Taming a Fuzzer Using Delta Debugging Trails

Yuanli Pei, Arpit Christi, Xiaoli Fern, Alex Groce and Weng-Keen Wong
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97330, USA
Email: {peiy, christia, xfern, alex, wong}@eecs.oregonstate.edu

***Abstract*—*Fuzzers*, or random testing tools, are powerful tools for finding bugs. A major problem with using fuzzers is that they often trigger many bugs that are already known. The *fuzzer taming* problem addresses this issue by ordering bug-triggering random test cases generated by a fuzzer such that test cases exposing diverse bugs are found early in the ranking. Previous work on fuzzer taming first reduces each test case into a minimal failure-inducing test case using delta debugging, then finds the ordering by applying the *Furthest Point First* algorithm over the reduced test cases. During the delta debugging process, a sequence of failing test cases is generated (the "delta debugging trail"). We hypothesize that these additional failing test cases also contain relevant information about the bug and could be useful for fuzzer taming. In this paper, we propose to use these additional failing test cases generated during delta debugging to help tame fuzzers. Our experiments show that this allows for more diverse bugs to be found early in the furthest point first ranking.**

***Keywords*-software testing; automated testing; fuzzing; fuzzer taming; test-case reduction;**

I. Introduction

Random testing, or *fuzzing*, is an important tool for finding bugs in software systems, including compilers. Fuzzing is a process that automatically generates random input data, with the hope of exposing software vulnerabilities. It has been shown to be impressively effective at finding software bugs. For example, Csmith [1], a fuzzing tool for C compilers, has identified more than 450 previously unknown bugs. The fuzzing tool `jsfunfuzz` [2] identified more than 1700 previous unknown bugs in SpiderMonkey, the JavaScript engine used in Firefox [3]. Fuzzing was also used to test flight software for the Curiosity Mars Rover mission [4].

Although fuzzers are powerful bug-finding tools, their use suffers from several drawbacks. The first problem is that random test cases are more effective at finding bugs when they are large [5], but such large test cases are generally difficult to debug. Therefore, how to transform such test cases into *simple* and *meaningful* test cases becomes an important problem. This problem is usually solved using test case reduction techniques, such as *delta debugging* [6], an automated greedy approach for finding small failure-inducing test cases. Delta debuggers iteratively reduce large test cases to smaller ones that trigger the same failure (e.g., simplifying test cases in a way as binary search). They stop reducing test cases when a 1-minimal test case is found: 1 minimality means that removing any part of the remaining test case causes the failure to disappear. Essentially, each delta debugging iteration generates a smaller test case that retains the relevant parts of the original test required to induce the failure.

Another problem with using fuzzers is that they often generate many test cases triggering the same bug. This not only makes it hard to find rarely occurring bugs in a large set of failing tests, with many exposing the same fault, but it can also lead to re-discovering known and uninteresting bugs. Although it would be desirable to fix all bugs, some lower-priority bugs often remain unfixed for months or years in reality due to limited resources. Thus, if a fuzzer keeps on triggering such known bugs instead of potentially critical unknown bugs, its practical usefulness will be in doubt.

To make fuzzers more useful to developers, Chen et al. [7] proposed an approach to *taming fuzzer* that ranks test cases such that tests triggering more distinct bugs are ranked with higher priority. As such, when developers process the ranked list of test cases one by one, they will encounter fewer test cases corresponding to already known bugs.

To our knowledge, [7] is the only work that tries to solve the fuzzer taming problem. In [7], each test case is first reduced to a minimal failing test case using delta debugging. The ranking is then obtained by applying the *Furthest Point First* (FPF) algorithm [8] on the minimal test cases. While this method has shown success in finding diverse bugs, we hypothesize a more effective fuzzer taming process of using delta debugging. During the reducing process, delta debugging generates a set of failing test cases (the delta debugging trail), all of which trigger the same bug and contain relevant information for the bug. However, in previous work [7], only the minimal test cases are used and all the other non-minimal test cases are simply ignored.

In this paper, we aim to make use of the additional failure-inducing test cases generated by delta debugging to help fuzzer taming, rather than just using the minimal test cases. Specifically, each of the original (large) test cases is represented as a set of failing test cases reduced during delta debugging. Distance between any pair of the test cases is redefined over such *sets* of reduced test cases. We then apply FPF on these distances to find a ranking. We experimented on a data set of testing JavaScript engine containing 28 known bugs, and showed that it is beneficial to consider such additional failing test cases in the delta debugging trail.

II. Problem Statement

When a developer has a collection of test cases, each triggering some bug, he/she needs to examine these test cases to identify a set of distinct bugs to fix. If top-ranked test cases trigger more diverse bugs, the developer will find more bugs after examining the same number of test cases. With this in mind, our goal is to maximize the number of different bugs represented by test cases early in a ranking.

Formally, let $\mathcal{X} = \{X_1, \ldots, X_N\}$ be the pool of failing random test cases generated by a fuzzer. These tests can trigger up to C different types of bugs, where C is usually much smaller than N. Let $y_i \in \{1, \ldots, C\}$ be the bug triggered by test case X_i. All N test cases are thus classified into C classes based on the bug each triggers.

In our problem, we represent each X_i as *multiple test cases* produced during delta debugging. Namely, each $X_i = (x_{i1}, \ldots, x_{in_i})$ consists of a *set* of small test cases, where x_{i1} is the minimal reduced test case, x_{i2} is the next smallest test case, etc., and n_i is the total number of reduced test cases kept for each test case. All of the x_{ij}'s trigger bug y_i. Each x_{ij} is represented using a feature vector such as a histogram of tokens (meaningful character strings) from the test case or functions that are called while executing the test case.

Given such a pool of test cases, our goal is to find *a ranking π of the test cases in $\mathcal{X}$ with the largest diversity of classes among the top ranked instances*. Ideally, the top C test cases would trigger all C types of bugs, with each test case triggering one category. Then, by only examining the top C test cases, the developer would be able to examine all discovered bugs. In practice, usually the budget for examining test cases and fixing bugs is limited and diversity of only the top-k test cases is of the interest, where k is often small.

III. Methodology

Our approach involves two steps. The first step is to reduce each large failing test case into a set of small test cases using delta debugging. Then we apply FPF [8] on the sets of reduced test cases to find an ordering.

A. Reducing Large Test Cases

Given a large test case, a delta debugger reduces it to a minimal test case x_{i1}, and generates a trial of test cases $(x_{i1}, x_{i2}, \ldots)$ during the simplification. The minimal test case x_{i1} is often viewed as the most informative one about the failure [6]. Thus, we always keep x_{i1} in X_i.

For the other failing test cases, we choose whether to keep them in X_i based on two heuristics: 1) if x_{ij} is very close to the minimal test case x_{i1}, it should also be informative and should be kept; 2) if x_{ij} is too much larger than x_{i1}, then it possibly contains more noise than failure-relevant information, and should be discarded. Based on the heuristics, we propose the following policies in collecting the additional test cases:

(a) *Number of additional test cases:* For each test case, we collect at most M more test cases in addition to x_{i1}.
(b) *Closeness to minimal test case:* The test cases should be collected based on their closeness (in length/size) to the minimal test case x_{i1}.
(c) *Giving up early:* If the length of a test case x_{ij} is more than δ times of the length of x_{i1}, it should be discarded even if we haven't collected M test cases.

Policy (a) is to restrict the number of reduced test cases for the purpose of computational complexity. Policy (b) and (c) are based on the aforementioned heuristic 1 and 2 respectively.

B. Justification for Furthest Point First Approach

In our problem, generally we do not have any prior information about bug classes, such as class distribution. Usually, until a developer has examined failing tests, the set of bug classes $\{1, \ldots, C\}$ itself is unknown. Thus, our problem is purely unsupervised. To proceed, we make the following assumption:

Assumption: *The larger the distance between two test cases X_i and X_j is, the larger the probability that they belong to two different classes is, i.e., if $d(X_i, X_j) > d(X_i, X_\ell)$, where d is some distance metric, then $p(y_i \neq y_j) > p(y_i \neq y_\ell)$.*

The FPF algorithm works as follows. Let $\mathcal{Q}^k = \{X_{\pi(1)}, \ldots, X_{\pi(k)}\}$ be the set of top-k ranked test cases, where $X_{\pi(i)}$ is the i-th ranked one. Then FPF first selects the pair of test cases that is separated by the largest distance and puts them into $\mathcal{Q}$ in random order. Then the $k+1$-th ranked test case is chosen by

$$X_{\pi(k+1)} = \operatorname*{argmax}_{X_i \in \mathcal{X} \setminus \mathcal{Q}^k} \left\{ \min_{X_j \in \mathcal{Q}^k} d(X_i, X_j) \right\}. \qquad (1)$$

The objective (1) says that FPF greedily chooses the next test case as the X_i that maximizes the distance between it and all other members of $\mathcal{Q}$. Thus, after the next test case is added to $\mathcal{Q}$, the objective guarantees that the minimum distance between all pairs of test cases in $\mathcal{Q}$ is maximized. Under our assumption, the smaller the distance between a pair of test cases, the more likely the test cases are to belong to the same class. By maximizing the pairwise distance between all pairs in $\mathcal{Q}$, we hope to increase the probability that the members of $\mathcal{Q}$ all belong to different classes.

Recall that in our fuzzer taming problem, we would like to maximize the number of found bugs for every top-k $(1 \leq k \leq N)$ ranked list. Since we do not have information on bug labels beforehand, we could instead maximize the minimum pairwise distance for every top-k $(2 \leq k \leq N)$ ranked list ($k = 1$ is a trivial case). This is reasonable by the same argument as before. Namely, maximizing the minimum pairwise distance would correspondingly increase the probability of bug diversity. In that sense, the greedy FPF approach is an optimal solution for our ranking problem.

C. *Furthest Point First with Delta Debugging Trails*

In previous work [7], the distance between X_i and X_j is computed using the Euclidean distance between x_{i1} and x_{j1}, the corresponding minimal reduced test cases, i.e., $d(X_i, X_j) = d(x_{i1}, x_{j1})$ [1]. All other test cases generated during delta debugging are not used.

In this work, we make use of more test cases generated during the reduction process. Let $X_i = (x_{i1}, \ldots, x_{in_i})$ be a set of failing (small) test cases simplified from the same test case. We can view each X_i as a small cluster with the same class label. We propose to calculate distances from such sets of test cases (instead of a single minimal test) to obtain a more stable distance.

While there are many ways to compute distance between two sets X_i and X_j, we use the following two:

- *Single-linkage*, where the distances $d(X_i, X_j)$ is the distance of the nearest pair in X_i and X_j:

$$d(X_i, X_j) = \min_{1 \le u \le n_i, 1 \le v \le n_j} d(x_{iu}, x_{jv}).$$

- *Average-linkage*, where the distances $d(X_i, X_j)$ is the average pairwise distances between X_i and X_j:

$$d(X_i, X_j) = \frac{1}{n_i \cdot n_j} \sum_{1 \le u \le n_i, 1 \le v \le n_j} d(x_{iu}, x_{jv}).$$

We do not recommend using the *complete-linkage* distance, where $d(X_i, X_j)$ is the maximum pairwise distance between X_i and X_j. Such a distance tends to enlarge all the distances between X_i and X_j compared with single-linkage and average-linkage, and is sensitive to outliers. Maximizing the minimum pairwise distance using complete-linkage does not necessarily enlarge the overall distance between the two sets X_i and X_j. Thus, it is not potentially as useful as single-linkage or average-linkage for finding diverse bugs. With the definition of such distance calculations, we then apply FPF on the sets of reduced test cases.

IV. Experiments

A. *Experimental Setup*

1) Data Set: Our experiments are based on one of the data sets in previous published work [7]: using a fuzzing tool `jsfunfuzz` [2] and swarm testing [9] to generate test cases for SpiderMonkey 1.6, a version of Mozilla's JavaScript engine that contains an interpreter and several JIT compilers. We reduced the test cases using delta debugging and kept the test cases using the policies presented in Section III-A. The parameters are set as $M = 10$ and $\delta = 4$.

We extracted two type of features regarding lexical analysis and function coverage. Each lexical feature vector consists of an 885-dimensional histogram of tokens (meaningful character strings) for each test case. The function coverage sets consists of 1,469 dimensions, with the i-th feature representing the number of times the JavaScript engine called the i-th function while executing the test case. All of the features are normalized using *tf-idf* [10].

2) Baselines: We compare with two baselines strategies: *Random*, where the test cases are examined in random ordering, and *Reduced* [7], where FPF is applied on distances computed only using the minimal test cases. Past work [7] has shown that FPF is more effective than clustering techniques such as [11]. Thus, we do not compare against baseline strategies based on clustering techniques.

B. *Results and Discussions*

Figure 1 shows the results of all methods measured with bug discovery curves. A bug discovery curve [12] shows how quickly a ranking allows a developer examining the tests one by one to see at least one instance from each bug class. A curve that climbs rapidly is better than a curve that climbs more slowly.

From the results, we can see that FPF is generally better than the random strategy. The advantage is more significant using lexical features extracted from test cases. This demonstrates the advantage of using FPF in general.

For the lexical features (Figure 1(a) and 1(b)), the single-linkage approach performs similarly to just using the minimal test cases. One possible reason is that the nearest pairs are mostly the minimal test cases among the two sets. Thus, using single-linkage does not really differ much from using only the minimal test cases.

However, for function coverage features (Figure 1(c) and 1(d)), single-linkage with delta debugging trail data works much better than just using the minimal test cases at the beginning of the curve, demonstrating the benefits obtained from additional failing test cases.

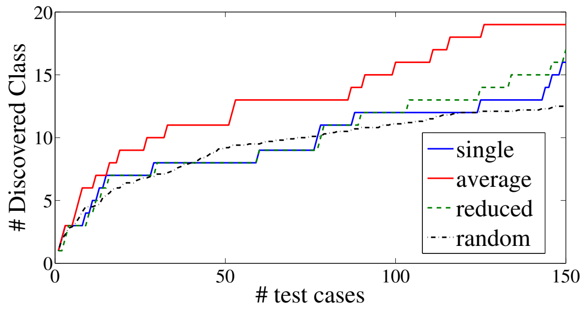
For both features, using average-linkage distance allows more diverse bugs to be found among the top-ranked test cases (Figure1(a) and 1(c)). This again shows the advantage of using additional delta debugging data to aid for finding a better ranking.

One drawback of our method is that, it does not perform as good as the base line *Reduced* in the later stage of the discovery curve. However, in practice the top-k ranked test cases are usually the most important ones. For the common case of small k, our method is preferable.

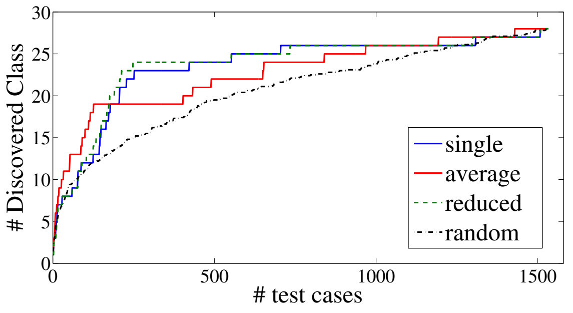
V. Conclusions

In this paper, we considered the problem of taming a fuzzer, where the goal is to rank a pool of bug-triggering

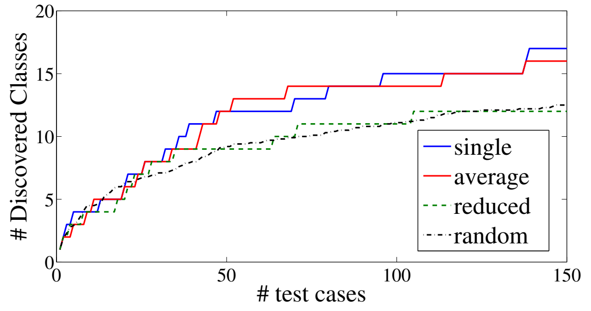
[1]In some cases, previous work also used Levenshtein edit distance, but this does not scale to large tests and is unlikely to work in some settings; our approach is more general.



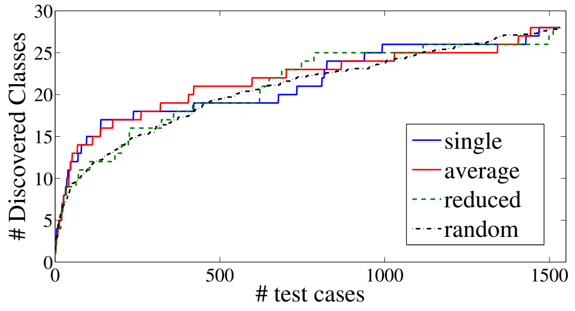
(a) Test case features, first 150 tests



(b) Test case features, all tests



(c) Function coverage features, first 150 tests



(d) Function coverage features, all tests

Figure 1. Number of discovered classes as a function of the number of examined test cases. *Single* and *Average* use sets of reduced test cases to compute distances; *reduced* use the reduced minimal test case. For *Single*, *Average*, and *reduced* furthest point first is applied to find the ranking.

test cases such that diverse bugs are found early in the ranking. We propose a method to better use delta debugging, a test case reduction technique, to improve fuzzer taming. In our method, each test case is first transformed into a set of test cases that trigger the same bug, gathered from delta debugging's trail of progressively reduced test cases. The distance between the original test cases is redefined based on these *sets* of test cases, using single-linkage or average-linkage distances. We then apply the FPF approach to rank test cases. Our experiment on Mozilla's JavaScript engine shows that using distances between sets of test cases generally allow us to find more bugs in the beginning of the ranking, implying a potential usefulness of the non-minimal failing test cases during the delta debugging.

References

[1] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011, pp. 283–294.

[2] J. Ruderman, "Introducing jsfunfuzz," Website, http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[3] ——, "Mozilla bug 349611," Website, http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[4] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu, "Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning," *Annals of Mathematics and Artificial Intelligence*, vol. 70, no. 3, pp. 315–348, 2014.

[5] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, "Random test run length and effectiveness," in *Automated Software Engineering, IEEE/ACM International Conference on*, 2008, pp. 19–28.

[6] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 183–200, 2002.

[7] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *PLDI*, 2013, pp. 197–208.

[8] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theoretical Computer Science*, vol. 38, pp. 293–306, 1985.

[9] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *ISSTA*, 2012, pp. 78–88.

[10] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[11] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *ICSE*, 2003, pp. 465–475.

[12] D. Pelleg and A. W. Moore, "Active learning for anomaly and rare-category detection," in *NIPS*, 2004, pp. 1073–1080.