A Guided Genetic Algorithm for Automated Crash Reproduction

Soltani, Mozhan; Panichella, Annibale; van Deursen, Arie

DOI
10.1109/ICSE.2017.27

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings of the 39th International Conference on Software Engineering (ICSE)

Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Guided Genetic Algorithm for Automated Crash Reproduction

Mozhan Soltani, Annibale Panichella, and Arie van Deursen

Report TUD-SERG-2017-006

TUD-SERG-2017-006

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication at the research track of the ACM/IEEE International Conference on Software Engineering (ICSE), held in Buenos Aires, May 2017.

A Guided Genetic Algorithm for Automated Crash Reproduction

Mozhan Soltani
Delft University of Technology
The Netherlands
m.soltani@tudelft.nl

Annibale Panichella
SnT Centre - University of Luxembourg
Luxembourg
annibale.panichella@uni.lu

Arie van Deursen
Delft University of Technology
The Netherlands
Arie.vanDeursen@tudelft.nl

***Abstract*—To reduce the effort developers have to make for crash debugging, researchers have proposed several solutions for automatic failure reproduction. Recent advances proposed the use of symbolic execution, mutation analysis, and directed model checking as underling techniques for post-failure analysis of crash stack traces. However, existing approaches still cannot reproduce many real-world crashes due to such limitations as environment dependencies, path explosion, and time complexity. To address these challenges, we present EvoCrash, a post-failure approach which uses a novel Guided Genetic Algorithm (GGA) to cope with the large search space characterizing real-world software programs. Our empirical study on three open-source systems shows that EvoCrash can replicate 41 (82%) of real-world crashes, 34 (89%) of which are useful reproductions for debugging purposes, outperforming the state-of-the-art in crash replication.**

***Keywords*-Search-Based Software Testing; Genetic Algorithms; Automated Crash Reproduction;**

I. Introduction

Manual crash replication is a labor-intensive task. Developers faced with this task need to reproduce failures reported in issue tracking systems, which all too often contain insufficient data to determine the root cause of a failure.

Hence, to reduce developer effort, many different automated crash replication techniques have been proposed in the literature. Such techniques typically aim at generating tests triggering the target failure. For example, record-replay approaches [1]–[5] monitor software behavior via software/hardware instrumentation to collect the observed objects and method calls when failures occur. Unfortunately, such techniques suffer from well-known practical limitations, such as performance overhead [6], and privacy issues [7].

As opposed to these costly techniques, *post-failure* approaches [6]–[12] try to replicate crashes by exploiting data that is available *after* the failure, typically stored in log files or external bug tracking systems. Most of these techniques require specific input data in addition to crash stack traces [6], such as core dumps [8]–[10], [13] or models of the software like input grammars [14], [15] or class invariants [16].

Since such additional information is usually not available to developers, recent advances in the field have focused on crash stack traces as the *only* source of information for debugging [6], [7], [12]. For example, Chen and Kim developed STAR [6], an approach based on backward symbolic execution. STAR outperforms earlier crash replication techniques, such as Randoop [17] and BugRedux [18]. Xuan et al. [12] presented MuCrash, a tool that updates existing test cases using specific mutation operators, thus creating a new pool of tests to run against the software under test. Nayrolle et al. [7] proposed JCHARMING, based on directed model checking combined with program slicing [7], [19].

Unfortunately, the state-of-the-art tools suffer from several limitations. For example, STAR cannot handle cases with external environment dependencies [6] (e.g., file or network inputs), non-trivial string constraints, or complex logic potentially leading to a path explosion. MuCrash is limited by the ability of existing tests in covering method call sequences of interest, and it may lead to a large number of unnecessary mutated test cases [12]. JCHARMING [7], [19] applies model checking which can be computationally expensive. Moreover, similar to STAR, JCHARMING does not handle crash cases with environmental dependencies.

In our previous preliminary study [20], we have suggested to re-use existing unit test generation tools, such as EvoSuite [21], for crash replication. To that end, we developed a *fitness function* to assess the capability of candidate test cases in replicating the target failure. Although this simple solution could help to replicate one crash not handled by STAR and MuCrash, our preliminary study showed that this simple solution still leaves other crashes as non-reproducible. These negative results are due to the large search space for real world programs where the probability to generate test data satisfying desired failure conditions is low. In fact, the classic *genetic operators* from existing test frameworks are aimed at maximizing specific coverage criteria [21] instead of exploiting single execution paths and object states that characterize software failures.

To address this challenge, this paper presents an evolutionary search-based approach, named EvoCrash, for crash reproduction. EvoCrash is built on top of EvoSuite [21], the well-known automatic test suite generation tool for Java. For EvoCrash we developed a novel *guided* genetic algorithm (GGA). It lets the stack trace guide the search, thus reducing the search space. In particular, GGA uses a novel generative routine to build an initial population of tests exercising at least one of the methods reported in the crash stack frames.

Furthermore, GGA uses new crossover and mutation operators to avoid the generation of futile tests that lack calls to failing methods. To further guide the search process, we developed a novel *fitness function* that improves the calculation of *stack trace distance* previously defined in [20], to assess candidate test cases.

The contributions of our paper are:

- A novel *guided* genetic algorithm (GGA) for crash reproduction that generates and evolves only tests that exercise at least one of the methods involved in the failure;
- EvoCrash, a Java tool implementing GGA that generates JUnit tests that developers can directly use for debugging purposes;
- An empirical study on 50 real-world software crashes involving different versions of three open source projects showing that EvoCrash can replicate 41 cases (82%), 34 (89%) of which are useful for debugging;
- A comparison of EvoCrash with three state-of-the-art approaches based on crash stack traces (STAR [6], MuCrash [12] and JCHARMING [7]).

Furthermore, we provide a publicly available replication package[1] that includes: (i) an executable jar of EvoCrash, (ii) all bug reports used in our study, and (iii) the test cases generated by our tool.

II. Related Work

Since our approach aims at crash reproduction using test generation, we start out by summarizing related work in the areas of automated crash replication and coverage-based unit test generation.

A. Automated Approaches to Crash Replication

Previous approaches in the field of crash replication can be grouped into three main categories: (i) *record-play* approaches, (ii) *post-failure* approaches using various data sources, and (iii) *stack-trace based post-failure* techniques. The first category includes the earliest works in this field, such as ReCrash [1], ADDA [2], Bugnet [3], and jRapture [5]. The aforementioned techniques rely on program run-time data for automated crash replication. Thus, they record the program execution data in order to use it for identifying the program states and execution path that led to the program failure. However, monitoring program execution may lead to (i) substantial performance overhead due to software/hardware instrumentation [6]–[8], and (ii) severe privacy issues since the collected execution data may contain sensitive information [6].

On the other hand, post-failure approaches [8]–[11], [15] analyze software data (e.g., core dumps) only after crashes occur, thus not requiring any form of instrumentation. Rossler et al. [8] developed an evolutionary search-based approach named RECORE that leverages from core dumps (taken at the time of a failure) to generate input data. RECORE combines the search-based input generation with a coverage-based technique to generate method sequences. Weeratunge et al. [13] used core dumps and directed search for replicating crashes related to concurrent programs in multi-core platforms. Leitner et al. [9], [10] used a failure-state extraction technique to create tests from core dumps (to derive input data) and stack traces (to derive method calls). Kifetew et al. [14], [15] used genetic programming requiring as input (i) a grammar describing the program input, and (ii) a (partial) call sequence. Boyapati et al. [16] developed another technique requiring manually written specifications containing method preconditions, post-conditions, and class invariants. However, all these *post-failure* approaches need various types of information that are often not available to developers, thus decreasing their feasibility.

To increase the practical usefulness of automated approaches, researchers have focused on crash stack traces as the only source of information for debugging. For instance, ESD [11] uses forward symbolic execution that leverages commonly reported elements in bug reports. BugRedux [18] also uses forward symbolic execution but it can analyze different types of execution data, such as crash stack traces. As highlighted by Chen and Kim [6], both ESD and BugRedux rely on forward symbolic execution, thus inheriting its problems due to *path explosion* and *object creation* [22]. To address these two issues, Chen and Kim [6] introduced STAR, a tool that applies backward symbolic execution to compute crash preconditions and generates a test using a method sequence composition approach.

Different from STAR, JCHARMING [7] uses a combination of crash traces and model checking to automatically reproduce bugs that caused field failure. To address the state explosion problem [23] in model checking, JCHARMING applies program slicing to direct the model checking process by reduction of the search space. Instead, MuCrash [12] uses mutation analysis as the underlying technique for crash replication. First, MuCrash selects the test cases that include the classes in the crash stack trace. Next, it applies predefined mutation operators on the tests to produce mutant tests that can reproduce the target crash.

In our earlier study [20], we showed that even coverage-based tools like EvoSuite can replicate some target crashes if relying on a proper fitness function specialized for crash replication. However, our preliminary results also indicated that this simple solution could not replicate some cases for two main reasons: (i) limitations of the developed fitness function, and (ii) the large search space in complex real-world software. The EvoCrash approach presented in this paper resumes this line of research because it used evolutionary search to synthesize a crash reproducible test case. However, it is novel because it utilizes a smarter fitness function and it applies an Guided Genetic Algorithm (GGA) instead of coverage-oriented genetic algorithms. Section III presents full details regarding the novel fitness function and GGA in EvoCrash.

B. Unit Test Generation Tools

A number of techniques and tools have been proposed in the literature to automatically generate tests maximizing specific code coverage criteria [17], [21], [24]–[27]. The main

[1] http://www.evocrash.org/

difference among them is represented by the core search algorithm used for generating tests. For example, EvoSuite [21] and JTExpert [27] use genetic algorithms to create test suites optimizing branch coverage; Randoop [17] and T3 [24] apply random testing, while DART [25] and Pex [26] are based on dynamic symbolic execution.

As reported in the related literature, such tools can be used to discover bugs affecting software code. Indeed, they can generate test triggering crashes when trying to generate tests exercising the uncovered parts of the code. For example, Fraser and Arcuri [28] successfully used EvoSuite to discover undeclared exceptions and bugs in open-source projects. Recently, Moran et al. [29] used coverage-based tools to discover android application crashes. However, as also pointed out by Chen and Kim [6] coverage-based tools are not specifically defined for crash replication. In fact, these tools are aimed at covering all methods (and their code elements) in the class under test. Thus, already covered methods are not taken into account for search even if none of the already generated tests synthesizes the target crash. Therefore, the probability of generating tests satisfying desired crash triggering object states is particularly low for coverage-based tools [6].

On the other hand, for crash replication, not all methods should be exploited for generating a crash: we are interested in covering only the few lines in those methods involved in failure, while other methods (or classes) might be useful only for instantiating the necessary objects (e.g., input parameters). Moreover, among all possible method sequences, we are interested only on those that can potentially lead to the target crash stack trace. Therefore, in this paper we developed a tool, named EvoCrash, which is specialized for stack trace based crash replication.

III. The EvoCrash Approach

According to Harman et al. [30], [31], there are two key ingredients for a successful application of search-based techniques. The first is the formulation of a proper fitness function to guide the search toward reaching the target, which in our case is a way to trigger a crash. The second ingredient consists of applying a proper search algorithm to promote tests closer to mimicking the crash, while penalizing tests with poor fitness values. The next sub-sections detail the fitness function as well as the genetic algorithms we designed in EvoCrash.

A. Crash Stack Trace Processing

An optimal test case for crash reproduction has to crash at the same location as the original crash and produce a stack trace as close to the original one as possible. Therefore, in EvoCrash we first parse the log file given as input in order to extract the crash stack frames of interest. A standard Java stack trace contains (i) the type of the exception thrown, and (ii) the list of stack frames generated at the time of the crash. Each stack frame corresponds to one method involved in the failure, hence, it contains all information required for its identification: (i) the method name; (ii) the class name, and (iii) line numbers where the exception was generated. The last frame is where the exception has been thrown, whereas the root cause could be in any of the frames, or even outside the stack trace.

From a practical point of view, any class or method in the stack trace can be selected as code unit to use as input for existing test case generation tools, such as EvoSuite. However, since our goal is to synthesize a test case generating a stack trace as close to the original trace as possible, we always target the class where the exception is thrown (last stack frame in the crash stack trace) as the main class under test (CUT).

B. Fitness Function

As described in our previous study [20], our fitness function is formulated to consider three main conditions that must hold so that a test case would be evaluated as optimal and have zero distance: (i) the line (statement) where the exception is thrown has to be covered, (ii) the target exception has to be thrown, and (iii) the generated stack trace must be as similar to the original one as possible. More formally, we use the following fitness formulation:

Definition 1. *The fitness function value of a given test t is:*

$$f(t) = 3 \times d_s(t) + 2 \times d_{except}(t) + d_{trace}(t) \qquad (1)$$

where $d_s(t)$ denotes how far t is to executing the target statement, i.e., the location of the crash; $d_{except}(t) \in \{0, 1\}$ is a binary value indicating whether the target exception is thrown or not; and $d_{trace}(t)$ measures the distance between the generated stack trace (if any) and the expected trace.

For the line distance $d_s(t)$, we use the two well-known heuristics *approach level* and *branch distance* to guide the search for branch and statement coverage [20]. The *approach level* measures the distance (i.e., minimum number of control dependencies) between the path of the code executed by t and the target statement. The *branch distance* uses a set of well-established rules [32] to score how close t is to satisfying the branch condition for the branch on which the target statement is directly control dependent.

If the target exception is thrown, $d_{except}(t) = 0$, then we proceed by calculating the trace distance, $d_{trace}(t)$, otherwise, the trace distance remains equal to the maximum value it can have, 1.0. To calculate the trace distance, $d_{trace}(t)$, in our preliminary study [20] we used the distance function defined as follows. Let $S^* = \{e_1^*, \ \ldots, e_n^*\}$ be the target stack trace to replicate, where $e_i^* = (C_1^*, m_1^*, l_1^*)$ is the i-the element in the trace composed by class name C_i^*, method name m_i^*, and line number l_i^*. Let $S = \{e_1, \ldots, e_k\}$ be the stack trace (if any) generated when executing the test t. The distance between the expected trace S^* and the actual trace S is defined as:

$$D(S^*, S) = \sum_{i=1}^{\min\{k,n\}} \varphi\left(\text{diff}(e_i^*, e_i)\right) + \mid n - k \mid \qquad (2)$$

where $\text{diff}(e_i^*, e_i)$ measures the distance between the two trace elements e_i^* and e_i in the traces S^* and S respectively; finally, $\varphi(x) \in [0, 1]$ is the widely used normalizing function $\varphi(x) = x/(x + 1)$ [32]. However, such a distance definition

has one critical limitation: it strictly requires that the expected trace S^* and the actual trace S share the same prefix, i.e., the first $\min\{k, n\}$ trace elements. For example, assume that the triggered stack trace S and target trace S^* have one stack trace element e_{shared} in common (i.e., one element with the same class name, method name, and source code line number) but that is located at two different positions, e.g., e_i^* is the second element in S ($e_{shared} = e_2$ in S) while it is the third one in S^* ($e_{shared} = e_3^*$ in S^*). In this scenario, Equation 2 will compare the element e_3^* in S^* with the element in S at the same position i (i.e., with e_3) instead of considering the closest element $e_{shared} = e_2$ for the comparison.

To overcome this critical limitation, in this paper we use the following new definition of stack trace distance:

Definition 2. *Let S^* be the expected trace, and let S be the actual stack trace triggered by a given test t. The stack trace distance between S^* and S is defined as:*

$$D(S^*, S) = \sum_{i=1}^{n} \min\{\text{diff}(e_i^*, e_j) : e_j \in S\} \quad (3)$$

where $\text{diff}(e_i^*, e_j)$ measures the distance between the two trace elements e_i^* in S^* and its closest element e_j in S.

We say that two trace elements are equal if and only if they share the same trace components. Therefore, we define $\text{diff}(e_i^*, e_j)$ as follows:

$$\text{diff}(e_i^*, e_i) = \begin{cases} 3 & C_i^* \neq C_i \\ 2 & C_i^* = C_i \text{ and } m_i^* \neq m_i \\ \varphi(\mid l_i^* - l_i \mid) & \text{Otherwise} \end{cases} \quad (4)$$

The score $\text{diff}(e_i^*, e_i)$ is equal to zero if and only if the two trace elements e_i^* and e_i share the same class name, method name and line number. Similarly, $D(S^*, S)$ in Equation 3 is zero if and only if the two traces S^* and S are equal, i.e., they share the same trace elements. Starting from the function in Equation 3, we define the trace distance $d_{trace}(t)$ as the normalized $D(S^*, S)$ function:

$$d_{trace}(t) = \varphi(D(S^*, S)) = D(S^*, S)/(D(S^*, S) + 1) \quad (5)$$

Consequently, $D(S^*, S)$ in Equation 3 is zero if and only if S^* shares the same trace elements with S. In addition, our fitness function $f(t)$ assumes values within the interval $[0, 6]$, reaching a zero value if and only if the evaluated test t replicates the target crash.

C. Guided Genetic Algorithm

In EvoCrash, we use a novel genetic algorithm, named GGA (Guided Genetic Algorithm), suitably defined for the crash replication problem. While traditional search algorithms in coverage-based unit test tools target all methods in the CUT, GGA gives higher priority to those methods involved in the target failure. To accomplish this, GGA uses three novel *genetic operators* that create and evolve test cases that always exercise at least one method contained in the crash stack trace, increasing the overall probability of triggering the target crash. As shown in Algorithm 1, GGA contains all main steps of a standard genetic algorithm: (i) it starts with creation of an initial population of random tests (line 5); (ii) it evolves such tests over subsequent generations using *crossover* and *mutation* (lines 12-20); and (iii) at each generation it *selects* the fittest tests according to the fitness function (lines 22-24). The main difference is represented by the fact that it uses (i) a novel routine for generating the initial population (line 5); (ii) a new crossover operator (line 15); (iii) a new mutation operator (lines 19-20). Finally, the fittest test obtained at the end of the search is optimized by post-processing (in line 26).

Algorithm 1: Guided Genetic Algorithm

```
Input: Class under test C
       Target call from the crash stack trace TC
       Population size N
       Search time-out max_time
Result: Test case t
1  begin
2      // initialization
3      M_crash ⟵ identify public methods based on TC
4      k ⟵ 0
5      P_k ⟵ MAKE-INITIAL-POPULATION(C, M_crash, N)
6      EVALUATE(P_k)
7      // main loop
8      while (best fitness value > 0) AND (time spent < max_time) do
9          k ⟵ k + 1
10         // generate offsprings
11         O ⟵ ∅
12         while | O | < N do
13             p1, p2 ⟵ select two parents for reproduction
14             if crossover probability then
15                 o1, o2 ⟵ GUIDED-CROSSOVER(p1, p2)
16             else
17                 o1 ⟵ p1
18                 o2 ⟵ p2
19             O ⟵ O ⋃ GUIDED-MUTATION(o1)
20             O ⟵ O ⋃ GUIDED-MUTATION(o2)
21         // fitness evaluation
22         EVALUATE(O)
23         P_k ⟵ P_{k-1} ⋃ O
24         P_k ⟵ select the N fittest individuals in P_k
25     t_best ⟵ fittest individual in P_k
26     t_best ⟵ POST-PROCESSING(t_best)
```

Initial Population. The routine used to generate the initial population plays a paramount role [33] since it performs sampling of the search space. In traditional coverage-based tools (e.g., EvoSuite [21] or JTExpert [27]) such a routine is designed to generate a well-distributed population (set of tests) calling as many methods in the target class as possible [21], which is not the main goal for crash replication.

For this reason, in this paper we use the novel routine highlghted in Algorithm 2 for generating the initial sample for random tests. In particular, our routine gives higher importance to methods contained in crash stack frames. Subsequently, if a target call, selected by the developer, is public or protected, Algorithm 2 guarantees that this call is inserted in each test at least once. Otherwise, if the target call is private, the algorithm guarantees that each test contains at least one call to a public caller method which invokes the target private call. Algorithm 2 generates random tests using the loop in lines 3-18, and requires as input (i) the set of public target method(s)

Algorithm 2: MAKE-INITIAL-POPULATION

```
Input: Class under test C
       Set of failing methods M_crash
       Population size N
Result: An initial population P_0
1  begin
2      P_0 ⟵ ∅
3      while | P_0 | < N do
4          t ⟵ empty test case
5          size ⟵ random integer ∈ [1;MAX_SIZE]
6          // probability of inserting a method involved in the failure
7          insert_probability ⟵ 1/size
8          while (number of statements in t) < size do
9              if random_number ⩽ insert_probability then
10                 method_call ⟵ pick one element from M_crash
11                 // reset the probability of inserting a failing method
12                 insert_probability ⟵ 1/size
13             else
14                 method_call ⟵ pick one public method in C
15                 length ⟵ number of statements in t
16                 // increase the probability of inserting a failing method
17                 insert_probability ⟵ 1/(size − length + 1)
18             INSERT-METHOD-CALL(method_call, t)
19         P_0 ⟵ P_0 ⋃ t
```

Algorithm 3: GUIDED-CROSSOVER

```
Input: Parent tests p_1 and p_2
       Set of failing methods M_crash
Result: Two offsprings o_1, o_2
1  begin
2      size_1 ⟵ | p_1 |
3      size_2 ⟵ | p_2 |
4      // select a cut point
5      μk ⟵ random number ∈ [0; 1]
6      // first offspring
7      o_1 ⟵ first μ × size_1 statements from p_1
8      o_1 ⟵ append (1 − μ) × size_2 statements from p_2
9      CORRECT(o_1)
10     if o_1 does not contain methods from M_crash then
11         o_1 ⟵ clone of p_1
12     // second offspring
13     o_2 ⟵ first μ × size_2 statements from p_2
14     o_2 ⟵ append (1 − μ) × size_1 statements from p_1
15     CORRECT(o_2)
16     if o_2 does not contain methods from M_crash then
17         o_2 ⟵ clone of p_2
```

M_{crash}, (ii) the population size N, and (iii) the class under test C. In each iteration, we create an empty test t (line 4) to fill with a random number of statements (lines 5-18). Then, statements are randomly inserted in t using the iterative routine in lines 8-18: at each iteration we insert a call to one public method either taken from M_{crash}, or member classes of C. In the first iteration, crash methods in M_{crash} (methods of interest) are inserted in t with a low probability $p = 1/size$ (line 7), where $size$ is the total number of statements to add in t. In the subsequent iterations, such a probability is automatically increased when no methods from M_{crash} is inserted in t (line 15-17). Therefore, Algorithm 2 ensures that at least one method of the crash is inserted in each initial test[2].

The process of inserting a specific method call in a test t requires several additional operations [21]. For example, before inserting a method call m in t it is necessary to instantiate an object of the class containing m (e.g., calling one of the public constructors). Creating a proper method call also requires the generation of proper input parameters, such as other objects or primitive variables. For all these additional operations, Algorithm 2 uses the routine INSERT-METHOD-CALL (line 18). For each method call in t, it sets the input parameters values by re-using objects and variables already defined in t, setting some input values to `null` (only for objects used as input parameters), or randomly generating new objects and primitive values.

Guided Crossover. Even if all tests in the initial population exercise one or more methods contained in the crash stack trace, during the evolution process—i.e., across different generations— tests can lose the inserted target calls. One possible cause for this scenario is represented by the traditional *single-point* crossover, which generates two offsprings by randomly exchanging statements between two parent tests p_1 and p_2. Given a random cut-point μ, the first offspring o_1 inherits the first μ statements from parent p_1, followed by $| p_2 | - \mu$ statements from parent p_2. Vice versa, the second offspring o_2 will contain μ statements from parent p_2 and $| p_1 | - \mu$ statements from the parent p_1. Even if both parents exercise one or more failing methods from the crash stack trace, after crossover is performed, the calls may be moved into one offspring only. Therefore, the traditional single-point crossover can hamper the overall algorithm.

To avoid this scenario, GGA leverages a novel *guided single-point crossover* operator, whose main steps are highlighted in Algorithm 3. The first steps in this crossover are identical to the standard single-point crossover: (i) it selects a random cut point μ (line 5), (ii) it recombines statements from the two parents around the cut-point (lines 7-8 and 12-13 of Algorithm 3). After this recombination, if o_1 (or o_2) loses the target method calls (a call to one of the methods reported in the crash stack trace), we reverse the changes and re-define o_1 (or o_2) as pure copy of its parent p_1 (p_2 for offspring o_2) (`if` conditions in lines 10-11 and 16-17). In this case, the mutation operator will be in charge of applying changes to o_1 (or o_2).

Moving method calls from one test to another may result in non well-formed tests. For example, an offspring may not contain proper class constructors before calling some methods; or some input parameters (either primitive variables or objects) are not inherited from the original parent. For this reason, Algorithm 3 applies a *correction* procedure (lines 9 and 15) that inserts all required objects and primitive variables into non well-formed offspring.

Guided Mutation. After crossover, new tests are usually mutated (with a low probability) by adding, changing and removing some statements. While adding statements will not affect the type of method calls contained in a test, the statement deletion/change procedures may remove relevant calls to methods in the crash stack frame. Therefore, GGA also uses a new *guided-mutation* operator, described in Algorithm 4.

[2]In the worst case, a failing method will be inserted at position $size$ in t since the probability $insert\_probability$ will be $1/(size - size + 1) = 1$

Algorithm 4: GUIDED-MUTATION

```
Input: Test t = ⟨s_1, ..., s_n⟩ to mutate
       Set of failing methods M_crash
Result: Mutated test t
1  begin
2      n ⟵| t |
3      apply_mutation ⟵ true
4      while apply_mutation == true do
5          for i =1 to n do
6              φ ⟵ random number ∈ [0; 1]
7              if φ ⩽ 1/n then
8                  if delete probability then
9                      delete statement s_i
10                 if change probability then
11                     change statement s_i
12                 if insert probability then
13                     insert a new method call at line i
14         if t contains method from M_crash then
15             apply_mutation ⟵ false
```

Let $t = \langle s_1, \ldots, s_n \rangle$ be a test case to mutate, the *guided-mutation* iterates over the test t and mutates each statement with probability $1/n$ (main loop in lines 4-15). Inserting statements consists of adding a new method call at a random point $i \in [1; n]$ in the current test t (lines 12-13 in Algorithm 4). This procedure also requires to instantiate objects or declare/initialize primitive variables (e.g., integers) that will be used as input parameters.

When changing a statement at position i (in lines 10-11), the mutation operator has to handle two different cases: (i) if the statement s_i is the declaration of a primitive variable (e.g., an integer), then its primitive value is changed with another random value (e.g., another random integer); (ii) if s_i contains a method or a constructor call m, then the mutation is applied by replacing m with another public method/constructor having the same return type while its input parameters (objects or primitive values) are taken from the previous $i-1$ statements in t, set to `null` (for objects only), or randomly generated.

Finally, removing a method call (lines 8-9 in Algorithm 4) requires to delete the corresponding variables and objects used as input parameters (if such variables and objects are not used by any other method call in t). If the test t loses the target method calls (i.e., methods in M_{crash}) because of the mutation, then the loop in lines 4-15 is repeated until one or more target method calls are re-inserted in t; otherwise the mutation process terminates.

Post processing. At the end of the search process, GGA returns the fittest test case according to our fitness function. The resulting test t_{best} can be directly used by developer as starting point for crash replication and debugging.

Since method calls are randomly inserted/changed during the search process, the final test t_{best} can contain statements not useful to replicate the crash. For this reason, GGA post-processes t_{best} to make it more concise and understandable. For this post-processing, we reused the *test optimization* routines available in EvoSuite [21], namely: *test minimization*, and *values minimization*. *Test minimization* applies a simple greedy algorithm: it iteratively removes all statements that do not affect the final fitness value. Finally, randomly generated input values can be hard to interpret for developers [34]. Therefore, the *values minimization* from EvoSuite shortens the identified numbers and simplifies the randomly generated strings [35].

TABLE I
REAL-WORLD BUGS USED IN OUR STUDY.

Project	Bug IDs	Versions	Exception	Priority	Ref.
ACC	4, 28, 35, 48, 53, 68, 70,77, 104, 331, 277, 411	2.0 - 4.0	NullPointer (5), UnsupportedOperation (1), IndexOutOfBounds, (1) IllegalArgument(1), ArrayIndexOutOfBounds, (2) ConcurrentModification, (1) IllegalState (1),	Major (10) Minor (2)	[6] [12]
ANT	28820, 33446, 34722, 34734, 36733, 38458, 38622, 42179, 43292, 44689, 44790, 46747, 47306, 48715, 49137, 49755, 49803, 50894 51035, 53626	1.6.1 - 1.8.2	ArrayIndexOutOfBounds (2), NullPointer (17), ArrayIndexOutOfBounds (1) StringIndexOutOfBounds (1)	Critical (2) Medium (14) Medium (14),	[6] [7]
LOG	29, 43, 509, 10528, 10706, 11570, 31003, 40212, 41186, 44032, 44899, 45335, 46144, 46271, 46404, 47547, 47912, 47957	1.0.2 - 1.2	NullPointer (17), InInitializerError (1)	Critical (1) Major (4) Medium (11) Enhanc. (1) Blocker (1)	[6] [7]

IV. EMPIRICAL STUDY

This section describes the empirical study we conducted to benchmark the effectiveness of the EvoCrash approach.

A. Definition and Context

The *context* of this study consists of 50 bugs from three real-world open source projects: `Apache Commons Collections`[3] (ACC), `Apache Ant`[4] (ANT), `Apache Log4j`[5] (LOG). ACC is a popular Java library with 25,000 lines of code (LOC), which provides utilities to extend the Java Collection Framework. For this library we selected 12 bug reports publicly available on `Jira`[6] submitted between October 2003 and June 2012, thus involving five different ACC versions. ANT is a large Java build tool with more than 100,000 LOC, which supports different built-in tasks, including compiling, running and executing tests for Java applications. For ANT we selected 20 bug reports submitted on `Bugzilla`[7] between April 2004 and August 2012 and that concern 10 different versions and sub-modules. Finally, LOG is a widely used Java library with 20,000 LOC that implements logging utilities for Java applications. For this library we selected 18 bug reports reported within the time windows between June 2001 and October 2009 and that are related to three different LOG versions. The characteristics of the selected bugs, including type of exception and priority, are summarized in Table I.

We selected these bugs as they have been used in the previous study on automatic crash reproduction when evaluating symbolic execution [6], mutation analysis [12], and directed model checking [7] and other tools [36], [37]. This

[3]https://commons.apache.org/proper/commons-collections/
[4]http://ant.apache.org
[5]http://logging.apache.org/log4j/2.x/
[6]https://issues.apache.org/jira/secure/Dashboard.jspa
[7]https://bz.apache.org/bugzilla/

selection covers crashes that involve the most common Java `Exceptions` [38], such as `NullPointerException` (77%), `ArrayIndexOutOfBoundsException` (8%), `IllegalStateException` and `IllegalArgument-Exception` (4%). Furthermore, the severity of these real-world bugs varies between *medium* (50%), *major* (36%) and *critical* (6%) as judged by the original developers.

B. Research Questions

To evaluate the effectiveness of EvoCrash we formulate the following research questions:

- **RQ$_1$:** *In which cases can EvoCrash successfully reproduce the targeted crashes, and under what circumstances does it fail to do so?* With this preliminary research question we aim at evaluating the capability of our tool to generate test cases (i) that can replicate the target crashes, and (ii) that are useful for debugging.
- **RQ$_2$:** *How does EvoCrash perform compared to state-of-the-art reproduction approaches based on stack traces?* In this second research question we investigate the advantages of EvoCrash as compared to the best known stack trace approaches previously proposed in the literature.

C. Experimental Procedure

We run EvoCrash on each target crash to try to generate a test case able to reproduce the corresponding stack trace. Given the randomized nature of genetic algorithms, the search for each target bug/crash was repeated 50 times in order to verify that the target crashes are replicated the majority of the time. In our experiment, we configured GGA by using standard parameter values widely used in evolutionary testing [21], [39], [40]:

- **Population size**: for GGA, we initially use a population size of 50 test cases. If the search reaches the timeout (30 minutes), we increment the population size by 25 and run EvoCrash once again until the population size reaches 300. If with population size of 300 EvoCrash cannot create a test case with fitness = 0.0 in 30 minutes, we specify the crash case as non-reproducible.
- **Crossover**: we use the novel *guided single-point* crossover with *crossover probability* set to 0.75 [21].
- **Mutation**: as mutation operator we use our *guided uniform mutation*, which mutates test cases by randomly adding, deleting, or changing statements. We set the *mutation probability* equal to $1/n$, where n is the length of the test case taken as input [21].
- **Search Timeout**: the search stops when a zero fitness function value is detected or when the timeout of 30 minutes is reached [40].

To address **RQ$_1$**, we apply the two criteria proposed by Chen and Kim [6] for evaluating the effectiveness of crash replication tools: *Crash Coverage* and *Test Case Usefulness*. According to the *Crash Coverage* criterion, a crash is covered when the test generated by EvoCrash results in the generation of the same type of exception at the same crash line as reported in the crash stack trace. Therefore, for this criterion we classified as *covered* only those crashes for which EvoSuite reached a fitness value equal to 0.0, i.e., when the generated crash stack trace is identical to the target one. In these cases, we also re-executed the generated tests against the CUT to ensure that the crash stack frame was correctly replicated.

For the *Test Case Usefulness* criterion, a generated test case by EvoCrash is considered useful if it can reveal the actual bug that causes the original crash. Therefore, we manually examined each crash classified as *covered* (using the coverage criterion) to investigate if it can reveal the actual bug following the guidelines in [6]. A test case *reveals a bug* if the generated crash trace includes the buggy frame (i.e., the stack element which the buggy method lies in [6]) or the frame the execution of which covers the buggy component. To assess usefulness of the tests, we carefully inspected the original developers' fixes to identify the bug fixing locations. Finally, *useful* tests have to reveal the origin of the corrupted input values (e.g., `null` values) passed to the buggy methods that trigger the crash [6]. This manual validation has been performed by two authors independently, and cases of disagreement were discussed.

To address **RQ$_2$**, we selected three state-of-the-art techniques, namely: STAR [6], MuCrash [12], and JCHARMING [7], [19]. These three techniques are modern approaches to crash replication for Java programs, and they are based on three different categories of algorithms: symbolic execution [6], mutation analysis [12], and model checking [7].

At the time of this submission, these three tools (either as executable jars or source code) were not available. Therefore, to compare our approach, we rely on their published data. Since the studies use different data sets, we cannot report data points for all subject systems. Thus, we compared EvoCrash with MuCrash for the 12 bugs selected from ACC that have also been used by Xuan et al. [12] to evaluate their tool. We compared EvoCrash with JCHARMING for the 8 bug reports that have been also used by Nayrolles et al. [7]. Finally, we compare EvoCrash with STAR for the 50 bugs in our sample that are in common with the study by Chen and Kim [6].

V. Experimental Results

This section presents the results of the empirical study we conducted to evaluate the effectiveness of EvoCrash in terms of crash coverage and test case usefulness. Moreover, we provide the first comparison results between the effectiveness of EvoCrash, STAR [6], MuCrash [12], and JCHARMING [7], as the state-of-the-art approaches based on crash stack traces.

EvoCrash Results (RQ1) As Table II illustrates, EvoCrash can successfully replicate the majority of the crashes in our dataset. Of the replicated cases, LOG-509 had the lowest rate of replications - 39 out of 50 - and 39 cases could be replicated 50 times out of 50. EvoCrash reproduces 10 crashes out of 12 (83%) for ACC, 14 out of 20 (70%) for ANT, and 17 out of 18 (94%) for LOG. Overall, it can replicate 41 (82%) out of the 50 crashes.

To assess the usefulness of the generated test cases, we used the same criterion that was used for STAR [6]. Based on

TABLE II
DETAILED CRASH REPRODUCTION RESULTS, WHERE **Y(YES)**, INDICATES THE CAPABILITY TO GENERATE A USEFUL TEST CASE, **N(NO)** INDICATES LACK OF ABILITY TO REPRODUCE A CRASH, **NU(NOT USEFUL)** SHOWS THAT A TEST CASE COULD BE GENERATED, BUT IT WAS NOT USEFUL, AND **'-'** INDICATES THAT DATA REGARDING THE CAPABILITY OF THE APPROACH IN REPRODUCING THE IDENTIFIED CRASH IS MISSING.

Project	Bug ID	EvoCrash	STAR [6]	MuCrash [12]	JCHARMING [7]
ACC	4	Y	Y	Y	-
	28	Y	Y	Y	-
	35	Y	Y	Y	-
	48	Y	Y	Y	-
	53	**Y**	Y	**N**	-
	68	N	N	N	-
	70	**Y**	**N**	**N**	-
	77	NU	NU	N	-
	104	N	Y	Y	-
	331	**Y**	**N**	Y	-
	377	**Y**	**N**	Y	-
	411	Y	Y	Y	-
ANT	28820	N	N	-	-
	33446	NU	NU	-	-
	34722	**Y**	**N**	-	-
	34734	**NU**	**N**	-	-
	36733	NU	NU	-	-
	38458	Y	Y	-	-
	38622	NU	Y	-	Y
	42179	**Y**	**N**	-	-
	43292	N	Y	-	-
	44689	**Y**	**NU**	-	-
	44790	Y	Y	-	-
	46747	N	N	-	-
	47306	N	N	-	-
	48715	N	N	-	-
	49137	**Y**	**NU**	-	-
	49755	Y	Y	-	-
	49803	Y	Y	-	-
	50894	**Y**	**NU**	-	-
	51035	N	N	-	-
	53626	**Y**	**N**	-	-
LOG	29	Y	Y	-	-
	43	N	N	-	-
	509	**Y**	**N**	-	-
	10528	**Y**	**N**	-	-
	10706	**Y**	**N**	-	-
	11570	Y	Y	-	Y
	31003	Y	Y	-	-
	40212	**Y**	**NU**	-	Y
	41186	Y	Y	-	Y
	44032	**Y**	**N**	-	-
	44899	**Y**	**N**	-	-
	45335	**Y**	**NU**	-	**N**
	46144	**Y**	**N**	-	-
	46271	NU	Y	-	Y
	46404	**Y**	**N**	-	-
	47547	Y	Y	-	-
	47912	**Y**	**NU**	-	Y
	47957	**NU**	Y	-	**N**

this, 34 (89%) of the replications were useful, as they included buggy frame. The remaining 13% non useful replications were mainly due to having dependency on data from external files which were not available during replication.

For ACC, there were two cases (ACC-68, and ACC-104) which were not reproducible by EvoCrash. For ACC-68, the class under test includes three nested classes, and the inner-most one was where the crash occurs. Currently, EvoSuite does not support instrumentation of multiple inner classes. For ACC-104, EvoCrash could replicate the case 4 times out of 50. This low ratio was due to the fact that calls to the input object and target method had to be made in a certain order to trigger the crash.

For ANT, 6 of the 20 cases (30%) are currently not supported by EvoCrash. For these cases, the major hindering factor was the dependency on a missing external `build.xml` file, which is used by ANT for setting up the project configurations. However, `build.xml` was not supplied for many of the crash reports. In addition, the use of Java reflection made it more challenging to reproduce these ANT cases, since the specific values for class and method names are not known from the crash stack trace.

```
java.lang.NullPointerException:
  at org.apache.tools.ant.util.SymbolicLinkUtils.
      isSymbolicLink(SymbolicLinkUtils.java:107)
  at org.apache.tools.ant.util.SymbolicLinkUtils.
      isSymbolicLink(SymbolicLinkUtils.java:73)
  at org.apache.tools.ant.util.SymbolicLinkUtils.
      deleteSymbolicLink(SymbolicLinkUtils.java:223)
  at org.apache.tools.ant.taskdefs.optional.unix.
      Symlink.delete(Symlink.java:187)
```

Listing 1. Crash Stack Trace for ANT-49137.

```
public void test0()  throws Throwable  {
  Symlink symlink0 = new Symlink();
  symlink0.setLink("");
  symlink0.delete();
}
```

Listing 2. Generated test by EvoCrash for ANT-49137.

For LOG, 1 of the 18 cases (5%) is not supported by EvoCrash. In this case, the target call is made to a static class initializer, which is not supported by EvoCrash yet.

Comparison to the State of the Art (RQ2) Table II shows the comparisons of EvoCrash to STAR, MuCRASH, and JCHARMING. Bold entries represent bugs which can be triggered by EvoCrash, yet one of the other techniques is not; Underlined entries represent bugs that EvoCrash cannot reproduce, while there is another technique that can. As can be seen, there are 22 (bold) cases in which EvoCrash outperforms the state of the art, and there are 2 (underlined) cases that EvoCrash cannot handle. Below we discuss these cases in more detail.

EvoCrash *vs.* STAR. As Table II presents, for ACC, EvoCrash covers all the cases that STAR covers except for ACC-104 (that was reflected on previously). In addition, EvoCrash covers 3 cases (25%) which were not covered by STAR due to the path explosion problem. For instance, in ACC-331, the defect exists in a private method, `least`, inside a for loop, inside the third `if` condition, which was too complicated for STAR. The case was complex to EvoCrash too, since this was one of the cases where we had to increase the population size (from 50 to 175).

For ANT, EvoCrash supports 7 cases (35%) which are not covered by STAR. Out of the 7, there are 3 cases, for which only EvoCrash can generate a useful test case. Listing 1 shows the crash stack trace for of these cases (ANT-49137). As reported in the issue tracking system of the project[8], in this case, the defect exists in the 4th stack frame. Thus, a useful test case should (i) make a call to the method `delete`, (ii) trigger a `java.lang.NullPointerException`, and (iii) yield a crash trace which includes the first stack frame, which is where the exception was thrown. As Listing 2 depicts,

[8]https://bz.apache.org/bugzilla/show_bug.cgi?id=49137

```
public void test0() throws Throwable {
    java.io.File v1 = (java.io.File) null;
    org.apache.tools.ant.util.SymbolicLinkUtils v2 =
         org.apache.tools.ant.util.SymbolicLinkUtils
        .getSymbolicLinkUtils();
    v2.isSymbolicLink((java.io.File) v1, (java.lang.
        String) null);
}
```

Listing 3. Generated test by STAR for ANT-49137.

```
java.lang.NullPointerException
  at org.apache.tools.ant.util.SymbolicLinkUtils.
      isSymbolicLink(SymbolicLinkUtils.java:107)
```

Listing 4. Generated Crash Stack Trace by STAR for ANT-49137.

the test case by EvoCrash creates an instance of `Symlink`, `symlink0`, adapts the state in `symlink0`, and ultimately makes a call to `delete`, which will result in generating the target crash stack trace with fitness equal to 0.0. On the other hand, as Listing 3 shows, the test case by STAR, makes an instance of `SymbolicLinkUtils`, which comes before the defective frame in the crash stack, and makes a call to the root method, `isSymbolicLink`. Consequently, only part of the target crash stack is generated by this test, which is shown in Listing 4. Since the defective frame is not revealed in the resulting crash trace, even though the root frame is covered, the test by STAR does not evaluate to useful according to the criteria set by STAR [6].

Other than ACC-104, ANT-43292 is the other case that is only reproducible by STAR. The main reason for this lies in an inheritance-related problem and how the current fitness function compares stack frames. In this case, the target method, `mapFileName`, is defined in `FilterMapper`, which extends `FileNameMapper`. However, the search can find better fitness values, using other subclasses of `FileNameMapper`, such as `FlatFileNameMapper`, because the implementation of `mapFileName` in these subclasses has lower complexity. To improve this case in the future, we plan to increase the strictness of the fitness function when it comes to distinguishing among subclasses and their inherited methods.

For LOG, EvoCrash covers all the cases that were covered by STAR. 6 of the LOG cases (33%) are only covered by EvoCrash. As an example, LOG-509 is among the cases which are only covered by EvoCrash. In this case, there is a need to interact with the file system, and in order to handle the interaction with the environment, EvoCrash benefits from the mocking mechanisms implemented in EvoSuite.

EvoCrash *vs.* MuCrash. As Table II shows, evaluation data for MuCrash is only available for ACC.[9] Except for ACC-104, EvoCrash covers all the ACC-cases that are covered by MuCrash. In addition, 3 cases (25%) are only covered by EvoCrash, though one of them is not marked as useful.

An example of a covered case is ACC-53, depicted in Listing 5. It requires that an object is added to an instance

[9]Since MuCrash is not publicly available we could not reproduce the data or add additional cases by ourselves.

```
java.lang.ArrayIndexOutOfBoundsException:
  at org.apache.commons.collections.buffer.
      UnboundedFifoBuffer$1.remove(
      UnboundedFifoBuffer.java:312)
```

Listing 5. Crash Stack Trace for ACC-53

```
Object object0 = new Object();
UnboundedFifoBuffer unboundedFifoBuffer0 = new
    UnboundedFifoBuffer();
unboundedFifoBuffer0.add(object0);
unboundedFifoBuffer0.tail = 82;
unboundedFifoBuffer0.remove((Object) null);
```

Listing 6. EvoCrash test for ACC-53

of `UnboundedFifoBuffer`, the `tail` index is set to a number larger than the buffer size, and then that the method `remove` is invoked. In addition, the order in which the methods are invoked matters. So, if the `tail` index would be set after `remove` is called, the target crash would not be replicated. As shown in Listing 6, EvoCrash synthetized the right method sequence and reproduced ACC-53.

EvoCrash *vs.* JCHARMING. As Table II shows, only few cases from ANT and LOG were shared with the cases used to evaluate JCHARMING. While 75% of the shared cases are covered both by EvoCrash and JCHARMING, there is substantial difference in the efficiency of the two approaches. On average, EvoCrash takes less than 2 minutes to cover the target crashes, whereas JCHARMING may take from 10 to 38 minutes to generate tests for the same cases.

For LOG-41186, 2 LOG cases out of 7 (29%) are only supported by EvoCrash. As an example, Listing 7 shows the crash stack trace for LOG-45335, which is one of the two cases covered only by EvoCrash. To generate a useful test for LOG-45335, as depicted in Listing 8, EvoCrash sets the `ht` state in `NDC` to `null`, and then makes a call to the static method `remove`, which is the buggy frame method.

VI. Discussion

We identify two possible directions for future work.

Interactive Search. It should be noted that since GGA strives for finding the fittest test case, thus discarding the ones with fitness $\neq 0.0$, the crash coverage and usefulness evaluation was performed on a set of EvoCrash tests with fitness equal to 0.0. However, considering the crash exploitability and usefulness criteria adopted from STAR [6], it could be possible that EvoCrash discarded tests with fitness between 0.0 and 1.0,

```
java.lang.NullPointerException:
  at org.apache.log4jb.NDC.remove(NDC.java:377)
```

Listing 7. Crash Stack Trace for LOG-45335.

```
public void test0()  throws Throwable  {
     NDC.ht = null;
     NDC.remove();
 }
```

Listing 8. The EvoCrash Test for LOG-45335.

which would actually conform to the aforementioned criteria. Considering the fitness function range, fitness values could be from 0.0 to 6.0, where 6.0 means a test case that does not reach the target line, therefore does not invoke the target method, and in turn, does not trigger the target exception. In contrast, fitness 0.0 means that the test covers the target line and method, and triggers the target exception. According to the definition of the fitness function (presented in Section III), when the fitness value is between 0.0 and 1.0, the target line and exception are covered, however, the stack trace similarity is not ideal yet. In this case, even though the target stack similarity is not achieved, crash coverage and test usefulness criteria could be covered. As the result, future work can provide interactive mechanisms through which the precision of the fitness function could be adjusted, so tests with fitness between 0.0 and 1.0 could also be accepted.

In addition, dependency on external files was a major factor that prevented EvoCrash from covering more cases. As described earlier, for some of the cases with environmental dependency, we increased the population size, which in turn led to successful generation of tests for some of the cases. Thus, if external files were to be provided by the bug reporters, then enabling developers to specify the external files, or adjust the population size through interactive mechanisms could be another possible direction for the future work.

Extending Comparisons. While to make the comparison among EvoCrash, STAR, MuCrash, and JCHARMING, we had to identify a subset of cases shared in the empirical evaluations of the techniques, we realize the need to extend the comparison between (i) EvoCrash and JCHARMING, and (ii) EvoCrash and MuCrash. To improve the comparison with JCHARMING, we would adopt the other projects that were targeted by JCHARMING, and evaluate EvoCrash against the identified cases for them. Considering the substantial performance difference between EvoCrash and JCHARMING, we also wish to statistically compare the efficiency of the tools. To do so, we would rely on availability of JCHARMING for experimentation. To improve the comparison with MuCrash, if additional evaluation data is published for the tool, or MuCrash becomes publicly available, we would extend the empirical study to increase the validity of the comparison results.

VII. Threats to Validity

With respect to external validity, the main threats arise from the focus on Java and open source. The use of Java is needed for our experiments due to the dependency on EvoSuite, yet we expect our approach to behave similarly on other languages such as Ruby or C#.

To maximize reproducibility and to enable comparison with the state of the art we rely on open source Java systems. We see no reason why closed source stack traces would be substantially different. As part of our future work we will engage with one of our industrial partners, mining their log files for frequent stack traces. This will help them create test cases that they can add to their test suite to reproduce and fix errors their software suffers from.

In order to facilitate comparison with earlier approaches we selected bugs and system versions that have been used in earlier studies, and hence are several years old. We anticipate that our approach works equally well on more recent bugs or versions as well, but have not conducted such experiments yet.

A finding of our experiments is that a key limiting factor for any stack-trace based approach is the unavailability of external data that may be needed for the reproduction. Further research is needed to (1) mitigate this limitation; and (2) identify a different data set of crashes focusing on such missing data, in order to further narrow down this problem.

With respect to internal validity, a key threat is in the evaluation of the crash coverage and usefulness of the generated test cases. In case EvoCrash generated a test with fitness = 0.0, we double checked the generated crash stack trace to ensure that the corresponding test correctly replicated the crash stack frame. Despite having taken the above procedures, it is still possible that we made errors in the inspections and evaluations. To mitigate the chances of introducing errors, we peer reviewed the tests and crashes. In addition, we make the EvoCrash tool, and the generated test cases publicly available [1] for further evaluations.

VIII. Conclusion

To increase developers' productivity while debugging, several approaches to automated crash replication have been proposed. However, the existing solutions have certain limitations that adversely affect their capability in covering more crash cases for real-world software projects. This paper presents EvoCrash, which is a search-based approach to crash replication based on using data from crash stack traces. EvoCrash applies a novel Guided Genetic Algorithm (GGA) as well as a smart fitness function, to search for a test case that can trigger the target crash and reveal the buggy frame in the crash stack trace. Our empirical evaluation shows that EvoCrash addresses the major challenges that were faced by three cutting-edge approaches, and thereby, outperforms them in automated crash reproduction.

The future work may take several directions, including: (i) enhancing the fitness function implemented in EvoCrash, (ii) extending the comparison between EvoCrash and the other techniques, which considerably would depend on the availability of the tools, and (iii), evaluating EvoCrash for industrial projects.

The implementation of EvoCrash, as well as the experimental data are publicly available[1].

Acknowledgment

This research was partially funded by the EU Project STAMP ICT-16-10 No.731529, the Dutch 4TU project "Big Software on the Run" and National Research Fund, Luxembourg FNR/P10/03.

References

[1] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *ECOOP 2008–Object-Oriented Programming*. Springer, 2008, pp. 542–565.

[2] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 261–270.

[3] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 284–295.

[4] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*. ACM, 2005, pp. 114–123.

[5] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, *jRapture: A capture/replay tool for observation-based testing*. ACM, 2000, vol. 25, no. 5.

[6] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE Tr. on Sw. Eng.*, vol. 41, no. 2, pp. 198–220, 2015.

[7] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 101–110.

[8] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, "Reconstructing core dumps," in *2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation*. IEEE, 2013, pp. 114–123.

[9] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, "Contract driven development= test driven development-writing test cases," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 425–434.

[10] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol, "On the effectiveness of test extraction without overhead," in *International Conference on Software Testing Verification and Validation (ICST)*. IEEE, 2009, pp. 416–425.

[11] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 321–334.

[12] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *ESEC/FSE*. ACM, 2015, pp. 910–913. [Online]. Available: http://doi.acm.org/10.1145/2786805.2803206

[13] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing multicore dumps to facilitate concurrency bug reproduction," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 155–166, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1735970.1736039

[14] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, "Sbfr: A search based approach for reproducing failures of programs with grammar based input," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 604–609.

[15] F. Kifetew, W. Jin, R. Tiellam, A. Orso, and P. Tonella, "Reproducing field failures for programs with complex grammar-based input," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, March 2014, pp. 163–172.

[16] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 123–133. [Online]. Available: http://doi.acm.org/10.1145/566172.566191

[17] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[18] W. Jin and A. Orso, "Bugredux: reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 474–484.

[19] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "A bug reproduction approach based on directed model checking and crash traces," *Journal of Software: Evolution and Process*, pp. n/a–n/a, 2016, jSME-15-0137.R1. [Online]. Available: http://dx.doi.org/10.1002/smr.1789

[20] M. Soltani, A. Panichella, and A. van Deursen, "Evolutionary testing for crash reproduction," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 1–4.

[21] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.14

[22] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, "Precise identification of problems for structural test generation," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 611–620.

[23] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[24] I. S. W. B. Prasetya, *T3, a Combinator-Based Random Testing Tool for Java: Benchmarking*. Cham: Springer International Publishing, 2014, pp. 101–110. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07785-7_7"

[25] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[26] N. Tillmann and J. De Halleux, "Pex: White box test generation for .net," in *Proceedings of the 2Nd International Conference on Tests and Proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792786.1792798

[27] A. Sakti, G. Pesant, and Y. G. Guhneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, March 2015.

[28] G. Fraser and A. Arcuri, "1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite," *Empirical Software Engineering*, vol. 20, no. 3, pp. 611–639, 2013.

[29] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 33–44.

[30] M. Harman, P. McMinn, J. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*. Springer, 2012, pp. 1–59.

[31] M. Harman, S. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, p. 11, 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379787

[32] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[33] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia, "Improving multi-objective test case selection by injecting diversity in genetic algorithms," *IEEE Trans. Software Eng.*, vol. 41, no. 4, pp. 358–383, 2015. [Online]. Available: http://dx.doi.org/10.1109/TSE.2014.2364175

[34] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, pp. 352–361.

[35] G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 362–369.

[36] H. Cibulski and A. Yehudai, "Regression test selection techniques for test-driven development," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 115–124.

[37] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: Object Capture-based Automated Testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 159–170. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831729

[38] R. Coelho, L. Almeida, G. Gousios, A. v. Deursen, and C. Treude, "Exception handling bug hazards in android," *Empirical Software Engineering*, pp. 1–41, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-016-9443-7

[39] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[40] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

TUD-SERG-2017-006
ISSN 1872-5392