Adaptive Random Testing for XSS Vulnerability

Chengcheng Lv[1], Long Zhang[2,3], Fanping Zeng[1], and Jian Zhang[2,3]

[1]School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
Email: lvcc@mail.ustc.edu.cn, zlong@ios.ac.cn, billzeng@ustc.edu.cn, zj@ios.ac.cn

***Abstract*—XSS is one of the common vulnerabilities in web applications. Many black-box testing tools may collect a large number of payloads and traverse them to find a payload that can be successfully injected, but they are not very efficient. Previous research has paid less attention to how to improve the efficiency of black-box testing to detect XSS vulnerability. To improve the efficiency of testing, we develop an XSS testing tool. It collects 6128 payloads and uses a headless browser to detect XSS vulnerability. The tool can discover XSS vulnerability quickly with adaptive random testing method. We conduct an experiment using 3 extensively adopted open source vulnerable benchmarks and 2 actual websites to evaluate the adaptive random testing method. The experimental results indicate that the adaptive random testing method can effectively improve the fuzzing method by more than 27.1% in reducing the number of attempts before accomplishing a successful injection.**

***Keywords*—XSS Vulnerability, Adaptive Random Testing, Fuzzing**

I. Introduction

Cross-site scripting attack (also known as XSS) is a well-known security vulnerability in web applications. In an XSS attack, attackers usually manipulate malicious content (malicious script) to disguise benign text, which can deceive a vulnerable web application. When executing a web application, the victim usually treats the malicious text as the legitimate code of the application, and the victim's browser inadvertently executes the malicious content [1]. In the report released by the Open Web Application Security Project (OWASP) [2] in 2017, XSS is listed as one of the top 10 web vulnerabilities.

For example, "*echo "".$userName."";*" is a piece of PHP code whose function is to display the user's name on the page. But when the user's name is "<script>alert('This is an XSS')</script>", the browser will execute the user's name as the page code and display "This is an XSS" in the window. Here, "<script>alert('This is an XSS')</script>" is called XSS payload.

XSS vulnerabilities can be divided into the following three types [1]:

- Reflected XSS attack: Reflected XSS attack is currently the most basic type of web vulnerability attack, which is also known as Type-I XSS attack or non-persistent XSS attack. When the victim clicks on a link containing malicious text (most commonly in HTTP query parameters), the server script parses malicious text into malicious code (i.e., reflected back), and the victim's browser executes it.
- Stored XSS attack: The stored XSS vulnerability is a variant of the cross-site scripting flaw, which is also known as Type-II or persistent XSS vulnerability, and attackers can exploit such vulnerability to attack web applications. An attacker can embed a malicious code into a vulnerable server through an application such as a forum and store it permanently. When a victim visits such an infected site, the malicious code is provided to the victim as part of the web page.
- DOM based XSS attack: DOM based XSS attack is a new sub-class of reflected XSS attacks, which is also known as Type-0 XSS attack. In DOM based XSS attacks, malicious data does not touch web servers. Instead, it is completely reflected by JavaScript code on the client side.

There are many black-box testing tools for detecting XSS vulnerability. They do not know the internals of the web application and use fuzzing techniques over the web HTTP requests [3]. The approaches that can detect XSS vulnerability are mainly divided into dynamic approach and static approach [4]. The static approach detects XSS vulnerability by analyzing the response data. The detection speed is fast but the false alarm rate is high and the alarms need to be confirmed manually. Therefore, the dynamic approach may be a better choice. It determines whether user input is being parsed as code based on the behavior of the program at runtime. The dynamic approach could detect XSS vulnerability more accurately, but consume more time and resources. At the same time, a website may have many different urls with risk of XSS vulnerabilities. Therefore, it is difficult for tools to perform a large number of test cases.

In this paper, we propose a dynamic detection tool with the method of adaptive random testing (ART) [5] to detect XSS vulnerability in web applications. We found that the reason why invalid payloads fail to be injected is that some keywords in payloads were filtered or converted, or the payloads do not satisfy the context so the browser could not execute malicious code. We have observed that effective payloads tend to cluster together. Moreover, there usually are some identical keywords in invalid payloads and effective payloads, some mutation in invalid payloads may result in successful injection. Therefore, after a payload fails to be injected, we can measure the distance between the failed payload and other payloads, then select the next payload that is most likely to be injected successfully to find vulnerability

2640-0715/19/$31.00 ©2019 IEEE
DOI 10.1109/APSEC48747.2019.00018

more quickly.

The main contributions of this paper are as follows.

- We convert each payload into a collection of words based on the defined rules and calculate the distance between two payloads.
- We find that in XSS testing, the distribution of effective payloads is uneven. Based on this observation, we develop an XSS testing tool. It collects 6128 payloads and use a headless browser to detect XSS vulnerability. The tool can discover XSS vulnerability quickly with the ART method.
- We conduct an experiment using 3 extensively adopted open source vulnerable benchmarks and 2 actual websites to evaluate the ART method. The experimental results indicate that there is a 27.1% improvement over Fuzzing method.

The rest of this paper is organized as follows. Section II describes our approach. Section III describes the implementation of the testing tool. Section IV evaluates our approach through experiments. Section V introduces related work. Section VI summarizes the paper.

II. Approach

Black-box testing is a common way to mitigate the threat of XSS vulnerability in web applications. Fuzzing is a popular effective black-box testing method to detect such vulnerabilities [6]. A security expert or an attacker often has a prepared collection of attack payloads in hand, and traverses these payloads to find an effective payload, which can successfully inject the web application. These tasks are very simple and easy to be automated, but the existing tools are inefficient [7].

To improve the efficiency of testing in the sense, adaptive random testing (ART) [8], [9] has been proposed. Based on the observation, we find that failure-causing inputs tend to be clustered together, ART tries to evenly spread the randomly generated test cases for improving the fault-detection capability [8], [9].

We observe that the distribution of effective payloads in the state space is often uneven and tend to cluster together. So we can try to improve efficiency with the method of adaptive random testing. We found that there usually are some identical keywords in invalid payloads and valid payloads, and some mutation in invalid payloads may result in successful injection. So when a payload cannot be successfully injected, we can select a payload with an appropriate distance from the invalid payload. It is equivalent to making a mutation on the invalid payload. The main problem is the distance measure between the payloads and how to select the next payload. We will explain separately below.

A. Distance Measure

Distance measure is mainly divided into word tokenizing and distance calculation.

1) Word tokenizing: In order to measure the distance between the payloads more accurately, we need to tokenize the XSS payloads for identifying sensitive strings and tags of HTML or JavaScript language [10]. So we define the rules as follows and use the Natural Language ToolKit [11] for processing the XSS payloads.

- The contents of single and double quotes, such as 'XSS'
- <>tag, such as '<script'
- parameter name, such as 'href='
- Function body, such as 'alert('
- Http/https link
- Common words composed of alphanumeric characters, such as 'javascript'
- Special encoding format, such as '\u003c'
- Special keywords, such as '\\'

2) Distance calculation: We have defined the rules for XSS payloads and eventually convert each payload into a collection of words. We use the ratio of the keywords shared by the two payloads to measure the similarity between the two payloads, that is, the distance between two payloads. In this section, we use the Jaccard distance [12], [13] to calculate the distance between two payloads. Suppose we are given two payloads P_i and P_j, which have word collection W_i and W_j. The Jaccard distance measures the similarity between two samples using the proportion of the different elements in the two sets. The formula is:

$$Distance(P_i, P_j) = Jaccard(W_i, W_j) = \frac{|W_i \cup W_j| - |W_i \cap W_j|}{|W_i \cup W_j|}$$

B. Payloads selection

The distribution of effective payloads in the state space is often uneven and tend to cluster together. So we propose an XSS payloads selection algorithm XSSART, which can improve efficiency with the method of adaptive random testing. We select a payload randomly as the first test case. When the payload cannot be successfully injected, increase the priority of the payloads whose distance from this invalid payload is in the interval [dl, dr] (The specific values of dl, dr are specifically determined in the next section). Then the highest priority payload would be selected as the next test case. The specific process of XSSART is shown in algorithm 1.

The input of the algorithm is the payloads collection Pc and the appropriate distance interval $[dl, dr]$. And the output of the algorithm is True or False, indicating whether an XSS vulnerability exists in the system. The $Rank$ value of a payload indicates the selected priority. First, the $Rank$ value of the payloads are all set to 0, and the $Candidate$ is set to all the payloads Pc (1, 2 lines of algorithm 1). Then a payload $P_{selected}$ is randomly selected from $Candidate$ and $P_{selected}$ is removed from Pc (4, 5 lines of algorithm 1). If $P_{selected}$ can be injected successfully, the algorithm will return $True$ (6, 7 lines of algorithm 1). If not, the algorithm will set $Candidate$ to $\emptyset$ and set Max_Rank to 0 (10 line of the algorithm). For each payload P_{tmp} in the Pc, if the distance between P_{tmp} and $P_{selected}$ is within the

interval $[dl, dr]$, the $Rank$ value of P_{tmp} will be increased by 1 (11, 12 lines of algorithm 1). If the $Rank$ value of the P_{tmp} is greater than Max_Rank , the algorithm will modify the value of Max_Rank and set $Candidate$ to $\emptyset$ (14, 15, 16 lines of algorithm 1). If the $Rank$ value of the P_{tmp} is equal to Max_Rank, P_{tmp} will be added to $Candidate$ (18, 19 lines of algorithm 1). After processing all the payloads in Pc, if $Candidate$ is not equal to the empty set, the algorithm will go back to the 4th line of the algorithm through the while loop and execute the next payload, otherwise, the algorithm will return False.

algorithm 1 Payloads selection

Input: Payloads collection : Pc
Input: Distance interval : dl , dr
Output: $True/False$

1: $Rank[P_0, P_1, \ldots, P_n] = \{0\}$
2: $Candidate = Pc$
3: **while** $Candidate \neq \emptyset$ **do**
4: randomly select $P_{selected}$ from Pc
5: $Pc \leftarrow Pc - \{P_{selected}\}$
6: **if** $P_{selected}$ can be injected successfully **then**
7: **return** $True$
8: **else**
9: $Candidate \leftarrow \emptyset, Max\_Rank = 0$
10: **for** all $P_{tmp} \in Pc$ **do**
11: **if** $dl \leq Distance(P_{selected}, P_{tmp}) \leq dr$ **then**
12: $Rank[P_{tmp}] += 1$
13: **end if**
14: **if** $Rank[P_{tmp}] > Max\_Rank$ **then**
15: $Candidate \leftarrow \emptyset$
16: $Max\_Rank = Rank[P_{tmp}]$
17: **end if**
18: **if** $Rank[P_{tmp}] = Max\_Rank$ **then**
19: $Candidate \leftarrow Candidate \cup \{P_{tmp}\}$
20: **end if**
21: **end for**
22: **end if**
23: **end while**
24: **return** $False$

III. Implementation

We develop a prototype tool that implements the approach mentioned in Section II in the python 3.5.4 environment. We use the PhantomJS [14] based browser to determine if the injection was successful based on the behavior of the program. If the content in the payloads is executed as page code, we believe that there is an XSS vulnerability at this site. The tool supports the detection of Reflected XSS vulnerability and Stored XSS vulnerability.

The overview of the tool is shown in Figure 1. The user needs to provide url, cookies and other information for testing. Each time the tool selects a payload with the method of ART, and the headless browser PhantomJS uses the selected payload and the information provided by users to construct a http request and sends it to the target server. After receiving the response, the tool runs the response code to determine if the XSS injection was successful based on the behavior of the browser. If it is not, select the next payload to execute, otherwise, output the effective payload and stop running.

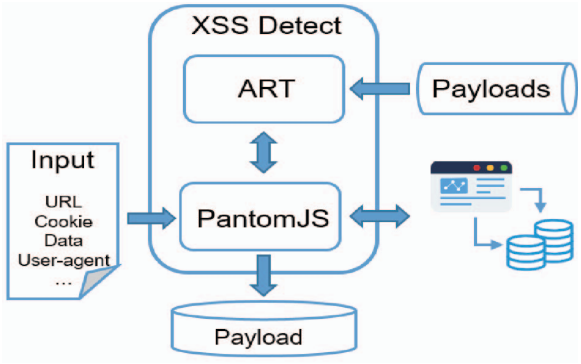

Fig. 1. Overview of the tool

At the same time, we collect high quality payloads from the following open source tools and websites.

- Xenotix[1] : an open source tool with 1630 payloads.
- XSSfork[2] : an open source tool with 71 payloads.
- Burp_suit[3] : a tool with 96 payloads.
- Fuzzdb[4] : a collection with 243 payloads
- foospidy[5] : a collection with various payloads

We keep only one item which appears multiple times and collect a total of 6,128 payloads.

IV. Experimental Evaluation

We test the efficiency of the ART method on open source vulnerable benchmarks and actual website applications. We select open source benchmarks Web for Pentester[6], Damn Vulnerable Web Application (DVWA)[7] and WAVSEP[8]. They are all well-known penetration test walkthrough environments. Web for Pentester has 7 reflected websites, which are recorded as WFP_1, WFP_2, WFP_3, WFP_4, WFP_5, WFP_6, and WFP_7. DVWA has a reflected injection point and a stored injection point with four different levels of security. The level 'Impossible' can't inject any payloads, so we have 6 websites that are recorded as DVWA_R_1, DVWA_R_2, DVWA_R_3, DVWA_S_1, DVWA_S_2 and DVWA_S_3. WAVSEP has 73 different websites. Considering the number of websites in the first two systems, we select the first 7 websites, recorded as WAVSEP_1, WAVSEP_2, WAVSEP_3, WAVSEP_4, WAVSEP_5, WAVSEP_6 and

[1]https://github.com/ajinabraham/OWASP-Xenotix-XSS-Exploit-Framework
[2]https://github.com/bsmali4/XSSfork
[3]http://portswigger.net/burp
[4]https://github.com/fuzzdb-project/fuzzdb
[5]https://codeload.github.com/foospidy/payloads/zip/master
[6]https://pentesterlab.com/exercises/web_for_pentester
[7]http://www.dvwa.co.uk/
[8]https://github.com/sectooladdict/wavsep

WAVSEP_7. We chose a course selection system as the actual website for the test. The site has two different XSS injection points. If exploited by an attacker, it can be detrimental to the teachers and students who use the site. We record these two injection points as School_1 and School_2. We tested these sites with the payloads. The results are shown in Table I.

We can see that the number of valid payloads is different in different websites. Some are dense (such as WFP_1 and DVWA_R_1) and some are sparse (such as WFP_7 and WAVSEP_2). How the effective payloads are distributed on these different sites and how efficient the ART method is will be discussed below.

TABLE I
PAYLOADS INJECTION RESULTS

Site	Total Payloads	Valid Payloads	Ratio
School_1	6128	587	0.096
School_2	6128	256	0.042
WFP_1	6128	1256	0.205
WFP_2	6128	908	0.148
WFP_3	6128	814	0.133
WFP_4	6128	450	0.073
WFP_5	6128	359	0.059
WFP_6	6128	131	0.021
WFP_7	6128	63	0.010
DVWA_R_1	6128	1444	0.236
DVWA_R_2	6128	1082	0.177
DVWA_R_3	6128	531	0.087
DVWA_S_1	6128	1505	0.246
DVWA_S_2	6128	1055	0.172
DVWA_S_3	6128	535	0.087
WAVSEP_1	6128	1484	0.242
WAVSEP_2	6128	16	0.003
WAVSEP_3	6128	583	0.095
WAVSEP_4	6128	65	0.011
WAVSEP_5	6128	174	0.028
WAVSEP_6	6128	272	0.044
WAVSEP_7	6128	261	0.043
Average	6128	628	0.102

A. Distribution of effective payloads

We count the proportion of payloads injected successfully at different distances between invalid payloads and effective payloads. The result is shown in Figure 2. Here, we use the relative distance. For a target payload, we sort the other payloads according to the Jaccard coefficient with the target payload, and divide the order by the value of the total number of payloads as the distance value, so that the distance from the target payload is evenly distributed.

The line in Figure 2 represents the average proportion of effective payloads. We can see that, for effective payloads, the closer the payloads are, the more successfully they can be injected. But for invalid payloads, as the distance increases, the proportion of payloads injected successfully increases first and then decreases. Therefore, we can say that effective payloads cluster together and selecting a payload with an appropriate distance from the invalid payload can increase the probability of successful injection.

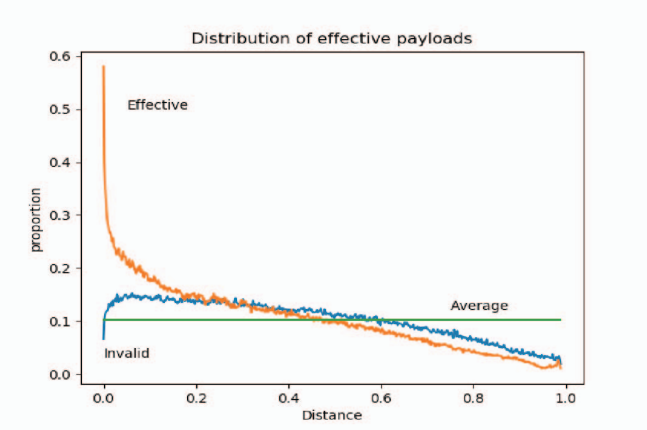

Fig. 2. Distribution of effective payloads.

B. XSSART Vs Fuzzing

From the previous section we know that for invalid payloads, as the distance increases, the proportion of payloads injected successfully increases first and then decreases. So we can select the next one based on the average distance between invalid payloads and effective payloads. We use the 7 websites of Web for Pentester as a "training set" to find a appropriate distance, to test the efficiency of the ART method on other different benchmarks and actual websites. We continue to use the relative distance, and the average distance between invalid payloads and effective payloads in the 7 websites of Web for Pentester is 0.391. We increase the priority of the 1/4 payloads each time, so we can set the distance interval to [0.265,0.515].

TABLE II
XSSART VS FUZZING

Site	Fuzzing	XSSART	Ratio
School_1	10.24	6.7	34.6%
School_2	**23.83**	**11.88**	**50.5%**
WFP_1	4.86	4.27	12.1%
WFP_2	6.76	5.68	16.0%
WFP_3	7.4	5.94	19.7%
WFP_4	**13.72**	**9.03**	**34.2%**
WFP_5	17.09	15.35	10.2%
WFP_6	**46.48**	**34.5**	**25.8%**
WFP_7	99.79	90.54	9.3%
DVWA_R_1	4.25	3.72	12.5%
DVWA_R_2	5.63	4.71	16.3%
DVWA_R_3	**11.4**	**8.4**	**26.3%**
DVWA_S_1	4.08	3.67	10.0%
DVWA_S_2	5.69	5.01	12.0%
DVWA_S_3	**11.61**	**8.45**	**27.2%**
WAVSEP_1	4.1	3.69	10.0%
WAVSEP_2	**358.84**	**173.56**	**51.6%**
WAVSEP_3	10.69	6.88	35.6%
WAVSEP_4	**92.73**	**45.73**	**50.7%**
WAVSEP_5	**34.82**	**20.67**	**40.6%**
WAVSEP_6	**22.2**	**12.11**	**45.5%**
WAVSEP_7	**23.48**	**12.65**	**46.1%**
Average	37.26	22.41	27.1%

In the XSS detection, it is significant to find the first effective payload. We stop testing once that we find a payload which can be injected successfully, and record the

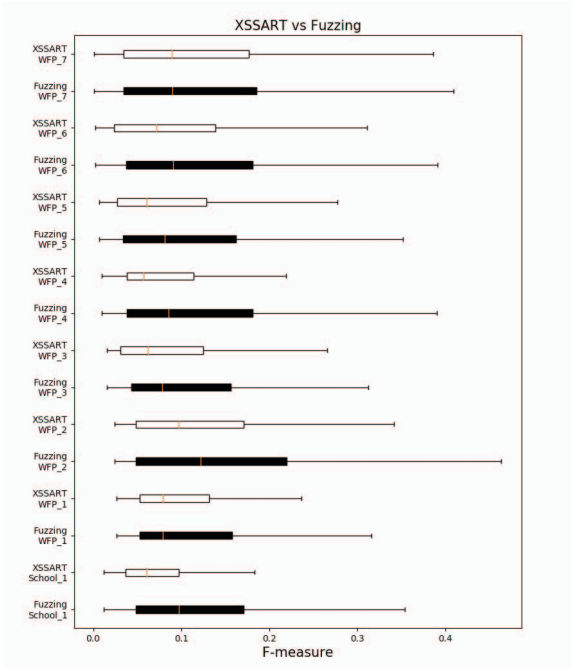

Fig. 3. ART Vs Fuzzing

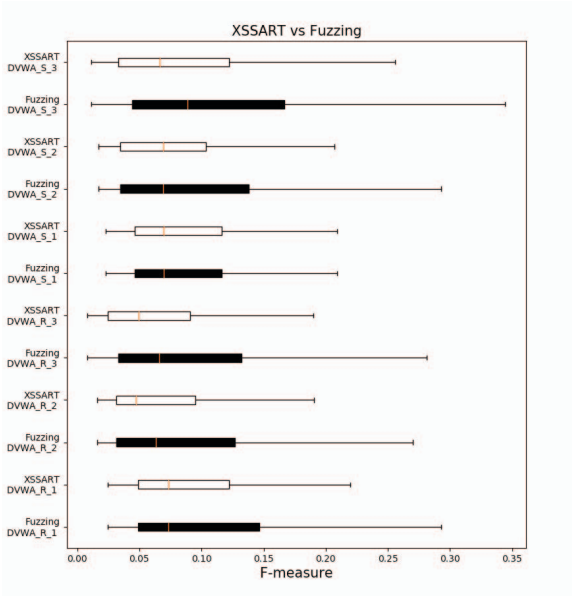

Fig. 4. ART Vs Fuzzing

number of executions of payloads to evaluate the ART method. This evaluation is called F-measure, a commonly used metric, which is defined as the expected number of test cases to detect the first failure [8], [9]. We use F-measure to compare XSSART with the Fuzzing method. Here Fuzzing method for XSS detection means that each time select an unexecuted payload for testing until the vulnerability is discovered. To avoid sample bias, we tested each website 1000 times and count the average as the last result. Finally, we calculate the ratio($(Fuzzing-ART)/Fuzzing*100\%$) to evaluates how much efficiency XSSART improves. The results are shown in Table II.

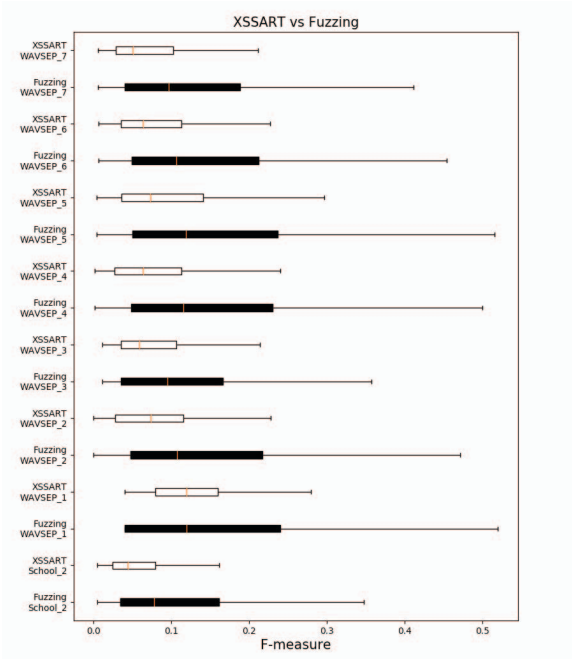

Fig. 5. ART Vs Fuzzing

As we can see in the table, the average efficiency of the ART method is superior to the Fuzzing method in the test of all 22 websites. The average increase is 27.1%, and the highest increase is 51.6%. Moreover, the more sparse the effective payloads are, the more efficient of ART is.

In order to better compare ART and Fuzzing, we use the box-whisker plots to represent the data of those websites, as shown in Figure 3, Figure 4 and Figure 5. We divide the F-measure of each website by the maximum of the two methods, and normalize the F-measure to between 0 and 1. There are median, maximum, and minimum values of F-measure. It can be seen that among the 22 XSS vulnerabilities, the minimum value of the F-measure of XSSART and Fuzzing is not much different, which means that the best case (the least number of payloads you need to try) in two methods is very close. But for the maximum value of the F-measure, XSSART is significantly smaller than Fuzzing. Among the 21 vulnerabilities, the maximum value of XSSART is less than Fuzzing, and the maximum value of the two methods in the remaining vulnerability (DVWA_S_1) is close. It indicates that the worst case of XSSART (the most payloads you need to try) is much better than Fuzzing.

Therefore, we can say that the method of ART can detect XSS vulnerabilities more effectively than the method of Fuzzing.

V. Related work

Academia and industry researchers have proposed many approaches to detect XSS attacks, we summarize the main work in the field related to this paper.

Omer Tripp et al. [15] proposed a learning approach to test XSS and realized it in XSS Analyzer. XSS Analyzer generates XSS attack vectors based on a context-free grammar rule. It can learn the constraints in the grammar rules through invalid attack vectors, i.e., whose words cannot be included in the attack vector to bypass the defense mechanism of the web application. Experimental results show that XSS Analyzer outperforms several competing algorithms, including a mature commercial algorithm featured in IBM Security AppScan Standard V8.5-by a far margin. But there are still shortcomings in XSS Analyzer. It applies learning to individual literals only. So it would consume a large quantity of HTTP requests and can't capture complex input constraints involving multiple literals simultaneously. XSS Analyzer has recently been integrated into the latest version of AppScan (V8.6) instead of that algorithm.

Bozic J et al. [7], [16] introduced a combined approach, which comprises the area of combinatorial testing with the emphasis on test case generation for XSS attacks and the attack pattern-based testing technique for test case execution against web applications. The generated strings were automatically tested against three web applications with different combinatorial interaction strength with highly promising initial results. In the next work, they revised an input grammar for combinatorial generation of test inputs and made use of constraints for another test suite. The experimental results show that setting constraints inside the input model results in significantly improved attack vectors. Next, Simos et al. [17] use a modification of a combinatorial testing-based fault localization method to identify XSS-inducing combinations, which can help to better understand the root cause of an XSS vulnerability and provide insights about how to fix a flawed sanitization function. Combinatorial test generation method may be an alternative approach for web security testing. The approach can discover the vulnerability by traversing the combinatorial test cases, and we can also apply the ART method here to find the vulnerability more quickly.

Tang Z et al. [10] proposed a Webmail XSS fuzzer called L-WMxD (Lexical based Webmail XSS Discoverer), which works on a lexical based mutation engine. L-WMxD is an active defense system to discover XSS vulnerability before the Webmail application is online for service. Unlike our method which uses payloads collected in advance to discover the vulnerability, they make different rules corresponding to different filtering strategies of Webmail server. The engine is initialized by normal JavaScript code called seed. Then, rules are applied to the sensitive strings in the seed which are picked out through a lexical parser. After that, the mutation engine issues multiple test cases. Newly-generated test cases are used for XSS test.

Bates et al. [18] proposed a client-side XSS filter named Auditor, which is embedded by default in Google Chrome. Auditor can block scripts after HTML parsing but before execution by inserting an interface between the browser's HTML parser and the JavaScript engine. Compared to the previous method, auditor is faster and more accurate.

Guo et al. [19] proposed a method of XSS vulnerability detection using optimal attack vector repertory. This method develops an XSS attack vector grammar to generate XSS attack vectors automatically. It first generates basic XSS attack vector repertory with attack vector pattern repertory and resource repertory. Then it applies mutation rule to generate the final XSS attack vector repertory. Finally, it uses machine learning optimization to reduce the size of the attack vector library. Experimental results show that the approach makes a good performance in detecting XSS vulnerability in web applications.

Goswami et al. [20] proposed a client-server based architecture to support XSS attack detection. An initial checking for the vulnerability is carried out at the client machine to decide whether to drop the request directly or send it to the proxy for further processing. The data collected at the proxy level undergoes steps such as preprocessing, feature extraction, feature selection, etc. to detect the attack using an unsupervised approach. This approach effectively balances the load between the client and the server.

There are also many tools for testing XSS vulnerability. Since white-box testing tools (such as Jovanovic et al. [21]) require access to source code, more tools use the black-box testing method to test XSS vulnerability by constructing http requests.

Xenotix [22] is a penetration testing tool developed exclusively to detect and exploit XSS vulnerability with zero false positives. It uses live payload reflection-based XSS detection via powerful triple browser rendering engines, including Trident, WebKit, and Gecko. Xenotix apparently has a large XSS payload database, allowing effective XSS detection and WAF bypass. The defects of Xenotix are also obvious, which uses the browser instead of the requests library, so it will consume more memory and time. So it is necessary to study how to find XSS vulnerability with fewer attempts.

VI. Conclusion and Future Work

Black-box testing is a common way to mitigate the threat of XSS vulnerability in web applications. Many black-box testing tools may collect a large number of payloads and traverse them to find a payload that can be successfully injected. But when there are many different URLs in the system or the vulnerability is deeply hidden, it is not very efficient. Previous research did not focus on how to improve the efficiency of black-box testing to detect XSS vulnerability.

In this paper, we convert the attack payloads into word vectors and use the Jaccard coefficient to measure the distance between the two payloads. We observe the distribution of effective payloads, and develop a tool for detecting XSS vulnerability with the ART method. The tool collects 6128 payloads from some open source tools and websites and uses a headless browser to detect XSS vulnerability. We conduct an experiment using 3 extensively adopted open source vulnerable benchmarks and 2 actual websites to evaluate the ART method. The experimental results indicate that the ART method can effectively improve the fuzzing method by

more than 27.1% in reducing the number of attempts before accomplishing a successful injection.

We have developed a black-box testing tool to detect XSS vulnerability and current work can still be improved. DOM-based XSS attacks are out of our range, and we will deal with them in the next stage. The payloads we collect are partially redundant and not comprehensive enough. In the future work, we will focus on the more comprehensive and efficient XSS payload generation approaches, which can detect XSS vulnerability more thoroughly. At the same time, we will try to extract features from payloads and use machine learning clustering algorithms to select more effective payloads.

ACKNOWLEDGMENT

This work is supported partly by the National Key R&D Program of China 2018YFB2100303, 2018YFB0803400; the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (CAS), Grant No.QYZDJ-SSW-JSC036; and National Natural Science Foundation of China (NSFC) under grant 61772487.

REFERENCES

[1] U. Sarmah, D. Bhattacharyya, and J. Kalita, "A survey of detection methods for XSS attacks," Journal of Network and Computer Applications, 2018.

[2] T. OWASP, "Top 10-2017 the ten most critical web application security risks," URL: owasp.org/images/7/72/OWASP Top 10-2017%28en, vol. 29, 2017.

[3] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for sql injection and XSS attacks," in 13th Pacific Rim international symposium on dependable computing (PRDC 2007). IEEE, 2007, pp. 365-372.

[4] H. Choi, S. Hong, S. Cho, and Y.-G. Kim, "Hxd: Hybrid XSS detection by using a headless browser," in 2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT). IEEE, 2017, pp. 1-4.

[5] J. Chen, L. Zhu, T. Y. Chen, R. Huang, D. Towey, F.-C. Kuo, and Y. Guo, "An adaptive sequence approach for oos test case prioritization," in 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2016, pp. 205-212.

[6] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). ACM, 2007, pp. 1-1.

[7] J. Bozic, D. E. Simos, and F. Wotawa, "Attack pattern-based combinatorial testing," in Proceedings of the 9th international workshop on automation of software test. ACM, 2014, pp. 1-7.

[8] T. Chen, D. H. Huang, and F.-C. Kuo, "Adaptive random testing by balancing," in Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). ACM, 2007, pp. 2-9.

[9] T. Y. Chen, T. Tse, and Y.-T. Yu, "Proportional sampling strategy: a compendium and some insights," Journal of Systems and Software, vol. 58, no. 1, pp. 65-81, 2001.

[10] Z. Tang, H. Zhu, Z. Cao, and S. Zhao, "L-wmxd: Lexical based webmail XSS discoverer," in 2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2011, pp. 976-981.

[11] E. Loper and S. Bird, "NLTK: the natural language toolkit," arXiv preprint cs/0205028, 2002.

[12] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," City, vol. 1, no. 2, p. 1, 2007.

[13] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in Proceedings of the international multiconference of engineers and computer scientists, vol. 1, no. 6, 2013, pp. 380-384.

[14] PhantomJS, "Scriptable headless browser," https://phantomjs.org/, 2018.

[15] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: a learning approach to web security testing," in Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, 2013, pp. 347-357.

[16] J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler, and F. Wotawa, "Attack pattern-based combinatorial testing with constraints for web security testing," in 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, 2015, pp. 207-212.

[17] D. E. Simos, K. Kleine, L. S. G. Ghandehari, B. Garn, and Y. Lei, "A combinatorial approach to analyzing cross-site scripting (XSS) vulnerabilities in web application security testing," in IFIP International Conference on Testing Software and Systems. Springer, 2016, pp. 70-85.

[18] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters," in Proceedings of the 19th international conference on World wide web. ACM, 2010, pp. 91-100.

[19] X. Guo, S. Jin, and Y. Zhang, "XSS vulnerability detection using optimized attack vector repertory," in 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. IEEE, 2015, pp. 29-36.

[20] S. Goswami, N. Hoque, D. K. Bhattacharyya, and J. Kalita, "An unsupervised method for detection of XSS attack," IJ Network Security, vol. 19, no. 5, pp. 761-775, 2017.

[21] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in 2006 IEEE Symposium on Security and Privacy (SP''06). IEEE, 2006, pp. 6-pp.

[22] A. Abraham, "Detecting and exploiting XSS with xenotix XSS exploit framework," https://www.exploit-db.com/docs/english/21223-detecting-and-exploiting-xss-vulnerabilities-with-xenotix-xss-exploit-framework.pdf, 2012.