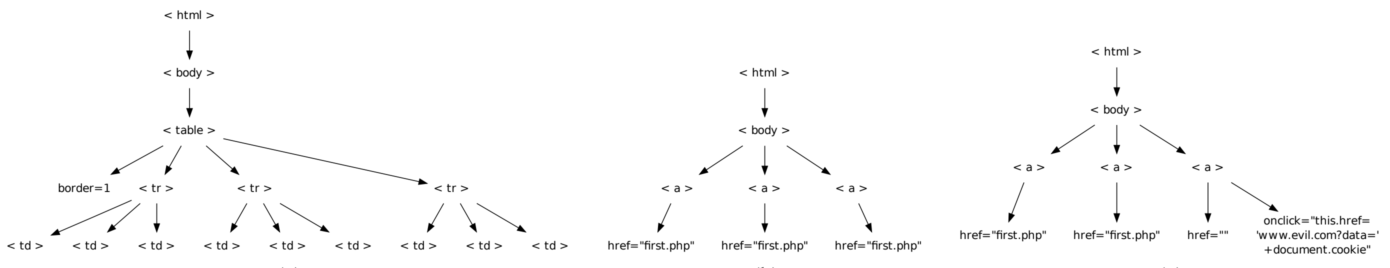

Figure 2. Parse trees of output pages for the running example. Trees (a) and (b) represent safe executions. Tree (c) represents an injection attack.

4 Preliminary Results

A preliminary experimentation has been conducted using SVM-light-TK[3] version 1.5 as kernel machine. This tool extends SVM-light tool[4] with kernel support, by implementing 7 different kernel methods:

- Standard (Tree) Kernel (SK) [3],
- Sub Tree Kernel (STK) [16],
- Subset Tree Kernel (SSTK) [4],
- Subset Tree Kernel (SSTK) with bag-of-words (BOW) feature [17],
- Partial Tree Kernel (PTK) [11],
- Partial Tree Kernel with no leaves (uPTK) [12] and
- String Kernel (StrK) [13].

We tested the feasibility of our approach on a case study application, a simple web application from which the running example of Figure 1 has been extracted. It consists of a single PHP script of 37 lines of code which represents a typical pattern of a dynamic web page. It implements two different functionalities, according to the value of an input parameter (generating a table or a sequence of links). The script contains two XSS vulnerabilities.

3470 test cases have been generated for the application under test. The test cases have been manually filtered into 600 safe executions and 60 code injection attacks, to respect a 1:10 proportion among the two classes[5]. This corpus of data has been randomly split in two parts, 50% for training and 50% for assessment. While splitting data, we took care of splitting attacks uniformly between the two parts.

Tuning of cost-factor value has been achieved with the following procedure. Initially, only 80% of the training data (270 test cases, training set) has been used to build an initial model. The remaining 20% (60 test cases, tuning) have been used to tune the cost-factor. We used the initial model to classify the tuning set by changing iteratively the cost-factor value from 1 to 50. We selected the optimal cost-factor value as the one that shown the best trade off between precision and recall in classifying the tuning data set. In case of identical results, cost-factor value that corresponds to the shortest execution time has been chosen.

Eventually, the final model has been constructed by using all the training data (training set and tuning set) for learning, applying the optimal cost-factor value. After the learning phase, performances of the final security oracle have been assessed on the assessment data set.

[3] http://disi.unitn.it/moschitti/Tree-Kernel.htm
[4] http://www.joachims.org/
[5] Generating attacks is usually harder than generating safe tests

Kernel	Optimal Cost-factor	Precision	Recall	F-measure
SK	1	100%	78%	88%
STK	20	7%	100%	13%
SSTK	1	100%	78%	88%
SSTK + BOW	1	100%	78%	88%
PTK	8	100%	17%	30%
uPTK	7	100%	39%	56%
StrK	1	100%	0%	0%

Table 1. Experimental results.

Table 1 reports experimental results collected on the case study applications for the 7 different kernel methods. The first column contains the name of the kernel method used, while the second column reports the optimal cost-factor value that has been chosen to run the experiment. Third, fourth and fifth columns report precision, recall and F-measure, obtained by running the classifier on the assessment data set. Results obtained by running String Kernel (StrK) method have been added for completeness, despite the fact that the generated parse trees do not exactly fit the intended input format for this method (trees instead of sequences of characters).

The best results have been achieved by three methods, SK, SSTK and SSTK + BOW. By running these methods, reported precision and recall have been 100% and 78% respectively, meaning that all the test cases that have been classified as attacks (18) are real attacks, while 5 attacks have been classified as safe tests. After manual inspection, we discovered the reason for not obtaining 100% recall. Despite the attack library [14] contains several distinct HTML syntactic elements, we noticed that the training set contained no instances of attacks with the same HTML syntactic structure used in the 5 misclassified attacks. A richer training set, containing at least one instance of any possible syntactic form of attacks, would have improved the performance of the oracle.

Despite this limit, however, the classifier assigned to the misclassified attacks a prediction value that was closer to the positive class (true attacks) rather than to the negative class (safe test cases). So, a revision of the classification threshold may be beneficial.

Among the other tree kernel methods, the best results have been obtained by uPTK. 9 attacks out of 23 have been classified in the correct way, achieving a high precision (100%) but a fairly low recall (39%). PTK method performed slightly worse, obtaining equal precision (100%) but even lower recall (17%). In fact, just 4 attacks have been correctly recognized by this method.

The remaining two methods, StrK and STK, reported the worst performance. In case of StrK, poor results were expected since the input format adopted is not perfectly suitable for this method, as the method classified all the candidates as safe test cases (no attacks reported means 100% precision and 0% recall). For STK instead, we observed an unstable behavior with respect to different cost-factor values. For cost-factor values lower or equal than 8, all the objects in the data set are classified as safe test cases (100% precision and 0% recall) while, for values greater than 8, all the tests are classified as attacks (low precision and 100% recall).

5 Related Works

A fundamental problem of security testing is deciding about successful attacks, i.e. when a test case is able to inject malicious code and reveal a defect. Initially, checking code injection was a manual task delegated to programmers. For instance, in the work by Tappenden et al. [15], security testing is approached with an agile methodology using HTTP-unit, while verification of test outcomes is a manual task.

Other approaches provide a higher level of automation. In [9], 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. A first stage of the oracle adopts dynamic taint analysis to verify if tainted data are used in a sink, while a 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 [10, 6], the oracle consists in checking if a response page contains the same <*script*> tag passed as input. McAllister et al. [10] adopt a black-box approach to detect XSS vulnerabilities. Data collected during the interaction with real users are subjected 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.

The paper by Halfond et al. [6] presents a complete approach to identify XSS and SQLI vulnerabilities in Java web applications. (1) Input vectors are identified and grouped together with their 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. The oracle detects if the response page contains the same script tag that was injected in the input data.

Limiting the check to injected script tags guarantee a high precision, but recall may be low, because of vulnerabilities depending on other tags may not be detected by these oracles. Our approach is more general, because it relies on structural differences among safe executions and attacks, that are general enough to capture different forms of code injection.

6 Conclusion

In this paper, we presented a preliminary investigation on using kernel methods for implementing a security oracle for web applications. The proposed security oracle has been assessed on a simple PHP application, with good performances in terms of precision and recall. From this initial experiment, we learned which tree kernel methods are the most appropriate to use in this domain. Moreover, we identified promising directions on how to improve our approach in terms of (1) more complete training sets and (2) customized classification threshold for the kernel methods.

As future works, we intend to experiment with customized kernel methods to improve the performance of the security oracle. Moreover, we plan to move on to real-world web applications (possibly written using different programming languages), to assess our approach also on bigger and realistic web applications.

Acknowledgements

The research described in this paper has been partially supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the grants #247758: ETERNALS – Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, and #288024: LIMOSINE – Linguistically Motivated Semantic aggregation engiNes.

REFERENCES

[1] A. Avancini and M. Ceccato, '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).

[2] S Christey and R A Martin, 'Vulnerability type distributions in cve', Technical report, The MITRE Corporation, (2006).

[3] Michael Collins and Nigel Duffy, 'Convolution kernels for natural language', in *Advances in Neural Information Processing Systems 14*, pp. 625–632. MIT Press, (2001).

[4] Michael Collins and Nigel Duffy, 'New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron', in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pp. 263–270, Stroudsburg, PA, USA, (2002). Association for Computational Linguistics.

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

[6] William G. J. Halfond, Shauvik Roy Choudhary, and Alessandro Orso, 'Improving penetration testing through static and dynamic analysis', *Software Testing, Verification and Reliability*, **21**(3), 195–214, (2011).

[7] Yao-Wen Huang, Chung-Hung Tsai, D.T. Lee, and Sy-Yen Kuo, 'Non-detrimental web application security scanning', in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp. 219 – 230, (nov. 2004).

[8] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic, 'Secubat: a web vulnerability scanner', in *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pp. 247–256, New York, NY, USA, (2006). ACM.

[9] A. Kieyzun, P.J. Guo, K. Jayaraman, and M.D. Ernst, 'Automatic creation of sql injection and cross-site scripting attacks', in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 199 –209, (may 2009).

[10] Sean McAllister, Engin Kirda, and Christopher Kruegel, 'Leveraging user interactions for in-depth testing of web applications', in *Recent Advances in Intrusion Detection*, eds., Richard Lippmann, Engin Kirda, and Ari Trachtenberg, volume 5230 of *Lecture Notes in Computer Science*, 191–210, Springer Berlin / Heidelberg, (2008).

[11] Alessandro Moschitti, 'Efficient convolution kernels for dependency and constituent syntactic trees', in *Proceedings of the 17th European conference on Machine Learning*, ECML'06, pp. 318–329, Berlin, Heidelberg, (2006). Springer-Verlag.

[12] Aliaksei Severyn and Alessandro Moschitti, 'Large-scale support vector learning with structural kernels', in *ECML/PKDD (3)*, pp. 229–244, (2010).

[13] John Shawe-Taylor and Nello Cristianini, *Kernel Methods for Pattern Analysis*, Cambridge University Press, New York, NY, USA, 2004.

[14] N. Surribas. Wapiti, web application vulnerability scanner/security auditor, 2006-2010.

[15] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, 'Agile security testing of web-based systems via httpunit', in *Agile Conference, 2005. Proceedings*, pp. 29 – 38, (july 2005).

[16] S.V.N. Vishwanathan and A.J. Smola, 'Fast kernels on strings and trees', in *In proceedings of Neural Information Processing Systems*, (2002).

[17] Dell Zhang and Wee Sun Lee, 'Question classification using support vector machines', in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, SIGIR '03, pp. 26–32, New York, NY, USA, (2003). ACM.