Boosting Fuzzing Performance with Differential Seed Scheduling

Chung-Yi Lin
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
daniel810736@gmail.com

Chia-Wei Tien
Cybersecurity Technology Institute
Institute for Information Industry
Taipei, Taiwan
emmily@iii.org.tw

Chun-Ying Huang
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan
chuang@cs.nctu.edu.tw

***Abstract*—Fuzzing is a common technique used to perform automated vulnerability discovery. Fuzzing performance could be improved by various means. In this paper, we discuss the impacts of seed scheduling, and propose differential seed scheduling to maximize fuzzing performance by increasing the number of crashes identified within a limited time. Differential seed scheduling works for grey-box fuzzers that generate seeds based on runtime code coverage measurement. It attempts to evaluate the value of fuzzing seeds and selectively pick the best one to achieve balance between fuzzing effectiveness and efficiency. Our contribution is four-fold. First, we proposed differential seed scheduling to improve overall fuzzing performance. Second, we implemented AFLExplorer by integrating differential seed scheduling with the open-source American Fuzzy Lop (AFL) fuzzer. Third, we conducted in-depth experiments with AFLExplorer to show the effectiveness and the efficiency of seed scheduling. Our evaluations showed that AFLExplorer can discover up to 90% more unique crashes compared with a vanilla fuzzer. Last, we reported newly identified vulnerabilities to the authors of the tested applications, had them fixed, and 15 common vulnerabilities and exposures (CVE) numbers were assigned as of writing of this paper.**

***Index Terms*—Fuzz testing; greybox fuzzing; hamming distance; software security**

I. Introduction

Managing vulnerabilities is the most critical work for security engineers. According to statistics, the percentage of critical vulnerabilities has been increasing since 1988 [1]. Thus far, the computer security resource center of NIST[1] has collected more than 90K vulnerabilities in the national vulnerability database [2]. Statistics obtained by Flexera Software [3] show that 17,147 vulnerabilities were discovered in 2,136 applications from 246 vendors in 2016. This number represents a 33% increase in the past five years and a 6% increase from 2015 to 2016. These observations also reflect the fact that managing vulnerabilities could be increasing tough for security engineers, especially when attackers relentlessly attempt to discover new vulnerabilities.

Automated discovery of vulnerabilities could be very helpful for security engineers. It is obvious that discovering vulnerabilities manually would cost too much time and human resources. Therefore, many techniques have been developed for automated discovery of vulnerabilities. Among all the techniques, fuzz testing (or fuzzing) and symbolic execution are two of the most commonly used techniques. Initially, fuzzing [4] was used as an automatic testing approach used to understand the reliability of UNIX tools [5]. A tool was said to be more reliable if fewer software bugs were identified in it. Since then, fuzzing has been developed as a general-purpose technique for discovering bugs in software programs. For simplicity, we call a tool based on fuzzing techniques a fuzzer. A fuzzer usually starts with generating test inputs by mutating a set of given inputs [6]. The generated inputs are then fed to a program under testing and the interaction of the program with the inputs is monitored. The basic idea of fuzzing is that feeding different inputs to a program would trigger different execution paths inside the program. When many execution paths are triggered, the paths containing a bug could lead to a crash, which can be used as an indication of a possible vulnerability[2]. As a result, the effectiveness of a fuzzer relies hugely on the generated inputs. One drawback of fuzzers is that they trigger only shallow paths in a program under testing. This is because fuzzers might be unaware of program designs, and the inputs are mutated blindly based on the strategies implemented in fuzzers. These blindly generated inputs may be useful for finding shallow paths (and shallow bugs), but finding deep paths inside a program could be very arduous.

The other commonly used technique for automated vulnerability discovery is symbolic execution [7], [8], [9], [10], [11], [12]. Symbolic execution attempts to explore different execution paths that a program could take for different inputs. Instead of blindly mutating the inputs, symbolic execution analyzes the variables and statements of a program and then enumerates the inputs that would explore all possible paths in a program. Despite the effectiveness of this method, exploring all the paths could result in path explosion, which would make infeasible the application of this method to large programs. Researchers [13], therefore, suggested limiting the execution time of path exploration to achieve a balance between efficiency and effectiveness for symbolic-execution-based program testing. Compared with symbolic execution, fuzzing

[1]NIST stands for National Institute of Standards and Technology.

[2]A vulnerability is often caused by a bug, but a bug does not always lead to a vulnerability.

offers a significant advantage in terms of testing speed because it does not require solving constraints and, therefore, runs much faster in general. Many recent vulnerabilities have been exposed using fuzzers instead of symbolic-execution-based tools. Although fuzzers are efficient, programs nowadays tend to be increasingly complex, and fuzzers would require smart inputs to test a program thoroughly. Hence, recent fuzzers remain incapable of dealing with complex programs, which means there is considerable scope for improvement regardless of whether the focus is on efficiency or effectiveness.

One major challenge in fuzzing is *selecting the best input to perform fuzzing*. A fuzzer keeps generating new inputs from initial inputs or existing inputs when performing tests. In fuzzing, an input is often called a seed[3] because it is often used to generate other possible inputs by using various mutation strategies. This means a fuzzer often must maintain a pool of inputs. The size of the pool keeps growing, and a fuzzer must select inputs iteratively from the pool to perform tests. The selected seeds could greatly affect fuzzing performance. For example, consider two seeds s_1 and s_2, and a simple program that contains a main function (program entry point) and two functions F and G. Suppose the main function calls F, and F calls G. Given that s_1 triggers a path that only traverses the main function, and s_2 triggers a path that traverses both the main function and function F. It is trivial that selecting s_2 would be better because this selection would lead to a higher coverage on both the main function and the function F. A bug in the main function would be detected by both s_1 or s_2. However, a bug in function F would be detected only by s_2, regardless of the number of times s_1 is fuzzed. Furthermore, suppose inputs s_1' and s_2' are generated by mutating s_1 and s_2, respectively. It might be easier for s_2' to reach G compared with s_1'. If a fuzzer randomly selects a seed from a pool of seeds, it could perform many meaningless and redundant tests for s_1 and inputs similar to s_1. Sometimes, it could be even worse if a fuzzer always selects a seed based on the seeds' sequence number. In real-world test cases, there could be thousands of seeds in a pool, and the selected seeds, therefore, becomes much more critical to fuzzing performance. Based on the observations, a smarter strategy is required to determine the seed that would be the best to perform fuzzing and mutation.

In this paper, we propose differential seed scheduling, a novel approach to seed rescheduling. Differential seed scheduling aims to maximize the number of unique crashes discovered in fuzzing. It can be applied to grey-box fuzzers. Differential seed scheduling attempts to increase testing coverage by selecting proper inputs from a pool of currently available seeds and, therefore, discover more possible vulnerabilities while satisfying a given time constraint. In this manner, redundant fuzzing workloads can be eliminated, and fuzzing performance can be improved. Our contribution is four-fold. First, we propose differential seed scheduling to improve the overall fuzzing performance. Second, we implement AFLExplorer by integrating the proposed differential seed scheduling with the open-source American Fuzzy Lop (AFL) fuzzer. Third, we use AFLExplorer to perform fuzzing on several applications and show its effectiveness by discovering more unique crashes. Finally, we report a number of identified new vulnerabilities to the authors of the tested applications. Our evaluations show that AFLExplorer can discover about 90% more unique crashes compared with previous improvements [14], [15].

The remainder of this paper is organized as follows. The detailed design of differential seed scheduling is introduced in Section III. Section IV introduces AFLExplorer, an implementation of differential seed scheduling. In addition, the effects of system parameters are discussed and the evaluation results are presented. Research works relevant to differential seed scheduling are discussed in Section II. Finally, our concluding remarks are given in Section V.

II. Related Work

A. Type of Fuzzing

Fuzz-testing tools can be classified into two groups: mutational fuzzing [16] and grammar-based fuzzing [17]. Mutational fuzzing often generates new inputs by mutating seeds with various strategies such as bit flipping, byte flipping, subtly increasing or decreasing integer values in seeds, and even randomly generating a new seed by mixing two existing seeds. Grammar-based fuzzing generates seeds from a specification, which means it obeys the format of the application strictly. Details of the two classes are as follows.

Mutational Fuzzing. Godefroid et.al [18] implemented SAGE, a white-box fuzzed with a novel directed-search algorithm to maximize the number of new test inputs generated from each symbolic execution. SAGE boosts performance by systematically negating the constraints in a given path one by one, in conjunction with the prefix of the path constraint leading to it, and attempts to be solved by a constraint solver. In this manner, a single symbolic execution can generate multiple test inputs.

Haller et.al [19] combined taint tracking, program analysis, and symbolic execution to find buffer overflow and underflow. Their method focuses on arrays in a loop to decrease complexity and uses taint analysis to determine which bytes influence the array index and then executes the program symbolically. This leads to finding two previous undocumented buffer overflow bugs.

Grammar-Based Fuzzing. Research on grammar-based fuzzing started in the 1970s [20], and it can be divided into two categories, random [21], [22] and exhaustive generation [23].

Godefroid et.al [24] enhanced white-box fuzzing of complex structured input applications with a grammar-based specification of their valid inputs. A novel dynamic test generation algorithm in which symbolic execution directly generates grammar-based constraints is presented. Their result shows that compared with other white-box fuzzers, their grammar-based white-box fuzzing provides 28% additional code coverage to their dataset. A few grammar-based fuzzing schemes use other methods. Dewey et al. [25] used constraint logic programming (CLP) for program generation. Using CLP,

[3]We use the terms "input" and "seed" interchangeably in this paper.

testers can write declarative predicates specifying interesting programs, including syntactic features and semantic behaviors. The results show that the CLP-based approach performs better than stochastic grammars for generating interesting programs.

B. Seed Selection

Improving the efficiency of fuzzing has been an issue for decades. Several research works have been devoted to improving fuzzing performance based on different seed selection approaches. One optimization approach [26], [27] is to precompute the value of seeds before fuzzing and select the top-notch seeds. According to Miller's report [28], a 1% increment in code coverage leads to a 0.92% increment in the number of bugs found. Robert et al. [26] evaluated six seed-selection algorithms, and the algorithm that outperformed the other algorithms selected seeds by maximizing code coverage. Different from these works, differential seed scheduling does not select a specific portion of seeds before fuzzing, but it progressively selects a proper seed from the existing seeds during fuzzing.

Another seed-selection improvement in AFLFast [15] is seed selection based on the number of times a seed is fuzzed before and the one that exerciser lower-frequency paths. The concept of AFLFast is to focus on low-frequency paths to discover more paths and thus increase efficiency.

Woo et al. [29] investigated how to schedule the fuzzing of program-seed pairs and maximize the number of unique bugs found within a limited time. They constructed a mathematical model of the problem by using multi-armed bandit algorithms and evaluated 26 existing online scheduling algorithms. In addition, they computed a seed's energy and ends up having the tendency towards generating more crashing inputs for already known errors.

C. Fuzzing Boosting

In addition to seed scheduling, several other techniques have been proposed to improve fuzzing performance. AFLFast [15] evaluates several power strategies to control the number of inputs generated from a seed. The objective is to limit the amount of fuzzing in each round to a reasonable number. Compared with the original AFL, AFLFast introduced two constant and four monotonous power schedules instead of always assigning a constant high amount of energy in each round. Hence, AFLFast is more likely to distribute correct energy to seeds in order to trigger crashes.

VUzzer [30] argues that most grey- and black-box fuzzers tend to be application-agnostic, which makes them unavailable to explore bugs that lie in deeper levels. Hence, VUzzer improves fuzzing performance by analyzing the control and data-flow features of the application to obtain interesting properties of the input. Angora [31] attempts to improve fuzzing performance without solving constraints. Instead, the authors propose to perform several program analyses including taint-tracking, context-sensitive branch counting, searching based on gradient descent, and input length exploration. It then performed seed mutations based on the analyzed results. Driller [32] combines fuzzing and concolic execution to overcome the difficulty of finding deeper bugs while avoiding path explosion. The core idea of Driller is that inexpensive fuzzing is used for exercising the compartments of an application, and concolic execution is used for generating inputs that satisfy the complex checks required for separating the compartments. Within Driller, fuzzing focuses on exploring interesting program activities. If fuzzing fails to find new paths, Driller switches to concolic execution for the loops and inner checks, and then switches back to fuzzing. Wang et al. [27] targeted programs that take highly structured files as input. Usually, they are processed in the following stages: syntax parsing, semantic checking, and application execution. Deep bugs are often hidden in the application execution stage, and most seeds will be rejected at the syntax parsing stage, which makes it difficult to trigger bugs. Wang et al. [27] implemented a novel data-driven seed generation approach called Skyfire. Skyfire cooperates with AFL to overcome this difficulty. The seeds generated by Skyfire are fuzzed by AFL, and the results show improvements of 20% and 15% in line coverage and function coverage, respectively.

Cha et al. [33] focused on optimizing the mutation ratio in mutational fuzzing to maximize the probability of finding bugs in black-box fuzzing given a seed and an application. They derived a mathematical framework to model their approaches. Their tool, called SYMFUZZ, found an average of 37.2% more crashes than other black-box fuzzers in eight applications within the same time. Kargén et al. [34] worked on program mutation at the binary-code level. They presented a novel approach to automatic test case generation by adding to and subtracting from the result of a computation and switching bitwise `AND`/`OR` operators. Their approach found a total of 16 crashing inputs and 8 unique bugs among them.

III. Differential Seed Scheduling

A. Design

This section explains the design of differential seed scheduling. For ease of computation and implementation, runtime code coverage is preserved in a bit string of length n initialized to all zeros. Visited blocks or tuples are marked in the bit string based on the involved block identifiers. In AFL, a fixed size character array called "trace bits" is used as the bit string to maintain runtime code coverage in a shared memory region.

Once runtime code coverage is preserved in the bit string, the effectiveness of each seed can be justified based on the bit string generated by it. The assumptions of differential seed scheduling are as follows:

1) A seed producing higher runtime code coverage would mark more bits in the bit string; and
2) Seeds traversing different paths in the same program would produce different bit strings.

Based on these assumptions, differential seed scheduling measures the hamming distances between bit strings produced by different seeds. It then assigns priorities to seeds based on the differences in the measurement results. Differential

Algorithm 1 *ChooseNext* function for differential seed scheduling.

Input: All available seeds in queue Q, $Q = \{s_1, s_2, \ldots, s_n\}$
Input: Queue B stores the corresponding bit string for each seed in Q, $B = \{b_1, b_2, \ldots, b_n\}$
1: $H = \phi$
2: **for** $i = 1$ **to** $|Q|$ **do**
3: **for** $j = 1$ **to** $|Q|$ **except** i **do**
4: $h_{ij} = hamming\_distance(b_i, b_j)$
5: add h_{ij} to H
6: **end for**
7: **end for**
8: $t = choose\_best(Q, H, F, S)$
Output: Recommended seed t

seed scheduling replaces the *ChooseNext* function introduced in AFL [15] with its own implementation, described in Algorithm 1. Instead of the single parameter input Q used in the *ChooseNext* function, the *ChooseNext* function used in differential seed scheduling takes two inputs Q and B, which store all currently available seeds and the corresponding coverage bit strings of each seed in Q, respectively. The bit string is available after an input in Q is fuzzed. Therefore, the bit string for each fuzzed seed can be preserved in B by slightly modifying the core Algorithm of the AFL fuzzer.The for-loop in Algorithm 1 computes the hamming distance for all possible pairs of bit strings and stores the results in H. At the end of the algorithm, the *choose_best* function is called to pick the recommended seed from queue Q based on the hamming distances stored in H. Algorithm 1 is the simplified *ChooseNext* function for illustration. Although it looks like the complexity involved in computing H is $O(|Q|^2)$, several methods can be used to minimize the computation cost. For example, the values of h_{ij} and h_{ji} would be identical, and therefore, half of the hamming distance computations can be eliminated. Alternatively, H may be implemented with global scope and pairwise hamming distances can be updated incrementally without computing everything from scratch.

The key function in Algorithm 1 is the *choose_best* function. In addition to the seed queue Q and seed coverage bitmaps H, the *choose_best* function requires a *scoring function* F and a *seed selector* S for making recommendations. The details of F and S are covered in Sections III-B and III-C, respectively.

B. Scoring Function

Differential seed scheduling uses two scoring functions for the *choose_best* function. One is based on the average hamming distance (**avg**) and the other on the maximum hamming distance (**max**). The details are as follows.

*1) Scoring based on Average Hamming Distance (**avg**):* The **avg** scoring function is defined as follows. Given inputs Q and H from Algorithm 1, the *choose_max* function computes score v_i of each seed s_i in Q by using the equation

$$v_i = \frac{\sum_{j=1, j\neq i}^{|Q|} h_{ij}}{|Q| - 1}. \quad (1)$$

Once we have v_i for all seeds in Q, the *choose_max* function returns the seed with the highest score. The rationale behind the equation is that the *choose_max* function attempts to find a seed that produces the most dissimilar runtime coverage compared with the coverages produced by *all other existing seeds*.

*2) Scoring based on Maximum Hamming Distance (**max**):* The **max** scoring function is defined as follows. Given inputs Q and H from Algorithm 1, the *choose_max* function computes score v_i of each seed s_i in Q by using the equation

$$v_i = \max_{\substack{1\leq j\leq |Q| \\ j\neq i}} h_{ij}. \quad (2)$$

Once we have v_i for all seeds in Q, the *choose_max* function returns the seed with the highest score. The rationale behind the equation is that the *choose_max* function attempts to find a seed that produces the most dissimilar runtime coverage compared to the coverage produced by *one specific seed*.

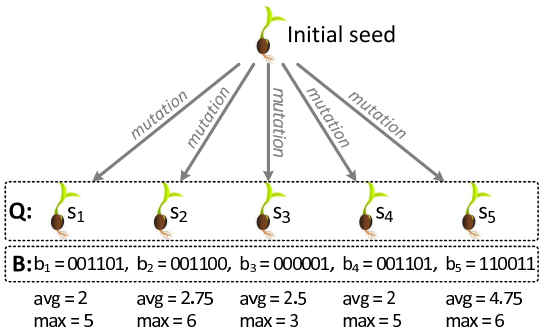

Fig. 1: Simple example to demonstrate how **avg** and **max** score functions are computed.

A simple example to show how **avg** and **max** scores are computed is given in Figure 1. An initial seed is mutated to generate five seeds. After fuzzing the five seeds, suppose all of them are added to Q and the corresponding coverage bit strings are preserved in B as well. The *choose_next* function is then called in the next iteration. Instead of choosing s_1 in Q, differential seed scheduling evaluates all-pair hamming distances for all bit strings in B and selects the seed with the highest score. In this example, the **avg** score of s_1 can be evaluated as $(\sum_{j=2}^{4} h_{1j})/4$, which is $(1 + 2 + 0 + 5)/4 = 2$. Similarly, the **max** score of s_1 can be evaluate as $\max_{2\leq j\leq 5} h_{1j}$, which is $\max(1, 2, 0, 5) = 5$. Once the scores of all seeds in Q are obtained, differential seed scheduling chooses s_5 and s_2 as the seeds for the next iteration, depending on whether the **avg** or the **max** scoring function is used.

C. Seed Selector

Although computing hamming distances is helpful for recommending seeds with distinguishable coverage, relying only

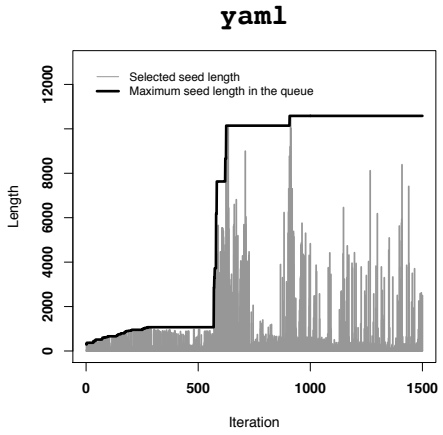

Fig. 2: Growth of seed length in differential seed scheduling without using a selector.

on hamming distances could seriously degrade fuzzing performance. An influential factor governing fuzzing performance is the length of the selected seeds. In general, a program would require more time to process a longer seed (input). As a result, the use of longer seeds could lead to slower fuzzing. Figure 2 shows the maximum length of seeds and the length of selected seeds for each iteration. The fuzzed program was `yaml`. The experimental results were collected from a fuzzing process that implemented differential seed scheduling with the **avg** scoring function but without a selector. Therefore, the fuzzer always selected the seed with the highest score (coverage) from the seed queue. There were 37 initial seeds, and the average length and the maximum length of the initial seeds were 103 bytes and 413 bytes respectively. The thick black line in Figure 2 shows that the maximal length of seeds in the queue increases dramatically from several hundreds of bytes to several thousands of bytes. However, the dark grey area indicates that the selected seed lengths are independent of the maximal length. In addition, the running time for the first 1000 iterations was approximately 10 minutes on our test machine, but the running time for the last 500 iterations was about 1 hour. This result also reflects that working with longer seeds would greatly affect fuzzing efficiency. Based on the observations, differential seed scheduling considers two seed selectors to manage seed selection and attempts to achieve a balance between fuzzing effectiveness and fuzzing efficiency.

1) Selector based on Length Growth Constraint (λ): The objective of the λ-selector is to achieve a balance between fuzzing efficiency and fuzzing effectiveness. Because of possible performance degradation due to longer seeds, the λ-selector selects a seed with the highest contribution (score) and fits in an adaptive length constraint. Without a length growth constraint, it is possible that a fuzzer could always select longer seeds from the pool, while ignoring the short but effective seeds. The length growth constraint adaptively updates the length upper bound for seed selection. The λ-selector evaluates the average length μ and the standard deviation σ of all seeds in the seed queue Q first and then sets the upper bound to $\mu + \lambda\sigma$. Only the seeds shorter than $\mu+\lambda\sigma$ are considered for selection. The upper bound is always updated when a new seed is appended to Q. Because a selected seed is used for mutating new seeds, the upper bound of length based on μ and σ would keep increasing, and the parameter λ can be used to control the preferred seed length growth rate. Setting a smaller λ leads to a smaller length growth rate and setting a larger λ leads to a higher growth rate.

2) Selector based on Normalized Length Contribution ($norm$): The $norm$-selector selects the seed with the highest per-unit contribution than others. Compared with the λ-selector, one benefit of the $norm$-selector is that users need not configure any parameter. The obtained score of each seed in Q is normalized by dividing its corresponding seed length. The seed with the highest normalized score is then selected as the seed to perform the test in the next iteration. The rationale behind this design is straightforward. Suppose two seeds have similar coverage but one is shorter and the other is longer. Given that mutation strategies are selected randomly, and the number of mutation operations performed for each of the seeds is similar, the effect on coverage would be more prominent if the mutations were to be performed against the shorter seed.

A simple example to show how the $norm$-selector works is as follows. Assume the seeds in Figure 1 have lengths of $L = \{4, 3, 1, 1, 4\}$. In this example, the **avg** and the **max** scores of s_1 is 0.5 and 1.25, respectively. As a result, instead of selecting s_5 and s_2 as the seeds for the next iteration, differential seed scheduling selects s_3 or s_4, respectively, depending on whether the **avg** or the **max** function is used.

We further performed an experiment to compare the preferences of the two selectors. Figure 3 shows the seed length selected by the λ-selector with λ=0.125 and that selected by the $norm$-selector. The fuzzed program was `yaml`. The experimental results were collected from a fuzzing process that implemented differential seed scheduling with the **avg** scoring function and the corresponding selectors. For ease of comparison, the seed lengths selected by the $norm$-selector are plotted as negative values in the figure. The plots in Figure 3 shows that the two evaluated selectors have very different seed-selection preferences. While the $norm$-selector tends to increase the selected seed lengths monotonically, the λ-selector could realize a very diverse selection even when a relatively smaller λ is used.

IV. Evaluation

To highlight the performance improvements of seed scheduling, we evaluated the performance of differential seed scheduling by integrating it with the vanilla open-source grey-box AFL fuzzer and compared performance against the original AFL and one of its variants, AFLFast [15]. Our integrated fuzzer is called AFLExplorer. We first briefly introduce the environment setup we used to conduct the experiments in Section IV-A. We then describe the performance evaluations of differential seed scheduling and AFLExplorer by designing various experiments in the rest of the sub-sections.

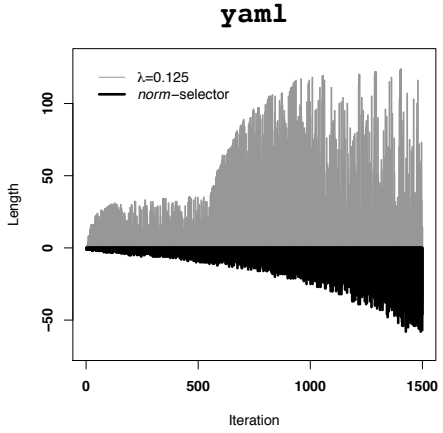

Fig. 3: Comparison of seed lengths selected by the λ-selector (λ=0.125) and the $norm$-selector.

TABLE I: List of fuzzed application programs.

Program	Version	Description
`cxxfilt`	`2.26.1`	Decode C++ and Java symbols.
`copac2xml`	`6.2`	Convert a COPAC file to a MODS XML file.
`isi2xml`	`6.2`	Convert an ISI file to an XML file.
`end2xml`	`6.2`	Convert an EndNote-XML file to a MODS XML file.
`yaml`	`0.5.3`	A human-readable data serialization language processor.
`abcm2ps`	`8.13.16`	Convert ABC to music sheet in PostScript or SVG format.
`mp3gain`	`1.5.2`	Analyze and adjust the volume of mp3 files.
`objdump`	`2.26.1`	Display information from object files.
`tcpdump`	`4.6.2`	Capture and dump network traffic.
`tcptrace`	`6.6.7`	A TCP connection analysis tool.

A. Experiment Setup

The experiments were performed on a machine with an Intel® Xeon® E5-2630 v4 CPU and 128GB of memory. The host operating system was Ubuntu Linux version 16.04. We implemented AFLExplorer by integrating differential seed scheduling with AFL version 2.33b. We used the same version of AFL[4] for performance comparisons. In addition, we compared the performance of AFLExplorer against AFLFast [15], which is available on github[5]. We used version 2.33b to ensure the comparisons are fair for all the involved experiments. The programs used for performing the evaluation experiments are summarized in Table I. Most of the selected programs have been extensively tested in the past. For example, `tcptrace` and `tcpdump` have been evaluated by Rawat et al. [30]; `cxxfilt` and `objdump` have been evaluated by Böhme et al. [15]; `mp3gain` has been evaluated by Rebert et al. [26]; and `copac2xml`, `isi2xml`, `end2xml`, and `abcm2ps` have been evaluated by Cha et al. [33]. The programs listed in the first five rows accept text-based inputs. The programs listed in the last five rows accept binary-based inputs.

B. Is Differential Seed Scheduling Effective?

We evaluated the effectiveness of differential seed scheduling by using the programs listed in Table I. We used the same set of programs to evaluate AFL and AFLFast on the same machine. Each selected program was fuzzed using all available fuzzing configurations (tools and parameters) for at least six rounds. Each round lasted for 8 hours. The numbers of identified unique crashes in each round were collected, averaged, and are presented later in this subsection. For all programs except `tcptrace` and `end2xml`, we used a single line feed (LF) control character as the input. For `tcptrace` and `end2xml`, we used a small pcap file and an xml file provided by AFL because the fuzzers failed to generate new seeds by using a single LF character as the initial seed for the two programs. We used the default settings of the AFL fuzzer. No additional parameter was passed to the `afl-fuzz` program. For AFLFast, we passed an additional "`-p fast`" parameter to the fuzzer. The additional parameter was recommended by the authors of AFLFast to achieve the best performance. For AFLExplorer, various combinations of scoring functions (**avg** and **max**) and selectors (the λ-selector with a λ ranging from 0.125 to 1.5 and the $norm$-selector) were evaluated. In total, there were 16 different configuration combinations used for AFLExplorer.

TABLE II: Percentage of considered seeds controlled by parameter λ.

λ	Estimated % of considered seeds (normal distribution)	Actual % of considered seeds (`yaml`)
0.125	54.98%	80.80%
0.25	59.87%	81.87%
0.50	69.15%	83.33%
0.75	77.34%	84.13%
1.00	84.13%	85.93%
1.25	89.44%	87.40%
1.50	93.32%	89.67%

We did not evaluate AFLExplorer with a large λ parameter because a large λ would have led to inclusion of too many seeds for consideration, thus negating the benefits of using the λ-selector. For example, assuming normal distribution of seed lengths, λ=1.5 would lead to the inclusion of 93.32% of seeds in the queue for consideration. For a rough estimate of the number of seeds included for consideration, refer to the numbers presented in Table II. In addition, we compared the estimated ratio against real-world cases. We fuzzed `yaml` for a period and counted the ratio of seeds that would be considered for selection based on the given λ value. There were total 2600 seeds in the queue after 1500 seeds were fuzzed. Even if a small λ value of 0.125 were to be used, differential seed scheduling would still consider more than 80% of the seeds in the queue.

Table III summarizes the measured average unique crashes in the effectiveness evaluations. For each fuzzed program, the top five results are highlighted with an asterisk. In most cases, AFLExplorer outperforms AFL and AFLFast. In addition, AFLExplorer can discover 1700% more unique crashes compared with AFL and AFLFast. The number was obtained from `end2xml`, where AFLFast identified 1.4 unique crashes and AFLExplorer (a/0.125) identified 26 unique crashes. If we omit the test cases in which fewer than 50 unique crashes were identified, AFLExplorer can discover 90% more unique

[4]Downloaded from http://lcamtuf.coredump.cx/afl/releases/.

[5]Downloaded from https://github.com/mboehme/aflfast.

TABLE III: Summary of effectiveness evaluations.

	AFL	AFLFast	AFLExplorer (Differential seed scheduling)**														
			m/0.125	m/0.25	m/0.50	m/0.75	m/1.00	m/1.25	m/1.50	a/0.125	a/0.25	a/0.50	a/0.75	a/1.00	a/1.25	a/1.50	a/norm	m/norm
cxxfilt	271	570*	310	359*	155	246	155	367*	201	442*	305	301	331	175	219	251	207	399*
copac2xml	27	62*	86*	78*	11	30	15	33	14	23	39	34	47	41	27	28	120*	98*
isi2xml	21	91*	24*	3	30*	20	12	13	23	15	10	5	12	2	5	4	171*	125*
end2xml	0.2†	1.4†	19*	44*	6	15	5	15	8	26*	22*	15	7	4	19*	8	6	12
yaml	130	125	171*	153	137	117	109	89	100	203*	180*	158	130	130	140	147	217*	208*
abcm2ps	413	527	706	741	876	825	842	968	1163	1471*	1439*	1374*	1263*	1174	1297*	1176	706	968
mp3gain	125	136	146*	136	134	82	133	143*	83	126	145*	134	132	135	111	142*	141*	136
objudmp	15	43	40	58*	57*	21	52*	68*	70*	25	24	33	17	14	7	11	47	51
tcpdump	70	114	223*	146*	167*	73	127	58	84	97	170*	59	112	218*	42	108	132	131
tcptrace	301*	280	290	304*	297*	291*	276	271	278	290	274	249	249	248	258	280	293*	269

* Top 5 players. ** For AFLExplorer, the configuration annotations are: m=**max** scoring function; a=**avg** scoring function; /{$number$}=λ value of the λ-selector; and /$norm$=$norm$-selector. † The numbers presented in this table are the averaged numbers from multiple repetitive benchmarks.

crashes compared with AFL and AFLFast. The number was obtained from `copac2xml`, where AFLFast identified 62 unique crashes and AFLExplorer (a/$norm$) identified 120 unique crashes. Furthermore, the following observations were made based on the numbers presented in Table III.

1) *The **max** scoring function has equivalent or slightly better performance than the **avg** scoring function.* The numbers presented in the table show that there is no remarkable performance difference between the two scoring functions. Therefore, we recommend users to start with the one they prefer.
2) *The use of a larger λ negatively impacts fuzzing effectiveness.* As mentioned before, a larger λ could negate the benefits of using the selector. For most of the cases presented in the table, setting a large λ does not help in terms of the number of identified unique crashes. The best cases are usually observed when a λ=0.125 or 0.25. If a user plans to work with the λ-selector, we would recommend starting with a smaller λ.
3) *The performance of the $norm$-selector is comparable with the best cases of using the λ-selector.* The numbers presented in the last two configurations in the table show that working with the $norm$-selector helps achieve competitive performance. The two configurations have similar performance to $m/0.125$ and $a/0.125$, and outperform AFL and AFLFast in most cases. If there are resource constraints in terms of performing fuzz testing, we would recommend generic users to start with either the $a/norm$ or the $m/norm$ configuration.

In conclusion, the performance improvement brought about by differential seed scheduling is because AFL and AFLFast do not take cautious while selecting seeds. Hence, there remain chances that they keep fuzzing seeds with low diversity, that is, selecting seeds in the pool with only minor coverage changes. In the case of AFLFast, it has been proven that changing the *energy* would improve the performance by reducing redundant tests on invaluable seeds. By contrast, AFLExplorer can traverse more paths earlier. This is advantageous because the generated seeds are valuable and fuzzers can devote more resources to fuzz valuable seeds.

C. Does AFLExplorer Find Any Real-World Issues?

Commonly, many crashes are reported by a fuzzer, but it is not clear if the identified crashes have real impacts on the community. In this subsection, we perform crash triage analysis against the crashes reported by AFLExplorer and determine whether there is any new finding reported by AFLExplorer in a shorter time. All the results presented in this section were found by AFLExplorer within one week. The crash triage analysis was performed for all selected programs in this study. In addition, we performed additional fuzz testings for several popular open-source programs including `exiv2` and tools from `ncurses` library. We collected thousands crashes reported by both AFLExplorer (**avg**/$norm$) and AFLExplorer (**max**/$norm$) fuzzers. For all reported crashes, we leveraged the open-source `exploitable gdb` plugin to perform the initial categorization. `exploitable` uses several heuristics to assess exploitability based on crash location, memory operation, and signals triggered by the input. It then categorizes each crash into classes including *exploitable*, *probably exploitable*, or *unknown*. Although the analysis result may be not accurate, it is fast, simple, and informative.

TABLE IV: Summary of new issues identified by AFLExplorer.

Program	Issue	Reported	Fixed	CVE ID
ncurses	Null pointer deref	yes	yes	CVE-2018-10754
exiv2	Integer underflow	yes	pending	CVE-2018-10772
abcm2ps	Stack buffer overflow (x2)	yes	yes	CVE-2018-10753,10771
abcm2ps	Memory access violation (x3)	yes	yes	no
mp3gain	Integer overflow	yes	yes	CVE-2018-10776
mp3gain	Memory access violation	yes	yes	CVE-2018-10777
mp3gain	Buffer overflow	yes	yes	CVE-2018-10778
bibutils	Memory access violation (x3)	yes	yes	CVE-2018-10773,10774,10775
avconv	Memory access violation (x2)	yes	pending	CVE-2018-11102,11224
libming	Buffer overflow	yes	yes	CVE-2018-11226
liblouis	Use after free	yes	yes	CVE-2018-11410
libtiff	Memory access violation	yes	yes	CVE-2018-10963

Based on the initial categorization, we further analyzed crashes that were classified as *exploitable* and *probably exploitable*. For these critical crashes, we manually checked each crash to see whether it matches an entry in existing common vulnerabilities and exposures (CVE). We attempted to report an issue to the authors of the program if a crash did not belong to a known CVE entry. Although the selected programs have been well-evaluated in past research works, Table IV lists new issues identified by AFLExplorer, types of these issues, and status of these issues. All listed issues have been reported to the authors, and many of them have been fixed by the authors. At the time of writing this paper, 18 issues were reported, of which 15 of them were fixed and 15 CVE numbers were assigned in 2018. The corresponding CVE numbers can be also found in the table.

V. Conclusion

In this paper, we proposed differential seed scheduling, a novel approach to schedule the order of seeds for improving fuzzing performance. Differential seed scheduling selects a seed based on the coverage differences between seeds. The run-time code coverage of each seed is marked in a bit string and evaluated by measuring distances between all available seeds in the current pool. Differential seed scheduling further adaptively controls the length growth rate of the selected seeds to balance effectiveness and efficiency. We implemented AFLExplorer by integrating differential seed scheduling with the vanilla grey-box fuzzer AFL and compared the performance of AFLExplorer against AFL and AFLFast. The evaluatins showed that a good seed scheduling algorithm could dramatically improve the overall fuzzing performance. It also showed that AFLExplorer discovers up to 90% more unique crashes compared with AFL and AFLFast under the same constraints. Last, we identified several new issues in the fuzzed programs, reported the issues to the authors of these programs, and had them fixed.

Acknowledgement

This paper is support in part by Ministry of Science and Technology under the grant No. MOST 107-2221-E-009-028-MY3 and the "IoT System Testing and Verification Project (2/3)" of the Institute for Information Industry which is subsidized by the Ministry of Economic Affairs of Republic of China. We would also like to thank the anonymous reviewers for their insightful and constructive feedbacks.

References

[1] Y. Younan,, "25 years of vulnerabilities: 1988-2012," Sourcefire Vulnerability Research Team, Tech. Rep., 2013.

[2] National Vulnerability Database, "NVD Data Feeds," Computer Security Resource Center, 2017.

[3] Flexera Software, "Vulnerability Review," 2017, https://resources.flexerasoftware.com/web/pdf/Research-SVM-Vulnerability-Review-2017.pdf.

[4] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[6] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*. USENIX Association, 2000.

[7] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *arXiv:1610.00502*, vol. abs/1610.00502, 2016.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2011, pp. 265–278.

[9] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[10] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operating Systems Design and Implementation OSDI*. USENIX Association, 2008, pp. 209–224.

[11] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of the 18th International Conference on Static Analysis*. Springer-Verlag, 2011, pp. 95–111.

[12] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 504–515.

[13] M. B. ohme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions Software Engineering*, vol. 42, no. 4, pp. 345–360, 2016.

[14] M. Zalewski, "American fuzzy lop," 2014, http://lcamtuf.coredump.cx/afl/.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based grey-box fuzzing as Markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. ACM, 2016, pp. 1032–1043.

[16] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope a checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 497–512.

[17] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium*. USENIX, 2012, pp. 445–458.

[18] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, 2012.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. USENIX Association, 2013, pp. 49–64.

[20] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, Dec. 1970.

[21] E. Sirer and B. Bershad, "Using production grammars in software testing," in *Domain-Specific Language*, 1999.

[22] W. M. McKeeman, "Differential testing for software," *Digital Technical*, vol. 10, no. 1, Dec. 1998.

[23] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 134–143.

[24] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008, pp. 206–215.

[25] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. ACM, 2014, pp. 725–730.

[26] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, 2014, pp. 861–875.

[27] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 579–594.

[28] C. Miller, "Fuzz by number." in *CanSecWest Applied Security Conference*, 2008.

[29] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *ACM Conference on Computer and Communications Security*. ACM, 2013, pp. 511–522.

[30] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Network Distributed Security Symposium NDSS*, 2017.

[31] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 711–725.

[32] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Network Distributed Security Symposium NDSS*. The Internet Society, 2016.

[33] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 725–741.

[34] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 782–792.