Refining Fuzzed Crashing Inputs for Better Fault Diagnosis

Kieun Kim
kieun@cbnu.ac.kr
Chungbuk National University (CBNU)
Cheongju, Republic of Korea

Seongmin Lee
seongmin.lee@mpi-sp.org
Max Planck Institute for Security and Privacy (MPI-SP)
Bochum, Germany

Shin Hong
hongshin@cbnu.ac.kr
Chungbuk National University (CBNU)
Cheongju, Republic of Korea

ABSTRACT

We present DiffMin, a technique that refines a fuzzed crashing input to gain greater similarities to given passing inputs to help developers analyze the crashing input to identify the failure-inducing condition and locate buggy code for debugging. DiffMin iteratively applies edit actions to transform a fuzzed input while preserving the crash behavior. Our pilot study with the Magma benchmark demonstrates that DiffMin effectively minimizes the differences between crashing and passing inputs while enhancing the accuracy of spectrum-based fault localization, highlighting its potential as a valuable pre-debugging step after greybox fuzzing.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

greybox fuzzing, test input generation, debugging, fault localization

ACM Reference Format:
Kieun Kim, Seongmin Lee, and Shin Hong. 2025. Refining Fuzzed Crashing Inputs for Better Fault Diagnosis. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3696630.3731436

1 INTRODUCTION

Greybox fuzzers [2, 6, 7] generate new test inputs by repeatedly applying random mutations to existing test inputs, while guiding the random mutation process to maximize code coverage. When a fuzzer generates a crashing input, the project maintainers diagnose the failure for debugging. Bisection aids fault diagnosis by identifying bug-inducing commits if the fuzzing infrastructure supports conducing fuzzing across program versions [1]. Otherwise, the maintainers need to inspect the input data to determine which aspects of the input trigger the failure. In addition, they analyze the code coverage or execution traces produced by the input to locate code elements that contribute significantly to the crash [5, 8].

A key challenge in fault diagnosis is that fuzzed crashing inputs are often difficult for human maintainers to analyze, since these fuzzed inputs typically differ significantly from valid program inputs due to the accumulation of random mutations. Moreover, since greybox fuzzers generate these inputs while aiming to maximize code coverage, fuzzed crashing inputs typically explore diverse program features, many of which are unrelated to the crashes.

To aid fault diagnosis after fuzzing, we present DiffMin, a technique that refines a fuzzed crashing input into a crashing input with greater similarity to a given passing input. Given pair of crashing and passing inputs, DiffMin first identifies edit actions to convert the crashing input to the passing input. Subsequently, DiffMin iteratively applies the edit actions until the crash disappears. As a result, DiffMin derives a refined crashing input that shares more aspects with the passing input, than the original fuzzed input. We suspect that refined crashing inputs would be better to diagnose.

We have conducted a pilot study to demonstrate the efficacy of DiffMin with the Magma benchmark [3]. We found that DiffMin effectively transforms fuzzed crashing inputs into alternative crashing inputs having greater similarities with given passing inputs. We also found that, for 3 out of 4 programs, the accuracies of spectrum-based fault localization are improved when refined inputs are used.

2 DIFFMIN

DiffMin, as shown in Algorithm 1, takes as input a program under test (P), and both a crashing input (c) and a passing input (p). The given passing input (p) serves as a reference to which the crashing input (c) must be transformed to achieve similarity. The key idea behind DiffMin is to define a series of edits by computing the differences between the two given inputs and iteratively applying each edit while preserving the same crashing behavior. Note that DiffMin is different from delta debugging as DiffMin focuses on minimizing the difference between a crashing input and a passing input rather than merely reducing the size of the crashing input.

Algorithm 1: DiffMin

Input: P, program under test; c, crashing input; p, passing input
Output: c_{min}, a refined crashing input

$c_{min} \leftarrow c$;
do
  $Edits \leftarrow GetEdits(p, c_{min})$;
  $c' \leftarrow \bot$;
  foreach $e \in Edits$ **do**
    $c_e \leftarrow EditApply(c, e)$;
    if $P(c) = P(c_e)$ /* *crashing preserved* */ **then**
      if $c' = \bot \lor EditDist(p, c_e) < EditDist(p, c')$ **then**
        $c' \leftarrow c_e$;
      end
    end
  end
  if $c' \neq \bot$ **then**
    $c_{min} \leftarrow c'$;
  end
while $c' \neq \bot$;
return c_{min}

FSE Companion '25, June 23–28, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/25/06
https://doi.org/10.1145/3696630.3731436

Table 1: Target Buggy Programs from Magma [3]

Bug ID	Num. Initial Seeds	Avg. Initial Seed Size	Time to Crash	Crashing Input Size
PNG006	4	312 bytes	4 min	384 bytes
PNG007	4	312 bytes	230 min	281 bytes
XML003	1196	505 bytes	36 min	33024 bytes
XML009	1196	505 bytes	104 min	233 bytes

Starting with the given crashing input (Line 1), DiffMin iteratively refines the crashing input by applying one edit that minimizes the lexical difference and reproduces the same crash (Lines 2-16). At each iteration, DiffMin represents both inputs as byte strings and uses the Hirshberg's algorithm [4] to find an optimal sequence alignment between the two byte strings which minimizes their Levenshtein distance. Given sequence alignment, DiffMin derives a set of possible edits ($Edits$) by defining each substring insertion, deletion or replacement as an edit operation (Line 3). Once possible edits are defined, DiffMin iterates over these edits (Lines 5-12) to apply each edit (Line 6) and finds one that reduces the Levenshtein distance (Line 8) most while reproducing the same crash (Line 7). If such edit exists (c'), DiffMin updates the latest minimized crashing input (Lines 13-14) and takes another iteration (Line 16). If there exists no single edit preserving the same crash behavior, DiffMin returns the latest minimized crashing input as output (Line 17).

3 PILOT STUDY

We conducted a pilot study to evaluate whether DiffMin effectively refines fuzzed crashing inputs and assists in fault diagnosis tasks. Specifically, we designed this study to answer two questions: (1) To what extent does DiffMin reduce the lexical distance between a fuzzed crashing input and a passing input (RQ1)? and (2) Does SBFL results improve when refined crashing inputs are used instead of fuzzed crashing inputs (RQ2)? We selected four buggy programs from Magma [3] as target programs (Table 1). These four programs were randomly sampled from the 138 buggy programs available in Magma. Since each Magma target program contains multiple bugs, we configured each program to include only the target bug by disabling the other bugs and enabling the bug-specific test oracle (i.e., canary). We obtained one crashing input by running AFL++, which took between 4 and 230 minutes.

To answer RQ 1, we applied DiffMin for each initial seeds (i.e., passing inputs) and compared the Levenshtein distances between the initial seeds and the fuzzed crashing inputs (labeled as $Dist(p,c)$), as well as the distances between the initial seeds and the DiffMin results (labeled as $Dist(p,c_{min})$). Table 2 shows the minimal, average, and maximal Levenshtein distances (in bytes) with all initial seeds. These results clearly show that DiffMin significantly reduces the lexical distances in most cases. For the four programs, the ratios of the average distances with the DiffMin results to the average distances with the fuzzed crashing inputs are 14%, 90%, 45% and 40%, respectively. For example of XML003, DiffMin transforms the fuzzed crashing input having 33024 bytes into a 174-bytes crashing input subject to a 90-bytes passing input.

To answer RQ 2, we conducted SBFL with three set-ups: (1) use all initial seeds (passing tests) and the fuzzed crashing input (labeled as *fuzz* in Table 3), (2) use all initial seeds and the ddmin [9] result of the fuzzed crashing input (*ddmin*), (c) use all initial seeds and all crashing inputs refined by DiffMin with the initial seeds (DiffMin). Table 3 shows the best ranks of a buggy line (statement-level) and a buggy function with Op2. The result shows that DiffMin improves statement-level rankings for three programs, and function-level rankings for two programs, suggesting that DiffMin can substantially contributes to improving SBFL accuracy.

Table 2: RQ 1. Lexical Distance Reduction

	$Dist(p,c)$			$Dist(p,c_{min})$		
	Min	Avg	Max	Min	Avg	Max
PNG006	341	348	356	5	52	123
PNG007	151	240	320	137	217	285
XML003	72	32681	45533	72	14777	33023
XML009	7	1663	140462	4	667	106468

Table 3: RQ 2. SBFL Result (Op2)

	Statement-level			Function-level		
	fuzz	*ddmin*	DiffMin	*fuzz*	*ddmin*	DiffMin
PNG006	70	70	**50**	16	16	**14**
PNG007	70	70	**58**	**12**	**12**	**12**
XML003	116	89	**29**	14	12	**4**
XML009	95	**94**	**94**	**12**	**12**	**12**

The pilot study demonstrates that DiffMin effectively minimizes the differences between crashing and passing inputs while enhancing the accuracy of spectrum-based fault localization, highlighting its potential as a valuable pre-debugging step after greybox fuzzing. In future work, we will explore the following three directions: (1) The crashing inputs that DiffMin can discover are limited by the possible edit operations defined through lexical comparison. We will explore alternative refinement algorithms that define and apply edits using different strategies. (2) We used only lexical distance as the complexity measure. We will explore different metrics considering different fault diagnosis methods. (3) We will conduct comprehensive empirical evaluations on how crashing input refinements influence the accuracies of spectrum-based fault localization. We will explore different strategies for selecting fuzzing inputs and generating crashing inputs to better understand the effectiveness of the proposed technique.

Acknowledgement. This work is partially supported by the NRF of Korea grants funded by the Korea government (2021R1A5A1021944, RS-2024-00465145), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence strategy-EXC 2092 CASA-390781972.

REFERENCES

[1] Rui Abreu, Franjo Ivančić, Filip Nikšić, Hadi Ravanbakhsh, and Ramesh Viswanathan. 2021. Reducing time-to-fix for fuzzer bugs. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[2] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *USENIX workshop on offensive technologies*.

[3] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[4] Daniel S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM* 18, 6 (1975), 341–343.

[5] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[6] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[7] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[8] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[9] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* 28, 2 (2002), 183–200.