Towards Security Testing with Taint Analysis and Genetic Algorithms

Andrea Avancini
Fondazione Bruno Kessler–IRST
Trento, Italy

Mariano Ceccato
Fondazione Bruno Kessler–IRST
Trento, Italy

ABSTRACT

Cross site scripting is considered the major threat to the security of web applications. Removing vulnerabilities from existing web applications is a manual expensive task that would benefit from some level of automatic assistance. Static analysis represents a valuable support for security review, by suggesting candidate vulnerable points to be checked manually. However, potential benefits are quite limited when too many false positives, safe portions of code classified as vulnerable, are reported.

In this paper, we present a preliminary investigation on the integration of static analysis with genetic algorithms. We show that this approach can suggest candidate false positives reported by static analysis and provide input vectors that expose actual vulnerabilities, to be used as test cases in security testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Security testing, Genetic algorithms, Taint analysis, Cross site scripting

1. INTRODUCTION

Web applications are publicly exposed to attacks that can compromise their security. Exploitation of their vulnerabilities could lead to serious consequences such as denial of service, sensitive data disclosure, frauds and information loss.

According to recent studies (e.g., [4]) the most frequent occurring vulnerability is cross site scripting (XSS). This vulnerability is due to inadequate validation on user-provided data, allowing the injection of code snippets (script) in the web page under attack. Malicious code is then executed by the browser of legitimate users, causing possible disclosure of users' sensitive data (e.g., authentication tokens or credit card numbers).

Expert review is the most reliable approach to reveal and fix vulnerabilities, however it is time consuming and expensive. Static analysis [3] should support this task by proposing candidate vulnerabilities as starting point for manual review.

Nevertheless, static analysis fails to precisely evaluate all the cases where run-time information is required (e.g., pointers and reflective calls). In these cases, static analysis keeps a conservative approach, possibly causing safe cases to be reported as vulnerabilities. Whenever false alarms are too numerous, a serious limitation is posed to the potential benefit. In fact, the expert should spend valuable time in order to browse a large portion of code to filter the results of the tool. Moreover, static analysis usually reports just vulnerable points, it does not show how the vulnerability can be exploited. The definition of test cases is left to the manual intervention of the security engineer.

In this paper, we present an approach to improve static analysis by integrating genetic algorithms. Taint analysis [10] is used to identify possibly vulnerable execution paths (*target paths*) in the application control flow, by statically detecting when data coming from tainted sources (i.e., from a possible attacker) are not sanitized before been used in sensitive statements. Then evolutionary algorithms are resorted to define security test cases, those application inputs that make the execution flow traverse target paths. Even if these test cases do not implement actual attacks, they demonstrate that (and how) input validation can be practically bypassed. Tests represent a starting point for understanding and patching high priority security issues. Such issues would require immediate attention by the security expert, because an attacker could develop a successful attack on top of them.

In case the proposed approach is not able to provide a test case that executes a specific target path, either the path is too hard for a genetic algorithm or the path has no solution at all, because it is infeasible (the path will never be executed). Issues in this second category should be inspected manually, after high priority ones have been fixed.

After some background on cross site scripting and taint analysis on Section 2, the evolutionary approach is presented in Section 3. Then, Section 4 presents the tool implementation and experimental results. After commenting the related works (Section 5), Section 6 closes the paper.

2. STATIC ANALYSIS

2.1 Cross site scripting

Cross site scripting vulnerabilities are caused by improper validation of input data (e.g., coming from the user). Data may contain HTML fragments that could flush to the web page, altering the resulting content such that malicious code is injected. When executed by the user browser, such code may disclose sensitive data to third parties.

Figure 1 shows a portion of PHP code that contains a reflected XSS vulnerability. This page receives some parameters (i.e., *firstname* and *surname*) from a previous page, and displays their values. PHP provides the function *htmlspecialchars* to sanitize strings. It changes special HTML characters (e.g. "<", ">" and """) in their encoded form ("<", ">", and """), safe when used in a web page. However, the example fails in using it properly.

```
1  $a = $_GET["firstname"];
2  $b = $_GET["surname"];
3  if ( strpos($a, "<script") ) {
4      $a=htmlspecialchars($a);
   }

5  if (isset($b))
6      $go_on_b = true;
   else
7      $go_on_b = false;

8  if ( $go_on_b ) {
9    $b=htmlspecialchars($b);
   }

10 echo $a;          //sink
11 if ( $go_on_b ) {
12     echo $b;      //sink
   }
```

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

Values of parameters *firstname* and *surname* are assigned respectively to variables *$a* and *$b* (on lines 1 and 2), they are later printed on the page (lines 10 and 12). Printing *$b* is safe, because whenever it is assigned a value, it is also sanitized (line 9). Conversely, the value of *$a* is only conditionally sanitized (line 4), but the condition on line 3 fails to cover all the dangerous cases. The vulnerability can be exploited by an execution path that reaches the sink statement on line 10, while skipping sanitization on line 4.

An example of attack vector is represented by passing a value for *firstname* equal to *click here*, because it alters the page structure. A brand new link (the <a> tag) is injected, pointing to an external web site controlled by the attacker (i.e., *evil.php*). In case such link is triggered by the legitimate user, his/her cookie is sent to the attacker site. With the stolen cookie, the attacker can pretend to be the legitimate user, hijack his/her session and access his/her sensitive data on the web-site under attack.

2.2 Taint analysis

Taint analysis tracks the tainted/untainted status of variables throughout the application control flow. A vulnerability is reported whenever a possibly tainted variable is used in a sensitive (sink) statement without been validated. In the case of XSS [21], tainted values are those that come from the external world (data base and user input) and sinks are all the print statements that append a string into the web page. Tainted status is propagated on assignments and expressions, and it is blocked on sanitization operations (e.g., function *htmlspecialchars* in PHP).

Node	IN[n]	GEN[n]	KILL[n]	OUT[n]
1	ϕ	{$a}	ϕ	{$a}
2	{$a}	{$b}	ϕ	{$a, $b}
3	{$a, $b}	ϕ	ϕ	{$a, $b}
4	{$a, $b}	ϕ	{$a}	{$b}
5	{$a, $b} $\cup$ {$b}	ϕ	ϕ	{$a, $b}
6	{$a, $b}	ϕ	{$go_on_b}	{$a, $b}
7	{$a, $b}	ϕ	{$go_on_b}	{$a, $b}
8	{$a, $b}	ϕ	ϕ	{$a, $b}
9	{$a, $b}	ϕ	{$b}	{$a}
10	{$a, $b} $\cup$ {$a}	ϕ	ϕ	{**$a**, $b}
11	{$a, $b}	ϕ	ϕ	{$a, $b}
12	{$a, $b}	ϕ	ϕ	{$a, **$b**}

Table 1: Taint analysis result for the running example, vulnerabilities are *$a*@10 and *$b*@12.

Taint analysis is formulated as a flow analysis [17] problem, where the information been propagated in the control flow graph is the set of variables holding tainted values. The information generated and killed at each node (statement) n can be defined as follow:

$$GEN[n] = \{v \mid \textit{statement } n \textit{ assigns a tainted value to } v\} \quad (1)$$

$$KILL[n] = \{v \mid \textit{statement } n \textit{ sanitizes the value of } v\} \quad (2)$$

The flow propagation is in the forward direction, with union as the meet operator at junction nodes. Flow analysis terminates when the following equations produce the least fixpoint:

$$IN[n] = \bigcup_{p \in pred(n)} OUT[p] \quad (3)$$

$$OUT[n] = GEN[n] \cup (IN[n] \setminus KILL[n]) \quad (4)$$

where *pred(n)* indicates the set of nodes that precede immediately n in the control flow graph. A vulnerability is reported when a tainted value reaches a sink statement:

$$n \textit{ is a sink for variable } v \quad (5)$$

$$v \in OUT[n] \quad (6)$$

Figure 2 shows the control flow graph for the running example, while Table 1 reports flow values computed by taint analysis. On node 1, *$a* is assigned a tainted value so OUT[1], the set of tainted variables at node 1, is {*$a*}. This set propagates to the successor (see Figure 2), node 2, as IN[2]. On node 2, *$b* is assigned a tainted values, so OUT[2]={*$a*,*$b*} propagates to node 3 as IN[3]. Node 3 does not change flow values, so IN[3]=OUT[3]={*$a*,*$b*} propagates to the successor. Since node 4 sanitizes *$a*, such variable is removed from OUT[4] (it is killed) and only *$b*

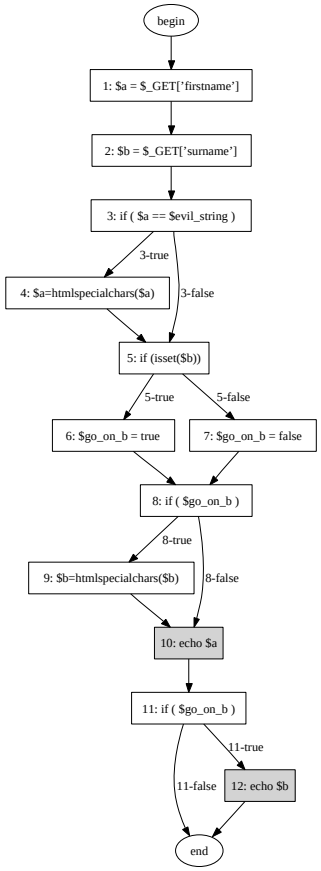

Figure 2: Control flow graph for the running example.

propagates. Node 5 is a special case, because it has two incoming edges, from nodes 3 and 4. On node 5, incoming flow (IN[5]) is computed as the union of outgoing flows from the predecessors (nodes 3 and 4):

$$IN[5] = OUT[3] \cup OUT[4] = \{\$a, \$b\} \cup \{\$b\} = \{\$a, \$b\}$$

Since static analysis is not able to tell which one of the two branches would be executed and there exist at least one flow that skips sanitization, the worst case is assumed and *$a* is conservatively considered tainted on node 5.

Even if later, on node 9, *$b* is sanitized, a similar argument holds for node 10, two incoming flows should be merged:

$$IN[10] = OUT[8] \cup OUT[9] = \{\$a, \$b\} \cup \{\$a\} = \{\$a, \$b\}$$

Since there exists a potential path where *$b* is not sanitized (*8-false* branch in Figure 2), *$b* can not be considered safe, so it is propagated as tainted on node 10.

When flow analysis reaches the fix point, OUT on sink nodes is inspected. In the example, equations (5) and (6) are satisfied in two cases, so two vulnerabilities are reported. They are *$a* on statement 10 and *$b* on statement 12.

While the vulnerability on *$a* is correct (a possible exploitation is discussed earlier), the result for *$b* is a false positive. In fact, the sink statement (line 12) depends on the condition *$go_on_b* = `true`, whereas skipping sanitization (line 9) requires *$go_on_b* = `false`. These two opposite conditions never hold simultaneously, so the path containing *8-false* and *11-true* is infeasible. The reported vulnerability can not be exploited by an attacker.

A more accurate identification of the infeasible paths would have required more complex analysis. Instead of going for unnecessary complicated (but still inaccurate) static analysis, an alternative approach is to stay simple and conservative, but generate test cases as input vectors for feasible paths in a second step, using genetic algorithms.

3. GENETIC ALGORITHMS FOR PATH SENSITIZATION

The act of finding a set of solutions to a path predicate expression is defined as *path sensitization* [1]. Genetic algorithms are resorted to find those inputs for the application that make its execution traverse target paths, identified by taint analysis. If such inputs can be found we have that:

1. The corresponding vulnerable path is feasible, it corresponds to a vulnerability that exposes the application to exploitation by attackers; and
2. Such inputs represent a test case. The application currently does not pass such test case, it should be fixed so as to pass the test.

```
1 population = generateRandomPopulation();
2 for (T in targetPaths) {
3     while ( not covered(T) AND
                    attempt < maxTry ) {
4         selection = select(population);
5         offspring = crossOver(selection);
6         population = mutate(offspring);
7         attempt = attempt + 1;
      }
  }
```

Figure 3: Genetic algorithm for path sensitization

Intuitively, a genetic algorithm evolves a set of solutions by combining together the good ones in the hope of generating better ones, until the optimum solution is found. The details of the algorithm are shown in Figure 3. An initial population of random set of input values (line 1), is evolved until the maximum number of trials are completed or the solution is found (line 3). On each evolution iteration, a subset of the population is selected (line 4) to form the next population, by giving more chances to those individuals that are more likely to generate the final solution, i.e., they have a better value of the *fitness function*. Selected individuals are paired to generate offspring by *crossing over* their chromosomes (line 5) and by *mutating* them (line 6), in the hope of generating better solutions.

Chromosomes.

Individuals are represented as chromosomes. They contain the input values for the page under analysis. A chromosome is a set of pairs, each pair contains a parameter name and a parameter value. For example, the URL *"page.php?firstname=john&surname=smith"* corresponds to the chromosome {(*firstname*, *john*), (*surname*, *smith*)}. While parameter names can be found among parameters used in the web page source code, parameter values are random strings.

Fitness function.

The fitness function corresponds to the *approach level*, i.e. the amount of branches from the target path that are executed when the application is run with the inputs from the current individual. The solution for the target path is found when an individual is able to traverse 100% of the required branches. The more an individual is near to this condition, the higher value of fitness function will have.

Selection.

At each iteration a sample of the population is selected for evolution, the probability of selecting an individual for the next generation is proportional to the value of its fitness function. In other words, input values more near to cover the target path are more likely to be selected for contributing to the new generation. Then the new generation is subject to mutation for a possible better improvement of the fitness function.

One point cross over.

When two individuals are selected for crossing over, their chromosomes are divided in two pieces. Two brand new individuals are generated by recombining two halves together. In the subsequent example, chromosomes A and B have been split. C is the result of joining the first part of A with the second part of B, while D is the union of the remaining two parts:

Example:
$A : \{(firstname, john), (surname, smith), (\mathbf{age}, \mathbf{23})\}$
$B : \{(firstname, mark), (address, mainstreet), (\mathbf{job}, \mathbf{teacher})\}$
$\downarrow$
$C : \{(firstname, john), (surname, smith), (\mathbf{job}, \mathbf{teacher})\}$
$D : \{(firstname, mark), (address, mainstreet), (\mathbf{age}, \mathbf{23})\}$

In case, after crossing over, the same parameter appears twice in a chromosome (possibly with different values), one of them is randomly removed, keeping the chromosome valid.

Mutation of parameter value.

The value of a parameter is randomly changed. One pair in the chromosome is chosen with uniform probability and its parameter value is changed in two alternative ways. Either (1) one character of the string is randomly selected and substituted with a random character or (2) a random string is concatenated to the existing parameter value. In the first example the value of parameter *surname* is selected for mutation, its first character is changed from "*s*" to "*x*". In the second example the other alternative modification is resorted. The random string "*x3scr*" is appended to the parameter value.

Example:
$A_1 : \{(firstname, john), (surname, \mathbf{s}mith)\}$
$\downarrow$
$A_2 : \{(firstname, john), (surname, \mathbf{x}mith)\}$

$A_1 : \{(firstname, john), (surname, smith)\}$
$\downarrow$
$A_3 : \{(firstname, john), (surname, smith\mathbf{x3scr})\}$

Insertion of a new parameter.

A new pair is added to the chromosome. The parameter name is randomly selected among the available parameter names and its value is a randomly generated string. In the subsequent example the new chromosome A_2 is created by concatenating a new pair to the chromosome A_1.

Example:
$A_1 : \{(firstname, john), (surname, smith)\}$
$\downarrow$
$A_2 : \{(firstname, john), (surname, smith), (\mathbf{age}, \mathbf{3e})\}$

Removal of existing parameter.

A pair is randomly selected from the chromosome and removed from it. In the example, the second pair is removed, resulting in a new chromosome.

Example:
$A_1 : \{(firstname, john), (\mathbf{surname}, \mathbf{smith})\}$
$\downarrow$
$A_2 : \{(firstname, john)\}$

String generation.

All the constant strings that appear in the page source code are collected and stored into a pool. When a new random string is required, such string is either chosen from the constant pool (probability 1/2) or randomly generated (probability 1/2). In the latter case, the following algorithm is resorted. A character is randomly selected from a set containing 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 the second character has been added, 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 decades exponentially. In general the probability of generating a string of length n is $1/2^{n-1}$.

4. EXPERIMENTAL RESULTS

The approach has been implemented in a prototype and empirically evaluated on a case study.

4.1 Implementation

Taint analysis has been performed using Pixy [10], an open source tool for static analysis of PHP code. Pixy reports those control flow paths that reach vulnerable sinks and skip sanitization. They are target paths for the subsequent genetic search.

The application under analysis is instrumented. Probes are inserted on branches so that when a branch is traversed, the corresponding probe is triggered. Information about the traversed branches is stored in the resulting web page as an easily recognizable annotation.

The genetic algorithm described in Section 3 has been implemented in Java and run on the instrumented application. Parameters of the genetic algorithm have been set up according to what proposed in literature [8]. In particular an elitist approach has been adopted, with the 10% of the best individuals kept alive across generations. The population is composed of 70 individuals and evolution has been run over 500 generations. The cross-over probability has been set to $P_c = 0.7$ and mutation probability to $P_m = 0.01$.

The genetic algorithm works as a client application that simulates a web-browser. HTTP requests are sent to the instrumented server. Requests encode individuals as URLs, with parameter values passed by GET. For each request, a page is elaborated and returned by the server, together with data on traversed branches, to be used for the computation of the fitness function.

4.2 Empirical data

The proposed approach has been validated on a case study application, PhpNuke[1] version 6.9. It is an open source content management system implemented in PHP, with a persistent back-end on MySql. It contains 1,046 PHP source files for a total of 157,000 lines of code.

Among the XSS vulnerabilities reported by taint analysis, some are due to data stored in the data base, (persistent XSS) so they are out of the scope of the present investigation and they have not been considered. The remaining 9 cases have been analyzed with the genetic algorithm. Vulnerabilities, listed in the first column of Table 2, are named after the tainted variable that reaches an *echo* statement (variable *$confirmNewUser* causes three distinct vulnerabilities).

As sanity check, our approach has been compared with random testing, to show that our approach at least performs better than random. 50,000 random test cases have been generated and evaluated on the case study. Experimental results are summarized in Table 2. For each vulnerability, the table reports how many branches, among the ones in the target path (second column), have been covered by test cases from the genetic search (third column) and by random testing (fourth column).

While random testing succeeded just in one case, in four cases out of nine our approach managed to find a test case that traverses the full target path. When the genetic algorithm converged, a solution was always found in less than 50 iterations, suggesting that our estimation for the maximum attempts (i.e. 500) was appropriate. In these cases, the algorithm was able to find proper input values that satisfy fairly complex boolean conditions on decision points, so as to trigger the corresponding expected branches.

For the other five vulnerabilities, no solution could be found after 500 iterations. In particular, for *confirmNewUser1*, *confirmNewUser2* and *my_headlines*, the search was not able to go beyond a particularly hard branch. Such branch depends on complex constraints on input values, that our approach was not able to satisfy. For *finishNewUser*, the problem involved some checks performed by the application on the password selected during registration. The password was not only required to be long and complex but also equal to the string specified in the *password confirmation* field. For *edithome*, path conditions depended on values stored on the data base by other pages not considered in the search.

4.3 Considerations

In this experiment, the most baseline approach has been adopted, a simple fitness function and very generic mutation operators. In fact, the intended objective was to study how a genetic algorithm works in the elementary case. By identifying limitations and weak points, considerations are formulated on how to improve such basic approach.

[1] http://phpnuke.org/

Run-time configuration.

No simplifications or reductions are imposed on the code in order to be analyzed by the proposed approach. Assuming that the fully installed application runs in a production-like environment, all the identified input values are true positives, they are security test cases that should guide security patches. However, such test cases are strongly connected to the run-time set-up. In fact, in case the application is configured in a different way, different paths could be executed for the same input values. For this reason it is important to perform the analysis on the final configuration. For instance, in case the web application adopts a plug-in oriented architecture, the analysis should be repeated whenever new plug-ins are installed.

To improve the present approach, configuration could be considered as an additional dimension to be explored by the generic algorithm, so as to search for particularly dangerous application configurations that would be missed by the present analysis.

Fitness function.

A fairly basic fitness function has been used for this first assessment. The fitness function only counts how many branches are hit by an individual among the branches required by the target path (*approach level*). Even if quite simple, this measurement shown good performance, in many cases populations were able to converge to a solution.

Currently, if two individuals deviate from the target path at the same branch, they are given the same fitness function. However, an higher score should be given to the one that is more near to satisfy the missed branch condition. A more advanced fitness function should take into account the *branch distance*.

A more sophisticated fitness function is expected to bring better results than the experimented baseline. Solutions could be found for those populations that currently do not converge and, for those ones that do converge, solutions could be found faster, requiring less iterations.

Mutation operators.

Mutation operators are of limited help. Though they are very generic and mutation probability is very low, it is very unlikely that they could bring improvements on the fitness function, especially for strings with complex syntax. More specific mutation operators could be based on syntax definition, so as to keep a value valid while mutating it.

Local optima.

Sometimes, the initial population contains potentially good individuals that are very similar to a solution, because they differ just for few characters. However, such similarity is not reflected by the fitness function, so they are discarded because other individuals are selected to form the next generation. An early excessive reward to bad individuals makes the population lose valuable genetic data and the search terminates in a local optimum, thus failing to find the final solution (global optimum).

Specific mutation operators should be adopted to make the population leave local optima and a different fitness function should be designed to avoid over-fitting.

Vulnerability	Target branches	Genetic search		Random	
		Covered br.	Solved?	Covered br.	Solved?
confirmNewUser1	4	2	no	1	no
confirmNewUser2	4	2	no	1	no
confirmNewUser3	3	3	yes	1	no
finishNewUser	4	2	no	2	no
userinfo	2	2	yes	1	no
mail_password	5	5	yes	1	no
userinfo	2	2	yes	2	yes
edithome	3	2	no	2	no
my_headlines	5	2	no	1	no

Table 2: Empirical results on the case study.

Constrained strings.

Occasionally, input strings correspond to structured data having a specific syntax, such as date, values from predefined set (e.g., enumerations) or numeric values. Currently, syntax is not considered when generating string values. The probability that a random string hits a valid value is very low and many evolution cycles are spent just to search for correct values. We tried to limit this problem, by reusing string values from the constant strings available in the source code of the web page. Taking into consideration expected syntax when generating random strings may bring major improvements.

Soundness.

Our approach is sound in the sense that (1) test cases are generated only for feasible paths and (2) test cases actually trigger execution paths where a tainted variable is used in a sink statement. However, test cases trigger vulnerable paths, they do not trigger (exploit) vulnerabilities. Test cases are not instances of actual XSS attacks, as they are not meant to inject code in the web pages. Their purpose is to show the way input data can skip validation.

In order to generate test cases that are actual attacks, an additional condition should be verified at the sink statement, to require that the content of the tainted is valid java-script code. This has not been implemented and is left as future work.

Completeness.

The approach is not complete, in some cases no test could be generated but we can not tell whether the problems were due to infeasible paths or to the limitations of the current approach. In these cases, our approach does not differ from static analysis and manual inspection is required.

5. RELATED WORKS

The adoption of static analysis for identifying vulnerabilities was initially proposed as a way to support manual inspection [3]. Originally called type-state analysis [18], taint analysis has been largely adopted to detect inadequate or missing input validation, resulting in cross site scripting [22] [10], SQL-injection [9] and buffer overflow [16] vulnerabilities. In order to mitigate inaccuracy of pure taint analysis due to conservativity, more sophisticated analyses have been integrated, such as string analysis [22], program slicing [11], points-to analysis [12] and model checking [20]. The present paper takes a different direction, instead of going for a more complex but still inaccurate static analysis, candidate vulnerabilities reported by static analysis are subject to path sensitization to filter likely false positives and to find concrete instance of true positives.

Instead of statically searching for security faults, other approaches resort on monitoring the application (often calling it *dynamic analysis*) while it runs in production. For instance, in [5] the application execution is monitored by several anomaly reasoners and a firewall blocks those interactions classified as abnormal. The principal drawback in monitoring is represented by run-time and memory overhead. Program slicing has been proposed by Walter et al. [2] to limit code instrumentation only to the actual vulnerable part.

Genetic search has been used on procedural code [14], to generate new test cases and improve coverage, considering the distance from an uncovered structural properties (e.g., a branch or a def-use chain) as fitness function. This approach has been extended [19] to test object oriented code, by searching not only for input values, but also for a method invocation sequence. Del Grosso et al. [8] applied genetic algorithms to identify tests to expose buffer overflows using a complex fitness function. Tests are searched with high statement coverage, that execute lots of vulnerable statements and deeply nested code, and that write data as near as possible to the buffer boundary. An evolutionary inspired approach has been used also by Zulkernine et al. [7] on security, for a totally different purpose. Network scanner rules are searched that maximize the ability in revealing and classifying intrusions.

Concolic execution [15, 23, 13, 6] is an alternative popular approach that mix symbolic and concrete execution to generate test cases. The code under test is executed initially on random inputs and symbolic constrains on are collected at run-time on assignments and decision points (branches). When a branch is traversed that diverges from the target flow, the condition of such branch is negated and added to constraints collected so far and sent to a solver. The solver computes new input values, that make the execution avoid the wrong branch. Concolic execution suffers the limits of the adopted solver, that often (to guarantee decidability) is able to reason only on liner constraints.

6. CONCLUSION

In this paper, we presented a preliminary investigation on combining static analysis with a genetic algorithm to support security testing. Proper test cases have been gener-

ated for actual vulnerabilities, as input values that make the application traverse vulnerable control-flow paths. A tool prototype has been implemented and applied on a real PHP application. The proposed approach shown good performance in generating security tests for reflected cross site scripting vulnerabilities.

By observing the experimental results, considerations have been drown on promising directions for improvement. As future work, we intend to improve our approach, for example by using a more advanced fitness function that includes branch distance and by extending mutation and crossover operators. Moreover, we intend to investigate whether the presented technique is effective on a broader set of vulnerabilities, including persistent and DOM-based cross site scripting, SQL-injection and cross site request forgery.

7. REFERENCES

[1] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[2] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 39–50, New York, NY, USA, 2008. ACM.

[3] B. Chess and J. West. *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

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

[5] C. Criscione and S. Zanero. Masibty: an anomaly based intrusion prevention system for web applications. In *Black Hat Europe 2009*, 2009.

[6] D. E. Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[7] R. H. Gong, M. Zulkernine, and P. Abolmaesumi. A software implementation of a genetic algorithm based approach to network intrusion detection. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005. Sixth International Conference on*, pages 246–253, May 2005.

[8] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Computers and Operations Research*, 35(10):3125 – 3143, 2008. Special Issue: Search-based Software Engineering.

[9] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. 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*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[11] J. Krinke. Information flow control and taint analysis with dependence graphs. In *3rd International Workshop on Code Based Security Assessments (CoBaSSA 2007)*, pages 6–9, 2007.

[12] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 271–286, Berkeley, CA, USA, 2005. USENIX Association.

[13] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[14] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verifications, and Reliability*, 9:263–282, September 1999.

[15] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference*, pages 263–272, New York, NY, USA, 2005. ACM.

[16] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[17] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two approaches to interprocedural data flow analysis, pages 189–233. Prentice Hall, 1981.

[18] R. Strom and D. Yellin. Extending typestate checking using conditional liveness analysis. *Software Engineering, IEEE Transactions on*, 19(5):478–485, May 1993.

[19] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM.

[20] L. Wang, Q. Zhang, and P. Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 165–173, Sept. 2008.

[21] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[22] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 171–180, May 2008.

[23] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, New York, NY, USA, 2008. ACM.