FuzzerAid: Grouping Fuzzed Crashes Based On Fault Signatures

Anonymous Author(s)

ABSTRACT

Fuzzing has been an important approach for finding bugs and vulnerabilities in programs. Many fuzzers deployed in industry run daily and can generate an overwhelming number of crashes. Diagnosing such crashes can be very challenging and time consuming. Existing fuzzers typically employ heuristics such as code coverage or call stack hashes to weed out duplicate reporting of bugs. While these heuristics are cheap, they are often imprecise and end up still reporting many "unique" crashes corresponding to the same bug. In this paper, we present *FuzzerAid* that uses *fault signatures* to group crashes reported by the fuzzers. Fault signature is a small executable program and consists of a selection of necessary statements from the original program that can reproduce a bug. In our approach, we first generate a fault signature using a given crash. We then execute the fault signature with other crash inducing inputs. If the failure is reproduced, we classify the crashes into the group labeled with the fault signature; if not, we generate a new fault signature. After all the crash inducing inputs are classified, we further merge the fault signatures of the same root cause into a group. We implemented our approach in a tool called *FuzzerAid* and evaluated it on 3020 crashes generated from 15 real-world bugs and 4 large open source projects. Our evaluation shows that we are able to correctly group 99.1% of the crashes and reported only 17 (+2) "unique" bugs, outperforming the state-of-the-art fuzzers.

ACM Reference Format:
Anonymous Author(s). 2022. *FuzzerAid*: Grouping Fuzzed Crashes Based On Fault Signatures. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the recent years, we have seen an increasing number of vulnerabilities in programs that were exploited [10, 37]. Google's *Project Zero*, a team of security researchers that study zero-day vulnerabilities, recorded their most ever detected and disclosed vulnerabilities in 2021 [37]. 67% of the actively exploited zero-day vulnerabilities in 2021, detected by *Project Zero*, were from memory related bugs. Thankfully, the modern state-of-the-art (SOTA) fuzzers are adept at finding these types of bugs [4, 20]. Thus the companies, such as Microsoft and Google, invest heavily to develop and deploy effective fuzzers for daily uses. Open source platforms such as GitHub [8] and GitLab [6] also provide friendly integration to run fuzzers. However, even with large scale fuzzing services, like Google's OSS-Fuzz [38] and Microsoft's OneFuzz [7], using fuzzers to find and fix bugs still involves considerable manual efforts [16]. The fault diagnosis may hardly catch up to the speed at which new crashes are generated. As a result, critical and exploitable bugs can be left undiagnosed in the large number of crashes reported by the fuzzers.

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper, our goal is to develop approaches and tools that can help group fuzzed crashes and provide support for diagnosing them. Grouping fuzzed crashes is also called *crash deduplication* or reporting *unique crashes* [27]. In the past, the common approach for crash deduplication is to apply heuristic metrics [13–15, 21, 25, 34, 35, 39, 40, 43], such as call stack hashing [14, 15, 35, 39], instrumented coverage information [13, 21, 43] or dynamic symptoms [25, 34, 40] to compare the similarities of the crashes. For example, AFL [43] uses instrumented branch coverage, while CERT-BFF [14] and HonggFuzz [39] use call stacks. However, the coverage based metrics tend to inflate the number of "unique" fuzzed crashes [23, 27] where typically crashes that traverse different paths will be separated, independent of whether they trigger the same bug. Meanwhile, call stack based metrics have risks of misclassifications; the crashes generated from the same bugs are separated into different groups as their call stacks are different [11] or different bugs are grouped together because they have the same call stacks [34]. The dynamic symptoms based approaches, like the ones based on symbolic analysis [34] and automatically generated patches [25, 40], are more precise, but they often have a limited scope, e.g., applicable only to a certain type of crashes.

In this paper, we propose to use *fault signatures* to group the fuzzed crashes. A fault signature is an executable program that consists of "indispensable" statements that can reproduce the bug. As opposed to call stacks, failure symptoms, and coverage based metrics, the fault signature captures the root cause of a failure. Crashes grouped based on the same fault signature thus likely share the root cause and fix, and should be diagnosed together. We say these crashes are induced by the *same bug*.

Our approach consists of three components, namely *generating fault signatures*, *classify with fault signatures*, and *merging fault signatures*. Given a collection of fuzzed crashes (here each crash is associated with an input), we first ran a crash inducing input to get its dynamic trace. We then create an executable program from the trace and perform program reduction to generate an as-small-as-possible program, namely *fault signature*, that can reproduce the bug (meaning removal of any statement from this program can result in the bug not triggering). Since the fault signature only contains a subset of statements from the complete crashing trace, the two crashes that exercise different paths in the original program can be grouped into the same signature as long as the two share the subset of root cause statements. To classify the next fuzzed crash, we took its crash inducing input and ran with the generated fault signatures and see if the failure has been produced with any of the fault signatures; if so, we classify the fuzzed crash into a group labeled with the fault signature. These two steps are implemented in the components of *generating fault signatures* and *classify with fault signatures* respectively.

After all the crashes have been bucketed into the groups, each of which is labeled with a fault signature, we perform post-processing, namely *merging fault signatures*, to examine if any of the two fault signatures can be further grouped. Two fault signatures may share a set of statements that are root causes, but differ in the paths that lead to the root cause from the entry of the program. These fault signatures can be grouped. We applied a heuristic based matching between two fault signatures to group very similar fault signatures. Specifically, we measure how many common statements the two signatures share and whether the call stacks are similar when crashing the two different fault signatures with their respective crash inducing inputs.

We implemented our approach in a tool called *FuzzerAid* for C programs. We used 15 real-world bugs from 4 large open source projects to evaluate it. We generated a total of 3020 fuzzed crashes and compared *FuzzerAid* with 3 SOTA fuzzers, name *AFL*, *CERT-BFF* and *HonggFuzz*, and a total of 5 settings. We used developers' patches to establish the ground truth, similar to the approaches in [27, 40]. Our results show that we correctly group 2995 (99.1%) of the 3020 fuzzed crashes and didn't misclassify any crashes into a wrong fault group. We grouped the fuzzed crashes from 15 different bugs into 17 (+2) groups while the next best baseline reported 40 groups, and other baselines reported 100+ or even 1000+ groups. Considering it is very time-consuming to diagnose a bug, our approach thus has a great potential to improve the productivity of bug diagnosis and fix associated with fuzzing tools.

In summary, we make the following contributions in the paper:

(1) we proposed to use *fault signatures* to group fuzzed crashes, and our intuition is that fault signatures capture the root causes and thus can more accurately classify the crashes;
(2) we designed an algorithm, consisting of *generating fault signatures*, *classify with fault signatures*, and *merging fault signatures*, to automatically group fuzzed crashes, and
(3) we implemented our tool *FuzzerAid* for C projects, and evaluated it with real-world bugs and large open source projects. Our results show that our approach can correctly group the crashes and significantly outperformed the SOTA widely deployed fuzzers.

2 MOTIVATION

In this section, we provide a few simple examples to explain the challenges of grouping fuzzed crashes and why existing fuzzing deduplication methods [13, 14, 21, 35, 39, 43] are not sufficient.

In Figure 1a, we show a code snippet that contains a null-pointer dereference at line 3 adapted from [27]. Here, the pointer p at line 2 is not initialized, and then the dereference at the next line can lead to crash. This bug can be triggered independent of the condition outcome at line 8, as both the paths $\langle 6, 7, 8, 1, 2, 3\rangle$ and $\langle 6, 7, 9, 1, 2, 3\rangle$ calls into the bug function. Due to the presence of two distinct paths, a coverage based heuristic, e.g., used in AFL, will classify the fuzzed crashes for this bug into two separate groups. However, in this case, the branch at line 8 is not important for triggering the bug.

In our approach, we will take a crashed input and collect its dynamic trace. We then reduce the trace to be able to reproduce

```
 1 void bug() {
 2   int *p;
 3   int value = *p; // Null Pointer Deference
 4 }
 5 int main (int argc, char* argv[]) {
 6   if (argc >= 2) {
 7     char b = argv[1][0];
 8     if ( b == 'a' ) bug();
 9     else bug();
10   }
11   return 0;
12 }
```

(a) A null pointer dereference with different paths

```
1 void bug() {
2   int *p;
3   int value = *p;
4 }
5 int main (int argc, char* argv[]) {
6   if (argc >= 2) {
7     bug();
8   }
9 }
```

(b) *FuzzerAid*: *Fault signature* for the bug in Figure 1a

Figure 1: deduplication based on code coverage can fail

the bug. We call such a program that contains only statements that contribute to the failure the *fault signature*. For Figure 1a, we generate the fault signature shown in Figure 1b. Here, lines 7–9 in the original program are replaced with a single call to the function bug. To group the fuzzed crashes, we run the crash inducing inputs with this fault signature. If the failure is reproduced with the fault signature, we group the fuzzed crash; if not, we will generate a new fault signature that can represent the crash. In this example, we can group any crashes triggered along the branch at line 8 (inputs that start with a) or along the one at line 9 (inputs that do not start with a) into the same fault signature.

In the second example shown in Figure 2a, we provide two different null pointer difference bugs. Bug1, marked at line 5, will trigger an incorrect pointer dereference at line 3 after the pointer is freed at line 5. Meanwhile, Bug2, marked at line 17, will trigger a null pointer dereference at line 3 along the path $\langle 17, 18, 8, 4, 6, 3\rangle$. The unitialized pointer at line 17 will be dereferenced.

The crashes for Bug1 can traverse different paths and lead to different call stacks at the crash site, as shown in Figure 2b. The first two lines indicate that Bug1 can be triggered by calling foo (at line 14), bug (at line 8), and trigger (at line 6); or by calling bar (at line 15), bug (at line 9), and trigger (at line 6). Since the two crashes have two different call stacks, the call stacks based approach can fail to group them and will consider them as different bugs.

Similarly, the crash for Bug2 can be triggered by calling foo (at line 18), bug (at line 8), and trigger (at line 6). As shown in Figure 2b, the call stacks for Bug1-Crash1 and Bug2-Crash1 are the same. Therefore, the call stack based deduplication methods can mistakenly group the two different bugs (they have different causes and require different fixes) into in one group.

```c
#include<stdlib.h>
#include<stdbool.h>
void trigger(char *s) { int value = *s; }
void bug(char *s, bool b1) {
  if (b1) free(s); // Bug 1
  trigger(s);
}
void foo(char *s, bool b1) { bug(s, b1); }
void bar(char *s, bool b1) { bug(s, b1); }
int main (int argc, char* argv[]) {
  if (argc >= 2) {
    char *s = malloc(100);
    char b = argv[1][0];
    if ( b == 'a' ) foo(s, true);
    else bar(s, true);
  } else {
    char *p; // Bug 2
    foo(p, false);
  }
  return 0;
}
```

(a) `null pointer dereference` with different call stacks

```
Bug1:
  Bug1-Crash1   main→foo→bug→trigger
  Bug1-Crash2   main→bar→bug→trigger
Bug2:
  Bug2-Crash1   main→foo→bug→trigger
```

(b) call stacks for the bugs in Figure 2a

```c
 1 Sig1 ===========
 2 void trigger(char *s) { int value = *s; }
 3 void bug(char *s, bool b1) {
 4   free(s); trigger(s);
 5 }
 6 void foo(char *s, bool b1) { bug(s, b1); }
 7 int main (int argc, char* argv[]) {
 8   if (argc >= 2) {
 9     char *s = malloc(100);
10     char b = argv[1][0];
11     if ( b == 'a' )  foo(s, true);  } }
12
13 Sig2 =============
14 void trigger(char *s) { int value = *s; }
15 void bug(char *s, bool b1) {
16   free(s); trigger(s);
17 }
18 void bar(char *s, bool b1) { bug(s, b1); }
19 int main (int argc, char* argv[]) {
20   if (argc >= 2) {
21     char *s = malloc(100);
22     char b = argv[1][0];
23     if ( b == 'a' ) ;
24     else bar (s, true); } }
25
26 Sig3 =============
27 void trigger(char *s) { int value = *s; }
28 void bug(char *s, bool b1) {
29   trigger(s);
30 }
31 void foo(char *s, bool b1) { bug(s, b1); }
32 int main (int argc, char* argv[]) {
33   if (argc >= 2) {
34   } else {
35     char *p;
36     foo(p, false);
37   }
38 }
```

(c) *FuzzerAid*: *Fault signatures* for the bugs in Figure 2a

Figure 2: deduplication based on call stacks can fail

Using our approach for this example, we will first generate fault signatures for the crashes, one at a time, shown in Figure 2c, `Sig1` at lines 2–11 for `Bug1-Crash1`, `Sig2` at lines 14–24 for `Bug1-Crash2`, and `Sig3` at lines 27–38 for `Bug2-Crash1`. We can see that in the fault signatures, the statements that weren't contributing to the bug's manifestation have been removed.

Each fault signature can represent one path or a set of paths that lead the crashes. Since two sets of paths may contain the same root cause, we further merge fault signatures to be the same fault group. In this example, we observe that `Sig1` and `Sig2` differ only at the branch `b=='a'` (highlighted in yellow and also see line 11 and lines 23–24), and we can merge them to be the same fault group. The merged fault signatures then can classify all fuzzed crashes that manifest Bug1 into a single group. Similarly, our approach will determine that `Sig3` has no close relation with `Sig1` and `Sig2` in terms of branches and the shared statements. We thus classify it as a separate fault group.

The above examples show that our fault signature based grouping potentially is more accurate than coverage based and call stack based approaches that are popularly used in the current fuzzers. In Sections 3 and 4, we provide the details on how we generate fault signatures, how we group the fuzzed crashes based on the fault signatures, and how we further merge fault signatures into fault groups.

3 OUR APPROACH

In this section, we first give an overview of *FuzzerAid*. Next, we provide a detailed explanation to help understand what is a *fault signature*. We then present the three main components of *FuzzerAid*.

3.1 An Overview

Figure 3 presents an overview of our workflow. *FuzzerAid* takes as input a collection of crashes from the fuzzers. Each crash is associated with an input. Each input will be run through a set of fault signatures created so far. If the failure is triggered with some fault signature, the fuzzed crash is put in a bucket labeled with the corresponding fault signature. This step is implemented in the component named *Classify with Fault Signature*.

If the fuzzed crash input did not trigger any failures matching with existing fault signatures created so far, we run the input in the original program using PIN [30] and collect its dynamic trace. We then patch the dynamic trace to generate an executable program. In the next step, we use a program reduction tool, C-Reduce, to reduce the executable program into the *fault signature*. C-Reduce ensures that the fault signature can still reproduce the failure at the same location with the same failure symptoms while statements not relevant to the failure reproduction are removed. This step is implemented in the component named *Generate Fault Signatures*.

Once all the crashes and their inputs are labeled with the fault signatures, we perform the step of *Merge Fault Signatures* and apply heuristics and similarity metrics to group fault signatures that likely originated from the same root cause. The resulting fault groups, representing grouped fuzzed crashes, and their corresponding fault signatures will be presented as the output of *FuzzerAid*.

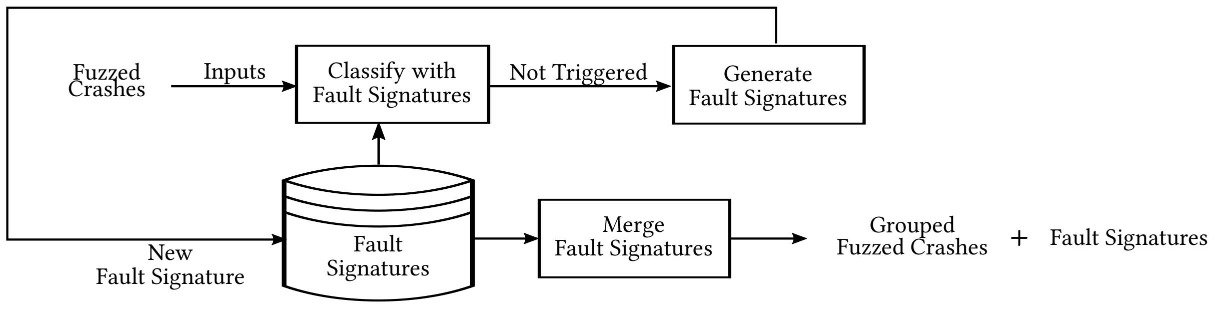

Figure 3: Overview of *FuzzerAid*

```
1  void HTMLlineproc0(...) {
2    struct readbuffer *obuf = h_env->obuf;
3    struct table_mode *tbl_mode = NULL;
4    if (obuf->table_level >= 0) {
5      int pre_mode = (obuf->table_level >= 0) ?
6      tbl_mode->pre_mode : obuf->flag; // Null Pointer
7    };
8  }
9  void loadHTMLstream(...) {
10   ...
11   while ((lineBuf2 = StrmyUFgets(f))->length) {
12     HTMLlineproc0(lineBuf2->ptr, &htmlenv1, internal);
13   }
14 }
15 Buffer *loadHTMLBuffer(...) {
16   ...
17   loadHTMLstream(f, newBuf, src, newBuf->bufferprop &
         BP_FRAME);
18 }
19 Buffer *loadGeneralFile(...) {
20   ...
21   proc = loadHTMLBuffer;
22 }
23 int main(int argc, char **argv, char **envp) {
24   ...
25   newbuf = loadGeneralFile(url, NULL, NO_REFERER, 0,
         request);
26 }
```

Figure 4: Fault Signature for a `Null Pointer Dereference` in `w3m`

3.2 Fault Signatures

Fault signature can be viewed as a minimized version of the original program consisting of only the statements necessary for triggering a particular bug. Ideally, a fault signature can reproduce the same bug for all the inputs that can trigger the bug in the original program. The statements in a fault signature include two parts: (1) those that trigger the bug and (2) those that set up the necessary conditions, e.g., parsing the input, for reaching the buggy location.

As an example, consider the `null pointer dereference` bug [1] in `w3m-0.5.3`. Even though the entire program consists of 80K lines of code, we generated a fault signature consisting of less than 100 lines. It can trigger the bug using the crash inducing inputs from the original programs. A simplified version of the fault signature is shown in Figure 4. The `null pointer dereference` occurs at line 6, while trying to access the variable `tbl_mode`. This variable was previously initialized as `NULL` at line 3 and not updated since then. Hence, lines 2–8 in Figure 4 can be considered as the statements that trigger the bug, and lines 9–26 are necessary to set up the conditions to reach the bug.

While the lines actually triggering the bug (lines 2–8 in Figure 4) mostly remain the same between different crash inducing inputs, the statements leading to it (lines 9–26) can be different. In another words, a bug can be triggered when executing different paths (e.g., in the region of lines 9-26), but these paths can share a root cause (lines 2–8). As a concrete example, consider the two execution paths (`Bug1-Crash1` and `Bug1-Crash2` from Figure 2b) for the bug shown in Figure 2a. The statements triggering this bug are in lines 3–7, while lines 8–15 provide the necessary conditions to reach the bug location.

It is difficult to enumerate all the different ways to reach a program point. Hence, producing an ideal fault signature that can reproduce the bug for all the crash inducing inputs is hard. However, since we have access to some inputs responsible for triggering the crashes, it is easy to create a fault signature based on one input with respect to its path, and then determine if other inputs can crash the same paths. Therefore, we chose to generate such fault signatures in *Generate Fault Signature* to group the fuzzed crashes.

3.3 Generate Fault Signatures

In order to generate fault signatures, we need to identify statements that are necessary for triggering a bug. Any statements that are not executed during a bug's manifestation are not necessary. So as a first step, we ran the crash inducing input with the original program to collect its dynamic trace. We used PIN, a dynamic binary instrumentation framework, to achieve this. The dynamic trace information collected using PIN is more resilient to call stack corruptions as opposed to the traditional methods [1].

The statements collected using dynamic trace typically don't include lines representing static information such as variables definitions, structure initialization and switch case headers. Or in other words, it is not possible to generate an executable fault signature directly using just the dynamic trace. Therefore, as the second step, we extracted all the functions that had a statement present in the dynamic trace. This takes care of missing local variable definitions and switch case headers. To get the global variable definitions and structure initialization, we extracted all global variables, macros,

[1]https://github.com/tats/w3m/issues/18

```
1  void HTMLlineproc0(...) {
2    Lineprop mode;
3    int cmd;
4    struct readbuffer *obuf = h_env->obuf;
5    int indent, delta;
6    struct parsed_tag *tag;
7    Str tokbuf;
8    struct table_mode *tbl_mode = NULL;
9    if (w3m_debug) {
10     HTMLlineproc1()
11   }
12   tokbuf = Strnew();
13   if (obuf->table_level >= 0) {
14     char *str, *p;
15     int is_tag = FALSE;
16     int pre_mode = (obuf->table_level >= 0) ?
17     tbl_mode->pre_mode : obuf->flag; // Null Pointer
18     // Rest of if block
19   }
20   else { // Else block }
21   // Rest of the function
22 }
23 void HTMLlineproc1 () {...}
24 str StrNew () {...}
```

Figure 5: Example of the statements removed for creating the Fault Signature in Figure 4

header file includes, and structure initialization using tools like srcML [18]. We made an executable program from these extracted information using the compilation and linker flags obtained via tools like Bear [31]. This program, even though not minimal, is a reduced version of the original program capable of triggering the original bug.

As the final step, we used C-Reduce [36] to remove statements not required for triggering the bug to generate fault signatures. C-Reduce, by default, uses a set of 135 passes to minimize the program, which also include transformations such as renaming variables and function names and merging control branches. We developed a custom configuration of C-Reduce by removing 45 passes to suit our needs. Figure 5 shows an example of the reduction (highlighted in red) when using our configuration for producing the fault signature shown in Figure 4. Only the statements involving the variables `obuf` and `tbl_mode` are necessary for reproducing the `null pointer dereference` bug at line 17. Hence, all the statements not related to the two variables till the fault's manifestation (at line 17) are removed (lines 2, 3, 5–7, 9–12, 14, 15) by C-Reduce. Since the statements after triggering the bug are also unnecessary, they also got removed (lines 18–21). We also remove the entire functions that were only used in the removed statements (lines 23 and 24), like `HTMLlineproc1` (used at line 10) and `StrNew` (used at line 12).

3.4 Classify with Fault Signatures

Although the fault signature generation starts with one crash inducing input, after trace minimization and removing unnecessary statements, the fault signature can crash a set of failure inducing inputs that exercise the same path and the paths that only differ in the removed statements. It thus can be used to group a set of crash inducing inputs.

We ran crash inducing inputs with the existing fault signatures. If the same failure occurs (meaning it triggers the same bug type at the same source code location with the same call stack as the original input used to produce the fault signature), we classify the fuzzed crash into the group labeled with this fault signature. Otherwise, we take the input that can not yet be categorized and generate another fault signature.

When running a fuzzed crash with a fault signature, we may encounter failures that do not match the original crashes, e.g., entering an infinite loop or hanging indefinitely. Therefore, we set a configurable timeout (1 minute in our evaluation) when running fault signatures to classify whether a valid failure is triggered. We also encountered some flakiness when running with fault signatures caused by the nondeterminism in the software execution. Hence, we repeated running any fuzzed crash that failed to trigger a bug for a fixed number of times (10 times in our evaluation).

3.5 Merge Fault Signatures

Our fault signature is created from the dynamic trace generated using a single crash inducing input. As discussed in Section 3.3, these fault signatures don't necessarily cover all the statements that can lead to the program state from which the bug can be triggered. Thus, during generation and classification of fault signatures, it is possible to create multiple fault signatures for the same bug, each of which represent a scenario of reaching the bug location. For example, see `Sig1` and `Sig2` for `Bug1` in Figure 2. In order to further group all the fuzzed crashes from `Bug1` into one group, we develop a technique to cluster fuzzed crashes associated with these fault signatures.

Our considerations are twofold. First, we want to group fault signatures of the same root cause (while a root cause can cover a segment/set of statements), and thus we should consider the similarity/overlap between the fault signatures. Second, we also consider the paths that lead to the crash site when merging the fault signatures. We observed that different bugs may fail at the same location, but two crashes that traverse very different paths before reaching the same location likely have different root causes. For example, in Figure 2, `Sig3` from `Bug2` shares the bug manifestation statements (line 3 in Figure 2a) with `Sig1` and `Sig2` from `Bug1`. However, the actual root causes (line 17 for `Bug2` and line 5 for `Bug1` in Figure 2a) along the paths leading to the buggy state (lines 18,4–7 for `Bug2` and lines 8–16 for `Bug1` respectively) are very different.

Specifically, to measure the similarity between two fault signatures, we used the Levenshtein's edit distance between them. See Equation 1, where Sim_{Sig} is the similarity score, $MAXSize$ returns the maximum size in lines of code of the two fault signatures, S_1 and S_2, and $LDistance$ is the Levenshtein's edit distance between the two signatures.

$$Sim_{Sig} = \frac{MAXSize(S_1, S_2) - LDistance(S_1, S_2)}{MAXSize(S_1, S_2)} \quad (1)$$

We also measured the similarity in the paths leading to the failure location using call stacks generated by running the crash inducing inputs with the fault signatures. Specifically, we used Equation 2, where Sim_{Call} is the similarity score, $COMMON$ is the number of functions that are shared between two call stacks CS_1 and CS_2, and $MAXSize$ is the maximum size in count of call stacks.

$$Sim_{Call} = \frac{COMMON(CS_1, CS_2)}{MAXSize(CS_1, CS_2)} \quad (2)$$

The final similarity score used to decide whether two fault signatures should be merged or not is shown in Equation 3, which is the average of the two similarity scores.

$$Sim_{Score} = \frac{Sim_{Sig} + Sim_{Call}}{2} \quad (3)$$

4 PUT TOGETHER: THE ALGORITHM

In Algorithm 1, we present our algorithm for grouping fuzzed crashes. The algorithm takes as input C_I, a collection of inputs that can lead to the crashes in a fuzzer, and generates the fault groups, each of which represents a bug and has the corresponding fault signature(s). The crashes can be generated from different runs of the same fuzzer or even from different fuzzers.

Algorithm 1 Grouping fuzzed crashed using fault signatures

```
1:  INPUT: $C_I$ (Fuzzed crashes)
2:  OUTPUT: $F_g$ (Fault groups), $F_s$ (Fault signatures)
3:  Initialize $F_s$                              ▷ Fault signatures
4:  Initialize $F_g$                              ▷ Fault groups
5:  for all $c \in C_I$ do
6:      if $\exists s \in F_s$ and $s \rightarrow c$ then
7:          Add $c$ to $s.Crashes$
8:      else
9:          $t \leftarrow$ Dynamic Trace ($c$)
10:         $s_{new} \leftarrow$ Generate Signature ($t$)
11:         Add $s_{new}$ to $F_s$
12:     end if
13: end for
14: $worklist \leftarrow F_s$
15: while $worklist \neq \emptyset$ do              ▷ Merge Fault Signatures
16:     Remove $s_c$ from $worklist$
17:     Initialize $G_{sc}$                        ▷ Fault group for $s_c$
18:     for all $s_{wl} \in worklist$ do
19:         $score \leftarrow$ Compute Similarity ($s_c, s_{wl}$)
20:         if $score \geq$ threshold then
21:             Add $s_{wl}$ to $G_{sc}$
22:         end if
23:     end for
24: end while
25: return $F_g$
```

Lines 3–13 implements the components of *Generate fault signatures* and *Classify fault signatures* specified in Figure 3. Lines 14–25 implements the *Merge fault signatures*. Specifically, at lines 3 and 4, we initialize fault signatures (F_s) and fault groups (F_g) respectively. The initialization can either set them as empty sets or using existing fault signatures and fault groups from previous fuzzing campaigns or a different fuzzer. Lines 5–13 loop through all the fuzzed crashes (C_I) given as the inputs to create and test with fault signatures. At line 6, the current fuzzed crash (c) is checked against all the existing fault signatures to see if there exist a fault signature (s) that can lead it to crash. In case such an existing fault signature is found, at line 7, we add the crash into the group represented by the fault signature. If we are unable to find such a fault signature, we create a new one at lines 8–12. To create a fault signature, we run the program with the crashing input to collect its dynamic trace (t) at line 9. This trace is used to create a new fault signature at line 10, by removing the statements in the trace until the failure cannot be reproduced. The newly created signature (s_{new}) is added to other fault signatures at line 11.

Once we have generated the fault signatures that can classify all the fuzzed crashes, we further merge the fault signatures into fault groups at lines 15–24. For this, we first create a work list ($worklist$) consisting of all the fault signatures at line 14. We initialize a fault group (GS_{sc}) at line 18 for s_c. This can either be empty or be an existing fault group given at line 4. We then traverse the work list at lines 18–23. For each fault signature in the work list, we compute similarity scores ($score$) at line 19 and compare it with the other fault signatures in the work list. The $score$ is computed as the average of (1) normalized Levenshtein's edit distance and (2) normalized percent of matching functions in the crashing call stack between two fault signatures. If this similarity score is above a set threshold, then we add that fault signature (s_{wl}) to the fault group (G_{sc}) at line 21.

Once the `worklist` is empty, we finish grouping all the fault signatures into fault groups. We return this group of fault groups, that represent "unique" crashes from the fuzzed crashes, as the output of our algorithm at line 25.

5 EVALUATION

Our evaluation aims to answer two research questions:

- **RQ1:** Can we correctly group crashes generated by the fuzzers?
- **RQ2:** How effective is our technique compared to the SOTA methods?

5.1 Experimental Setup

5.1.1 Implementation. We implemented *FuzzerAid* for C programs using Clang [28], srcML [18], SQLite [24], PIN [30], C-Reduce [36] and Rust [32]. Specifically, we used Clang, srcML and SQLite to create a database containing the function names and their line numbers at a file level granularity for each benchmark. Then we used this database with PIN to collect statement level dynamic traces for crash inducing inputs and generated an executable program. We used C-Reduce to minimize the executable programs into fault signatures. We used Rust to implement the similarity comparison between traces and between fault signatures. We used a cost of 1 for all the edits when computing the Levenshtein's distance. The threshold for grouping two *Fault signatures* (line 20 in Algorithm 1) was set as "0.7".

5.1.2 Subject Selection. To answer the research questions and demonstrate that our techniques are applicable in practice, we aim to use the benchmarks that (1) are real-world open-source C programs, (2) preferably contain multiple real-world bugs in each program, so that we can evaluate if our approach can separate the crashes from different bugs, (3) the bugs and their patches are known, so we can have a ground truth to compare against, (4) the bugs can be triggered by the fuzzers, so we can generate the crash corpus, and (5) can be handled by our baseline methods, so we can compare with them.

We searched for the readily available benchmarks based on the above 5 criteria in fuzzing literature [12, 13, 21, 27, 29, 34, 40]. We also went through the list of programs at AFL's website [2]. As a result, we collected all the C projects (3 out of 6 total projects) provided by [40], namely `w3m`, `sqlite` and `libmad`, and `libarchive` from AFL's website. We were not able to use the other 3 benchmarks from [40], namely `PHP`, `R` and `Conntrackd`, as they were either implemented using multiple languages (PHP, R) or the bug/patch were in a non C file. Similarly, for C only benchmarks with multiple bugs on the AFL's websites, such as *audiofile* and `libxml`, we were either not able to get the crashing input and the minimal patch, or they had the bug/patch in non C files.

Through the above process, we collected a total of 15 known bugs from 4 large real-world C projects. Specifically, we used 4 bugs from `w3m`, a text-based web browser, 8 bugs from `sqlite`, a database software, 1 bug from `libmad`, an MPEG audio decoding library, and 2 bugs from `libarchive`, a multi-format archive and compression library. Four `sqlite` bugs from [40] were not included due to the problems of reproducing them with PIN. The projects, their sizes[2], and the bugs are listed in the first three columns in Table 1.

5.1.3 Fuzzer selection. To demonstrate the effectiveness of our techniques and compare with meaningful baselines, we looked for fuzzers that (1) are open source, (2) are well documented, (3) have been widely used in research or industrial settings, (4) applied different methodologies for deduplicating fuzzed crashes (so we can compare with different approaches of deduplication used in practice), and (5) could work with our benchmarks. In case of fuzzers with similar deduplication methodologies, we picked the one that reported more bugs, cited by more references, and that are easier to work with our benchmarks. In the end, we used AFL [43] to generate crashes, and used the deduplication methods implemented in AFL, CERT-BFF [14] and Honggfuzz [39] as our baselines. Specifically, AFL uses branch (edge) coverage and coarse grained branch-taken hit counter to determine unique fuzzed crashes. Only fuzzed crashes associated with execution paths that involves new edges or doesn't visit the common edges are kept after deduplication. CERT-BFF uses hashes generated from last N calls (frames) in the call stack to determine uniqueness. Any fuzzed crashes sharing the same call stack hash is discarded during the deduplication process. HonggFuzz, on the other hand, uses the information at the crash location (fault address, last known PC instruction and last 7 frames in call stack) to deduplicate the fuzzed crashes.

[2]LOC calculated using tokei, https://github.com/XAMPPRocky/tokei

5.1.4 Experimental design for RQ1. In RQ1, our goal is to evaluate the correctness of the grouping made by *FuzzerAid*. Specifically, we aim to discover (1) if we can correctly group all the fuzzed crashes from the same bug, (2) if we would incorrectly mix crashes from different bugs and put them in one group, and (3) if we would fail to group any fuzzed crashes.

To establish the ground truth, we propose to generate the crashes for the multiple known bugs and see if we can group crashes caused by the same bug and separate the crashes generated from different bugs. The challenge we face is that given an arbitrary given seed, the fuzzers may not trigger the known bugs or trigger them within a reasonable time window. To set up this experiment, we used a special configuration of the fuzzers together with the bug patches from the developers to achieve the goal. Specifically, our approach is to run AFL in the "Crash Exploration Mode [3]". It takes a known fuzzed crash inducing input and uses the traditional feedback and genetic algorithms to quickly generate a corpus of crashes that explore different paths that can lead to similar crash state. We found that this approach is likely to generate crashes that trigger the given bug. In our experiments, we found only a total of 28 among thousands of crashes that belong to some unknown bugs.

Our setup is as follows. First, we generate crash corpus for individual know bugs using the above approach. We ran AFL for 2 hours per bug. Given 2 hours, some bug generated more than 2K fuzzed crashes, some bug only generated less than 100 crashes. To balance the crashes from different bugs, we used at most 250 crashes from each bug (randomly select 250 if there are more than 250) and mixed them as a mixed crash corpus. Since this approach is heuristic, we also used the developers' patch to further validate whether the generated crashes are indeed from a known bug and which known bug it belongs to. Using developers' patch to determine ground truth [27, 40], is based on the assumption that if an input I crashes a program P, but no longer crashes it after applying patch P, we can associate I with the bug for P [16]. Further, if two inputs I_1 and I_2 both crash P, but disappear with patch P, then both I_1 and I_2 are caused by the same bug (given that the patches are "minimal" [5]). As a result of validation, each crash is labeled with a bug ID and the ones do not match any existing bugs are labeled as unknown. We then apply *FuzzerAid* to group these crashes that we know the ground truth.

To evaluate the correctness, we used as metrics of the NO. of fuzzed crashes that were (1) correctly grouped with fault groups for a bug, (2) incorrectly grouped with fault groups from a different bug, (3) weren't grouped to any fault group. We also recorded the number of fault signatures and fault groups created for each bug to measure the usefulness of the grouping.

5.1.5 Experimental design for RQ2. In RQ2, we compared *FuzzerAid* with the three SOTA real-world fuzzers regarding their capabilities of deduplicating crashes. In the following, we present the setups of the fuzzers used in comparison:

For AFL, we ran *afl-cmin* on all the fuzzed crashes generated for each benchmark. It finds the smallest subset of fuzzed crashes that still exercises the full range of instrumented data points as the original fuzzed crash corpus. The remaining fuzzed crashes are reported as the deduplicated fuzzed crashes for *AFL*.

Benchmark	Size (KLOC)	Bug ID	Crashes	Fault Sig	Group	Correct	Incorrect	Missed
w3m	**80.4**	1	250	1	1	250	0	0
v0.5.3		2	352	4	1	351	0	1
		3	250	4	1	250	0	0
		4	139	2	1	115	0	24
Sub Total		**4**	**991**	**11**	**4**	**966**	**0**	**25**
SQLite	**313.3**	5	285	5	1	285	0	0
v3.8.5		6	191	14	1	191	0	0
		7	240	5	1	240	0	0
		8	113	2	1	113	0	0
		9	226	2	1	226	0	0
		10	237	5	2	237	0	0
		11	250	4	1	250	0	0
		12	236	3	2	236	0	0
Sub Total		**8**	**1778**	**40**	**10**	**1778**	**0**	**0**
libmad v0.15.1b	**18.0**	13	99	2	1	99	0	0
Sub Total		**1**	**99**	**2**	**1**	**99**	**0**	**0**
libarchive	**207.2**	14	67	1	1	67	0	0
v3.1.0		15	85	1	1	85	0	0
Sub Total		**2**	**152**	**2**	**2**	**152**	**0**	**0**
Total	**618.9**	**15**	**3020**	**55**	**17**	**2995**	**0**	**25**

Table 1: Result of RQ1: Evaluating Grouping Correctness

We ran *CERT-BFF* on all the fuzzed crashes for each benchmark, with `backtracelevels` set to 5 (*BFF-5*). This gives us deduplicated fuzzed crashes based on the uniqueness of the last 5 frames (function calls) on the stack. Similarly, we also performed deduplication using the last frame (crashing function) of the call stack by setting `backtracelevels` to 1 (*BFF-1*). We chose these two configurations because *BFF-5* represents the default deduplication used by *CERT-BFF*, and *BFF-1* is used as a baseline in the related work [40].

For *HonggFuzz*, we ran the fuzzed crashes for each benchmarks with the `instrument` option enabled. This gave us deduplicated fuzzed crashes determined using a combination of code coverage, call stack, and crash site information [9]. Then we used the `noinst` mode (*HonggFuzz-S*) to obtain deduplicated fuzzed crashes determined using only call stack and crash site information. We chose these two configurations because *HonggFuzz* represents the default deduplication of the fuzzer and *HonggFuzz-S* is also used as a baseline in the related work [40].

In the experiment, we first collect a set of crashes for a project, e.g. 991 for w3m shown in Table 2. We then run a baseline tool, e.g. AFL, to deduplicate the crashes. The number of groups reported by the tool is listed under the columns of each baseline's `SubTotal` row, e.g., 490 for AFL in Table 2. We then use the developer's patch to determine how many groups were reported for each bug, e.g. 109 for Bug 1 for AFL.

5.1.6 Running the experiments. The initial crash corpus generation and the crash deduplication for the baseline fuzzers were run on a VM with 64-bit 32 core Intel Haswell processors. The *FuzzerAid* experiments were conducted on a VM with 64-bit 12 core Intel Haswell processors. Both the VMs had with 32 GB memory and were running CentOS 8.

5.2 Results for RQ1: Grouping Correctness

Table 1 shows the result for RQ1. Each row corresponds to a known bug labeled with *Bug ID*. The column *Crashes* lists the number of fuzzed crashes generated for each bug using the approach presented in Section 5.1.4. The crashes reported in this column are post-processed using the developer's patch. The *Fault Sig* and *Group* columns provide the number of fault signatures and the number of fault groups generated for the known bug using *FuzzerAid*. Under *Correct* column we list the number of fuzzed crashes that were correctly grouped in one of the fault groups generated for the bug. Similarly, the *Incorrect* column lists the number of fuzzed crashes from unrelated bugs that were incorrectly grouped under one of the fault groups generated for the bug. Any fuzzed crash that *FuzzerAid* failed to group under any fault group is reported under *Missed*.

Our results indicate that among the total 3020 fuzzed crashes for which we know the ground truth, *FuzzerAid* correctly grouped 2995 (99.1%) fuzzed crashes. For 3 benchmarks (`sqlite`, `libmad` and `libarchive`), we correctly classified 100% (2029) of their fuzzed crashes. While we were unable to classify 25 (0.08%) fuzzed crashes, we didn't misclassify any fuzzed crash into unrelated fault groups. The 25 fuzzed crashes we missed for `w3m` (1 from *Bug 2* and 24 from *Bug 4*) were due to fault signature generation failure caused by the

Benchmark	Bug ID	Crashes	FuzzerAid	AFL	BFF-5	BFF-1	Honggfuzz	Honggfuzz-S
w3m	1	250	1	109	2	1	49	49
	2	352	1	208	2	1	9	8
	3	250	1	113	89	7	14	17
	4	139	1	60	4	3	8	7
Sub Total	**4**	**991**	**4**	**490**	**97**	**12**	**80**	**81**
SQLite	5	285	1	179	14	5	2	3
	6	191	1	43	22	8	11	12
	7	240	1	81	7	4	1	1
	8	113	1	43	2	3	1	1
	9	226	1	60	3	1	1	1
	10	237	2	58	4	1	2	2
	11	250	1	134	1	1	2	2
	12	236	2	62	4	2	2	2
Sub Total	**8**	**1778**	**10**	**660**	**57**	**25**	**22**	**24**
libmad	13	99	1	58	4	2	5	6
Sub Total	**1**	**99**	**1**	**58**	**4**	**2**	**5**	**6**
libarchive	14	67	1	24	0	0	1	1
	15	85	1	44	1	1	1	1
Sub Total	**2**	**152**	**2**	**68**	**1**	**1**	**2**	**2**
Total	**15**	**3020**	**17**	**1276**	**159**	**40**	**109**	**113**

Table 2: Results of RQ2: Comparing *FuzzerAid* against SOTA fuzzer deduplication

project specific hard-coded dynamic functions. This implementation issue of *FuzzerAid* could be improved in the future.

FuzzerAid generated a total of 17 fault groups for the 15 known bugs. For 3 benchmarks (`w3m`, `libmad` and `libarchive`), we reported the same number of groups as the ground truth. The 2 extra fault groups (highlighted in red) generated for `sqlite` (one for *Bug 10* and one for *Bug 12*) missed the clustering threshold by a very small margin (difference of 0.08% and 0.4% respectively).

We reported a total of 55 fault signatures for the 15 bugs sized between 52 LOC to 340 LOC. The number of fault signatures can indicate the number of different important paths (or scenario) in which a particular bug can manifest. Of particular interest is *Bug 6* from `sqlite`, which produced 14 fault signatures from *just* 191 fuzzed crashes. The relatively high number of fault signatures may be an indicator that this bug can be crashed from a variety of scenarios and thus likely more important.

Using AFL "Crash Exploration Mode", our crash corpus also included 28 fuzzed crashes that do not belong to any known bugs, which we discovered using the developers' patches (See Section 5.1.4). *FuzzerAid* is able to successfully separate them into different groups from the known bugs.

5.3 Results for RQ2: Comparing Against SOTA

Table 2 shows the result for RQ2. Similar to Table 1, each row corresponds to a known bug. We label them using the same assigned *Bug ID* in Table 1. For all the crashes listed under *Crashes*, the grouping results from *FuzzerAid* and our baselines are listed under *FuzzerAid*, *AFL*, *BFF-5*, *BFF-1*, *Honggfuzz* and *Honggfuzz-S* respectively.

Our results show that *FuzzerAid*'s grouping is the same as the ground truth, except that we generated two additional groups for the *SQLite* bugs. We generated a total of 17 groups for 15 bugs, compared to 40 from *BFF-1*, 109 from *Honggfuzz*, 113 from *Honggfuzz-S*, 159 from *BFF-5*, and 1276 from *AFL*. Considering the challenges and cost of diagnosing a bug, our precise grouping techniques and improvement over the baselines indeed have practical values.

AFL used a conservative approach and applied branch coverage information to group the crashes of the same paths, thus it generated the most group. On the other hand, call stack hashes based approach of CERT-BFF and Honggfuzz were able to greatly reduce the number of groups, but with the risk of misclassification. For example, when we inspected the correctness of the grouping, we found that both *BFF-1* and *BFF-5* incorrectly classified all the fuzzed crashes from the two bugs of `libarchive` into one single group. See the numbers in the row of *libarchive* highlighted in red.

5.4 Summary

In our evaluation, *FuzzerAid* is able to correctly group 2995 out of 3020 (99.1%) fuzzed crashes without any incorrect classification. We were also the closest to ground truth in terms of grouping with 17 fault groups reported instead of ground truth's 15. The next closest baseline (BFF-1) reported 40 groups (2.35 times more) while still misclassifying fuzzed crashes from one group for `libarchive`.

The trace generated by PIN for creating the fault signatures varied between 2.03 M lines to 9.5 K lines. Using the program reduction techniques, the fault signatures used to group crashes range betwen 52 LOC to 340 LOC. Such fault signatures provided fault localization information and potentially help developers focus on a small set of statements for bug understanding and diagnosis.

5.5 Threats To Validity

Internal Threat to Validity: One of the important challenges of evaluating deduplication of fuzzing results is that we need to have ground truth for grouping. To simulate this setting, our approach is to take known bugs and configure the fuzzers to generate only crashes for the known bugs. This approach can detect whether we are able to group crashes of the same bug together. Meanwhile, to evaluate that we do not mistakenly group crashes from one bug to other groups, we mixed all the crashes from known bugs and see whether the grouping is correct. We also consider the fact that using AFL "Crash Exploration Mode" may generate additional crashes from unknown bugs. We used the developers' patches to fix each bug and observe if the crashes disappear. We also used a similar approach to validate if there are any misclassification in the groups generated by *FuzzerAid* and other baselines.

External Threat to Validity: To evaluate if our techniques can be generally applicable in practice, we used 15 different bugs from 4 real-world large open-source projects. These projects range from 18 KLOC to 313 KLOC and cover a variety of software, e.g., text based web browser and audio library. We also selected 3 SOTA widely used fuzzers and their 5 total different settings as baselines to understand if our approach indeed advances the SOTA. Although more crashes, bugs, software, and baselines can be useful for further confirming the generality of our approaches, our current results do provide confidence that our approach is promising and can be useful.

6 RELATED WORK

The STOA fuzzers [13–15, 21, 26, 35, 39, 43] use either a coverage based [13, 21, 43] or call stack based [14, 15, 26, 35, 39] heuristics to determine uniqueness of the fuzzed crashes and report the deduplicated fuzzed crashes. Boehme et al. [13] extended AFL to direct the fuzzing towards a specific target, while Gan et al. [21] improved AFL to more uniquely determine the branch coverage when fuzzing. Even though both carried out additional (manual) verification when reporting "unique" bugs, they didn't change AFL's underlying deduplication method of using branch coverage. Similarly, most of the call stack based approaches used the same hashing method proposed by Molar et al. [33] with varying number of calls (frames) used for the hashing. Of particular interest are `SYMFUZZ` [15] that uses "safe stack hash", that only considered non-corrupted call stacks, and `VUzzer` [35] that uses the last 10 basic blocks along with call stack to generate hashes to prevent classifying fuzzed crashes from different bugs into the same group. These deduplication methods are tightly coupled to their respective fuzzers and hard to use independently. In contrast, our method is agnostic of the method used for generating the fuzzed crashes. Since we capture the root cause of the bug in the fault signatures, we are also less prone to the over counting and misclassification of "unique" bugs found in the coverage and stack hash based methods [27].

There are also grouping methods developed independent of the fuzzers [16, 25, 34, 40]. The closest related work is by Chen et al. [16] that calculated a "distance" between fuzzed crashes to capture their static and dynamic properties and used a machine leaning to rank them. Homes et al. [25] and van Tonder et at. [40] grouped fuzzed crashes based on their responses to the mutations of the program. They hypothesize that if the behavior of two fuzzed crashes change similarly (change from crashing to not crashing) due to the same mutation, then they are more likely to be the same bug. Pham et al. [34] proposed using symbolic constraints on input paths to group fuzzed crashes. They have limited applicability due to the reliance on symbolic execution to generate the constraints. Cui et al. [19] and Molar et al. [33] proposed call stack similarity to group fuzzed crashes. They are more prone to misclassification as discussed above. We have considered these related work as baselines, however, they are either tailored for specific applications like [16, 25] or limited in the types of bugs [40] or benchmarks [34] they can handle, while others [19, 33] are very similar to our current baselines.

There have also been work on performing fault localization for fuzzers. Blazytko et al. [11] is a representative work in this area. Similar to our approach of generating crash corpus, they used a known crashing input from a fuzzed crash to generate similar inputs to observe the dynamic state of the program. These are used to generate predicates similar to the input path constraints generated by symbolic execution to isolate the root cause. Variations of delta debugging [17, 22, 41, 42] have also been used for fault location. Both Christi et al. [17] and Vince et al. [41] proposed reducing the fuzzed crashes, similar to our approach in generating fault signatures, before trying to localize bugs in order to improve their accuracy. The main difference to our work is that they only identify or rank root causes for a crash, but do not generate executable fault signatures and use them to group fuzzed crashes.

7 CONCLUSIONS AND FUTURE WORK

This paper presents a heuristics based approach for deduplicating fuzzed crashes. As opposed to the use call stacks, code coverage, and failure symptoms based approaches, our approach uses *fault signatures* to group fuzzed crashes. A fault signature captures the necessary statements that allow the bugs to be reproduced. Crashes grouped based on a fault signature thus likely share the root causes and fixes. We developed an algorithm and a tool that consist of the three components, *generating fault signatures*, *classifying with fault signatures* and *merging fault signatures*. We evaluated our approach on 3020 fuzzed crashes against the ground truth we set up from 15 real-world bugs and patches and from 4 different open-source projects. Our results show that our approach correctly grouped 99.1% of 3020 fuzzed crashes and generated 17 groups for 15 bugs. Our approach significantly outperformed the deduplication methods offered by 3 SOTA fuzzers, namely AFL, BFF and HonggFuzz, which reported 40-1276 groups. Considering diagnosing a crash can be challenging and time-consuming, we believe our tool can significantly improve the debugging productivity for fuzzing. In the future, we will explore the further usage of fault signatures for fault localization and automated patch generation/verification. We will also experiment our approach for grouping fuzzed crashes from different program versions and from different fuzzers.

REFERENCES

[1] 2013. https://insights.sei.cmu.edu/blog/one-weird-trick-for-finding-more-crashes/
[2] 2014. https://lcamtuf.coredump.cx/afl/
[3] 2014. https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html
[4] 2015. https://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html
[5] 2015. http://www.pl-enthusiast.net/2015/09/08/what-is-a-bug/
[6] 2020. https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/
[7] 2020. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/
[8] 2021. https://google.github.io/clusterfuzzlite/running-clusterfuzzlite/github-actions/
[9] 2021. https://github.com/google/honggfuzz/blob/master/docs/USAGE.md#output-files=
[10] 2022. Looking back at the zero-day initiative in 2021. https://www.thezdi.com/blog/2022/1/20/looking-back-at-the-zero-day-initiative-in-2021
[11] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. 235–252. https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko
[12] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (May 2021), 79–86. https://doi.org/10.1109/MS.2020.3016773
[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020
[14] CERT. 2010. CERT Basic Fuzzing Framework (BFF). *URL: https://github.com/CERTCC/certfuzz (visited on 01/10/2022)* (2010).
[15] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, USA, 725–741. https://doi.org/10.1109/SP.2015.50
[16] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, 197–208. https://doi.org/10.1145/2491956.2462173
[17] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 184–191. https://doi.org/10.1109/ISSREW.2018.00005
[18] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance*. 516–519. https://doi.org/10.1109/ICSM.2013.85
[19] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 820–831. https://doi.org/10.1145/2884781.2884844
[20] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. *arXiv:2103.11518 [cs]* (Mar 2021). http://arxiv.org/abs/2103.11518 arXiv: 2103.11518.
[21] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040
[22] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause Reduction for Quick Testing. In *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*. 243–252. https://doi.org/10.1109/ICST.2014.37
[23] 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 (Nov 2020), 1–29. https://doi.org/10.1145/3428334 arXiv: 2009.01120.
[24] Richard D Hipp. 2022. SQLite. https://www.sqlite.org/index.html
[25] Josie Holmes and Alex Groce. 2018. Causal Distance-Metric-Based Assistance for Debugging after Compiler Fuzzing. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 166–177. https://doi.org/10.1109/ISSRE.2018.00027
[26] Ulf Kargén and Nahid Shahmehri. 2015. 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 782–792. https://doi.org/10.1145/2786805.2786844
[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804
[28] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04)*. IEEE Computer Society, USA, 75.
[29] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (Sep 2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476
[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (Jun 2005), 190–200. https://doi.org/10.1145/1064978.1065034
[31] Many. 2017. Build EAR: a tool that generates a compilation database. https://github.com/rizsotto/Bear. Accessed: 2017-05-07.
[32] Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT '14*. ACM Press, Portland, Oregon, USA, 103–104. https://doi.org/10.1145/2663171.2663188
[33] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium (SSYM'09)*. USENIX Association, USA, 67–82.
[34] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing Failing Tests via Symbolic Analysis. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Marieke Huisman and Julia Rubin (Eds.). Springer, 43–59. https://doi.org/10.1007/978-3-662-54494-5_3
[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2017.23404
[36] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104
[37] Ryan. 2022. The More You Know, The More You Know You Don't Know. https://googleprojectzero.blogspot.com/
[38] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. (2017). https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany
[39] Robert Swiecki. 2017. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URL: https://github. com/google/honggfuzz (visited on 06/21/2017)* (2017).
[40] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 612–622. https://doi.org/10.1145/3238147.3238200
[41] Dániel Vince, Renáta Hodován, and Ákos Kiss. 2021. Reduction-assisted Fault Localization: Don't Throw Away the By-products!:. In *Proceedings of the 16th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, 196–206. https://doi.org/10.5220/0010560501960206
[42] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, 52–63. https://doi.org/10.1145/2635868.2635906
[43] Michal Zalewski. 2017. American fuzzy lop. *URL: URL http://lcamtuf.coredump.cx/afl (visited on 01/10/2022)* (2017).