The 49th Annual IEEE International Carnahan Conference on Security Technology
ICCST 2015

Browser Fuzzing by Scheduled Mutation and Generation of Document Object Models

Ying-Dar Lin[1] Feng-Ze Liao[1] Shih-Kun Huang[1] Yuan-Cheng Lai[2]
[1] Dept of Computer Science, National Chiao Tung University, Taiwan
[2] Dept. of Information Management, National Taiwan University of Science and Technology, Taiwan

***Abstract*—Internet applications have made our daily life fruitful. However, they also cause many security problems if these applications are leveraged by intruders. Thus, it is important to find and fix vulnerabilities timely to prevent application vulnerabilities from being exploited. Fuzz testing is a popular methodology that effectively finds vulnerabilities in application programs with seed input mutation. However, it is not a satisfied solution for the web browsers. In this work, we propose a solution, called scheduled DOM fuzzing (SDF), which integrates several related browser fuzzing tools and the fuzzing framework called BFF. To explore more crash possibilities, we revise the browser fuzzing architecture and schedule seed input selection and mutation dynamically. We also propose two probability computing methods in scheduling mechanism which tries to improve the performance by determining which combinations of seed and mutation would produce more crashes. Our experiments show that SDF is 2.27 time more efficient in terms of the number of crashes and vulnerabilities found at most. SDF also has the capacity for finding 23 exploitable crashes in Windows 7 within five days. The experimental results reveals that a good scheduling method for seed and mutations in browser fuzzing is able to find more exploitable crashes than fuzzers with the fixed seed input.**

Keywords —browser fuzzing, black-box fuzzing, vulnerabilities, exploits, mutation, scheduling, document object model, DOM

I. INTRODUCTION

Internet applications have made our daily life fruitful. However, they also cause many security problems if these applications are leveraged by intruders. Take some historical events as examples. It could make nuclear centrifuges lose control[1] and shut down banking systems unexpectedly[2]. The intrusions are often due to the vulnerabilities in application programs. Intruders try to find vulnerabilities in an application and exploit it to get the access they need, i.e., using malformed inputs to take control of the victim system. It is therefore important to find and fix vulnerabilities as soon as possible to prevent vulnerabilities from being exploited[3].

Fuzzing is a popular methodology that effectively finds vulnerabilities in application programs[4, 5]. The mechanism is to constantly create an input, feed it into an application, and observe whether the application terminates or crashes. The reason why we are interested in finding an input that crashes a target application is that if it could crash the target, then it also has the chance to exploit the target. Hence, how to create the inputs more effectively to find crashes becomes a critical issue not only for intruders but also for application developers.

In general, fuzzing can be classified into white-box fuzzing and black-box fuzzing, depending on how they create the inputs[6]. White-box fuzzing would analyze the applications to ensure that the inputs it creates are effective in finding crashes in applications. That is, an application always has many branches and then forms many possible execution flows. With the enormous input space, it would be possible to create virtually infinite inputs to exhaust all execution flows. However, white-box fuzzing tries to create less inputs to cover more possible execution flows. Recently, some solutions have been proposed. CRAXfuzz[7] identifies specific sensitive functions in advance. If an application uses some identified functions, it would determine whether the application has security vulnerabilities. If the answer is "yes", it would also generate specific inputs that cause the application to crash. Coverset[8] proposes six algorithms to compute and selects the proper inputs that could maximize the code coverage. Nevertheless, white-box fuzzing is not perfect in practice. Because of too many execution flows and complex constraints in an application, it could be imprecise. Moreover, it takes much time to analyze the application. Hence, an alternative way is to randomly generate inputs without analyzing applications, which is called black-box fuzzing.

Black-box fuzzing uses a seed from a data pool to generate the next input. The idea behind the black-box fuzzing is simple. First, it selects a seed, i.e., a template that feeds to the application, and then uses a method to mutate it. The general method to mutate is to change some bytes of the original seed to generate an input. Second, it feeds the input, the mutated seed, to the application and checks the result of the application. Hence, black-box fuzzing would rather generate inputs directly than take time to analyze the application. It is an effective and simple method to find vulnerabilities. *zzuf*[9] is an example that mutates seeds by changing some bytes and generates inputs to feed to the application. Moreover, it uses the mutation ratio as parameters to decide how many bytes should be changed. Basic Fuzzing Framework (BFF)[10] is a tool that improves the performance of black-box fuzzing. It is based on *zzuf* as the mutation method with a seed recycling strategy. It records the results in the past runs and determines which seed to select from the seed pool next time. The objective is to increase the number of crashes found. BFF is Linux-based, while Failure Observation Engine (FOE)[11] is Windows-based version of BFF. *fuzzSim*[12] tries to find an optimal seed selection algorithm based on BFF to get more number of crashes found.

Although the above solutions can be used to find vulnerabilities in applications, it might not be the best solution to find any types of applications. Take Web browser as an

Authorized licensed use limited to: National Chiao Tung Univ.. Downloaded on May 12,2020 at 06:21:20 UTC from IEEE Xplore. Restrictions apply.

example. The input fed into a browser is text-based, so it would not be efficient if we do not follow their text well-formed rules to mutate the seed. Thus, there exists an extra problem if we want to find vulnerabilities for a browser. Besides, the report from Symantec 2013[13] says that the number of browser vulnerabilities found was 351 in 2011 and increased to 891 in 2012. In 2013, there were also 591 vulnerabilities found. It said that there existed some vulnerabilities even at the time of reporting. There are some solutions to browser fuzzing, targeted only at browsers, such as *bf3*[14], *crossfuzz*[15] and *ndujafuzz* [16]. *bf3* tries to generate inputs by adding a random element with a repeated random string in an empty structure, the input which only has a header. *crossfuzz* and *ndujafuzz* use document object model (DOM)[17] to implement the mutation process. That is, they would rather constantly change their structures and values in the DOM than just change some bytes in the seed. The method mentioned worked well with fuzzing efficiently formerly. However, they have been released for a long time without frequent updates and may not be able to find new vulnerabilities, because the vulnerabilities they could find have already been fixed. Thus, we'd like to propose a solution that uses an improved way to generate inputs and determine whether the method would find more crashes or vulnerabilities based on the results of recent runs.

In this work, we propose a solution, named scheduled DOM fuzzing(SDF), which integrates the concept of browser fuzzing, black-box fuzzing, and the architecture of BFF. To get more possibilities, we improve the architecture of the original browser fuzzing and make seed and mutation changeable. The primary contributions to our work are described as follows.

- We propose an improved architecture that make seed and mutation changeable, and check recent runs to explore the execution paths in better possibilities.
- We propose a method to increase the performance by determining which combinations of seed selection and mutation would find more crashes.

II. Background

In this section, we review the techniques of black-box fuzzing and discuss how to improve it. Then, we introduce the techniques of browser fuzzing.

A. Black-box fuzzing

Black-box fuzzing is known as dynamic randomized-input for functional testing that has been used to find application vulnerabilities since 1990s[18]. There are two important components: seed and mutation, for fuzzing to generate input successfully. A seed is a template that feeds to the application after using mutation to randomly change the seed. We would generate the input based on the corresponding seed and mutation. In addition, it would also use some seeds as a seed pool to increase the variety of results as we want. After the application is fed with the input, we would inspect whether it crashes or not. For example, if we want to fuzz a media player using *zzuf*, the seed inputs must be kinds of media files. We would select a seed and use mutations in *zzuf* to change some bytes of the media data.

After that, we feed the media player the mutated media and inspect whether media player crashes or not. Hence, we would find some crashes including possible vulnerabilities. Ideally, there are many possible seeds that can be mutated and we can generate more crashes as you want. In fact, it would be restricted by the limited set of seeds and it would use designated seeds for specific vulnerabilities to find crashes easily. At the same time, it might ignore some seeds that may potentially lead to a crash in the fuzzing process.

B. Improvement of Black-box Fuzzing

Although the technique of black-box fuzzing is simple and also effective, there is a problem that it doesn't care about how to select a seed from the seed pool. Hence, it would select the seed input randomly. In order to improve the performance of black-box fuzzing, the concept of scheduling seed selection is proposed. The use of scheduling is to give seeds with selected probability or score. Hence, black-box fuzzing can select proper seeds based on the selected probabilities or score. We have discussed some of the work. BFF designs an architecture which provides the mechanism, i.e., scheduling, to select a proper seed according to the past experience. It computes the rate, as the number of crashes found divided by the total time, to select the seed. Because of the rate as a belief metric, it produces the situation that if more crashes are found with a seed, there is a higher possibility to select the seed again.

Likewise, *fuzzSim* is to improve the performance of BFF, so it uses five different scores of computation and five selected probabilities to compute and determine which seed to select. The objective is to take less time to find the crash you have found before.

C. Browser fuzzing

Browser fuzzing is a specific way to test a browser based on the input format of the browser. The input of the browser is document-based and the structure needs to follow certain rules. According to the rules, browser fuzzing could randomly generate executed instructions in a controllable way. Recently, there are two types of browser fuzzing, i.e. static and dynamic, which are used to test browsers[19]. In static browser fuzzing, the input is randomly generated before fuzzing and then browsers execute the input to test whether they will crash or not. For the example of the browser fuzzer 3(called *bf3*), it is a static browser fuzzing which adds one element and assigns a random sequence as the value in a seed. Therefore, it could generate lots of input that can feed to the browser. However, it is not an effective way to find crashes because the input may have similar or simple structures.

In dynamic browser fuzzing, they usually use JavaScript to constantly mutate the structure of the seed which is an html file. Hence, the result obtained from a mutated seed is unpredictable.

There is a random number to decide the executed procedures every time, so we would not know the result in advance. It would not stop until the browser terminates or crashes. Besides, it would employ document object model (DOM) to change the structure of the seed dynamically. There is a number of tools based on dynamic browser fuzzing. *crossfuzz* generates extremely long-winding sequences of DOM

Authorized licensed use limited to: National Chiao Tung Univ.. Downloaded on May 12,2020 at 06:21:20 UTC from IEEE Xplore. Restrictions apply.

operations, inspecting returned objects, doing recursion, and creating circular node references that stress to test garbage collection mechanisms. *ndujafuzz* shows an evolutional approach of browser fuzzing that relies on some DOM interfaces introduced by W3C DOM Level 2 and Level 3 specifications.

TABLE I shows a comparison of two types of browser fuzzing. On the issue of resource consumed, static browser fuzzing generates a lot of inputs and the input needs to execute one by one. Dynamic browser fuzzing could use one input to generate different conditions of executed instructions when browser consume a certain type of inputs. So, static browser fuzzing needs extra resource spent on opening and closing browser when fuzzing the browser. Moreover, the key issue in static browser fuzzing is how to generate more effective input to crash the target program. Because how to change is determined after generating the initial input, it is critical to have a good method for input generations. However, dynamic browser fuzzing has only one seed and tries to generate different input, so the key issue is how to reconstruct the input more effectively.

TABLE I. COMPARISONS BETWEEN STATIC AND DYNAMIC BROWSER FUZZING

	Static	**Dynamic**
Frequency to load into browsers	More	Less
Resource consumed	More	Less
Important point	Input generation	Input reconstruction
Record condition of crash	Easier	Much difficult

III. PROBLEM STATEMENT

We propose a browser fuzzing framework that is able to find the crashes efficiently by mixing with the solutions of the black-box browser fuzzing. We highlight the terminology and assumption, and then discuss the problem statement.

A. Terminology and Assumptions

We describe the notation in Table II. In fuzzing category, S and Mut stand for the component of browser fuzzing and $Input$ is the pair of S and Mut. In the experiment, we would feed in_p to browser b, and let b run for tout second before terminate b or terminate b by crashing. Then, we would update the CRS after that and compute PS and PM by different scheduling mechanisms again. In final, after scheduling the seed and mutation, fuzzing the b and update CRS for t, it would finish the mission.

In the problem, though input in browser won't include just the html file only, we will focus on html files in this paper to reduce the complexity of the problem. Besides, though the more time you fuzz the browser, the more crashes you would find, we will fuzz in fixed time to obtain each performance result.

B. Problem Statement

This work is intended to find an efficient browser fuzzing architecture. The problem statement is as follows. We assume that given browser b as the target, we would use the mentioned probability computing method to compute PS and PM. And then, using PS and PM to schedule s_n and mut_m from S and Mut to generate the in_p for b. We would update CRS after executing in_p and compute PS and PM again. We would schedule other seed and mutation and generate another in_p again. It would continue until executing t. In final, we would get the performance by comparison the $|CRS|$.

TABLE II. NOTATIONS DESCRIPTION

Categories	Notations	Descriptions
Fuzzing	$S = \{ s_n \mid n \geq 1\}$	The set of the seed s_n that would become the input after the seed combines with a mutation.
	$Mut = \{ mut_m \mid m \geq 1\}$	The set of mutation mut_m that would become the input after the mutation combines with a seed.
	$Input = \{in_p \mid in_p \in (S, Mut), p \geq 1 \}$	The set of the input in_p, the input means the pair of seed and the mutation.
	$PS = \{ ps_n \mid n \geq 1 \}$	The set of the seed's probabilities ps_n, the possibility is used by schedule mechanism to select a seed as a time.
	$PM = \{ pm_m \mid m \geq 1 \}$	The set of the seed's probabilities pm_m, the possibility is used by schedule mechanism to select a mutation as a time.
	t	A given time for fuzzing.
	t_{out}	A given timeout for executing a browser b.
	b	The browser the fuzzing executed
Result	$CRS = \{ crs_{n,m} \mid n \geq 1, m \geq 1 \}$	The set of result $crs_{n,m}$, $crs_{n,m}$ represents the number of the crash that found after executing $in_p = (s_n, mut_m)$.

IV. SCHEDULED DOM FUZZING:

A. Overview

Unlike the structure of traditional browser fuzzing just with single seed and mutation to find the crashes, our SDF method use dynamic structure of seed and change the mutation scheme. We select a random seed and mutation parameter for the fuzzing process. Because of this, we select a pair of the seeds and mutation used by existing browser fuzzing tools. In this case, we may just fuzz as same as the original browser fuzzing. We would further create a pair that is new and may not be used to test before, so we may be able to find more different results due to the dynamic pair of seeds and mutations. We might create more different results and find some unknown crashes even with existing solutions.

Although it might create more and unknown possibilities, it could also create many useless or redundant results from some pairs. It will take much time and sacrifice the performance.

We therefore use the concept of scheduling in black-box fuzzing to increase the performance by selecting proper seed and

Authorized licensed use limited to: National Chiao Tung Univ.. Downloaded on May 12,2020 at 06:21:20 UTC from IEEE Xplore. Restrictions apply.

mutation. From the past simulation experience, we would determine the better selection method.

Our algorithm showing in Fig. 1 presents the whole design. We have to generate some seeds *S* and mutations *Mut* by ourselves or from the existing systems. We would use the probabilities *PS* and *PM* to schedule the seed and mutation. How to compute *PS* and *PM* is the critical part for scheduling mechanism, because probability *PS* and *PM* would determine which seed and mutation will get more possibility to be selected. The probability computing method will be described later.

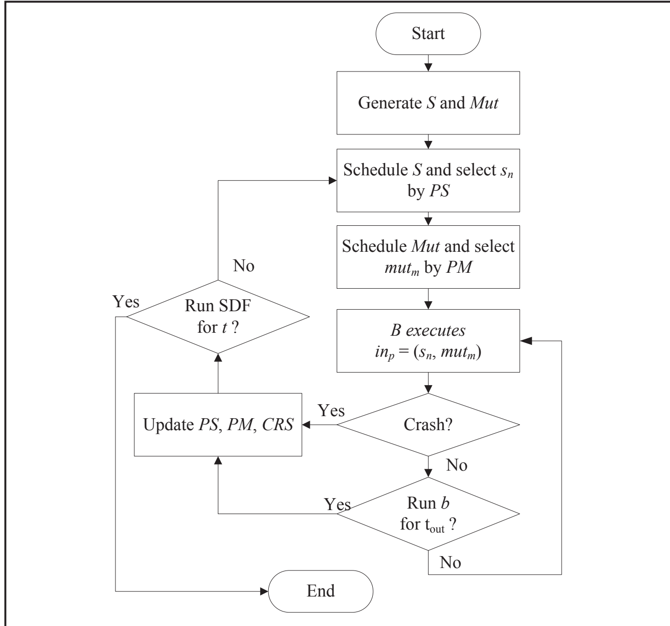

Fig. 1. Our algorithm

After generating in_p, we will let browser *b* executes with in_p and observe whether *b* would crash in t_{out}. We would record the result by updating *CRS*, *PS* and *PM* after executing *b*.

To sum up, we have two probability computing methods for scheduling mechanism to select the seed and mutation from pools, and then use the pair as the input for feeding to the browser. In the final phase, we would record the result regardless of crashing or not to affect the decision in the future.

B. Seed and Mutation Modules

In the architecture, the seed input will be in a set that includes more than one seed, and the mutation. We will introduce a scheduling mechanism for selection by combining the seed and mutation to form the input for browser. In this case, seed will be mutated with the preset rules when browser executing. However, because the result of dynamic browser fuzzing will not be known, and our mutation is a type of dynamic browser fuzzing, the pair combined of the same seed and mutation will not create the same result. Hence, we will create lots of input that is based on the dynamic browser fuzzing. The more seeds and mutations we provide, much more crash input and possibilities we will obtain. We can find more crashes due to this changing scheme.

C. Scheduling Mechanism for Selection

In the mechanism of how to schedule, we compute selected probability for seed and mutation, i.e. *PS* and *PM*. We think how to select mutation is more important than seed. So, we only determine a method to compute *PS*. The method to compute *PS* is called weighted random. It is a method with more crashes found to have the higher priority. We think the seed with more crashes will find other crashes again. We compute the selected probability of kth seed as

$$ps_k = \frac{\sum_{m=1}^{|Mut|} crs_{k,j}}{\sum_{i=1}^{|S|}\sum_{j=1}^{|Mut|} crs_{i,j}}.$$

In the scheduling of mutation, we use four mechanisms, including two history non-cared methods and two history-cared methods. In history non-cared methods, we use normal random selection and round robin selection. It doesn't matter the result of the mutation in history. In history-cared methods, we record the result after fuzzing the browser with the pair of mutation each time.

Weight random is a method with more crashes found to have the higher probability for scheduling mechanism of seed. We compute the selected probability of kth mutation as

$$pm_k = \frac{\sum_{i=1}^{|S|} crs_{i,k}}{\sum_{j=1}^{|Mut|}\sum_{i=1}^{|S|} crs_{i,j}}.$$

ε - weight random can be a trade of uniform random and weight random. It means that every mutation has a basic probability and gives the extra probability to the mutation to find more crashes. "ε" is a decimal faction that between 0 to 1 and means the portion of preservation for each mutation. If we want to compute the selected probability of kth mutation as

$$pm_k = \frac{\varepsilon}{n} + \frac{\sum_{i=1}^{|S|} crs_{i,k}}{\sum_{j=1}^{|Mut|}\sum_{i=1}^{|S|} crs_{i,j}} \times (1-\varepsilon).$$

V. Implementation

Our framework is based on the FOE fuzzing framework with a browser fuzzing mode to support the method of dynamic browser fuzzing. We preserve the original seed pool and create an additional mutation pool. We use the components of FOE including the seed selection method, debugger tool and crash triage method.

In practical situations, the seed of browser fuzzing is an html file with a clear structure. We can divide the seed pool into two groups. One group is a simple structure with less elements or with some specific elements. Another group includes some public web pages. The reason of collecting from well-known web pages is that the structure is more complicated than other small sites. The mutation of browser fuzzing is a script written in JavaScript. We use the mutation from *crossfuzz* and *ndujafuzz*. Especially, we also write a simple mutation. We try to combine different mutations of existing fuzzing tools with our own mutation method.

VI. Experiment and Result Analysis

We first present the performance of browser fuzzing in three metrics: throughput, crash input generation rate and the influence of optimal scheduling. Throughput and crash generation rate are used for measuring performance of the browser fuzzing in the same environment and time interval. On the optimal scheduling, we will find the proper scheduling mechanism for choosing suitable mutations.

Authorized licensed use limited to: National Chiao Tung Univ.. Downloaded on May 12,2020 at 06:21:20 UTC from IEEE Xplore. Restrictions apply.

A. Experimental Environment

Although the fuzzing method can be applied in different browsers, we use IE as our main target. We conduct the experiment on VMware 10.0.4 configured with a CPU and 1G RAM. We use the Windows XP SP3 distribution and Window 7 for our experiment. We also use the debugging function and crash triage from FOE.

In FOE, the configuration will set target application to IE and timer defined in configuration of FOE to 25 seconds. FOE will feed the input to the target application within 25 seconds and then terminate the application. We select IE 8 as our main version for fuzzing. Before fuzzing, we have to set the configuration by turning off the function of automatic recovery and turn on automatic execution of JavaScript.

B. Throughput

We compare four browser fuzzing types with IE8 in the same environment for five days. FOE uses a simple mutation method provided by *zzuf*. Mutation in *zzuf* will change a certain bytes in the seed as used in black-box fuzzing. The seed mutated by *zzuf* is the same as the one used by SDF. *crossfuzz* and *ndujafuzz* will use the original seed. We haven't changed the rules of *crossfuzz* and *ndujafuzz*, and the default configurations are remained. However, since there are some compatibility issues in different versions, we must fix them before the fuzzing process begins. All experiments are performed in FOE with our scheduling and mutation schemes, and all crashes are handled and analyzed by FOE. FOE will classify the crashes into several types, including probably, not exploitable, probably exploitable, exploitable and unknown situations by !exploitable[20].

TABLE III. THE THROUGHPUT OF FUZZING

	Total found Crashes	Useful crashes		
		Unknown	Probably exploitable	Exploitable
zzuf	2988	0	0	0
crossfuzz	26	2	3	13
SDF	50	2	6	12
SDF*	74	3	15	23

We also want to observe the differences with or without using the architecture. The architecture includes seed pool, scheduling mechanism and the timer for execution. We fix the used mutation and compare the results of using architecture or not. The used mutation in SDF* is the mutation of crossfuzz only. It includes the same mutation as crossfuzz, but the architectures are different.

The results of throughput in terms of total number of unique crashes found are shown in Table III. zzuf can only find unexploitable crashes and not as much effective as other browser fuzzing tools. As for crossfuzz, it finds more crashes classified as "unknown" and "probably exploitable". SDF finds about the same crashes as crossfuzz since SDF includes crossfuzz as a fuzzer and it should generates different inputs that others don't have but it doesn't behave as our expectation. However, SDF* finds more crashes than crossfuzz and SDF. The reason is that SDF doesn't include mutation of crossfuzz only, so it would use other mutation to fuzz IE. If other mutation can find crashes effectively, it just waste time and find fewer crashes in the end.

However, SDF* could find 2.27 (41/18) times more number of crashes than crossfuzz. The reason is that the crashes found by crossfuzz may have been fixed and SDF* should explore different paths for other potential inputs.

C. Rate of Useful Crash Finding

Other than the number of the crashes generated in a fixed time interval, we also care about the speed of crash finding. By observing the speed of crash finding, we could know how effective the fuzzing method is in a fixed time interval. However, because some crashes are not exploitable, they are not useful actually. So, we focus on the crashes that are exploitable and ignore the crashes that are probably not exploitable.

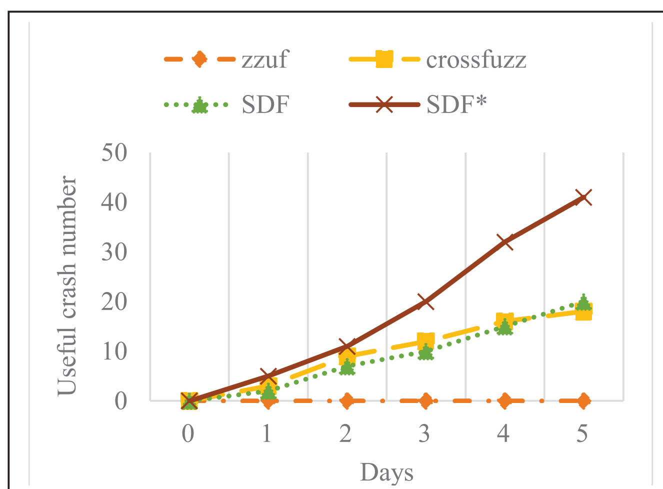

Fig. 2. Rate of crash finding

In Fig. 2, we show that zzuf does not find any exploitable crashes, so it is not an effective tool for fuzzing browser. SDF can find more useful crashes than crossfuzz in the beginning, and find more crashes within five days. We show that SDF may find another crashes from previously generated crashes. SDF selects the specific seed and mutation for speedup. Though crossfuzz won't recycle the former crashes found, it is good enough and find crashes constantly. It is efficient due to the improved architecture with scheduling and mutation schemes. The scheduling mechanism would learn and constantly find useful crashes.

D. Optimal scheduling

In this experiment, we evaluate the influence of scheduling for getting better fuzzing results. In Table IV, we find that if the scheduling algorithms ignore the mutation for the crash in the history record, we cannot get more crashes. We thus need a better algorithm to select the mutations. It is more efficient than using random or round robin scheduling methods.

Authorized licensed use limited to: National Chiao Tung Univ.. Downloaded on May 12,2020 at 06:21:20 UTC from IEEE Xplore. Restrictions apply.

TABLE IV. COMPARISON BETWEEN SCHEDULING ALGORITHMS

	Total found Crashes	Useful crashes		
		Unknown	Probably exploitable	Exploitable
Uniform random	34	2	7	4
Round robin	31	1	4	4
Weighted-Random	26	0	7	5
ε- Weighted-Random	27	0	4	4

E. Types of the useful crashes found

From the previous results, we have shown that we found useful crashes whatever is found by crossfuzz or SDF. In this, we would show the message from !exploitable. From Fig. 3, it is the result from SDF with the mutation of crossfuzz. We could see that most of the crashes are raised from access violations.

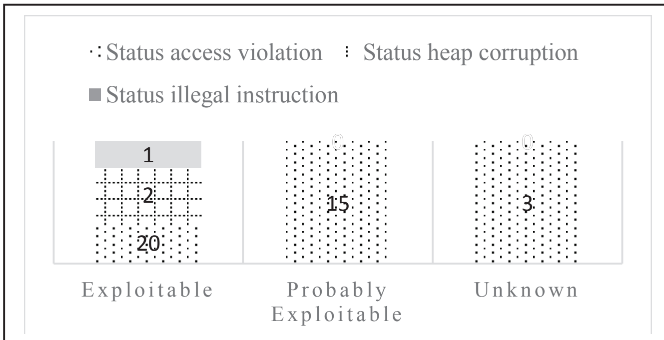

Fig. 3. The type of found crashes result

In the exploitable crashes, 17 crashes with the status "access violation" try to access DEP (data execution prevention), two crashes write in user mode and one crash tries to access on invalid control flows. In the probably exploitable cases, nine crashes with "access violation" find a faulty address which would be used for a branch and six crashes try to access DEP. In the unknown cases, one is to find the fault address would be later used to determine whether a branch is taken, one is read access violation and the other is write access violation in user mode.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we focused on browser fuzzing. By designing a new architecture for fuzzing, our solution is able to find 2.27 times of number of useful crashes more than the original fuzzing method. By changing the mutation scheme with different seed selection method, we successfully find crashes from different seed inputs. If you want to find crashes of browsers, the most important part is the mutation. How to determine whether mutation is good or not would depend on analyzing the structures from fuzzing. If you have a proper mutation, then using our architecture would efficiently find crashes. If you find more than one mutation that is able to find useful crashes, adding scheduling mechanism would reduce the time to find crashes and you would have more time to explore unknown crashes. Unfortunately, SDF has only one mutation that finds useful crash, so it is unobvious to see the effect of scheduling mechanism.

In the future, we will develop a reproducing mechanism for the crash input, because it is difficult to record persistent information for dynamic browser fuzzing. We would try to record them for analysis by keeping the parameters of mutation to reproduce the crash input. Besides, we could use this solution to fuzz different browsers and try to find previously unknown crashes. We can add other seeds and mutation methods to find more crashes and a better scheduling algorithm to better improve the SDF.

REFERENCES

[1] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Security & Privacy, IEEE,* vol. 9, pp. 49-51, 2011.
[2] R. Sherstobitoff and M. Itai Liba, "Dissecting Operation Troy: Cyberespionage in South Korea," ed: McAfee White Paper, 2013.
[3] W. L. F. W. A. Arbaugh, and J. McHugh, "Windows of vulnerability: A case study analysis," pp. 52-59, 2000.
[4] L. F. B. P. Miller, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM,* vol. 33, pp. 32-44, 1990.
[5] A. G. M. Sutton, and P. Amini, "Fuzzing: brute force vulnerability discovery," 2007.
[6] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on*, 2012, pp. 152-156.
[7] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, 2012, pp. 78-87.
[8] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, *et al.*, "Optimizing seed selection for fuzzing," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 861-875.
[9] *The ZZUF fuzzer*. Available: http://caca.zoy.org/wiki/zzuf
[10] W. Dorman, "CERT Basic Fuzzing Framework," 2010.
[11] *Failure Observation Engine (FOE)*. Available: http://www.cert.org/vulnerability-analysis/tools/foe.cfm?
[12] S. K. C. M. Woo, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," *in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security,* pp. 511-522, 2013.
[13] Symantec, "Internet Security Threat Report 2014," 2014.
[14] *Bf3*. Available: http://www.aldeid.com/wiki/Bf3
[15] M. Zalewski. (2011). *crossfuzz*. Available: http://lcamtuf.blogspot.tw/2011/01/announcing-crossfuzz-potential-0-day-in.html
[16] R. Valotta, "Taking Browsers Fuzzing To The Next (DOM) Level," 2011.
[17] W3C. *Document Object Model (DOM) Technical Reports*. Available: http://www.w3.org/DOM/DOMTR
[18] S. D. Cook and J. S. Brown, "Bridging epistemologies: The generative dance between organizational knowledge and organizational knowing," *Organization science,* vol. 10, pp. 381-400, 1999.
[19] A. Aphale. *Introduction to browser fuzzing*. Available: http://www.slideshare.net/null0x00/introduction-to-browser-fuzzing
[20] Microsoft. *!exploitable Crash Analyzer - MSEC Debugger Extensions*. Available: https://msecdbg.codeplex.com/

Authorized licensed use limited to: National Chiao Tung Univ.. Downloaded on May 12,2020 at 06:21:20 UTC from IEEE Xplore. Restrictions apply.