Web Application Vulnerability Fuzzing Based On Improved Genetic Algorithm

Xinshi Zhou[1,2] , Bin Wu[1,2]
1. College of Cyberspace Security, Beijing University of Posts and Telecommunications
2. National Disaster Recovery Technology Engineering Laboratory, Beijing University of Posts and Telecommunications
Beijing, China
nickzhou@bupt.edu.cn, binwu@bupt.edu.cn

***Abstract*—Web fuzzing has always been an effective way to detect web vulnerabilities. Normally, traditional web fuzzing method mainly use limited test cases or generate test cases based on certain rules, which cause web fuzzing slow and inefficient. To solve this problem, we present improved genetic algorithm with a new mutation method to generate test cases. And the concept of preset functional units is proposed: test cases are divided into different functional units to ensure that the semantic structure will not be damaged during crossover and mutation. The experimental results show that the improved algorithm can generate better test cases than the standard genetic algorithm (SGA) and the adaptive genetic algorithm (AGA) and also detect more web vulnerabilities.**

Keywords—web fuzzing test;genetic algorithm;functional units

I. INTRODUCTION

In recent years, the number of web vulnerabilities has increased greatly, and new exploits are dangerous and easy to use. According to the latest "Top 10 Most Critical Web Application Security Risks" released by Open Web Application Security Project (OWASP) [1], web application vulnerabilities cause huge losses to the national economy every year. Therefore, how to find vulnerabilities in web applications faster and more effectively has become the focus of attention for enterprises and organizations.

One of the most effective way for detecting software services vulnerabilities is fuzzing test. Fuzzing is a semi-automated or automated process that consists in sending test cases that are more likely to cause error, and monitoring the response returned by the application in order to determine if there are vulnerabilities [2]. Traditional fuzzing methods use predetermined test cases or generate test cases randomly. Although they are very automatic, they are inefficient, and their ability to find vulnerabilities is determined by the quality and number of test cases. But with the rise of evolutionary algorithms, fuzzing of applications which utilize evolutionary algorithms to intelligently guide input generation have seen tremendous success. In the field of system software testing, there are many fuzzing methods based on genetic algorithms that improve the testing efficiency [3~5]. But the genetic algorithm is not perfect, depending on the testing scenario, different problems (e.g. population precocity, slow convergence [6]) occur. We introduced the genetic algorithm to the field of web fuzzing and improved the genetic algorithm to make it more suitable for web vulnerability detecting.

The method proposed in this paper is mainly focus on detecting SQL injection attacks and Cross-Site Scripting (XSS) attacks in web application. By preprocessing the test case, the test case is divided into preset functional units to ensure that the syntax and aggressiveness of the original test case are not destroyed. We perform crossover and mutation operations based on the functional units, and set a new mutation method: when the test case with lower fitness value mutated, imitate the mutation strategy of the best test case in current generation, thereby increasing the aggressiveness of the test case with low performance and speeding up the entire fuzzing progress.

II. RELATED WORKS

There are many methods for web fuzzing. We choose some of them to discuss their pros and cons. R. Hammersland et al. proposed a method for semi-automatic generation of pseudo random test data for web application fuzzing [7], they created a way of writing attack scripts for fuzzing to specify how the applications should be attacked, but their goal was to crash web applications by sending pseudo random test data, not detect web vulnerabilities like SQL injection, and they required manual analysis of logs and responses; J. Bozic et al. and Wang et al. proposed methods to detect XSS respectively [8~9], they were both based on analyze attack pattern or syntax and semantics structures of the attack grammars to guide the generation of test cases. J. Bozic et al. used a general structure for XSS attack vectors and added constraints to XSS attack grammar to produce test cases with better quality. Wang et al. automatically learned the structure of attack vectors from practical data analysis to modeling a structure model of attack vectors. They all have a good analysis of the attack syntax, but still the generated test cases have certain limitations based on the analysis results; Duchene et al. detected XSS vulnerabilities by generating test inputs using a combination of model inference and genetic algorithm based fuzzing, they used grammar productions of an attack grammar for XSS as chromosome in genetic algorithm [10]. The limitations are that their strategy required an expert to manually write the attack grammar used to generate payloads for their genetic algorithm and only for XSS.

978-1-7281-4390-3/20/$31.00 ©2020 IEEE

III. Implementation of Improved GA in Web Application Vulnerability Fuzzing

In this section, we will discuss how to integrate genetic algorithms into web application vulnerability fuzzing, explain the preprocessing phase of test cases and the fitness function. Finally, we show how crossover and our new mutation methods work.

A. Test Case Preprocessing

Genetic algorithm is a heuristic algorithm that simulates the evolution of nature. It uses chromosomes for crossover and mutation operations to produce a better next generation [6]. In order to perform those operations, chromosomes encoding(preprocessing) is the first problem to be solved, and it is also a key step when designing genetic algorithms. Software or network protocol fuzzing with GA use test cases that are mainly displayed in the form of numbers or unordered meaningless strings, and those can be simply encoded using binary coding or floating-point number coding [3,4]. However, web fuzzing with GA uses string form test cases which have a certain semantic structure and meaning. If web fuzzing test cases are simply processed in binary coding, it will destroy the syntax of the test case during crossover and mutation, and also make fuzzing meaningless by sending corrupted test cases. The purpose of the preprocessing phase in web fuzzing is to make the test cases available in genetic algorithms without damage their semantic structure.

Whether it is SQL injections or XSS, they all have a certain semantic structure. Each components of the structure have different functions. We define and name these components as different preset functional units. These units are like genes in a chromosome. The preprocessing method is to divide an input test case according to the preset semantic functional units. Using this method, genetic algorithms can easily handle input test cases with semantic structure, crossover and mutation operations only need to operate on each functional unit. The principle of dividing functional units is that the syntax of the test case cannot be damaged.

1) SQL injection Test Case Preprocessing

We define an input SQL injection test case T_{sql} is composed of four different type of functional units (i.e. $U_{S0}, U_{S1}, U_{S2}, U_{S3}$):

$$T_{sql} = U_{S0} \cup U_{S1} \cup U_{S2} \cup U_{S3} \quad (1)$$

The details of each U_S is in Table 1.

For instance, Let T_{sql} be the following SQL injection test case: `x' OR 1=1--`. Suppose the following SQL query is injected in name filed: `SELECT * FROM users WHERE name='' and password=''`, the resulting SQL becomes `SELECT * FROM users WHERE name='x' OR 1=1--' and password=''`. As a result, the `name` filed is closed by `x'` which represent close unit in T_{sql}, and start a new SQL statement: `OR 1=1`, which make the `where` clause is evaluated to be true because the logical expression "`1=1`" is always true and the rest of the SQL query are commented out by the comment unit, i.e. "`--`". Also, T_{sql} has no Other function unit. Therefore, the preprocessing result of T_{sql} can be summarized as shown in Figure 1.

TABLE I. Preset Functional Units of SQL Injection Test Case

Symbol	Name	Description	Example
U_{S0}	Close unit	To Close the vulnerable SQL query and start SQL injection attack	`1' ‖ eETd" ‖ xDd3‘)`, etc.
U_{S1}	Logical expression unit	A logical expression component in a SQL injection test case	`or 2>1 ‖ or 'x'='x ‖ AND 1=0`, etc.
U_{S2}	Other function unit	Additional operators or commands in the SQL injection test case	`UNION select sleep(1) ‖ UnIoN select 1,'09ac5d' ‖ extractvalue(1, select @@version)`, etc.
U_{S3}	Comment unit	Comment out the remaining SQL query behind injected test case	`-- ‖ # ‖ /*`, etc.

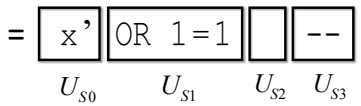


Fig. 1. Example of preprocessing result of a SQL injection test case

2) XSS Test Case Preprocessing

Same as SQL injection test case, an input XSS test case T_{xss} also is composed of four different type of functional units (i.e. $U_{X0}, U_{X1}, U_{X2}, U_{X3}$):

$$T_{xss} = U_{X0} \cup U_{X1} \cup U_{X2} \cup U_{X3} \quad (2)$$

These functional units are listed in Table 2.

TABLE II. Preset Functional Units of XSS Test Case

Symbol	Name	Description	Example
U_{X0}	Close unit	Tag delimiter at the beginning and end of XSS test case	'> ‖ "> ‖ <", etc.
U_{X1}	Tags unit	HTML tags in XSS test case	`<img src='abc'...> ‖ <iFrAme sRC='xxx'...>`, etc.
U_{X2}	Event unit	HTML events in XSS test case	`onload="..." ‖ onmouseover="..." ‖ onerror="..."`, etc.
U_{X3}	JavaScript code unit	JavaScript code in XSS test case	`alert("XSS") ‖ JaVaScRiPt:alert('XSS')`, etc.

Figure 2 shows an example of the preprocessing result of an input XSS test case T_{xss}:

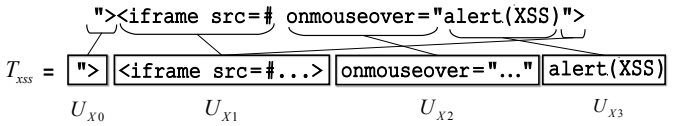


Fig. 2. Example of preprocessing result of an XSS test case

To be noticed, whether it is SQL injection or XSS, depending on the purpose and attack grammar of the input test case, it may not fully have all the functional units.

B. First Generation

We use the OWASP's SQL Injection Bypassing Cheat Sheet [11] and SuperCowPowers's "Data Hacking" GitHub repository [13] as the first generation of SQL injection test cases; We also use OWASP's XSS Filter Evasion Cheat Sheet [12] as XSS initial test cases. The test cases of the sample were chosen in order to cover a wide range of different SQL injection or XSS attacks.

C. Function of Fitness

The fitness function assesses how well a given test case is close to detect a vulnerability. In this paper, the fitness function should be universal, i.e. it should work for both SQL injection and XSS, and can be applied to other new web vulnerabilities in future work. The following evaluation principles should be considered:

- First, if a given test case successfully achieves a SQL injection or XSS, it is identified as a payload that can trigger the vulnerability. When such payload is found, the fitness value of this test case is set to the highest, and the fuzzing can be ended. For instance, a SQL injection test case triggered server error and successful return database fingerprint (e.g. "`MySQL: You have an error in your SQL syntax.`") or an XSS test case that successfully executed `alert()` script. This comprehensive judgment is represented by the symbol Σ.

- Furthermore, the similarity between input test cases and server-returned test cases (i.e. input test cases filtered by server) should be taken into account. Higher similarity indicates that test cases are more likely to be injected into the server. If a functional unit in test case is filtered by server, it is considered that this unit has not been successfully injected into the server. We use (3) to calculate similarity. Assuming that one of the four functional units in Figure 3 is filtered, then $U_{input}(T)=4$ and $U_{filtered}(T)=1$, its similarity is $\frac{3}{4}$.

$$S(T)=\frac{U_{input}(T)-U_{filtered}(T)}{U_{input}(T)} \tag{3}$$

- We compare the changes in the normal response page P_{normal} and the response page after sending test case $P_{injected}(T)$ to determine whether the test case has an attack effect on the target web application, such as chaos page content, unexpected error pages, etc. Page changes are calculated using Levenshtein Distance algorithm:

$$D(T)=\frac{LD(P_{normal},P_{injected}(T))}{len(P_{normal}(T))} \tag{4}$$

- We also observe the server response time. The longer the response, the more likely the server have vulnerabilities. This principle is mainly used for Time-Based SQL Injection:

$$t(T)=\frac{\left|t_{normal}-t_{injected}(T)\right|}{t_{normal}} \tag{5}$$

Finally, we propose the fitness function:

$$\begin{cases} Fit(T)=\Sigma+\alpha*S(T)+\beta*D(T)+\gamma*t(T) \\ \alpha+\beta+\gamma=1 \end{cases} \tag{6}$$

Each of these evaluation principles have weight to tune its impact when fuzzing different types of vulnerabilities. For example, server response time (5) does not have much impact on XSS fuzzing, so we can set its weight γ to approximately 0. And for SQL injection fuzzing, normally the server does not return filtered SQL injection test case, so α can be set to 0.3 or lower.

D. Crossover and mutation

1) Crossover

With the preprocessed test case, we can easily perform crossover on each functional unit. In the crossover process, only the contents of the same unit are exchanged, and no cross-unit exchange is performed to prevent the semantic structure from being destroyed.

Each unit have a certain probability of crossover. The crossover process can target one unit, and also can be able to crossover multiple units at the same time. The number of offspring is determined by the maximum number of functional units $U_{\max}$ in the parent test cases. Crossover operation can produce up to $2^{U_{\max}}-2(2\le U_{\max}\le 4)$ offspring. By setting, we can adjust the number of functional units to perform crossover and the number of offspring generated by parent test cases.

Let T_{x1} be the test case in Figure 2, and T_{x2} be the following test case:

```
">%22><SCRIPT>top[8680439..toString(30)](1)</SCRIPT>
```

Figure 3 shows an example of crossover process, in this example, we only allow one random unit (In this case, U_{X3}) to perform crossover and produce two children.

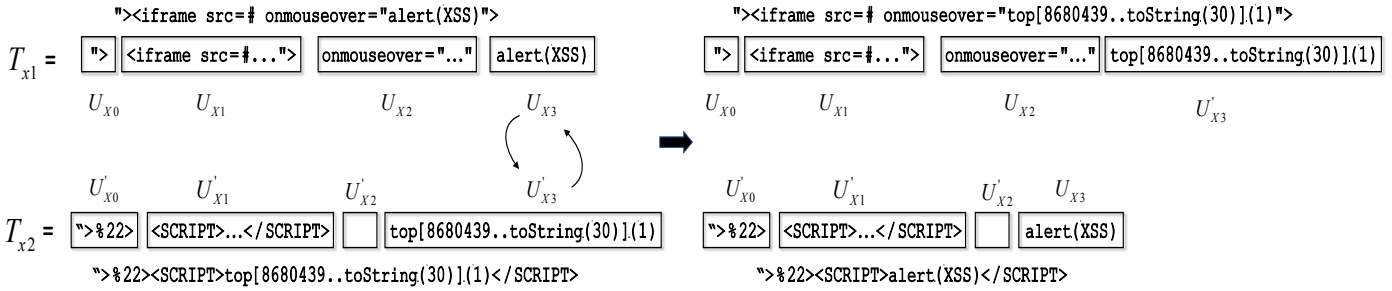


Fig. 3. An example of crossover process

2) Muation

Mutation can increase the diversity of the population and expand the current search space. In web fuzzing, mutation is mainly use mutation strategies to make test cases able to bypass the Web Application Firewall (WAF). We implemented several strategies for mutation, each mutation strategy can be used on multiple functional units, and also some functional units may not be able to use certain strategies. Some examples of strategies are shown in the Table. 3 below.

TABLE III. EXAMPLES OF STRATEGIES FOR MUTATION

Vulnerability	Mutation strategy	Example
SQL injection	Case Changing	' and 1=2 unIOn sELEecT * FRoM uSers --
	Repeated Keywords	1+UNunionION+SEselectLECT+1,2,3--,
	Add comment	1+un/**/ion+sel/**/ect+1,2,3--
XSS	Case Changing	JaVaScRiPt:alErt('XSS'),
	Encode	<img src=x onerror="&#0000106&#0000097&#00001 18...
	Malformed tags and obfuscation	<IMG """><SCRIPT> top[8680439..toString(30)](1)</SCR IPT>">

Like crossover, Mutation is also performed on each functional unit. Each one has a probability to mutate, and a mutation strategy is randomly selected. In standard mutation, only one functional unit in the test case is allowed to mutate. For the test cases with low fitness score, we propose a new mutation approach, which is to make them imitate the way how test cases with high fitness score mutated. Each functional unit of a test case with low fitness score no longer randomly chooses a mutation strategy to mutate, but uses the mutation strategy of a test case with high fitness score.

Let $T_{x1}=(U_{X0},S_1(U_{X1}),S_2(U_{X2}),U_{X3})$ be a test case with high fitness score, U_{X1} and U_{X2} were mutated using mutation strategy S_1 and S_2, respectively. And $T_{x2}=(S_2(U'_{X0}),S_2(U'_{X1}),U'_{X2},U'_{X3})$ is a test case with low fitness score, both U'_{X0}, U'_{X1} were mutated using S_2. Using our approach, T_{x2} will "learn from the master T_{x1}" and becomes $T_{x2}=(S_2(U'_{X0}),S_1(U'_{X1}),S_2(U'_{X2}),U'_{X3})$. If T_{x1} can bypass WAF or filter, then the probability of T_{x2} bypassing WAF will greatly increase, thereby increasing its threat and fitness score. Using this method, the worst individual will move towards to the best individual in search space, speeding up fuzzing test, but reducing the diversity. In order to maintain a certain diversity, we usually only allow one unit to learn and mutate, or manually adjust according to the situation.

IV. RESULTS AND ANALYSIS

In this section, we choose standard genetic algorithm and the adaptive genetic algorithm [14] as comparison objects. And target applications are some vulnerable web applications in OWASP's "Broken Web Application" Virtual Machine [15]. The test environment is Windows 7, Python v3.7.

A. Improved algorithm test

Each algorithm is run 20 times, with a maximum of 200 generations each time, using the same initial test cases for XSS.

Figure 4 shows average fitness score of standard genetic algorithm, adaptive genetic algorithm and the improved genetic algorithm from this paper. The average fitness of this paper's method is higher than AGA and SGA, and also convergence faster than the others. The high convergence speed can be attributed to the new mutation method that allows individuals with high fitness values to guide individuals with poor fitness values, whereas AGA is determined based on the overall state of the current population, improving the overall fitness value at a gentle speed. With a certain direction, individuals with poor fitness values can increase their fitness value faster and eventually speed up the convergence. The result indicates that the improved genetic algorithm has a higher chance of finding valid test cases.

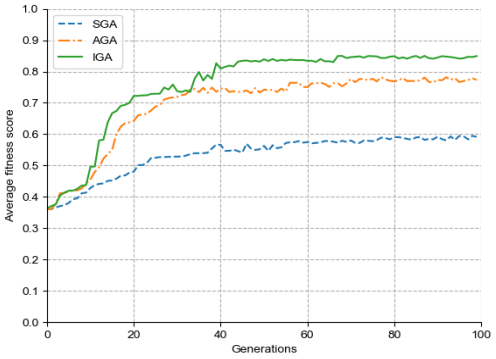
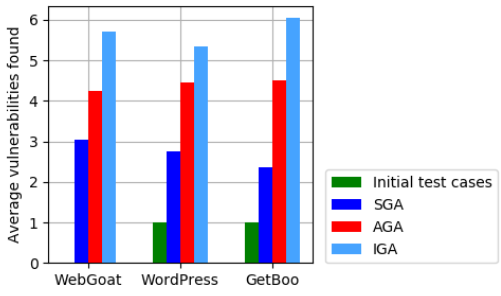


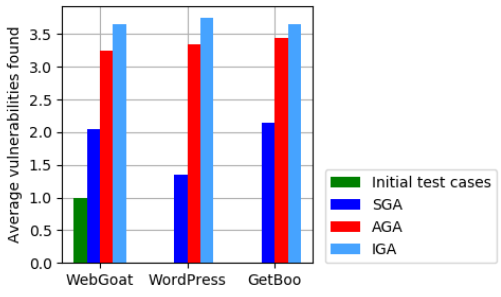
Fig. 4. Generations (x) and average fitness score(y)

B. *Web application vulnerability fuzzing test*

We used WebGoat v5.4, WordPress v2.0 and GetBoo v1.04 as test objects, and SGA, AGA and the initial test cases set as comparison. We collected potential vulnerabilities injection point in these applications in advance as input location. Each algorithm run 20 times for each web application and calculate the average number of vulnerabilities they found on each application.



(a) XSS fuzzing result



(b) SQL injection fuzzing result

Fig. 5. Web application(x) and average numbers of vulnerabilities found (y)

Based on the observation of Figure 5, the average number of vulnerabilities found by the improved genetic algorithm in this paper is higher than others. During the test, we found that the method of this paper has the fastest test speed compared to others. The same vulnerability was discovered several generations earlier than other methods. This phenomenon can be explained as that through the new mutation approach, when a test case successfully bypasses WAF or filter, other test cases will immediately imitate it and gain the identical capability to bypass the same WAF or filter, thereby increasing their chance of finding vulnerabilities. We also discovered that SGA has certain instability: the vulnerabilities found in previous runs are not found in the current run. This can be summarized as the premature phenomenon of SGA, and AGA has improved this problem by autonomously adjusting parameters. The improved algorithm in this paper is better than SGA when dealing with this issue due to the mutation method. However, it is not enough compared to AGA because the number of functional units participated in crossover and mutation is fixed and cannot be adaptively adjusted. This will be a primary subject of future work which allows the improved algorithm adaptively adjust the number of functional units involved in crossover and mutation according to the situation of each generation.

V. CONCLUSIONS

In this paper, we introduce genetic algorithm into web vulnerabilities fuzzing test. Through the analysis of attack semantics, functional units are preset, and test cases are preprocessed by being divided into different functional units, which make genetic algorithm better suited for web vulnerability fuzzing. Then, we propose a new mutation approach, which allows test cases with poor fitness values to mimic the mutation strategies of the test cases with high fitness values to mutate, so that genetic algorithm can produce more aggressive test cases. Finally, we verify the efficiency of the proposed improved algorithm from two perspectives: algorithm performance and vulnerability detecting. The future research work includes combining the idea of AGA and dynamically adjusting the number of functional units participated in crossover and mutation. In addition, we will apply more web vulnerabilities to the algorithm by analyzing its attack grammar and set their functional units.

REFERENCES

[1] OWASP, “OWASP Top 10-2017”, https://www.owasp.org/

[2] Michael Sutton, Adam Greene, and Pedram Amini. “Fuzzing: Brute Force Vulnerability Discovery”. Addison-Wesley Professional, 2007.

[3] “American fuzzy lop”, http://lcamtuf.coredump.cx/afl/

[4] Liu G.-H., Wu G., Zheng T., Shuai J.-M., Tang Z.-C. “Vulnerability analysis for X86 executables using Genetic Algorithm and Fuzzing”, In Proceedings of third International Conference on Convergence and Hybrid Information Technology, ICCIT 2008, 2 , art. no. 4682289 , pp. 491-497.

[5] Roger Lee Seagle Jr. “A framework for file format fuzzing with genetic algorithms.”, 2012.

[6] Mitchell Melanie. “An introduction to genetic algorithms”. Cambridge, Massachusetts, London, England, Fifth printing, 3:62–75, 1999.

[7] Hammersland, Rune and Einar Snekkenes. “Fuzz testing of web applications.”, 2008.

[8] J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler and F. Wotawa, "Attack Pattern-Based Combinatorial Testing with Constraints for Web Security Testing," 2015 IEEE International Conference on Software Quality, Reliability and Security, Vancouver, BC, 2015, pp. 207-212..

[9] Wang, Yi-Hsun, Ching-Hao Mao and Hahn-Ming Lee. “Structural Learning of Attack Vectors for Generating Mutated XSS Attacks.” TAV-WEB , 2010.

[10] F. Duchene, R. Groz, S. Rawat and J. Richier, "XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing," 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, 2012, pp. 815-817.

[11] OWASP, “SQL Injection Bypassing Cheat Sheet”, https://www.owasp.org/index.php/SQL_Injection_Bypassing_WAF

[12] OWASP, “XSS Filter Evasion Cheat Sheet”, https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[13] SuperCowPowers, “Data Hacking” , https://github.com/SuperCowPowers/data_hacking

[14] Jakobovic, Domagoj & Golub, M.. “Adaptive genetic algorithm. Journal of Computing and Information Technology.”, 1999, 7. 229-235.

[15] OWASP, “OWASP Broken Web Applications Project”, https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project