Exploring means to facilitate software debugging

SOLTANI, M.S.

2 Evolutionary Crash Reproduction

Software systems fail. These failures are often reported to issue tracking systems, where they are prioritized and assigned to responsible developers to be investigated. When developers debug software, they need to reproduce the reported failure in order to verify whether their fix actually prevents the failure from happening again. Since manually reproducing each failure could be a complex task, several automated techniques have been proposed to tackle this problem. Despite showing advancements in this area, the proposed techniques showed various types of limitations. In this paper, we present EvoCrash, a new approach to automated crash reproduction based on a novel evolutionary algorithm, called Guided Genetic Algorithm (GGA). We report on our empirical study on using EvoCrash to reproduce 54 real-world crashes, as well as the results of a controlled experiment, involving human participants, to assess the impact of EvoCrash tests in debugging. Based on our results, EvoCrash outperforms state-of-the-art techniques in crash reproduction and uncovers failures that are undetected by classical coverage-based unit test generation tools. In addition, we observed that using EvoCrash helps developers provide fixes more often and take less time when debugging, compared to developers debugging and fixing code without using EvoCrash tests.

2.1 Introduction

Despite the significant effort spent by developers in software testing and verification, software systems still fail. These failures are reported to issue tracking systems, where

they are prioritized, and assigned to responsible developers for inspection. When developers debug software, they need to reproduce the reported failure, understand its root cause, and provide a proper fix that prevents the failure. While crash stack traces indicate the type of crash and the method calls executed at the time of the crash, they may lack critical details that a developer could use to debug the software. Therefore, depending on the complexity of the reported failures and amount of available information about them, manual crash reproduction can be a labor-intensive task which negatively affects developers' productivity.

To reduce debugging effort, researchers have proposed various automated techniques to generate test cases reproducing the target crashes. Generated tests can help developers better understanding the cause of the crash by providing the input values that actually induce the failure and enable the usage of a debugger in the IDE with runtime data. To generate such tests, crash reproduction techniques leverage various sources of information, such as stack traces, core dumps, failure descriptions. As Chen and Kim [81] first identified, these techniques can be classified into two categories: record-replay techniques, and post-failure techniques. Record-replay approaches [**?**, **?**, 58, 169, 205] 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 [81], and privacy issues [171].

As opposed to these costly techniques, *post-failure* approaches [81, 150, 151, 171, 194, 215, 219] 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 [81], such as core dumps [150, 151, 194, 208] or software models like input grammars [136, 137] or class invariants [69].

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 [81, 171, 215]. For example, Chen and Kim developed STAR [81], an approach based on backward symbolic execution that outperforms earlier crash replication techniques, such as Randoop [177] and BugRedux [134]. Xuan et al. [215] presented MuCrash, a tool that mutates existing test cases using specific operators, thus creating a new pool of tests to run against the software under analysis. Nayrolle et al. [171] proposed JCHARMING, based on directed model checking combined with program slicing [171, 172].

Unfortunately, the state-of-the-art tools suffer from several limitations. For example, STAR cannot handle cases with external environmental dependencies [81] (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 [215]. JCHARMING [171,172] applies model checking which can be computationally expensive. Moreover, similar to STAR, JCHARMING does not handle crash cases with environmental dependencies.

This paper is an extension of our previous conference paper [203], where we presented EvoCrash, a search-based approach for the automated crash replication problem and built on top of EvoSuite [103], which is a well-known coverage-based unit test generator for Java code. Specifically, EvoCrash uses a novel evolutionary algorithm, namely Guided Genetic Algorithm (GGA), which leverages the stack trace to guide the search toward generating tests able to trigger the target crashes. GGA uses a generative routine to build an initial population of test cases, which exercise at least one of the methods reported in the crash stack frames (target methods). GGA also uses two novel genetic operators, i.e., namely *guided crossover* and *guided mutation*, to ensure that the test cases keep exercising the target methods across the generations. The search is further guided by a fitness function that combines coverage-based heuristics with a crash-based heuristic measuring the distance between the stack traces (if any) generated by the candidate test cases and the original stack trace of the crash to replicate.

We assess the performance of EvoCrash by conducting an empirical study on 54 crashes reported for real-world open-source Java projects. Our results show that EvoCrash can successfully replicate more crashes than STAR (+23%), MuCrash (+17%), and JCHARMING (+25%), which are the state-of-the-art tools based on crash stack traces. Furthermore, we observe that EvoCrash is not affected by the path explosion problem, which is a key problem for symbolic execution [81], and can mock environmental interactions which, in some cases, helps to cope with the environmental dependency problem.

Furthermore, we compare EvoCrash with EvoSuite to assess whether the crash replicated by our tools could be simply detected by classical coverage-based test case generators. The results of this comparison show that EvoCrash reproduced 85% of the crashes, while EvoSuite reproduced only 33% of them. For crashes reproduced by both EvoCrash and EvoSuite, on average, EvoCrash took 145 seconds while EvoSuite took 391 seconds. Thus, on average, EvoCrash is 170% more efficient than EvoSuite when they both reproduce crashes. These results show that coverage-based test generation lacks adequate guidance for crash reproduction. This in turn confirms the need for specialized search when the goal is to trigger specific software behavior rather than achieving high code coverage.

We also assess the extent of practical usefulness of the tests generated by EvoCrash during debugging and code fixing tasks. To this aim, we conducted a controlled experiment with 35 master students in computer science. The achieved results reveal that tests generated by EvoCrash increase participants' ability to provide fixes (+21% on average) while reducing the amount of time they spent to complete the assigned tasks (-15.36% on average).

The novel contributions of this extension are summarized as follows:

- A comparison of EvoCrash with EvoSuite, which is a test generation tool for coverage-based unit testing.
- A controlled experiment involving human participants; its results show that the usage of the tests aids developers in fixing the reported bugs while taking less time when debugging.
- We provide a publicly available replication package[1] that includes: (i) an executable jar of EvoCrash, (ii) all bug reports used in our study, (iii) the test cases generated by our tool, and (iv) anonymized experimental data as well as R scripts used to analyze the results from the controlled experiment.

The remainder of the chapter is structured as follows. Section 2.2 provides background on search-based software testing, in addition to describing the related work on the approaches to automated crash replication, unit test generation tools, and user studies in testing and debugging. Section 2.3 presents the EvoCrash approach. Section 2.4 and 2.5 describe the empirical evaluation of EvoCrash as well as the controlled experiment with human participants, respectively. Discussion follows in Section 2.6. Section 2.7 concludes the paper.

2.2 Background and Related Work

In this section, we present related work on automated crash reproduction, background knowledge on search-based software testing,related work in software testing and debugging which conducted experiments involving human participants.

2.2.1 Automated Approaches to Crash Replication

Previous approaches in the field of crash replication can be grouped into three main categories: (i) *record-replay* approaches, (ii) *post-failure* approaches using various

[1] **DOI**: 10.4121/uuid:001bb128-0a55-4a8d-b3f5-e39bfc5795ea

data sources, and (iii) *stack-trace based post-failure* techniques. The first category includes the earliest work in this field, such as ReCrash [58], ADDA [**?**], Bugnet [169], and jRapture [205]. In addition, [64] and [76] are recent record-replay techniques which are based on monitoring non-deterministic and hard-to-resolve methods (when using symbolic execution) respectively. The recent work on reproducing context-sensitive crashes of Android applications, MoTiF [114], also falls in the first category of record-replay techniques. The aforementioned techniques rely on program runtime 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 [81,171,194], and (ii) privacy violations since the collected execution data may contain sensitive information [81].

On the other hand, post-failure approaches [137, 150, 151, 194, 217, 219] analyze software data (e.g., core dumps) only after crashes occur, thus not requiring any form of instrumentation. Rossler et al. [194] developed an evolutionary search-based approach named RECORE that leverages 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. [208] used core dumps and directed search for replicating crashes related to concurrent programs in multi-core platforms. Leitner et al. [150, 151] 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. [136, 137] used genetic programming requiring as input (i) a grammar describing the program input, and (ii) a (partial) call sequence. Boyapati et al. [69] developed another technique requiring manually written specifications containing method preconditions, postconditions, and class invariants. However, the above mentioned *post-failure* approaches need various types of information that are often not available to developers, thus decreasing their feasibility. To address lack of available execution data for replicating system-level concurrency crashes, Yu et al. [217] propose a new approach called, DESCRY. DESCRY only assumes the existence of the source code of processes under debugging and default logs generated by the failed execution. This approach [217] leverages a combination of static and dynamic analysis techniques and symbolic execution to synthesize the failure-inducing input data and interleaving schedule.

To increase the practical usefulness of automated approaches, researchers have focused on crash stack traces as the only source of information available for debugging. For instance, ESD [219] uses forward symbolic execution that leverages commonly reported elements in bug reports. BugRedux [134] also uses forward symbolic execu-

tion but it can analyze different types of execution data, such as crash stack traces. As highlighted by Chen and Kim [81], both ESD and BugRedux rely on forward symbolic execution, thus inheriting its problems due to *path explosion* and *object creation* [214]. As shown by Braione et al. [70], existing symbolic execution tools do not adequately address the synthesis of complex input data structures that require non-trivial method sequences. To address the path explosion and object creation problems, Chen and Kim [81] introduced STAR, a tool that applies backward symbolic execution to compute crash preconditions and generates a test using a method sequence composition approach. Despite these advances in STAR, Chen and Kim [81] reported that their approach is still affected by the path explosion problem when replicating some crashes. Therefore, path-explosion still remains an open issue for symbolic execution.

Different from STAR, JCHARMING [171, 172] uses a combination of crash traces and model checking to automatically reproduce bugs that caused field failure. To address the state explosion problem [60] in model checking, JCHARMING applies program slicing to direct the model checking process by reduction of the search space. Instead, MuCrash [215] 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.

STAR [81], JCHARMING [171, 172], and MuCrash [215], have been empirically evaluated on a varying number of field crashes (52, 12, and 31, respectively) which were reported for different open source projects, including: Apache Commons Collections, Apache Ant, Apache Hadoop, Dnsjava, etc. The results of the evaluations are reported in the published papers, however, to the best of our knowledge, the tools are not publicly available.

A recent approach based on using crash stacks for reproducing concurrency failures, that violate thread safety of a class, is CONCRASH, proposed by Bianchi et al. [65]. As input, CONCRASH requires the class that violates thread safety and the generated crash stack trace. CONCRASH iteratively applies pruning strategies to search for test code and interleaving that trigger the target concurrency failure. Differently from our approach, CONCRASH targets only concurrency failures violating the thread-safety of a program [65], which represents the minority of failures reported in issue tracking systems [218]. For example, Yuan et al. [218] reported that only 10% of the failures in distributed data-intensive systems are due to multi-threaded inter-leavings. A later study by Coelho et al. [85] further reported that a large majority of failures in android apps are related to errors in programming logic and resource management, while concurrency accounts only for 2.9% of all failures.

In our earlier study [201], we investigated coverage-based unit testing tools like EvoSuite as a technology for replicating some 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 uses evolutionary search to synthesize a crash reproducing test case. However, it is novel because it utilizes a more effective fitness function and it applies a Guided Genetic Algorithm (GGA) instead of coverage-oriented genetic algorithms. Section 2.3 presents full details regarding the novel fitness function and the GGA in EvoCrash.

2.2.2 Search-based Software Testing

Search-Based Software Testing (SBST) is a sub-field of a larger body of work on Search-Based Software Engineering (SBSE). In SBSE, software engineering tasks are reformulated as optimization problems, to which different meta-heuristic algorithms are applied to automate them [122]. As McMinn describes [161], search optimizations have been used in a plethora of software testing problems, including structural testing [209], temporal testing [187], functional testing [73], and mutation testing [132]. Among these, structural testing has received the most attention so far.

Applying an SBST technique on a testing problem requires [117,161]: (i) a representation for the candidate solutions in the search space, and (ii) a definition for a fitness function. The *representation* of the solutions shall constitute elements which make it possible to encode them using some data structures [122] (e.g., vectors, trees). This is mainly because search optimization techniques rely on operators that manipulate the encoded elements to derive new solutions. In addition, the representation shall be accurate enough so that a small change in one individual solution represents a neighbor solution in the search space [122].

A *fitness function* (also called objective or distance function) is used to measure the distance of each individual in the search space from the global optimum. Therefore, it is important that this definition is computationally inexpensive so that it could be used to measure the distance of multiple individuals until the global optimum is found [122].

Furthermore, as described before, path explosion and object creation are open problems when using symbolic execution [70] [81]. Different from symbolic execution, search-based software testing uses *distance functions* to satisfy each condition of the

program in "isolation" [59], i.e., independently from which alternative path is taken to reach the condition to solve. Focusing on each condition at a time allows to address the path explosion problem but, on the other hand, it may fail to capture dependencies between multiple conditions in the programs as in the case of *deceiving* conditions [160]. Search-based approaches can be implemented to handle complex input data type by relying on the APIs of the SUT. Indeed, random sampling is used to create randomized tests containing object references through the invocation of constructors and randomly generated method sequences. The "quality" of the generated test input data is then assessed through test execution and measuring the distance to satisfy a given branch. The complexity of the input is then evolved depending on whether more complex data structures help or not satisfying the testing criterion.

Moreover, with regards to environmental interactions, Arcuri et al. [54] show that such interactions may inhibit the success of automated test generation. This is mainly due to two reasons: (i) the code that depends on the environment may not be fully covered, and (ii) the generated tests may be unstable. Arcuri et al. [54] showed that proper instrumentation in a search-based test generator can be used not only to synthesize the test inputs during the search process but also to control the environmental state. More specifically, *mocking strategy* can be used to isolate the execution of a class from its environmental dependencies.

Finally, meta-heuristics that have been used in SBST include hill climbing, simulated annealing, genetic algorithms, and memetic algorithms. The first two algorithms fall in the category of *local* search techniques since they evaluate single points in the search space at the time [122]. On the other hand, genetic algorithms are *global* search techniques since they evaluate a population of candidate solutions from the search space in various iterations [122]. Memetic algorithms hybridize the local and global algorithms. Therefore, in these techniques, the individuals of populations in a global search are also provided with the opportunity for local improvements [106]. Since genetic algorithms have been widely applied to software testing problems, in what follows, we provide a brief description of a classic genetic algorithm.

2.2.2.1 Genetic Algorithms

Genetic Algorithms (GAs) imitate evolutionary processes observed in nature. A GA starts by initializing a random population of individuals. When applied to test generation problems, individuals are typically test suites comprised of test cases [103], or test cases consisting of a sequence of statements [178]. After the first population is initialized, tests are executed against the program under test and the best ones are selected to form new individuals. This process continues until either an individual that

satisfies the search criterion is found, or the allocated resources to the search process are consumed.

To produce the next generation, the best individuals from the previous generation (*parents*) are selected (*elitism*) and used to generate new test cases (*offspring*). Offspring is produced by applying typical evolutionary operators, namely crossover and mutation, to the selected "fittest" individuals. Depending on whether the parent or the offspring scores better for the search criterion, one is selected to be inserted into the next generation.

To illustrate the evolutionary operators, let us consider as examples two test cases $T_1 = \{s_1, \ \ldots, s_m\}$ and $T_2 = \left\{s_1^*, \ \ldots, s_n^*\right\}$ selected from a given generation as parents. To generate offspring O_1 and O_2, first a random number α, called *the relative cut-point*, between 0.0 and 1.0 is selected. Then, the first offspring O_1 will contain the first $\alpha \times m$ statements from T_1 followed by the last $(1 - \alpha) \times n$ statements from T_2. Similarly, O_2 will contain the first $\alpha \times n$ statements from T_2 followed by $(1 - \alpha) \times m$ statements from T_1. Thus, each offspring inherits its statements (e.g., objects instantiations, methods calls) from both the two parents.

Newly generated test cases are further changed by applying a mutation operator. With mutation, either random new statements are inserted into the tests, or random existing statements are removed, or random input parameters are modified [178]. Both crossover and mutation are performed such that the resulting test cases will be compilable. For example, if a new object is inserted as a parameter, then before it is inserted it is declared and instantiated.

2.2.3 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 [18, 103, 112, 156, 159, 177, 184, 196, 206]. The main difference among them is represented by the core approach used for generating tests. For example, EvoSuite [103], JTExpert [196], and SAPIENZ [159] use genetic algorithms to create test suites optimizing code coverage; Randoop [177], T3 [184], Dynodroid [156], and Google Monkey [18] apply random testing, while DART [112] and Pex [206] 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 [101] successfully used EvoSuite to discover undeclared exceptions and bugs

in open-source projects. Recently, Moran et al. [165] used coverage-based tools to discover android application crashes. However, as also pointed out by Chen and Kim [81] 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 [81].

On the other hand, for crash replication, not all methods should be exploited for generating a crash: we are interested in covering only a few lines in those methods involved in the 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 design and evaluate a tool-supported approach, named EvoCrash, which is specialized for stack trace based crash replication.

2.2.4 User Studies in Testing and Debugging

In 2005, Sjøberg et al. [200] conducted a survey in which they studied how controlled experiments are conducted in software engineering, in the decade from 1993 to 2002. As they report, 1.9% of the 5453 scientific articles reported controlled experiments in which human participants performed one or more software engineering tasks. Later on, in 2011, Buse et al. [74] surveyed over 3000 papers, spanning over ten years, to investigate trends, benefits, and barriers of involving human participants in software engineering research. As Buse et al. [74] report, about 10% of the surveyed papers involved humans to evaluate a research claim directly. As they observed, the number of papers in software engineering which use human evaluations is increasing, however, they highlighted that papers specifically related to software testing and debugging rarely involved human studies.

In the area of software testing, Orso and Rothermel [176] conducted a survey among 50 software testing scholars, to provide an account of the most successful research in software testing, since the year 2000. In addition, they aimed at identifying the most significant challenges and opportunities in the area. Orso and Rothermel [176] argue that while prominent advances have been made in empirical studies on software testing, more user studies, in particular within an industrial context, are needed in which practical impact of research becomes apparent. Ang et al. [44] recently stud-

ied the progress that is made in the research community since 2011 to address the suggestions given by Orso and Rothermel [176]. As their study indicates, involving human evaluations in studies on automated debugging techniques remains mostly unexplored.

Recently, some research work in software testing and debugging started involving user evaluations include the following: [181], [188], [79], [108], [191], [109], and [180]. Parnin and Orso [181] performed a preliminary study with 34 developers to investigate whether and to what extent using an automated debugging approach may aid developers in their debugging tasks. In their results, Parnin and Orso [181] show that several assumptions made by automated debugging techniques (e.g., examining isolated statements is enough to understand the bug and fix it) do not hold in practice. Moreover, Parnin and Orso [181] also encourage the researchers to involve developers in their studies to understand how richer information such as test cases and slices may make debugging aids more usable in practice.

Ramler et al. [188] compared tool-supported random test generation and manual testing, involving 48 master students. Their findings are twofold: (i) the number of detected defects by randomly generated test cases is in the range of manual testing, and (ii) randomly generated test cases detect different defects than manually-written unit tests.

Ceccato et al. [79] performed two controlled experiments with human participants to investigate the impact of using automatically generated test cases in debugging. They show that using automatically generated test cases has a positive impact on the accuracy and efficiency of developers working on fault localization and bug fixing tasks. Furthermore, Fraser et al. [108], and [109] conducted controlled experiments with human participants to investigate whether automatically generated unit test cases aid testers in code coverage and finding faults. In their experiments, they provided JavaDocs to the participants and asked them to both produce implementations and test suites. Their results confirmed that while automatically generated test cases, designed for high coverage, do not help testers find bugs, they do aid in achieving high coverage when compared to the ones produced by human participants.

In addition, Rojas et al. [191] combined a controlled experiment with 41 students with five think-aloud observations to assess the impact of using the automated test generation tool, *EvoSuite*, in software development. Their results confirmed that using the tool leads to an average branch coverage increase of 13%, and 36% less time spent on testing, compared to when developers write tests manually. The results from their think-aloud observations with professional programmers confirmed the necessity to (i) increase the usability of the tool, (ii) integrate it better during development, and

(iii) educate developers on how to best use the tool during development.

To improve the comprehensibility of test cases which in turn could improve the number of faults found by developers, Panichella et al. [180] proposed *TestDescriber* which automatically generates summaries of the portions of the code that is exercised by individual test cases. To assess the impact of their approach, Panichella et al. [180] performed a controlled experiment with 33 human participants comprising of professional developers, senior researchers, and students. The results of their study show that using TestDescriber, (i) developers find twice as many bugs, and (ii) test case summaries improve the comprehensibility of test cases which were considered useful by developers.

To investigate and understand the practical usefulness of automatically generated crash-reproducing tests, we acknowledge the need for involving human practitioners in our line of research. Therefore, as the first step in this direction, we conducted a controlled experiment (described in Section 2.5) with master students in computer science to assess the impact of using the crash-reproducing unit tests generated by EvoCrash when performing debugging tasks.

2.3 The EvoCrash Approach

In the following, we present the Guided Genetic Algorithm (GGA) and the fitness function we designed in our search-based approach to automated crash reproduction.

Figure 2.1 shows the main steps of EvoCrash. EvoCrash begins by pre-processing a crash stack trace log in order to formulate the target crash to be reproduced. Next, EvoCrash applies a Guided Genetic Algorithm (GGA) in order to search for a test case that triggers the same crash. The search is over either when the test is found or when the search budget is over. If a crash reproducing test case is found, it goes through post-processing, a phase where the generated test is minimized and transformed into an executable JUnit test. In what follows, we elaborate on each of the above phases in more detail.

2.3.1 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 similar 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

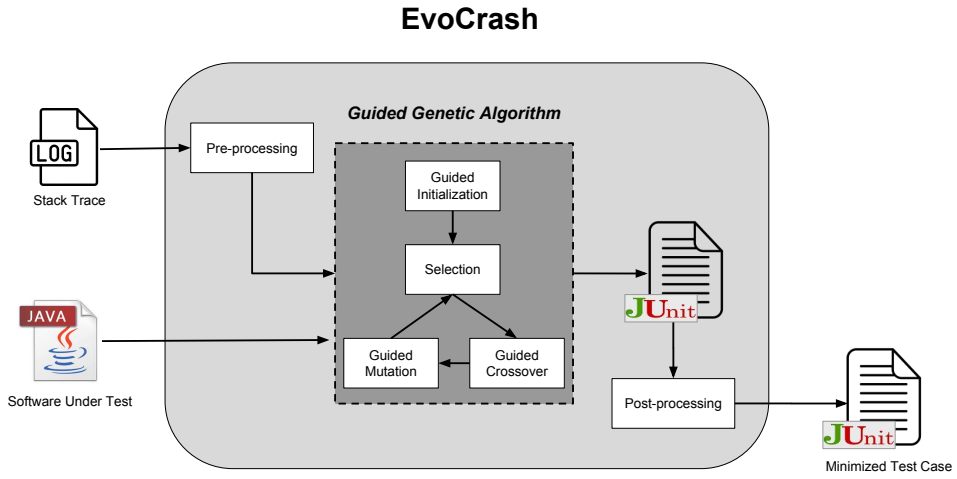

Figure 2.1: Overview of The Guided Genetic Algorithm in EvoCrash

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 and contains: (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 similar 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).

2.3.2 Fitness Function

In search-based software testing, the fitness function is typically a *distance function* $d(.)$, which is equal to zero if and only if the a test case satisfying a given criterion is found. As described in our previous study [201], we have to consider three main conditions in the definition of our distance for crash replication: 1. the line (statement) where the exception is thrown has to be covered, 2. the target exception has to be thrown, and 3. the generated stack trace must be as similar to the original one as possible.

Therefore, we first define three different distance functions for the three conditions above, one for each condition. Then, we combine these three distances into our final fitness function using the sum-scalarization approach. The three distance functions as well as the final one are described in details in the following subsections.

Line distance. A test case t that successfully replicates a target crash has to cover the line of the production code where the exception was originally thrown. To guide the search toward covering the target line, we need to define a distance function $d_s(t)$ for line coverage. To this aim, we use two heuristics that have been successfully used in white-box testing for branch and statement coverage [160,201]: the *approach level* and the normalized *branch distance*. The *approach level* measures the distance in the control flow graph (i.e., the minimum number of control dependencies) between the path of the production code executed by t and the target line. The *branch distance* uses a set of well-established rules [160] to score how close t is to satisfy the conditional expression where the execution diverges from the paths to the target line.

Exception distance. The exception distance is used to check whether the test case t triggers the correct exception. Hence, we define the exception distance d_{except} as a boolean function that takes a zero value if and only if the target exception is thrown; otherwise, d_{except} is set to one.

Trace distance. Several stack trace similarity metrics have been defined in the related literature [90], although for different software engineering problems. These metrics could be in theory used to define our trace distance. Dang et al. [57, 90] proposed a stack trace similarity to clusterize duplicated bug reports. Their similarity metric uses dynamic programming to find the *longest common subsequence* (i.e., sequence of stack frames) among a pool of stack traces. The clusters are then obtained by applying a supervised hierarchical clustering algorithm [90]. However, this similarity metric requires a pool of stack traces plus a training algorithm to decide whether two stack traces are related to the same crash. Artzi et al. [57] proposed some similarity metrics to improve fault localization by leveraging concolic testing. Their intuition is that fault localization becomes more effective when generating passing test cases that are similar to the test cases inducing a failure [57]. However, the similarity metrics proposed by Artzi et al. cannot be used in our context for two main reasons: (i) the test inputs inducing the target failure are not available (generating tests that replicate a crash is the actual goal of EvoCrash and not its input) and (ii) the similarity metrics are defined for input and path-constraints (i.e., not for stack traces).

To calculate the trace distance, $d_{trace}(t)$, in our preliminary study [201] we used the distance function defined as follows. Let $S^* = \left\{e_1^*, \ \ldots, e_n^*\right\}$ 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:

$$d_{trace}(t) = \sum_{i=1}^{\min\{k,n\}} \varphi\Big(\text{diff}(e_i^*, e_i)\Big) + \mid n - k \mid \tag{2.1}$$

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 $\varphi(x) = x/(x+1)$ [160]. 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 2.1 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 1. *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:*

$$d_{trace}(t) = \sum_{i=1}^{n} \min\Big\{\text{diff}(e_i^*, e_j) : e_j \in S\Big\} \tag{2.2}$$

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:

$$\text{diff}(e_i^*, e_i) = \begin{cases} 3 & \text{if } C_i^* \neq C_i \\ 2 & C_i^* = C_i \text{ and } m_i^* \neq m_i \\ \varphi\left(\mid l_i^* - l_i \mid\right) \in [0; 1] & \text{Otherwise} \end{cases} \tag{2.3}$$

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, $d_{trace}(t)$ in Equation 2.2 is zero if and only if the two traces S^* and S are equal, i.e., they share the same trace elements.

Table 2.1: Example of three different test cases with their corresponding distances and fitness function scores.

Test	d_s	d_{except}	d_{trace}	Fitness Function
t_1	0.14	1.00	2	$0.12 * w_1 + 1.00 * w_2 + 0.67 * w_3$
t_2	0.00	1.00	4	$0.00 * w_1 + 1.00 * w_2 + 4.00 * w_3$
t_3	0.00	0.00	5	$0.00 * w_1 + 0.00 * w_2 + 0.86 * w_3$

Final fitness function. To combine the three distances defined above, we use the *weighted-sum scalarization* [92].
Definition 2. *The fitness function value of a given test t is:*

$$f(t) = w_1 * \varphi(d_s(t)) + w_2 * d_{except}(t) + w_3 * \varphi(d_{trace}(t)) \tag{2.4}$$

where $d_s(t)$, $d_{except}(t)$, and $d_{trace}(t)$ are the three individual distance functions described above; $\varphi(.)$ is a normalizing function [160]; w_1, w_1, and w_3 are the linear combination coefficients.

Notice that in the equation above, the first and the last terms are first normalized before being summed up. This is because they have different orders of magnitude: the maximum value for $d_{trace}(t)$ corresponds to the total number of frames in the stack traces; $d_{except}(t)$ takes values in $\{0, 1\}$; while the maximum value of $d_s(t)$ is proportional to the cyclomatic complexity of the class under test.

In principle, the linear combination coefficients can be chosen such as to give higher priority to the different composing distances. In our context, meeting the three conditions for an optimal crash replication should happen in a certain order. In particular, executing the target line takes precedence over throwing the exception, and in turn, throwing the target exception takes priority over the degree to which the generated stack trace is similar to the reported one.

For example, let us consider the three test cases t_1, t_2, and t_3 reported in Table 2.1. In the example, t_1 does not cover the target line (i.e., $d_s(t_1) > 0$) and it throws an exception but not the target one; t_2 covers the target line but throws the wrong exception (i.e., $d_s(t_2) = 0$ and $d_{except} = 1.0$); finally, t_3 covers the target line (i.e., $d_s(t_2) = 0$), it throws the right exception (i.e., $d_{except} = 0$) but its trace similarity is larger than the one for t_1 (i.e., $d_{trace}(t_3) > d_{trace}(t_1)$). The distance values and the corresponding fitness function for the three tests are also reported in Table 2.1.

Now, let us suppose we decide to give larger priority to d_{trace} compared to the other distances, e.g., $w_1 = 0.05$, $w_2 = 0.05$, and $w_3 = 1$. By applying Equation 2, we

would obtain the following fitness scores:

$$f(t_1) = 0.05 * 0.12 + 0.05 * 1.00 + 0.67 \approx 0.7228$$
$$f(t_2) = 0.05 * 0.00 + 0.05 * 1.00 + 0.80 \approx 0.8500$$
$$f(t_3) = 0.05 * 0.00 + 0.05 * 0.00 + 0.86 \approx 0.8571$$

In other words, with these weights, t_3 has the largest (worst) fitness score although it is the closest one to replicate the target crash (it covers the target line and triggers the correct exception). Instead, t_1 and t_2 do not even cover the target line even though they have a better fitness than t_3. With the weights above, the corresponding fitness function $f(.)$ would misguide the search by introducing local optima. Therefore, our weights should satisfy the constraints $w_1 \geq w_3$ and $w_3 \geq w_1$, i.e., d_{trace} should not have larger a weight compared to the other distances.

Let us consider other three coefficients that satisfy the constraints above: $w_1 = 0.01$, $w_2 = 1$, $w_3 = 0.01$. The corresponding fitness values for the three tests in Table 2.1 are as follows:

$$f(t_1) = 0.01 * 0.12 + 1.00 + 0.01 * 0.67 \approx 1.0079$$
$$f(t_2) = 0.01 * 0.00 + 1.00 + 0.01 * 0.80 \approx 1.0080$$
$$f(t_3) = 0.01 * 0.00 + 0.00 + 0.01 * 0.86 \approx 0.0086$$

With these new weights, t_3 has the lowest (better) fitness value since both the two constraints $w_1 \geq w_3$ and $w_2 \geq w_3$ are satisfied. However, t_1 has a better fitness than t_2 although the latter covers the target line while the former does not. To avoid this scenario, our weights should satisfy another constraint: $w_1 \geq w_2 + w_3$.

In summary, choosing the weights for the function in Definition 2 consists in solving the following linear system of inequality:

$$\begin{cases} w_1 \geq w_2 + w_3 \\ w_1 \geq w_3 \\ w_2 \geq w_3 \end{cases} \tag{2.5}$$

In this paper, we chose as weights the smallest integer numbers that satisfy the two inequalities in the system above, i.e., $w_1 = 3$, $w_2 = 2$, $w_3 = 1$. With these weights, the fitness values for the test cases in the example of Table 2.1 become: $f(t_1) = 3.04$, $f(t_2) = 2.80$, and $f(t_3) = 0.86$. While choosing the smallest integers makes the interpretation of the fitness values simpler, we also used different integers in our preliminary trials. We did not observe any impact on the outcomes.

In general, with these weights, fitness function $f(t)$ assumes values within the interval $[0, 6]$; a value $3 \leq f(t) \leq 6$ indicates that a test t does not cover the target line; a

value $1 \leq f(t) < 3$ means that the test t covers the target line but does not throw the target exception; a zero value is reached if and only if the evaluated test t replicates the target crash.

2.3.3 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 2.1 (*please see the end of the chapter*), 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 post-processed (e.g., minimized) in line 26.

Initial Population. The routine used to generate the initial population plays a paramount role [179] since it performs sampling of the search space. In traditional coverage-based tools (e.g., EvoSuite [103] or JTExpert [196]) such a routine is designed to generate a well-distributed population (set of tests) that maximize the number of methods in the class under test C that are invoked/covered [103]. Instead, the main goal for crash replication is invoking the subset of methods M_{crash} in C that appear in the crash stack traces since they may trigger the target crash. Instead, the remaining methods can be still invoked with some random probability to instantiate objects (test inputs) or if they help to optimize the fitness function (i.e., decreasing the approach level and branch distance for the target line to cover).

For this reason, in this paper we use the novel routine highlighted in Algorithm 2.2 (*please see the end of the chapter*) 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 2.2 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 2.2 generates random tests using the loop in lines 3-18, and requires as input (i) the set of public target method(s) 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 2.2 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 [103]. 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 2.2 uses the routine INSERT-METHOD-CALL (line 18). For each method call in t, such a routine sets each input parameter as follows:

Case 1 It re-uses an object or variables already defined in t with a probability p=1/3;

Case 2 If the input parameter is an object, it sets the parameter to `null` with a probability p=1/3;

Case 3 It randomly generates an objects or primitive value with a probability p=1/3;

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 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.

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

To avoid this scenario, GGA leverages a novel *guided single-point crossover* operator, whose main steps are highlighted in Algorithm 2.3 (*please see the end of the chapter*). 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 2.3). 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 2.3 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 2.4 (*please see the end of the chapter*).

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 2.4). 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:

Case 1 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);

Case 2 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; the input parameters (objects or primitive values) are either (i) taken from

the previous $i-1$ statements in t, (ii) set to `null` (for objects only), (iii) or randomly generated. These three mutations are applied with the probability $p=1/3$. Therefore, they are equally probable and mutually exclusive for each input parameter.

Finally, removing a method call (lines 8-9 in Algorithm 2.4) also 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 a developer as a 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 needed 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 [103], 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 [41]. Therefore, the *values minimization* from EvoSuite shortens the identified numbers and simplifies the randomly generated strings [102].

2.3.4 Mocking Strategies

Since EvoCrash is built on top of EvoSuite, by default, EvoCrash inherits the mocking strategies implemented in EvoSuite [53–55]. Therefore, if reproducing a target crash requires environmental interactions involving system calls (e.g., System.currentTimeMillis), network connections (e.g., calls to java.net APIs) and file system (e.g., calls to java.io.File), EvoCrash benefits from the available mocking operators to reproduce the crash.

However, it is possible that reproducing a crash requires specific content as the result of the interaction with the environment. For example, it could be that specific content of an XML file is needed to reproduce a crash. In these cases, EvoCrash lacks support for finding the specific content needed to optimize the fitness function. This is an

open problem in automated test generation that calls for future work and is beyond the scope of this study.

2.4 Study I: Effectiveness

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

2.4.1 Research Questions

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

- **RQ$_1$:** *How does EvoCrash perform compared to coverage-based test generation?* EvoCrash is built on top of Evosuite, which is a coverage-based test generation tool for unit testing. Therefore, with this research question, we aim at investigating to what extent EvoCrash actually provides the expected benefits in terms of the number of reproduced crashes and test generation time compared to a classical coverage-based test generation approach.
- **RQ$_2$:** *In which cases can EvoCrash successfully reproduce the targeted crashes, and under what circumstances does it fail to do so?* With this 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$_3$:** *How does EvoCrash perform compared to state-of-the-art reproduction approaches based on stack traces?* In this research question, we investigate the advantages and disadvantages of EvoCrash as compared to the most recent stack trace based approaches to crash reproduction previously proposed in the literature.

For **RQ$_1$**, we selected EvoSuite [103] as a representative tool for state-of-the-art approaches for coverage-based unit testing. Our choice is guided by the fact that EvoSuite won the latest two editions of the SBST tool competition [115] [107] and achieved very competitive scores (i.e., code coverage and fault detection rate) compared to hand-written tests. Moreover, EvoCrash and EvoSuite share the same instrumentation engine, the test execution environment and the encoding schema for test cases. By default, EvoSuite uses the *Archive-based Whole Test Suite* generation approach (WSA) [193], which evolves test suites and optimizes multiple testing criteria.

The default coverage criteria are *line coverage, branch coverage, direct branch coverage, weak mutation, exception coverage, no-exception top-level method coverage, and output coverage*, which are described in detail by Rojas et al. [190]. Exception coverage is particularly important in our context: using WSA, when this criterion is enabled, EvoSuite stores in an *archive* all test cases (which compose candidate test suites) that trigger an exception when trying to maximize the other coverage criteria. Therefore, the final test suite produced from EvoSuite not only achieves higher code coverage but also contains all tests triggering some exceptions which were found during the generation process.

For the sake of our analysis, we conducted the experiments with EvoSuite using the default coverage criteria and targeting the same class tested by EvoCrash. First, we compare EvoSuite and EvoCrash in terms of crash replication frequency, i.e., the number of times each of the two techniques successfully reproduced a crash over 15 independent runs. A crash is covered, according to the *Crash Coverage* criterion by Chen and Kim [81], when the test generated by one tool triggers 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 EvoCrash reached a zero-fitness value, i.e., when the generated crash stack trace is identical to the target one.

While EvoCrash produces only one test for each crash, EvoSuite generates entire test suites. Thus, for the latter tool, we consider a crash as replicated if at least one test case within the test suite generated by EvoSuite is able to replicate the target crash. To further guarantee the reliability of our results, we re-executed the tests generated by EvoCrash and EvoSuite against the CUT to ensure that the crash stack frame was correctly replicated.

We also compared EvoSuite and EvoCrash in terms of search time required to replicate each crash. To this aim, during each tool run, we stored the duration of the time interval between the start of the search and the point in time where each test case (or test suite for EvoSuite) was generated. Then, the time to replicate each crash (if replicated) corresponds to the search time interval of the test case (or test suite) that successfully replicates it.

To address $\mathbf{RQ}_2$, we apply the two criteria proposed by Chen and Kim [81] for evaluating the effectiveness of crash replication tools: *Crash Coverage* and *Test Case Usefulness*. *Crash Coverage* is the same criterion used to answer $\mathbf{RQ}_1$. For the *Test Case Usefulness*, a test case generated by EvoCrash is considered *useful* if and only if it reveals the actual bug that causes the original crash. According to the guidelines in [81], 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 [81]) or the frame the execution

of which covers the buggy statement. The guidelines in [81] further clarify that in addition to generating the buggy frame, *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 [81]. This implies that if the buggy frame receives input arguments, then a *useful* test case must also generate at least one frame at a higher level than the buggy frame, through which we can observe how the input arguments to the buggy method are generated. Of course, if a) the stack trace has only one frame, or 2) the buggy method does not receive any arguments, then a *useful* test must only generate the buggy frame to be considered as useful.

To assess usefulness of the tests, we carefully inspected the original developers' fixes to identify the bug fixing locations. We manually examined each crash classified as *covered* (using the coverage criterion) to investigate if it reveals the actual bug following the guidelines in [81]. This manual validation has been performed by the first two authors independently, and cases of disagreement were discussed and resolved.

For **RQ3**, we selected three state-of-the-art techniques, namely: STAR [81], MuCrash [215], and JCHARMING [171, 172]. These three techniques are modern approaches to crash replication for Java programs, and they are based on three different categories of algorithms: symbolic execution [81], mutation analysis [215], and model checking [171].

At the time of writing this paper, STAR, MuCrash, and JCHARMING were not available (either as executable jars or source code). Therefore, to compare our approach, we rely on their published data. Thus, we compared EvoCrash with MuCrash for the 12 bugs selected that have also been used by Xuan et al. [215] to evaluate MuCrash. We compared EvoCrash with JCHARMING for the 13 bug reports that have been also used by Nayrolles et al. [171]. Finally, we compare EvoCrash with STAR for the 51 bugs in our sample that are in common with the study by Chen and Kim [81].

2.4.2 Definition and Context

As Table 2.2 presents, the *context* of this study consists of 54 bugs from seven real-world open source projects: `Apache Commons Collections`[3] (ACC), `Apache Ant`[4] (ANT), `Apache Log4j`[5] (LOG), `ActiveMQ`[6], `DnsJava`[7], and `JFreeChart`[8].

[3] `https://commons.apache.org/proper/commons-collections/`
[4] `http://ant.apache.org`
[5] `http://logging.apache.org/log4j/2.x/`
[6] `http://activemq.apache.org/`
[7] `http://www.dnsjava.org/`
[8] `http://jfree.org/jfreechart//`

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`[9] submitted between October 2003 and June 2012 and 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 21 bug reports submitted on `Bugzilla`[10] between April 2004 and August 2012 and that concern 10 different versions and sub-modules.

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 window between June 2001 and October 2009 and that are related to three different LOG versions.

ActiveMQ is a messaging and Integration Patterns server that is actively maintained by the Apache Software Foundation. ActiveMQ has 205000 LOC, and supports many cross-language clients written in Java, C, C++, C#, and PHP. We selected one case from ActiveMQ that was also used for evaluating JCHARMING.

DnsJava is an implementation of DNS in Java, which has more than 3000 LOC. It supports all defined record types (including the *DNSSEC* types), and unknown types. It can be used for queries, zone transfers, and dynamic updates. It includes a cache which can be used by clients, and a minimal implementation of a server. In addition, since it is written in pure Java, DnsJava is fully threadable. We selected one case from DnsJava, which was also used in the evaluation of JCHARMING [171, 172].

JFreeChart is a free Java chart library, with 310000 LOC, that could be used to display high-quality charts in both server-side and client-side applications. JFreeChart has a well-documented API and it has been maintained over a long period of time, since 2005. We also selected a case from JFreeChart to use for comparison with JCHARMING.

We selected this set of bugs as they have been used in the previous studies on automatic crash reproduction when evaluating symbolic execution [81], mutation analysis [215], and directed model checking [171] and other tools [82, 129]. The characteristics of the selected bugs, including type of exception and priority, are summarized in Table 2.2. These bugs cover crashes that involve the most common Java `Exceptions` [84], such as `NullPointerException` (74%), `ArrayIndexOutOf-`

[9] `https://issues.apache.org/jira/secure/Dashboard.jspa`
[10] `https://bz.apache.org/bugzilla/`

Table 2.2: The 54 real-world bugs used in our study.

Project	Bug IDs	Versions	Exceptions	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)	[81] [215]
ANT	28820, 33446, 34722 34734, 36733, 38458 38622, 41422, 42179 43292, 44689, 44790 46747, 47306, 48715 49137, 49755, 49803 50894, 51035, 53626	1.6.1 - 1.8.2	ArrayIndexOutOfBounds (3) NullPointer (17) StringIndexOutOfBounds (1)	Critical (2) Major (5) Medium (14)	[81] [172]
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)	[81] [172]
ActiveMQ	5035	5.9	ClassCastExecption (1)	Major (1)	[172]
DnsJava	38	2.1	ClassCastException (1)	N/A (1)	[172]
JFreeChart	434	1.0	NullPointerException (1)	N/A (1)	[172]

`BoundsException` (9%), `IllegalStateExcep-tion` and `IllegalArgumentException` (3%). Furthermore, the severity of these real-world bugs varies between *medium* (46%), *major* (37%) and *critical* (5%) as judged by the original developers.

50 of these cases come from the primary study we performed in [203]. In this extension to [203], we aimed at extending the comparison with JCHARMING via the cases reported in [172]. However, ultimately, we chose to discard several cases reported in [172], and extend the comparison with JCHARMING via only 4 new cases, for four main reasons:

1. In six cases, the exact buggy version of the target software was either unknown or not found. Consequently, the reported line numbers in stack traces did not match the source code. Since the fitness function (Section 2.3.2) is primarily designed based on the exact line numbers where the exceptions are thrown, we discarded such cases.

2. As Nayrolles et al. report [172], to make a trade-off between reproducibility and relevance of the test cases, after a number of incremental attempts, they arrived at the threshold of 80% for reproducing stack traces. Thus, in some cases they report *partial* coverage, which means that at least 80% of a stack trace could be reproduced in those cases. While this partial measure is relative to the size of the stack traces, in our case we need to have exact measure of the reproduced traces to compare the *usefulness* of the tests, as described in Section 2.4.1.

3. In two cases, `ActiveMQ-1054` and `ArgoUML-311`, the reported stack traces lack line numbers. Thus, considering how the fitness function works (Section 2.3.2), we could not apply EvoCrash on such cases.

4. Finally, one of the reported cases in [172], `Mahout-1594`, actually refers to an external problem in the configuration file. Thus, this case was not a valid crash case to be considered in this study.

2.4.3 Experimental Procedure

We run EvoCrash and EvoSuite on each target crash to try to generate a test case and test suite able to reproduce the corresponding stack trace. Given the randomized nature of genetic algorithms, we executed the tools multiple times to verify that the target crashes are replicated in most of the runs. For RQ_1, we ran EvoSuite and EvoCrash 15 times for each crash. For RQ_2 the search for each target bug/crash was repeated 50 times.

In our experiment, we configured both tools by using standard parameter values widely used in evolutionary testing [52, 103, 178]:

- **Population size**: we use a population size of 50 individuals as suggested in [103, 178]. In the context of EvoCrash, individuals are test cases whereas in the context of EvoSuite, individuals are test suites, containing one or more test cases.

- **Crossover**: For EvoCrash, we use the novel *guided single-point* crossover; in EvoSuite, the crossover operator is the classic *single-point* crossover [103]. In both cases, the *crossover probability* is set to $p_c 0.75$ [103].

- **Mutation**: EvoCrash uses our *guided uniform mutation*, which mutates test cases by randomly adding, deleting, or changing statements. EvoSuite uses the standard *uniform mutation*, which randomly adds, deletes, or changes test cases in a test suite. For both cases, we set the *mutation probability* equal to $p_m 1/n$, where n is the length of the test case/suite taken as input [103].

- **Search Timeout**: The choice of 10 minutes as the search budget is a common practice in studies on search-based test generation [52, 103, 178]:. In our preliminary experiments, we noticed that the number of reproduced crashes does not change after 10 minutes. Therefore, in both cases, the search stops when a zero-fitness function value is detected or when the timeout of 10 minutes is reached.

2.4.4 Comparison with Coverage-Based Test Generation

As Table 2.3 shows, EvoCrash reproduced 46 crashes (85%) out of 54, compared to 18 crashes (33%) that were reproduced by EvoSuite. In particular, 28 (52%) crashes out of 54 were reproduced only by EvoCrash. Other 18 crashes (33%) were reproduced by both EvoCrash and EvoSuite. Finally, for the remaining 8 cases (14%) both EvoCrash and EvoSuite failed to generate a crash reproducing test.

However, in those 18 cases where both EvoSuite and EvoCrash generate tests, the former always achieved a lower or equal reproduction rate compared to the latter, i.e., every crash was rarely reproduced out of 15 runs (e.g., ACC-53 in Table 2.3). Furthermore, EvoSuite took longer compared to EvoCrash to reproduce the same crashes. Indeed, EvoCrash took 145 seconds on average to reproduce the crashes, while EvoSuite required 391 seconds (+170%) to reproduce the same crashes on average.

Table 2.3: Crash reproduction results for comparing Archive-based Whole Test Suite generation (WSA) in EvoSuite and Guided Genetic Algorithm (GGA) in EvoCrash. The bold cases are the ones for which only EvoCrash could generate a test at least 8 times out of 15 runs.

		EvoCrash		**EvoSuite**	
Project	**Bug ID**	**avg. time**	**reproduction %**	**avg. time**	**reproduction %**
ACC	4	2	100%	314	100%
	28	1	100%	10	100%
	35	1	100%	50	100%
	48	40	100%	350	33%
	53	5	100%	377	66%
	68	600	0%	600	0%
	70	2	100%	407	33%
	77	98	100%	233	100%
	104	455	73%	600	0%
	331	100	73%	315	20%
	377	100	100%	335	13%
	411	153	80%	600	0%
	28820	600	0%	600	0%
	33446	10	100%	540	40%
	34722	59	100%	459	26%
	34734	45	100%	600	0%
	36733	32	86%	600	0%

	38458	43	86%	510	20%
	38622	33	100%	81	100%
	41422	220	66%	590	13%
	42179	56	93%	77	60%
	43292	600	0%	600	0%
	44689	32	100%	358	40%
	44790	15	100%	540	40%
	46747	600	0%	600	0%
	47306	600	0%	600	0%
	48715	600	0%	600	0%
	49137	90	100%	320	100%
	49755	30	100%	449	53%
	49803	10	100%	600	0%
	50894	42	100%	600	0%
	51035	600	0%	600	0%
	53626	105	100%	600	0%
	29	28	93%	301	6%
	43	600	0%	600	0%
	509	136	100%	600	0%
	10528	1	100%	3	100%
	10706	1	100%	35	100%
	11570	1	100%	129	100%
	31003	1	100%	9	93%
	40212	18	100%	472	26%
	41186	1	100%	27	100%
LOG	**44032**	3	100%	487	33%
	44899	42	100%	69	93%
	45335	10	100%	462	46%
	46144	20	93%	533	13%
	46271	3	100%	74	100%
	46404	59	100%	600	0%
	47547	3	100%	10	100%
	47912	38	93%	388	40%
	47957	5	100%	28	100%
ActiveMQ	**5035**	377	60%	600	0%
DnsJava	**38**	115	85%	481	13%
JFreeChart	**434**	389	53%	500	13%

The results above show that indeed the GGA in EvoCrash outperforms WSA in EvoSuite for crash reproduction in both the number of reproduced crashes and test generation times. The underlying explanation for such observations is that EvoSuite, using WSA, evolves test suites with the goal of maximizing code coverage. Assuming that line l is where the target exception e happens, if there is a test suite that includes a test case t_l that covers l, EvoSuite archives t_l and l, and proceeds by evolving test suites targeting only the remaining uncovered lines. The archived test case t_l that covers the target line l, by chance may or may not trigger e as well. Furthermore, since criterion *Exception* was included in the optimization criteria, if there exists a test suite that contains test case t_e which triggers an exception, EvoSuite would archive t_e. By chance, t_e may or may not trigger e on the target line l.

On the other hand, EvoCrash uses GGA, which customizes test generation for crash coverage. Therefore, the search is aimed for a test case that both covers the target line l, and triggers the target exception e. This means that even if a test t_i covers l, EvoCrash keeps t_i in the search process in order to evolve it until it can also trigger e.

Thus, this comparison highlights that while coverage-based test generation by EvoSuite may by chance detect crashes, using GGA is a more effective and efficient approach for crash reproduction.

2.4.5 Crash Reproduction Effectiveness

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.

Table 2.4: 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. The bold cases are the ones for which only EvoCrash could generated a test and the underlined ones are those where EvoCrash failed to produce a test at least 25 times out of 50 runs.

Project	Bug ID	EvoCrash	STAR [81]	MuCrash [215]	JCHARMING [171]
	4	Y	Y	Y	-
	28	Y	Y	Y	-
	35	Y	Y	Y	-
	48	Y	Y	Y	-
	53	**Y**	Y	**N**	-

ACC

	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	-
	28820	N	N	-	-
	33446	NU	NU	-	-
	34722	**Y**	**N**	-	-
	34734	**NU**	**N**	-	-
	36733	NU	NU	-	-
	38458	Y	Y	-	-
	38622	Y	Y	-	Y
	41422	**NU**	Y	-	**N**
	42179	**Y**	**N**	-	-
	43292	N	Y	-	-
ANT	**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**	-	-
	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	-	Partial
LOG	**44032**	**Y**	**N**	-	-
	44899	**Y**	**N**	-	-

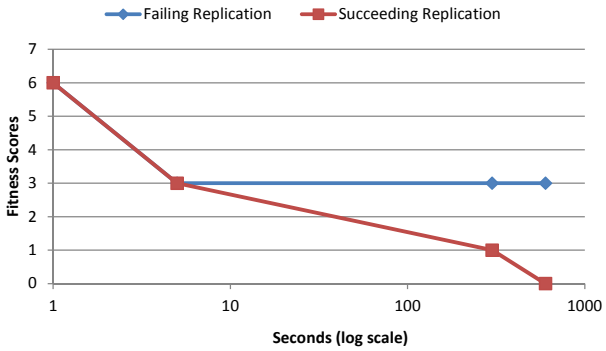

Figure 2.2: Fitness progress over time for both succeeding and failing runs of EvoCrash for ACC-104.

	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**
ActiveMQ	**5035**	**Y**	-	-	**N**
DnsJava	38	Y	-	-	Y
JFreeChart	434	Y	-	-	Y

EvoCrash Results (RQ2) As Table 2.4 illustrates, EvoCrash can successfully replicate the majority of the crashes in our dataset. 39 cases could be replicated 50 times out of 50 runs of EvoCrash. Of the replicated cases, LOG-509 had the lowest rate of replications - 39 out of 50. EvoCrash reproduces 11 crashes out of 12 (91%) for ACC, 15 out of 21 (71%) for ANT, and 17 out of 18 (94%) for LOG. Overall, EvoCrash can replicate 46 (85%) out of the 54 crashes.

To assess the usefulness of the generated test cases, as explained in Sub-section 2.4.1, we used the same criterion that was used for STAR [81]. Based on this, 38 (84%) of the replications were useful, as they included the buggy frame. The remaining 16% non-useful replications were mainly due to having dependency on data from external files which were not available during replication.

For ACC, ACC-68 was not reproducible by EvoCrash. In this case, the class under test includes three nested classes, and the inner-most one was where the crash occurs. We could not replicate this crash as EvoCrash relies on the instrumentation engine of EvoSuite, which does not currently support the instrumentation of multiple inner classes.

In addition, for ACC-104 [11], EvoCrash could replicate the case 42 times out of 50. The average time EvoCrash took for reproducing this case is 300 seconds. In this case, the defect lies on line *20* in Figure 2.1, where the shift operation does not correctly increment or decrement array indexes. In order to replicate this case, a test case shall meet the following criteria: 1) Make an object of the `BoundedFifoBuffer` class. 2) Add an arbitrary number of objects to the buffer. 3) Remove the last item from the buffer, and add arbitrary number of new items. 4) Remove an item that is not the last item in the buffer.

To understand why EvoCrash takes relatively longer to reproduce ACC-104, Figure 2.2 demonstrates the search progress during the failing and successful executions. As the Figure shows, during the failing executions, the fitness value quickly progresses to *3.0* and it remains unchanged until the search budget *(10 minutes)* is over. In these executions, a fitness value of *3.0* means that the target line, line *20* in Figure 2.1 is covered by the execution of the test cases. However, the target exception `ArrayIndexOutOfBounds` is not thrown at this line, which is why the fitness does not improve and remains *3.0* until the search time is consumed. On the other hand, during the successful runs, not only line *20* is covered, on average in five seconds, but also after 5 minutes, the target exception is thrown and generates the reported crash stack trace. As our results indicate, setting an object of `BoundedFifoBuffer` to the right state such that an arbitrary number of elements are added and removed in a certain order (as indicated previously) to throw the `ArrayIndexOutOfBounds` exception is a challenging task.

For ANT, six of the 20 crashes (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. For LOG, one 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.

[11] https://issues.apache.org/jira/browse/COLLECTIONS-104

```
java.lang.ArrayIndexOutOfBoundsException:
    at org.apache.commons.collections.buffer.BoundedFifoBuffer.remove(
    BoundedFifoBuffer.java:347)
```

Listing 2.2: Crash Stack Trace for ACC-104.

```
 public void remove() {
   if (lastReturnedIndex == -1) {
      throw new IllegalStateException();
   }

   // First element can be removed quickly
   if (lastReturnedIndex == start) {
      BoundedFifoBuffer.this.remove();
      lastReturnedIndex = -1;
      return;
   }

   // Other elements require us to shift the
       subsequent elements
   int i = lastReturnedIndex + 1;
   while (i != end) {
      if (i >= maxElements) {
         elements[i - 1] = elements[0];
         i = 0;
      } else {
         elements[i - 1] = elements[i];
         i++;
      }
   }

   lastReturnedIndex = -1;
   end = decrement(end);
   elements[end] = null;
   full = false;
   index = decrement(index);
}
```

Listing 2.1: Buggy method for ACC-104.

2.4.6 Comparison to State of the Art

This section discusses the results of the comparison between EvoCrash and the state-of-the-art approaches based on crash stack traces, namely STAR [81], MuCrash [215], and JCHARMING [171]. In Table 2.4, bold entries represent bugs which can be

triggered by EvoCrash, but not by at least one of the other techniques; Underlined entries represent bugs that EvoCrash cannot reproduce, while there is another technique that can. As can be seen, there are 23 (bold) cases in which EvoCrash outperforms the state of the art, and there are two (underlined) cases that EvoCrash cannot handle. Below we discuss these cases in more detail.

EvoCrash *vs.* STAR. As Table 2.4 presents, for ACC, EvoCrash covers all the cases that STAR covers. In addition, EvoCrash covers three 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 could not be handled by STAR.

For ANT, EvoCrash supports seven cases (35%) which are not covered by STAR. Out of the seven, there are three cases, for which only EvoCrash can generate a useful test case. Listing 2.3 shows the crash stack trace for one of these cases (ANT-49137). As reported in the issue tracking system of the project[12], 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.

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

Listing 2.3: Crash Stack Trace for ANT-49137.

As Listing 2.4 depicts, 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 2.5 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 2.6. 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

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

set by STAR [81].

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

Listing 2.4: Generated test by EvoCrash for ANT-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 2.5: Generated test by STAR for ANT-49137.

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

Listing 2.6: Generated Crash Stack Trace by STAR for ANT-49137.

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, `mapFile Name`, 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.

For LOG, EvoCrash covers all the cases that were covered by STAR. Six of the LOG cases (33%) are only covered by EvoCrash. As an example, for LOG-509 there is a need to interact with the file system in order to open a file, and in order to do so, EvoCrash benefits from the mocking mechanisms implemented in EvoSuite.

LOG-47912 (shown in Listing 2.7) is another example for which only EvoCrash successfully generated a useful test case. The buggy frame in this case is at level four, and the generated test by EvoCrash is at level five, which is shown in Listing 2.8. As the listing shows, in order to generate a test at this level, several complex objects need to be generated and set up first, until finally the call to *jULBridgeHandler0.publish(logRecord0);* is made. This example shows the capability of EvoCrash to

generate complex objects which may be needed to execute a particular execution path that leads to the target line where the target exception is thrown.

```
java.lang.NullPointerException:
    at org.apache.log4j.CategoryKey.(CategoryKey.java:32)
    at org.apache.log4j.Hierarchy.getLogger(Hierarchy.java:266)
    at org.apache.log4j.Hierarchy.getLogger(Hierarchy.java:247)
    at org.apache.logging.julbridge.JULLog4jEventConverter.convert(
    JULLog4jEventConverter.java:121)
    at org.apache.logging.julbridge.JULBridgeHandler.publish(
    JULBridgeHandler.java:49)
```

Listing 2.7: Stack Trace for LOG-47912.

```
public void test0() throws Throwable {
   Logger logger0 = Logger.getLogger("I}h}$.Xa|yA,YSXf");
   Hierarchy hierarchy0 = (Hierarchy)logger0.getLoggerRepository();
   JULLog4jEventConverter jULLog4jEventConverter0 = new
       JULLog4jEventConverter((LoggerRepository) hierarchy0,
       (JULLevelConverter) null);
   JULBridgeHandler jULBridgeHandler0 = new
       JULBridgeHandler((LoggerRepository) hierarchy0,
       jULLog4jEventConverter0);
   Level level0 = Level.SEVERE;
   LogRecord logRecord0 = new LogRecord(level0, "");
   jULBridgeHandler0.publish(logRecord0);
}
```

Listing 2.8: Generated test by EvoCrash for LOG-47912.

EvoCrash *vs.* MuCrash. As Table 2.4 shows, evaluation data for MuCrash is only available for ACC.[13] Except for ACC-104, EvoCrash covers all the ACC-cases that are covered by MuCrash. In addition, three 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 2.9. It requires that an object is added to an instance 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 2.10, EvoCrash synthetized the right method sequence and reproduced ACC-53.

[13]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 2.9: 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 2.10: EvoCrash test for ACC-53

EvoCrash *vs.* JCHARMING. As Table 2.4 shows, we have 12 cases to derive comparisons between EvoCrash and JCHARMING. While 75% of the 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 (based on the published results) JCHARMING may take from 10 to 38 minutes to generate tests for the same cases.

Among the LOG cases, two out of seven (29%) are only supported by EvoCrash. As an example, Listing 2.11 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 2.12, 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.

Among the other cases, two of them are only supported by EvoCrash, `ANT-41422`, and `ActiveMQ-5035`. The former is a `NullPointerException`, and the latter is a `ClassCastException`.

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

Listing 2.11: Crash Stack Trace for LOG-45335.

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

Listing 2.12: The EvoCrash Test for LOG-45335.

2.4.7 Threats to Validity

In this section, we outline various possible threats to the validity of the empirical evaluation we conducted.

External Validity. The main threats arise from the focus on Java and open source projects. 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.

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 a systematic 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.

Internal Validity. A key threat to the internal validity 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 rerun the generated test against the SUT to double checked that the generated crash stack trace correctly replicated the target crash stack. 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 tests and crashes. In addition, we make the EvoCrash tool, and the generated test cases publicly available for further evaluations.

2.5 Study II: Usefulness for Debugging

To assess the degree to which generated crash-reproducing tests are useful during debugging, we conduct a controlled experiment. The experiment aims to address the following:

- **RQ_4:** *Do participants who use EvoCrash tests more often locate defects compared to participants who do not use EvoCrash tests?* With this research question, we aim to understand whether using the generated tests by EvoCrash helps locate defects.
- **RQ_5:** *Do participants who use EvoCrash tests more often provide fixes compared to participants who do not use EvoCrash tests?* With this research question, we aim to investigate whether using the generated test by EvoCrash helps fixing defects.
- **RQ_6:** *Do participants who use EvoCrash tests spend less time than participants who do not use EvoCrash tests?* With this research question, we aim to analyze the impact of using the generated tests by EvoCrash in the amount of time the participants took to deliver fixes.

2.5.1 Task Selection

To select the crash cases to be used in the debugging tasks, we considered the following selection criteria: (i) From the 54 crashes we used in the empirical evaluation (Section 4.4), we selected those crashes which signal the two common types of exceptions in Java programs [84], namely: `NullPointerException`, and `IllegalArgumentException`; (ii) We filtered out stack traces which have less than four stack frames, since locating and fixing the related bug would be very simple; (iii) To avoid cases that would be overly complicated to fix, we selected cases for which the original fixes (delivered by the original developers) are provided for the classes that were included in the stack traces. (iv) We ensured that the JavaDoc documentation is available for all classes appearing in the stack traces and could serve as specification for the participants. Finally, (v) considering the usefulness criterion (described in Section 2.4.1), we opted for including both a *useful* and *not useful* crash-reproducing unit test case.

As the result, we selected ACC-48 (with a useful test), and LOG-47957 (with a not useful test) to be the target cases. Listing 2.13 and 2.14 show the stack traces for the two cases.

```
java.lang.IllegalArgumentException: Initial capacity must be greater than 0
   at org.apache.commons.collections.map.AbstractHashedMap.
     (AbstractHashedMap.java:142)
   at org.apache.commons.collections.map.
     AbstractHashedMap.(AbstractHashedMap.java:127)
   at org.apache.commons.collections.map.AbstractLinkedMap.
   (AbstractLinkedMap.java:95)
   at org.apache.commons.collections.map.LinkedMap.
     (LinkedMap.java:78)
   at org.apache.commons.collections.map.TransformedMap.
     transformMap(TransformedMap.java:153)
   at org.apache.commons.collections.map.TransformedMap.putAll
     (TransformedMap.java:190)
```

Listing 2.13: Crash Stack Trace for ACC-48; Fixed at frame 5 (line 153) and tested at frame 6 (line 190).

```
java.lang.NullPointerException:
   at org.apache.log4jb.net.SyslogAppender.append(SyslogAppender.java:251)
   at org.apache.log4jb.AppenderSkeleton.doAppend(AppenderSkeleton.java:230)
   at org.apache.log4jb.helpers.AppenderAttachableImpl.appendLoopOnAppenders
     (AppenderAttachableImpl.java:66)
   at org.apache.log4jb.Category.callAppenders(Category.java:203)
   at org.apache.log4jb.Category.forcedLog(Category.java:388)
   at org.apache.log4jb.Category.info(Category.java:663)
```

Listing 2.14: Crash Stack Trace for LOG-47957; Fixed and tested at frame 1 (line 251).

The original fixes for `ACC-48` and `LOG-47957` were provided for the frame levels five and one, respectively. In addition, the tests from EvoCrash for these cases were targeted for the frame levels six and one, respectively.

2.5.2 Experiment Participants

We invited 35 master students in computer science from the Delft University of Technology to participate in the study. Table 2.5 presents the level of formal education the participants have in Java programming. Table 2.6 presents the degree to which the participants have industrial experience in software engineering. Moreover, Table 2.7 summarizes the degree to which the participants were familiar with the JUnit testing framework.

Table 2.5: Participants' Education in Java Programming

Self-educated	Formal Education		
	Basic	Intermediate	Advanced
5.71%	28.57%	45.71%	20%

Table 2.6: Participants' Industrial Experience

No exp.	≤ 2 years	3-5 years	5-10 years
42.85%	34.28%	20%	2.85%

2.5.3 Experiment Procedure

Before conducting the experiment, the participants received an introduction to the tasks to perform. The students had two weeks within which, at some point they were to start performing the experiment and deliver the results. Notice that to avoid any bias, we made sure participants were neither aware of the research questions of our study nor which crashes (name and id) were used as subjects of the experiment.

The participants were asked to debug and fix the classes involved in the two bugs ACC-48, and LOG-47957 starting from the corresponding crash stack traces. Each participant had to perform one bug fixing task using the crash-reproducing test from EvoCrash (e.g., ACC-48), while for the other one (e.g., LOG-47957) we did not provide the test from EvoCrash. To address potential bias due to learning effects, we assigned the tasks to have a balanced number of participants that performed the first task with and without the EvoCrash test. Therefore, we randomly grouped students in four different groups, whose configurations are shown in Table 2.8.

Once participants started performing the experiment at some point within the two weeks, they were asked to complete three stages in the context of the experiment: (i) filling a pre-test questionnaire that we used to collect data about participants' background, (ii) performing the first debugging task and filling the corresponding post-test questionnaire, and (iii) performing the second debugging task and filling a second post-test questionnaire. While the time to complete the first stage was unbounded, for the remaning two stages we restricted the amount of time participants could spend on each task following the guidelines by [212]. In particular, participants had 45 minutes for each task, which includes: (i) reading the instructions, (ii) cloning a Maven project from GitHub, and (iii) performing the corresponding debugging task. Each debugging task consists of (i) locating the defect that trigger the target crash, (ii) providing the code fix, (iii) running the existing test suite and adding new tests if needed. The participants could finish the tasks in less than 45 minutes if they were

Table 2.7: Participants' Familiarity with the JUnit Framework

Unfamiliar	Basic	Average	Advanced
37.14%	25.71%	28.57%	8.57%

Table 2.8: Configuration of the Participant Groups

Group	Task 1		Task 2	
	Bug	EvoCrash	Bug	EvoCrash
I	ACC-48	Yes	LOG-47957	No
II	ACC-48	No	LOG-47957	Yes
III	LOG-47957	Yes	ACC-48	No
IV	LOG-47957	No	ACC-48	Yes

sure that (i) the identified bug location is correct, and (ii) the provided fixes prevent the crashes to incur again and do not break the existing test suite. Controlling the time allowed to prevent that too little or too long time would be spent by participants on each task.

To prepare the projects on GitHub, we selected the versions of `Apache Commons Collections`, and `Apache Log4j` that were specified in the bug reports for ACC-48 and LOG-47957. Both projects were already Maven projects, so we imported them into Eclipse, and made sure the tests were run with no particular difficulties. For those tasks where the test from EvoCrash was provided, we included the tests in the projects, and added their path (packages) in the instructions provided to the participants.

As the first task reached the time out, or the participants completed the task within 45 minutes, they would proceed to the follow-up post-test questionnaire. To make sure the participants do not take time at this point to keep working on the task, we allowed 10 minutes to be spent on answering the questions. The second task followed the same procedure as the first one, after which the assignment would be completed. At the end, the participants had to send the artifacts they produced (including any test cases, or fixes) via e-mail to the first author. Furthermore, we used the online platform: *https://www.qualtrics.com* to collect the results of the questionnaires.

Before conducting the experiment, the last two authors performed the tasks to assess their feasibility and correctness in advance. We also conducted three pilot studies with external researchers within the software engineering research group at Delft University of Technology. The feedback we received from the pilot studies were used to improve both the questionnaires and the instructions for the tasks. Data points from dry-runs and pilot studies are not included in our analysis of the results.

2.5.4 Data Analysis

The original location of the defects and the patches provided by the developers for both cases represent our *golden answers* (oracle) as for the defect locations and fixes.

To answer $\mathbf{RQ_3}$, we compared the bug locations which were pointed to by the participants with the locations in our golden set. For example, for ACC-48, the defect could be fixed at two different frame levels in the stack trace, namely: (a) in the `transformMap` method at the 5th frame level in the stack trace reported in Listing2.13, and (b) in the `putAll` method at the 6th level. However, it is important to target the `transformMap` routine as the location for the underlying defect, and not the `putAll` routine. This is because `putAll` is an API call whereas `transformMap` is a private routine to which other routines make calls as well. Therefore, `transformMap` is the root location where the defect must be fixed otherwise the crash could recur. In cases where the participants targeted `putAll` as the buggy location, we marked their answers as incorrect.

For what concerns $\mathbf{RQ_4}$, we ran the fixes given by the participants to assess whether they prevented the crashes from recurring. If so, then we manually analyzed the content of the fixes. For example, in case of the fixes given for LOG-47957, we accepted every fix which pointed to checking for `null` references at the right location in the source code.

For what regards $\mathbf{RQ_5}$, we utilized the data that was provided by the online platform for collecting the data related to the time participants took to deliver the fixes. The data measured the point in time when the participant started a task (by reading the instructions), and the point in time when the participant completed the task before proceeding to answering the subsequent questions.

2.5.5 Statistical Analysis

To assess the effect of using the EvoCrash tests on the ability of participants to locate and fix the defects, we used the *odds ratio* measure [50] since the data is binary distributed, i.e., the defect is correctly located (or fixed) or not. For this test, we use a 95% confidence interval and we computed it for each debugging task (ACC-48, and LOG-47957) separately. In addition, to determine the significance of the findings, we used the Fisher's exact test, which is can be used for small sample sizes [50]. We considered $\alpha = 0.05$ for the Type I error. Significant p-values (i.e., lower than 0.05) indicate that participants with EvoCrash tests were able to correctly locate and

fix defects more frequently compared the participants who performed the same task (e.g., ACC-48) without the EvoCrash tests.

To measure the effect of using EvoCrash tests on the amount of time the groups took to complete each task, we used the Vargha-Delaney Â statistic [207]. We selected this effect size measure since it is well-suited for numerical data distributions [207], such as the time in seconds. Values of $\hat{A}<0.50$ indicate that participants with the EvoCrash tests spend less time than the participants without the EvoCrash tests to complete the same task; values of $\hat{A}>0.50$ indicates the opposite scenario, i.e., participants with the EvoCrash tests spent more time to complete the assigned tasks; $\hat{A}=0.50$ when there is no difference between the participants who performed the tasks with and without EvoCrash. The effect size can be classified as one of the four different levels [207]: *negligible* ($\hat{A}\geq0.44$), *small* ($0.36\leq\hat{A}<0.44$), *medium*($0.29\leq\hat{A}<0.36$), or *large* ($\hat{A}\leq0.29$). For a given task, we also test whether the difference (if any) between the groups with and without EvoCrash were statistically significant by using the non-parametric Wilcoxon Rank Sum test with $\alpha = 0.05$ for the Type I error. Significant p-values imply that there is significant difference in the amount of time the participants take when performing the debugging tasks with and without EvoCrash.

2.5.6 Analysis of the Results

In this section, we present the results of the controlled experiment with student participants. Table 2.9 summarizes the results regarding assessing the impact of using the tests from EvoCrash on the ability of the participants in locating the defects and providing fixes for them. As Table 2.9 indicates, one of 35 students, corresponding to one of the groups II or III in Table 2.8, did not deliver the debugging tasks. Thus, the number of participants in these groups is 34. On the other hand, all participants corresponding to groups I and IV in Table 2.8 delivered the debugging tasks. Therefore, the total number of participants in Table 2.9 is 35. In what follows, we discuss the results and thereby answer RQ_4, RQ_5, and RQ_6, respectively.

Table 2.9: Results of RQ_4 and RQ_5 grouped by tasks ("With" = with EvoCrash tests, "Without" = without EvoCrash tests).

Metrics	ACC-48		LOG-47957	
	With	Without	With	Without
No. of correct bug locations	13	13	12	9
No. of incorrect bug locations	5	3	4	8
No. of undelivered tasks	0	1	1	0
No. of correct bug fixes	10	3	8	6
No. of incorrect bug fixes	8	13	8	11
No. of undelivered tasks	0	1	1	0

2.5.6.1 RQ_4: Impact of EvoCrash Tests on Locating Defects

As Table 2.9 shows, in the case of ACC-48, the number of participants who located the defect correctly, using the test from EvoCrash, is the same as the number of participants who did not use the test from EvoCrash. The number of participants who failed to locate the defect, with and without using the EvoCrash test, are five and three, respectively. In the case of LOG-47957, 12 participants, using the test from EvoCrash, and nine participants without using the test correctly located the defect. The number of participants who failed to locate the defect, with and without the EvoCrash test, are four and eight, respectively.

To assess the impact of using EvoCrash on locating the underlying defects for each of the debugging task, we used the odds ratio and Fisher's exact test as explained in Section 2.5.5. For ACC-48, the odds ratio is 0.63, thus, indicating that the test case from EvoCrash did not help the participants in locating the underlying defect. Moreover, the Fisher test further confirms that there is no statistically significant difference between the two groups (p-value = 0.86). For LOG-47957, the odds ratio is 2.66, suggesting that the test from EvoCrash helped the participants in locating the underlying defect more often than the participants who did not use the test. However, these results are not statistically significant in this case either (p-value=0.14).

RQ_4: EvoCrash helps participants in locating the defect for LOG-47957, while in the case of ACC-48 we did not observe such an impact. In either case, the differences are not statistically significant.

2.5.6.2 RQ_5: Impact of EvoCrash Tests on Fixing Defects

As Table 2.9 shows, in the case of ACC-48, the number of participants who provided acceptable fixes are 10 when using the test from EvoCrash and three without the test. In addition, eight and 13 participants, with and without the test from EvoCrash respectively, failed to provide an acceptable fix for ACC-48. In the case of LOG-47957, eight participants, using the test from EvoCrash, and six participants without using the generated test provided acceptable fixes. The number of participants who failed to provide acceptable fixes, with and without using the test is eight and 11, respectively.

To assess the impact of using EvoCrash on the ability of participants in providing fixes, we computed the odds ratio for each debugging task, separately. In addition, we used the Fisher's exact test for significance.

For ACC-48, the odds ratio is 5.41. This indicates that the test case generated by EvoCrash increased the participants' ability to provide fixes when performing such a debugging task. According to the Fisher test, the differences are statistically significant (p-value=0.03). We further note, based on the usefulness criterion described in Section 2.4.1 we labeled the test generate by EvoCrash as useful for debugging.

For LOG-47957, the odds ratio is 1.83. Based on these measures, we observed that using the test from EvoCrash increased the participants' ability to provide correct fixes. However, such an improvement is not statistically significant as suggested by the Fisher test (p-value=0.30). These results are in line with the results of Study I, where we labeled the test generated by EvoCrash as *not useful* according to the usefulness criterion described in Section 2.4.1).

RQ_5: Using a test from EvoCrash, that is *useful* according the usefulness criterion in Section 2.4.1, increases developers' ability in fixing defects when debugging. In addition, our results suggest that using a test from EvoCrash, that is *not useful* according to the usefulness criterion in Section 2.4.1, also increases developers' ability in fixing defects when debugging. However, in the latter case, the difference is not statistically significant.

2.5.6.3 RQ_6: Impact of EvoCrash Tests on Debugging Time

The box-plots in Figure 2.3 show the distribution of time participants took to perform each task. In the case of ACC-48, the median for the group which did not use the EvoCrash test is 1565 seconds, while the median for the other group, using the EvoCrash test, is 1064 seconds (-32%). In the case of LOG-47957, the median for the group which did not use the EvoCrash test is 2700 seconds, while the median for the other group, using the EvoCrash test, is 2037 seconds (-25%). Thus, in both cases, the medians for the group which used the tests from EvoCrash are lower than the median for the group which did not use the EvoCrash tests.

To verify whether such differences are statistically significant or not, we used the non-parametric Wilcoxon test for each debugging task (ACC-48, and LOG-47957) as described in Section 2.5.5. As effect size measure, we used the Vargha-Delaney Â statistics.

For ACC-48 and LOG-47957, the obtained Â scores are 0.28 (medium) and 0.30 (medium), respectively. The differences between the groups with and without the

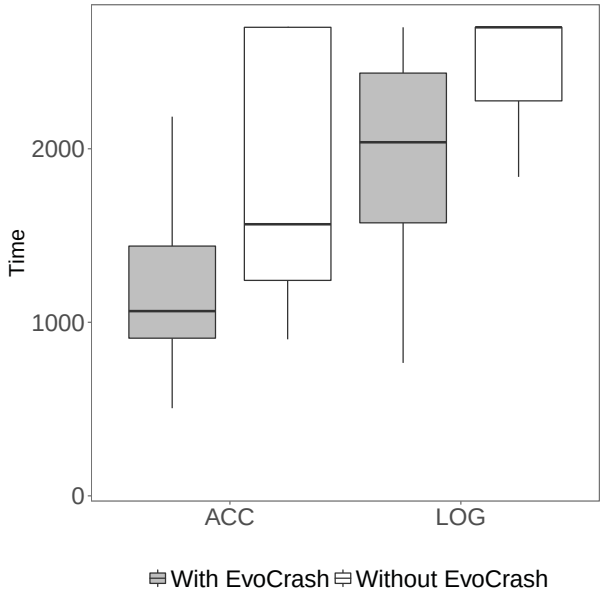

Figure 2.3: Amount of time participants took to perform each task, with and without the tests from EvoCrash.

EvoCrash test are also statistically significant according to the Wilcoxon test, which returns p-values of 0.03 and 0.04 for `ACC-48` and `LOG-47957`, respectively. Based on the results above, we conclude:

RQ$_6$: Developers using the tests from EvoCrash take significanlty less time when debugging, compared to those not using the EvoCrash tests.

2.5.7 Threats to Validity

In this section, we outline various possible threats to the validity of the controlled experiment we conducted.

Internal Validity. To reduce factors that could affect the causal relations under scrutiny, we randomly assigned the tasks to the participants. Regarding the ability of the participants in locating the defects and fixing them, it could be that not being familiar with the source code negatively affects the degree to which the participants were able

to locate and fix the defects. To mitigate this impact: i) We made sure Java-Doc documentation is available for the target projects to be debugged, and ii) We checked with the pilot studies whether the given time for each task was reasonable, and whether the available documentation was sufficient to perform them.

Moreover, we conducted the experiment remotely from the participants, which implies that they would do the experiment at their own discretion. Using the online platform, we made sure the participants are mandated to perform the tasks in the specified order, and within the specified time limit. In addition, the participants could only answer each follow up questionnaire after they had completed each task.

External Validity. One factor that could affect the generalizability of the study could be the student participants of the experiment. Different studies [124, 166] show that if students are familiar with performing the tasks of the experiment, then they would perform similar to participants from industry. Over 50% of the participants declared to have at least 2 years of industrial experience, and basic familiarity with the JUnit framework. In addition, by giving an introductory lecture we further tried to familiarize the students and thereby, mitigate possible threats to the generalizability of the experiment results.

Furthermore, we analyzed only two types of exceptions in the experiment. As described in Section 2.5.1, to select these types we considered a number of criteria, including how often they occur, the stack trace sizes, and whether they are overly complex or overly simple cases to debug. We deliberately opted for only two exceptions in order to i) maintain statistical power in the analysis, and ii) avoid introducing fatigue and learning effects to the participants.

Construct Validity. Threats to this type of validity concern the degree to which the conducted experiment measures what is intended to be measured. We used the online platform to measure the amount of time each participant took to complete the debugging tasks. Since the experiment was done remotely, we did not fully observe how the participants spent the debugging time they took. While by limiting the debugging time and providing the questions after each task was completed we tried to control the experiment flow, it is possible that the participants did not spend the entire time on the debugging tasks.

Conclusion Validity. We conducted the experiment with 35 master students. In the experiment, each task was performed by at least 16 students. While 16 is not a large number as for the size of each group, it still yields sufficient statistical power to assess the impact of using EvoCrash tests on the number of fixed bugs (when the test is useful for debugging), as well as the amount of time it takes to finish the debugging tasks. Regarding assessing the impact of EvoCrash tests on the ability of developers

in locating defects, our experiment shows preliminary results, and therefore indicates the need for further future investigation.

We support our findings by using appropriate statistical tests (namely: The Fisher's exact test, odds ratio measure, Wilcoxon Rank Sum test, and the Vargha-Delaney Â statistic) to assess the impact of using EvoCrash tests in debugging.

2.6 Discussion and Lessons Learnt

Interactive Search. It should be noted that since GGA strives for finding the fittest test case, thus discarding the ones with fitness > 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 [81], it could be possible that EvoCrash discarded tests with fitness between 0.0 and 1.0, 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 4.2.2), 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. 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. Therefore, if external files were to be provided by the bug reporters, then enabling developers to specify the external files could be another possible direction for the future work.

Extending Comparisons. Towards extending the empirical evaluation, we aimed at adopting the crash cases reported in [172] in order to make a larger comparison with JCHARMING. However, due to various reasons, ultimately we managed to adopt four cases to this end. While the new cases provide a bigger picture, we are still interested to expand the comparisons among the recent tools for automated crash reproduction. This aim would be facilitated if the tools become publicly available.

In addition, we acknowledge the need for extending the empirical evaluation of EvoCrash to crashes from recent industrial projects. In such projects scale, complexity, and type of the generated crashes may vary, which may indicate new research dimensions to consider for improving our search-based crash reproduction approach.

Controlled Experiment. To analyze the impact of EvoCrash tests in debugging, we selected two common exceptions in Java programs, `NullPointerException`, and `IllegalArgumentException`. This is while various types of exceptions may impose different levels of complexity in debugging, and thus, the impact of crash reproducing tests may vary in each case. Therefore, future studies could adopt more common exceptions in Java programs, and assess the impact of EvoCrash tests per exception type.

In addition, our experiment results showed that using *useful* EvoCrash tests helps developers fix bugs and take less time in debugging. While using such tests helped the participants locate the given defect, the observed impact was not statistically significant. To be able to locate the root cause of a given failure, having upfront understanding and knowledge about the defective source code may be another important factor that can impact the ability of a developer in localizing a given defect. Therefore, future studies may assess the impact of having up-front knowledge of source code and its correlation with using crash reproducing tests in debugging.

2.7 Conclusions

Several approaches to automated crash replication have been proposed to aid developers when debugging. However, these approaches report several challenges such as path explosion and handling environmental dependencies in practice. We propose a new approach, EvoCrash, to automated crash reproduction, via a Guided Genetic Algorithm (GGA). Our empirical evaluation on 54 real-world crashes shows that GGA addresses the path explosion problem. Furthermore, thanks to the mocking mechanisms in EvoSuite, some crashes involving environmental interactions were reproduced. However, handling environmental dependencies (such as content of a required file) remain to be a challenge for EvoCrash. We acknowledge the need for further empirical evaluations on more recent and industrial cases. The result of such evaluations may help identify the areas where we can improve our search-based crash reproduction technique.

In addition, we compare effectiveness and efficiency of EvoCrash with EvoSuite as a whole test suite generation approach to coverage-based test generation. Our results

confirm that the provided guidance in GGA is necessary for effectively and efficiently reproducing the crashes.

Moreover, we report from a controlled experiment with 35 master students in computer science, in which we assessed the impact of using EvoCrash tests in practice. Based on the results of the controlled experiment, we observed that: i) Our data regarding the impact of EvoCrash tests on the ability of developers in locating defects is preliminary. Therefore, our results show need for further future investigation in this regard. ii) Using a useful test from EvoCrash when debugging, developers can provide fixes more often, compared to when debugging without using such tests. Finally, iii) using EvoCrash tests reduces the amount of time developers take when debugging.

Algorithm 2.1 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  M_crash ⟵ identify public methods based on TC
3      k ⟵ 0
4      P_k ⟵ MAKE-INITIAL-POPULATION(C, M_crash, N)
5      EVALUATE(P_k)
6      // main loop
7      while (best fitness value > 0) AND (time spent < max_time) do
8          k ⟵ k + 1  // generate offsprings
9          O ⟵ ∅
10         while | O |< N do
11             p_1, p_2 ⟵ select two parents for reproduction
12             if crossover probability then
13                 o_1, o_2 ⟵ GUIDED-CROSSOVER(p_1, p_2)
14             else
15                 o_1 ⟵ p1
16                 o_2 ⟵ p2
17             O ⟵ O ⋃ GUIDED-MUTATION(o_1)
18             O ⟵ O ⋃ GUIDED-MUTATION(o_2)
19         // fitness evaluation
20         EVALUATE(O)
21         P_k ⟵ P_{k−1} ⋃ O
22         P_k ⟵ select the N fittest individuals in P_k
23     t_best ⟵ fittest individual in P_k
24     t_best ⟵ POST-PROCESSING(t_best)
```

Algorithm 2.2 MAKE-INITIAL-POPULATION

Input: Class under test C

Set of failing methods M_{crash}

Population size N

Result: An initial population P_0

```
25 begin
26     P_0 ⟵ ∅
27     while | P_0 |< N do
28         t ⟵ empty test case  size ⟵ random integer ∈ [1;MAX_SIZE]
29         // probability of inserting a method involved in the failure
30         insert_probability ⟵ 1/size
31         while (number of statements in t) < size do
32             if random_number ⩽ insert_probability then
33                 method_call ⟵ pick one element from M_crash
34                 // reset the probability of inserting a failing method
35                 insert_probability ⟵ 1/size
36             else
37                 method_call ⟵ pick one public method in C
38                 length ⟵ number of statements in t  // increase the probability of inserting a failing method
39                 insert_probability ⟵ 1/(size − length + 1)
40             INSERT-METHOD-CALL(method_call, t)
41         P_0 ⟵ P_0 ⋃ t
```

Algorithm 2.3 GUIDED-CROSSOVER

Input: Parent tests p_1 and p_2

Set of failing methods M_{crash}

Result: Two offsprings o_1, o_2

```
42 begin
43     size_1 ⟵| p_1 |
44     size_2 ⟵| p_2 |
45     // select a cut point
46     μ ⟵ random number ∈ [0; 1]
47     // first offspring
48     o_1 ⟵ first μ × size_1 statements from p_1
49     o_1 ⟵ append (1 − μ) × size_2 statements from p_2
50     CORRECT(o_1)
51     if o_1 does not contain methods from M_crash then
52         o_1 ⟵ clone of p_1
53     // second offspring
54     o_2 ⟵ first μ × size_2 statements from p_2
55     o_2 ⟵ append (1 − μ) × size_1 statements from p_1
56     CORRECT(o_2)
57     if o_2 does not contain methods from M_crash then
58         o_2 ⟵ clone of p_2
```

Algorithm 2.4 GUIDED-MUTATION

Input: Test $t = \langle s_1, \ldots, s_n \rangle$ to mutate

Set of failing methods M_{crash}

Result: Mutated test t

```
59 begin
60     n ⟵| t |
61     apply_mutation ⟵ true
62     while apply_mutation == true do
63         for i =1 to n do
64             ϕ ⟵ random number ∈ [0; 1]
65             if ϕ ⩽ 1/n then
66                 if delete probability then
67                     delete statement s_i
68                 if change probability then
69                     change statement s_i
70                 if insert probability then
71                     insert a new method call at line i
72         if t contains method from M_crash then
73             apply_mutation ⟵ false
```