Syntest-ACR: Automated Crash Reproduction for JavaScript

Philip Oliver, Jens Dietrich, Craig Anslow, Michael Homer
Victoria University of Wellington
New Zealand

***Abstract*—Automated Crash Reproduction (ACR) is an area of software testing research that aims to reproduce software crashes to improve developers' ability to debug programs. There has been little progress in applying ACR techniques to JavaScript, as the highly dynamic nature of JavaScript poses challenges for program analysis and synthesis. We present *SynTest-ACR*, the first tool for ACR in JavaScript, applying artificial intelligence techniques to evolve suitable reproduction cases. We have evaluated *SynTest-ACR* against the *CrashJS* dataset consisting of 453 crashes. As a baseline, we ported the state-of-the-art search-guiding fitness function from EvoCrash for Java, finding that it performs much worse when applied to JavaScript programs, and through comprehensively designing and evaluating alternative fitness functions more suitable for JS ACR we obtain an 18.9% increase in reproduction rate over this baseline for Syntest-ACR.**

I. Introduction

Automated Crash Reproduction (ACR) is the process of automatically generating test cases that reproduce a crash, usually by matching a stack trace produced by the initial crash [1]. The main goal of ACR is to provide developers with tests that can be used to debug the faulty program. Initial research in ACR focused on static analysis techniques such as symbolic execution to reproduce crashes [1]. More recently, there has been an increased focus on evolutionary techniques, particularly genetic algorithms [2]–[4]. There have also been advances in natural language processing and Large Language Models (LLMs) for bug reproduction [5], [6]. These bug reproduction techniques differ from ACR approaches as they do not aim to reproduce a stack trace [5]. Genetic approaches perform reasonably well, often using fitness functions derived from the work of Soltani *et al.* for EvoCrash [2], [7]. A fitness function guides an evolutionary search by measuring how well a candidate solution meets a goal, in this case the goal of reproducing a specific stack trace. However, to the best of our knowledge, there has been no research into applying these techniques to JavaScript, a language that has had substantial adoption outside its original browser-based embedded usage in the last decade.

The dynamic nature of JavaScript and its dynamic type system present several additional complexities above other languages that generally make tasks like program analysis and synthesis tasks challenging [8]–[11], including:

- A range of type coercions that both inhibit early detection of errors and result in ill-typed values being passed as arguments or stored.
- JavaScript objects are always meta-mutable — they can have fields and methods added or removed at any time, including to whole inheritance trees at once — and it is common to do so in real programs.
- Fields and methods of any object can not just be accessed by syntactic identifiers, but by indexing into them with a run-time string — again, legitimate uses of this are very common.
- The library ecosystem and culture encourages both broad and deep dependency trees, wherein having hundreds of dependencies for a relatively small program is not unusual, and so there are many more external API boundaries.
- Limited encapsulation boundaries: internal methods and fields are generally accessible, and often *are* accessed even by widely-used libraries.
- Very heavy use of first-class anonymous functions, which can convey little information in stack traces and are often inaccessible to test cases.
- No static typing, but cultural expectations of compliance with documented preconditions that are not enforced.
- The types of exceptions convey little information, with many distinct problems resulting in one of a small number of defined categories, such as "TypeError".

In other languages explored by ACR tools, such as Java and even Python, many of these are not possible or not common, or result in early and information-rich errors that can be exploited in ACR. JavaScript presents a steeper hill to climb but if anything *more* motivation to do so: precisely because of the high propensity for low-information errors arising from complex cross-component interactions, human-in-the-loop debugging of program crashes can be tedious and prolonged, while an ACR tool may be able to narrow the problem significantly relatively quickly.

As a part of this paper, we present the following major contributions:

- SynTest-ACR: The first ACR tool for NodeJS JavaScript programs and an evaluation on CrashJS.
- An evaluation of fitness functions for JavaScript ACR, some of which outperform the current state-of-the-art on JavaScript programs.

This work was supported by the New Zealand Science for Technological Innovation National Science Challenge through project RTVU2001 "Automating Automated Testing". The work of the first and second authors was supported by a gift by Oracle Labs Australia.

CrashJS [12] is the only existing dataset of stack traces and the JavaScript programs that produced them. For SynTest-ACR, we build on the existing work of Olsthoorn *et al.* [13] and Stallenberg *et al.* [14] on the SynTest framework. We have modified the SynTest JavaScript tool to reproduce crashes on NodeJS programs, thus creating the first ACR tool for JavaScript programs.

We first implemented the state-of-the-art EvoCrash fitness function as a reference point, and after observing its much poorer performance on JavaScript programs we explored other mechanisms to drive the search that could potentially perform better on this language. We identified 22 candidate fitness function terms, each of which measures some property of a program's execution against the stack trace being reproduced — for example, whether a certain line was reached — and serves as a building block in combination with others to form a full measurement of the reproduction case's suitability.

We used these terms to answer several research questions:
RQ1: How do discrete and continuous versions of terms perform?
RQ2: How much does the presence of coverage affect an ACR search?
RQ3: What performance do fitness terms have for ACR?
After evaluating these fitness terms, we selected 6 terms with high-performing capability of reproducing crashes and combined these to create fitness functions with 3 terms. This resulted in 20 fitness functions we evaluated to answer several more research questions:
RQ4: What combinations of fitness terms create a good fitness function?
RQ5: Do single or multi-objective searches result in higher rates of ACR?
RQ6: How do the reproduction rates compare to the state-of-the-art?

We found that we could reproduce more crashes with our fitness functions (25.2%) than the current state-of-the-art fitness function for ACR from the work of Soltani *et al.* on EvoCrash [15] (21.2%), with an improvement in partial reproductions of 6.6%.

II. Background and Related Work

There are several existing ACR tools for Java and Python which make use of fitness functions for crash reproduction. Most continue the work of Soltani *et al.* on EvoCrash [2], [7]. EvoCrash is an ACR tool for Java that uses EvoSuite as a test generator [16], [17]. As a part of EvoCrash, Soltani *et al.* produced a fitness function to guide the tool to reproduce crashes [15].

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

$d_s(t)$ is a measure of how close the test case (t) is to the location of the crash and is calculated using the approach level and branch distance. $d_{except}(t) \in [0, 1]$ is a boolean flag that measures if the target exception has been thrown. $d_{trace}(t)$ is a measure of the similarity between the target and generated stack traces. The fitness function gives values between 0 and 6, with a 0 value occurring only if the crash has been successfully reproduced. The intention of this weighted fitness function is to bias the search towards the correct crash location in the system under test, with the correct exception and stack trace similarity being less important in the EvoCrash approach.

In 2020, Derakhshanfar *et al.* presented Botsing [3]. Botsing, like EvoCrash, is a search-based crash reproduction tool for the Java programming language built on top of EvoSuite. Derakhshanfar *et al.* identified the work of Soltani *et al.* [2] as showing the ability of evolutionary search for crash reproduction. Botsing uses an evolutionary search to produce a test case replicating the crash behaviour [3]. Three fitness functions are implemented in Botsing to drive the evolutionary search. The single objective function from the work of Soltani *et al.* on EvoCrash is used as the default fitness function. The second fitness function provides a multi-objective approach which divides the first fitness function into separate parts as sub-objectives of the evolutionary search, also previously tested in EvoCrash. The final fitness function is one created by Rößler *et al.* [18] and iteratively checks that the test covers the frames in the stack trace and then checks the type of the exception which is thrown. To evaluate Botsing, the authors used JCrashPack [19] as a benchmark. Botsing was run on this benchmark and successfully reproduced 66 of the 124 (53.2%) crashes without using the seeding strategies and 70 (56.5%) with the model and test seeding mechanisms.

Beacon is a tool presented at the 2021 Workshop on Search-Based Software Testing and is heavily inspired by EvoCrash. Like EvoCrash, Beacon uses a guided genetic algorithm to reproduce crashes using the stack trace [4]. However, unlike EvoCrash, Beacon is implemented for the Python programming language. The main difference between Java and Python is that Python a dynamically typed programming language. As a part of developing Beacon, Bergel and Muñoz developed a genetic algorithm framework for Python called *Genyal*. This framework allowed Beacon to use a genetic algorithm without requiring the implementation of the algorithms in Beacon itself. In the usage of Genyal, the fitness function from EvoCrash has been adapted for Beacon and is given by the formula:

$$F_I = 2t_{ex} + t_{arg} + 2t_{fn}$$

F_I is the fitness of an individual, t_{ex} is 1 if the correct exception is thrown, t_{arg} is 1 if the exception message contains the correct string, and t_{fn} is 1 if a function is present in the crash stack trace. Genyal uses standard crossover, selection, and mutation operators for the genetic algorithm. A key difference to EvoCrash is that Beacon does not aim to reproduce the full stack trace directly, only the same exception at the same location.

Some of the most recent developments in ACR and bug localisation have been in the use of Large Language Models (LLMs) to generate test cases [5], [20]–[23]. LLMs are a form of artificial intelligence that use neural networks to perform natural language processing tasks such as classification or text generation. LLMs predict the next most probabilistic word for

text generation based on the data used in training the model. This can result in LLMs generating text which may syntactically make sense, but may present incorrect information. There are several approaches used in the literature applying LLM prompt engineering for test generation and bug reproduction. For example, Feng and Chen use a multi-step process in AdbGPT, where they use an LLM to extract information from a bug report and an Android APK file to identify available actions, action primitives, in-context learning, and chain-of-thought [23]. Available actions relate to Android gestures such as *tap* and *scroll*, while action primitives provide more information, combining these available actions with a component or a direction, and the LLM was fed a sample of bug reports and manually reproduced test cases to train on some expected outcomes. Another approach, Libro, by Kang *et al.* is much simpler [5]. Kang *et al.* construct a Markdown document which provides the title and description from a bug report and asks the LLM to "provide a self-contained example that reproduced this issue.", and provide the start of a test case in an open code block for the LLM to extend.

The SynTest framework is a series of tools for search-based test generation in JavaScript-based languages developed by the Software Engineering Research Group at the Technical University of Delft [13], [14]. Originally published as a tool for generating tests for Solidity smart contracts [13], SynTest has also been expanded to include a tool for generating tests for JavaScript programs [14]. As a part of SynTest-JavaScript (SJS), an unsupervised method for probabilistic type inference was created to allow for automatic test generation in the dynamically typed environment of JavaScript.

SJS has three phases to generate tests for any given test subject [14]. The first phase is static analysis. During this phase, SJS analyses the test subject and its dependencies to build Abstract Syntax Trees (ASTs). These ASTs are parsed to find identifiers and literals, which are then used to find relations between these and other language features such as classes, interfaces, and objects. These relations are used to infer type information. For example, if an identifier is assigned as a `string` literal, it can usually be inferred that the identifier is of type `string`. Stallenberg *et al.* defined 75 relations in 15 categories: *primary, left-hand side, increment/decrement, unary, arithmetic, relational, equality, bitwise shift, binary bitwise, binary logical, ternary, optional chaining, assignment, comma,* and *function*.

The second phase SJS uses is the novel unsupervised static type inference method. For literal elements, the process is as described before as the type can be trivially inferred. The process is more complex for other elements and requires consideration of all relations present for the element. A type score is assigned based on the number of hints present within the relations found in the first phase. For a given type, the score will be equal to the number of hints for that type there are within the relations divided by the sum of all scores for that element. The higher the result, the more likely the type to be used in a generated test case.

$$p(element, type) = \frac{score(element, type)}{\sum_{type_i} score(element, type_i)}$$

The final phase is test case generation. In this phase, SJS uses an evolutionary approach to generate test cases using DynaMOSA [24]. The type inference algorithm in stage 2 is incorporated into DynaMOSA in two ways: proportional type sampling, and ranking [14]. Proportional sampling allows different test cases in the genetic algorithm's population to have different types for the same input variable, proportional to the probability found in stage 2. Ranking selects only the type with the highest probability for each input variable. Once created, tests are evaluated by executing the tests and measuring the distance to unreached branches, calculated using approach level and normalised branch distance.

III. Methodology

To the best of our knowledge, there has been no research into the application of ACR principles to JavaScript programs. Most existing ACR tools for other languages are built with a test generation tool as the primary component. We elected to follow this approach and selected SynTest-JavaScript as the JavaScript test generation tool on which to base our approach.

We set out the following steps to construct a suitable ACR tool for JavaScript: 1) Selection of a dataset; 2) Creation of an ACR tool for NodeJS; 3) Identifying terms for use in fitness functions; 4) Evaluation of these terms; 5) Creation of more complex fitness functions out of them; 6) Evaluation of these fitness functions; 7) Comparison of the best-performing fitness functions with other tools. These are addressed in order below. The experimental code, results, and Syntest-ACR tool are available through https://doi.org/10.5281/zenodo.16345353.

A. CrashJS

We selected CrashJS as our dataset for this evaluation [12]. CrashJS is a collection of 453 NodeJS crashes across several areas and projects. There are four sets of crashes within CrashJS which are separated based on where the crashes were collected: 71 from GitHub, 17 from SecBench.js [26], 90 from BugsJS [27], and 275 from SynTest-JavaScript [14]. All 453 crashes were used as a dataset to evaluate fitness functions.

B. SynTest-ACR

To develop an ACR tool for JavaScript programs we extended SynTest-JavaScript to leverage the implementation of the genetic algorithm and test generation code already present. This involved implementing scaffolding around SJS to handle a stack trace–focused approach. Within the SynTest architecture, we created a library (`crash-reproduction-setup`) and a tool (`crash-reproduction`). The SynTest-ACR tool we developed mostly provides an interface to present crashes to the processes already implemented within the SynTest Framework. The SynTest-ACR runtime consists

of a stack trace (ST), crash environment (E), and optional coverage targets.

$$Runtime := (ST, E, Coverage)$$

C. Experiment Runners and Architecture

Due to the length of several of the experiments undertaken for this research, we utilised a grid-computing system implemented at our institution.

The grid consists of Linux machines with 64-bit AMD processors and at least 8GB of RAM. As the majority of machines in the grid are desktop computers there are some issues which can occur when power or network access is interrupted, and we also discovered that a small number of machines did not have NodeJS installed. Some data will be missing due to these limitations, but given the overall number of runs we regard these lost jobs as negligible.

For the experiments, SynTest used the default configuration. During the search, SynTest uses a timeout of 600 seconds, an allowance for 20 iterations of the evolutionary algorithm, which is DynaMOSA, and a timeout for each generated test of 5000 milliseconds. A seed is not specified, so each search will be randomised. SynTest is configured to extract and use constants found in the target program, with a probability of these being used instead of a random value of 0.5. We configured SynTest to allow illegal values, as these may be required to cause a crash. The primary evolutionary parameters were set at the default SynTest values: crossover, 0.7; mutation, 0.8. We do not explore tuning these values or configurations, rather leaving this to future research.

IV. Fitness Function Terms

A fitness function *term* maps a SynTest-ACR runtime to a value between 0 and 1 based on a specific aspect of a candidate crash reproduction's suitability.

$$fterm : Runtime \rightarrow [0, 1]$$

These terms are miniature fitness functions, intended to be composed with others into *composite fitness functions* (*fitness functions* for short), to be used for guiding a search to reproduce a crash. For this paper, we have studied fitness functions with a range of $[0, 3]$. We define fitness functions as a weighted sum of several fitness terms.

$$ffunc(Runtime) := \sum_{i=1}^{N} \alpha_i \cdot fterm_i$$

While terms alone often do not provide enough information to fully reproduce a crash, they do provide useful information in the process of reproducing a crash, such as identifying a reproduced exception or if an execution has reached a particular line. Analysing these terms will allow us to identify which terms are suitable for ACR, and a compositional approach allows us to make empirical decisions about which elements provide value in driving evolutionary ACR. We will derive our more complex fitness functions by evaluating the performance of the smaller terms on ACR for NodeJS programs. We also assess the impact of coverage metrics on each function term.

We further distinguish between *discrete* and *continuous* fitness terms. Discrete function terms have a finite domain as a range, while continuous terms map into an infinite domain (defined by a numerical interval). For example, matching an exception message could be a simple string comparison where the result is either `true` and `false` (or 0 and 1). This would be modeled as a discrete term. On the other hand, continuous string matching would make use of a similarity metric such as Levenshtein distance to allow for a range of values between 0 and 1.

A. Experiment

We implemented 22 fitness function terms to identify different aspects of post-crash analysis with suitable results for use in a larger fitness function. While several of these terms will be unsuitable for ACR, we include them for completeness and as they may introduce diversity to a search, potentially leading to different crashes being partially reproduced.

The fitness function terms we investigated were:

- T1: Correct exception raised
- T2: Correct exception raised on the correct line
- T3: Wrong exception raised on the correct line
- T4: Correct exception raised in the correct function
- T5: Wrong exception raised in the correct function
- T6: Correct exception raised from neighbouring function
- T7: Wrong exception raised from neighbouring function
- T8: Correct exception raised in the wrong function
- T9: Reached the line of the exception without crashing
- T10: Wrong exception with partial stack trace match
- T11: Executed some of the call hierarchy without crashing
- T12: Reached the line of the stack trace entry
- T13: Executed functions in the stack trace with no crash
- T14: Stack trace matched with fuzzy line numbers
- T15: Executed lines near a call to a stack trace function
- T16: Entered a branch containing a call to the stack trace
- T17: Entered a branch near a call from the stack trace
- T18: Called N functions from the stack trace
- T19: Executed N lines before a call to a stack trace function
- T20: Correct exception raised with partial stack trace match - implementation 1
- T21: Exception raised and the stack trace exception line is covered
- T22: Correct exception raised with partial stack trace match - implementation 2

Terms T1–3, T10, T20, T22 are disaggregated from the EvoCrash fitness function [2], [7], while T1, T4–5, T8 derive from Beacon's [4]. T6, T7, and T14 are generalisations that we believe to be novel, but obvious. T9, T11–13, T15–19, T21 are measures that make use of dynamic tracing information to collect analogous information to the other terms even in non-crashing cases, something possible in our system but not EvoCrash or Beacon. Terms which check for an exception

ensure the generated exception matches the target exception for a crash. Terms which check for the correct line ensure the generated exception is thrown from the same line as the target exception. A neighbouring function is one function above or below the correct function in the code file. Stack trace match is a distance metric to assess stack trace similarity. Fuzzy line number matching assesses if stack frames within the generated exception are within 10% of the code file's length of the target crash's stack frames. Stack trace entry is the program entry frame for the stack trace. T11 assesses whether any part of the stack trace has been executed at all. T20 and T22 differ only in their implementation; T22 further filters functions in the stack trace to ensure there are no test fixture functions, anonymous functions, or other erroneous information.

Using these fitness function terms we ran an experiment to discover the capability of these terms. To evaluate the capability of these terms, we included two post-search objectives in determining whether an execution had reproduced a crash: the EvoCrash fitness function, and a function that counts the stack trace frames that have been covered. These functions are only evaluated when the tool creates the final test suite, and do not steer the search.

During this first experiment, we also considered the impact of discrete and continuous fitness functions and enabled and disabled coverage targets within SynTest. Several terms we implemented for the experiment can use either discrete or continuous methods. For example, a term evaluating if a line is executed could produce a boolean (true 0, 1 false) or a continuous distance measure in $[0, 1]$ to the correct line. Terms with multiple branches (e.g. correct line and correct exception) may give 0.5 when matching one but not the other.

Another opportunity for discrete and continuous terms is matching exception messages. Many JavaScript exceptions embed the meaningful information into the textual message of a catch-all error class, so using only the error type is insufficient, and messages may contain embedded argument values or other varying details. An exception could then be matched using an equality operator, or fuzzy string-matching using Levenshtein distance for a more continuous result [28]. Fuzzy matching may be able to guide the search to generate the argument required to cause the expected crash.

We expect that the continuous terms are more likely to produce a good result as they are more capable of guiding the search for a reproducible crash. As described in the literature, discrete terms can lead to the flag problem, where the search space is restricted to two plateaus with no gradient to direct the search [29]. We implemented discrete and continuous versions for our T1–T8, all relating to exceptions and their line numbers. The remaining terms (T9–T22) are specifically discrete or continuous by design and are not considered in the comparisons made between discrete and continuous functions.

The coverage targets we have implemented are simply each function or file from the target stack trace. Using these additional targets can allow SynTest to optimise the search to reach these locations, but this may make identifying terms with the best results for ACR difficult. As such, we evaluated the capability of each term with and without coverage metrics enabled for the search.

The first aspect of the fitness function terms we evaluated was the performance difference between discrete and continuous term values to identify how these versions perform. The following experiments also present the analysis of terms with and without coverage targets enabled.

B. Discrete vs Continuous Terms

Tables I, II, III, and IV present different aspects of Terms 1 to 8, for which we evaluated discrete and continuous versions. The results presented are counts of targets covered rather than specific crashes. For the majority of crashes, there is only one suitable target explored. However, some crashes whose traces cross multiple files may have multiple targets in the search, as a test case may enter the stack trace at any point. When this occurs, one target typically has significantly better results. Furthermore, SynTest is unable to explore all crashes in CrashJS. For some crashes, SynTest is unable to create an initial population, while for others, there can be errors in the search, leading to no result for that crash. These tables also do not present non-matched crashes. For these reasons, the results in these tables do not sum to the 453 crashes present in CrashJS. As we are mostly interested in if a crash has been fully reproduced, our comparisons are between the fully matched results unless we explicitly state we are comparing the partial matches.

TABLE I
DISCRETE AND CONTINUOUS FITNESS TERMS PERFORMANCE ON CRASHJS WITHOUT COVERAGE

	Discrete		Continuous	
Function Term	Match	Part. Match	Match	Part. Match
T1	244	0	249	209
T2	247	255	245	254
T3	205	232	136	356
T4	226	315	260	270
T5	135	411	5	495
T6	181	297	279	232
T7	287	208	210	274
T8	0	331	0	500

Table I summarises the results for each of the eight fitness terms' performance over CrashJS with no coverage targets in SynTest enabled. The first observation from this table is that T1 (right exception raised) has a significant difference in partial matches between discrete and continuous versions. This is due to the discrete version simply checking for the correct exception message, with no possibility of partial matches, while the continuous version uses an edit distance instead. The other, multi-branch, terms can produce partial matches even in the discrete condition when one half matches without the other. For example, a term such as T2 "correct exception raised on correct line" will partially match if it finds an incorrect exception on the right line, and a full match requires that both the exception and line match.

The continuous version of T3 (wrong exception raised on the right line) tends to perform *worse* than the discrete version.

In the continuous version, exceptions with a partial match or small difference in line will be explored further, even once the correct line or exception has been reached, while in the discrete version the search will tend to solidify around exactly-matching lines or exceptions once landed on and not expend time looking nearby, as the dropoff from moving to an incorrect line is more drastic.

TABLE II
DISCRETE AND CONTINUOUS FITNESS TERMS POST-SEARCH EVOCRASH PERFORMANCE ON CRASHJS WITHOUT COVERAGE

	Discrete			Continuous		
Function Term	f=0	0¡f¡1	f≥1	f=0	0¡f¡1	f≥1
T1	55	164	239	63	158	237
T2	68	179	212	64	181	213
T3	6	56	352	7	61	386
T4	83	194	216	79	206	242
T5	49	152	295	24	111	356
T6	43	138	289	40	161	309
T7	7	55	388	10	48	388
T8	15	91	212	22	117	354

Table II shows the score from the EvoCrash fitness function on the test suite generated using each function term to guide the search. The EvoCrash fitness function is absent during the search and does not influence the search, so we use it only to measure whether a crash has been reproduced. $f = 0$ denotes fully reproduced crashes, $0 < f < 1$ denotes crashes which have the correct exception on the correct line but do not have a full stack trace reproduction, and $f \geq 1$ denotes non-reproduced crashes. As before, these values are totals of targets, rather than crashes. Some crashes have multiple targets, while other crashes have no targets executable by SynTest or targets which are only sometimes executable. Due to these reasons, the totals for each of these experiments do not total the 453 crashes in CrashJS.

T4 (right exception in right function) has the highest number of crashes reproduced for both discrete and continuous versions by a reasonable margin. Interestingly, the performance for T4 drops for the continuous version, although by a small amount potentially from the stochastic nature of the search.

T1 (right exception raised) and T2 (right exception raised on right line) have the next highest levels of full crash reproductions, including matching stack traces. These two terms also have high levels of partially reproduced crashes. For the discrete terms, T2 can fully reproduce more crashes than T1. This likely occurs as T2 has additional information about the crash location, and thus will guide the search more efficiently than simply searching for the correct exception.

T5 (wrong exception in the correct function) and T6 (correct exception in neighbouring function) both perform well for the discrete versions. T5 drops significantly for the continuous version, likely due to the term being more adept to steering the search. As T5 is actively searching for the wrong exception, the continuous version will steer the search away from reproducing a crash.

RQ1: Continuous fitness terms tend to guide the search towards their target more successfully than discrete terms. However, when verified against an existing reproduction metric, the numbers of crashes reproduced are similar.

C. Coverage in ACR Search

TABLE III
DISCRETE AND CONTINUOUS FITNESS TERMS PERFORMANCE ON CRASHJS WITH COVERAGE

	Discrete		Continuous	
Function Term	Match	Part. Match	Match	Part. Match
T1	235	0	252	178
T2	246	299	242	309
T3	209	335	136	411
T4	202	353	251	340
T5	73	176	6	602
T6	162	702	276	287
T7	291	246	208	299
T8	0	572	0	600

Table III summarises the same results as Table I, except with SynTest coverage targets enabled. Coverage targets are a set of functions extracted from the system under test, each added as a separate target for the search using the DynaMOSA algorithm [24]. Most of these results are similar to those without coverage: terms which are better as discrete or continuous remain that way with the coverage targets enabled. While the number of matched executions remains similar for discrete terms, the number of partial matches increases significantly with the addition of coverage targets, as coverage targets allow the search to improve without necessarily improving the score from the function term.

There are significant decreases in the number of matches for T5 (wrong exception in correct function). T5 likely decreases as the coverage targets aim to steer the search towards exploring files and functions in the stack trace. This means that T5 has less control over the search, and the coverage targets are steering the search towards the correct exception rather than providing the ability for T5 to explore wrong exceptions.

T8 is the most striking result from runs with and without coverage targets enabled, as there are 0 matches. This is not unexpected and means the correct exception has never been raised in the wrong function. As mentioned earlier, JavaScript exceptions can contain specific information in the error message which would be unlikely to occur elsewhere.

TABLE IV
DISCRETE AND CONTINUOUS FITNESS TERMS POST-SEARCH EVOCRASH PERFORMANCE ON CRASHJS WITH COVERAGE

	Discrete			Continuous		
Function Term	f=0	0¡f¡1	f≥1	f=0	0¡f¡1	f≥1
T1	69	149	262	68	164	300
T2	70	176	295	66	176	305
T3	31	77	328	29	87	428
T4	78	184	183	77	208	305
T5	73	176	234	53	137	417
T6	50	111	231	61	165	336
T7	26	405	102	29	69	406
T8	35	142	276	36	155	409

Table IV shows the performance of the EvoCrash fitness function on the test suite generated, similar to Table II but with coverage targets enabled. Unsurprisingly, significantly more crashes are reproduced for function terms which perform poorly without coverage enabled. As discussed previously, the coverage targets are likely overpowering the fitness terms and guiding the search to reproduce more crashes. This shows that coverage helps improve crash generation for terms that do not generally perform well for ACR. However, the number of crashes reproduced for terms which do perform well, such as T2 (right exception on right line), shows little difference with coverage enabled. The number of partial matches ($0 < f < 1$) increases with coverage enabled. Similar to the increase in partial matches for fitness terms, this likely occurs due to a more diverse population of tests.

When comparing Table II and Table IV, T1 and T2 perform similarly regardless of coverage targets. As the coverage targets add targets only present within the expected stack trace, this effectively will provide the search with guidance to find the location of the crash, much like T2 does.

Interestingly, T4 (right exception in right function) performs worse with coverage enabled, albeit by a small amount. As T4 aims to find the correct exception on the correct line, it is possible that the increased population diversity introduced by coverage targets takes computation time away from the tests which could evolve to reproduce the crash.

RQ2: Coverage tends to improve poor-performing fitness terms, but for high-performing fitness terms, coverage does not tend to improve the performance of the terms.

D. Function Term Suitability for ACR

Table V shows the results of matches for each term. These results are for the discrete versions of the terms and with coverage targets disabled. To be good for crash reproduction, a term must both expose a crash (good EvoCrash score), and also identify that it has done so (good term score). That is, it must be able to guide the evolution process effectively by pulling the set of generated tests in its direction, and that direction must result in crashes that resemble the target. Analysing these results, we can draw some conclusions about the ability of these fitness function terms' performance on the CrashJS dataset. Terms with a high proportion of 0 values include T2 (right exception on the correct line), T4 (right exception in right function), T20 (correct exception raised with partial stack trace match - implementation 1), T22 (correct exception raised with partial stack trace match - implementation 2). As T22 significantly outperforms T20 and these are two implementations of the same task, we exclude T20 due to time constraints.

Other terms with reasonable results could be used to guide a search. These may steer towards the right place, but not produce the crash itself, or towards the exception but not the location. T1 (right exception raised), T6 (wrong exception raised on right line), and T21 (exception line covered in exception) are promising here. As T1 is included in T2 and T2 outperforms T1, we exclude T1 due to time constraints. Including T1 and T20 in the combinatorial process in the next section would result in 2.8 times as many combinations and significantly increase the run time with limited benefit.

TABLE V
EVOCRASH AND TERM RESULTS PER TERM. MATCHES (0 VALUE), PARTIAL MATCHES (GREATER THAN 0 AND LESS THAN 1), AND NON-MATCHES (GREATER THAN 1) ARE SHOWN.

	EvoCrash Res.			Term Res.		
Term	f=0	0¡f¡1	f≥1	f=0	0¡f¡1	f≥1
T1	55	164	239	244	0	259
T2	68	179	212	247	255	1
T3	6	56	352	205	232	2
T4	83	164	216	226	315	1
T5	49	152	295	135	411	2
T6	43	138	289	181	297	1
T7	7	55	388	287	208	1
T8	15	91	212	0	331	1
T9	10	44	386	133	0	317
T10	9	41	439	65	386	45
T11	10	53	390	428	29	86
T12	23	51	262	166	0	172
T13	14	50	385	457	32	76
T14	10	52	420	334	149	17
T15	8	58	408	330	0	162
T16	13	71	383	154	0	324
T17	14	56	419	191	0	306
T18	8	48	447	0	0	507
T19	8	47	407	0	0	474
T20	99	35	356	99	358	39
T21	41	122	323	307	0	189
T22	106	89	257	205	245	1

Separately, we also ran the EvoCrash fitness function on the CrashJS dataset to evaluate its performance on JavaScript programs. With no coverage, we found the EvoCrash function capable of reproducing 96 crashes, or 86 and with coverage. This reduction in performance likely occurs due to the diversity introduced by the coverage. Meanwhile, our T22 (right exception with partial stack match) reproduces 106 crashes without coverage enabled.

An important note is that the EvoCrash function only applies to executions that crash. At the same time, a significant number of the other terms we implemented can be evaluated on non-crashing executions. These results correlate reasonably strongly with our fitness term results, showing that SynTest-ACR can effectively reach the line where an exception is thrown, regardless of whether the execution crashes.

RQ3: We identify terms T2, T4, T5, T6, T21, and T22 as the best performing terms for ACR.

V. COMBINING FUNCTION TERMS

In this section, we explore the use of high-performing fitness terms we identified in §IV and their performance for ACR across CrashJS. The following experiments assess combinations of our best fitness terms to identify suitable fitness functions. We also explore the capability of single and multi-objective searches using these functions for ACR in NodeJS. Furthermore, we assess the performance of SynTest-ACR and functions against the state-of-the-art.

A. Methodology

The primary focus of this experiment was to evaluate combinations of fitness terms with high performance over the experiment in §IV. The terms we selected were those with the highest number of crashes fully reproduced, as evaluated by the post-search EvoCrash fitness function. These terms are:

- T2: Correct exception raised on the correct line
- T4: Correct exception in correct function
- T5: Wrong exception in correct function
- T6: Correct exception raised from neighbouring function
- T21: Exception raised, and exception line is covered
- T22: Correct exception with partial stack trace match

These terms were used in combination with two others to create fitness functions with every combination of three terms, resulting in 20 functions to be evaluated. As we discovered in §IV, continuous functions tend to perform better than discrete functions, and thus, we use the continuous versions of these functions for this experiment. We evaluated each function twice: once as a single-objective function and once as a multi-objective function. We use a similar setup for these experiments as the fitness term experiment. The only differences were in the implementation of single and multi-objective functions.

For a single-objective search within SynTest, the terms were combined using the sum of the results for each term, effectively creating a traditional fitness function.

For a multi-objective search, we included each of the terms as individual objectives for the search.

VI. Results

We will look at the following results tables to analyse the capability of our fitness functions. Table VI shows the performance of the EvoCrash fitness function post-search for each of the 20 fitness functions in single and multi-objective searches.

Table VI allows us to see which functions can successfully reproduce a crash as defined by the EvoCrash fitness function. As the EvoCrash function requires the stack trace to be perfectly matched for a 0 value, we could also consider fitness between 0 and 1 as reproductions. These reproductions are presented as partial reproductions in Table VI. However, we could also consider a higher threshold as a complete reproduction, given that some of the stack frames for a crash may not need strict reproduction to provide a developer with the location of the bug which causes the crash. Table VII shows the number of crashes partially and wholly reproduced considering different thresholds.

As we can see in Table VI, these functions are successful at reproducing crashes in up to 101 and 114 targets for single and multi-objective searches, respectively. Table VI highlights the highest-performing functions (those with either single or multi-objective reproductions of more than 100), regarding full reproductions, for single-objective search. Interestingly, all these high-performing functions include T22 (correct exception raised with partial stack trace match). Another observation here is that the performance of these fitness functions tends to suffer in single-objective search with the presence of T5 (wrong exception in correct function) and T6 (correct exception in neighbouring function). These tests respectively steer the search away from the correct exception and the correct location. In a single-objective search, the exploration value they bring is are unable to overcome the misdirection they enforce.

TABLE VI
Post-Search EvoCrash Results for each Function. Matches (0 value), Partial Matches (greater than 0 and less than 1), and Non-Matches (greater than 1) are Shown. Functions with greater than 100 target reproductions are highlighted

	Single Objective			Multi Objective		
Function	f=0	0¡f¡1	f≥1	f=0	0¡f¡1	f≥1
T2, T6, T4	48	178	258	84	220	186
T2, T6, T5	21	102	359	77	230	175
T2, T6, T21	56	159	207	81	221	174
T2, T6, T22	68	88	297	105	197	183
T2, T21, T4	73	202	206	80	133	181
T2, T21, T5	18	111	366	66	222	185
T2, T21, T22	100	97	296	110	201	183
T2, T22, T4	99	90	279	114	197	182
T2, T22, T5	20	101	343	109	203	178
T2, T4, T5	21	113	360	76	238	178
T6, T21, T4	59	172	254	81	216	182
T6, T21, T5	21	88	370	69	205	206
T6, T21, T22	67	93	279	108	178	196
T6, T22, T4	70	92	322	111	180	180
T6. T22, T5	24	98	364	99	146	199
T6, T4, T5	20	105	360	81	186	180
T21, T22, T4	101	80	281	113	185	196
T21, T22, T5	26	100	367	109	165	218
T21, T4, T5	23	116	331	83	218	192
T22, T4, T5	16	101	373	111	179	199

The large discrepancy in crashes reproduced between single and multi-objective searches is striking for the functions (T2, T22, T5), (T21, T22, T5), and (T22, T4, T5), all containing T5 and T22. As T22 searches for the correct exception and T5 for the wrong one, changes to the exceptions will result in an increase in distance for one of these terms and a decrease for the other. This direct conflict between the results prevents the function being able to guide the search effectively.

In contrast, these functions do not experience such a significant drop in performance for multi-objective search, as SynTest will optimise each objective separately, with different tests aiming to meet each objective. This means that tests for the other objectives guide the search to reproduce these crashes, while tests for T5 are largely irrelevant, although early in the search they may cross over successful mutations.

Considering Table VII, with a sliding tolerance of acceptable EvoCrash crash reproduction fitness, the difference between single and multi-objective search becomes more pronounced. As the threshold of accepted reproductions increases, significantly more crashes become reproduced. However, as the tolerance increases, so does the gap between the number of reproductions for single and multi-objective searches. This indicates that multi-objective search may be significantly more useful for ACR, where even reproductions with a non-zero

TABLE VII
NUMBER OF REPRODUCTIONS USING POST-SEARCH EVOCRASH EVALUATION CONSIDERING A SLIDING SCALE OF REPRODUCTION VALUES.

	Single Objective			Multi Objective		
Function	0.25	0.5	0.75	0.25	0.5	0.75
T2, T6, T4	48	88	206	84	141	285
T2, T6, T5	21	36	111	77	121	282
T2, T6, T21	56	89	198	81	130	280
T2, T6, T22	68	84	149	105	137	281
T2, T21, T4	73	117	254	80	128	284
T2, T21, T5	18	39	117	66	107	263
T2, T21, T22	100	115	184	110	145	286
T2, T22, T4	99	106	177	114	148	289
T2, T22, T5	20	37	110	109	139	285
T2, T4, T5	21	43	121	76	132	292
T6, T21, T4	59	91	213	81	137	277
T6, T21, T5	21	38	98	69	110	252
T6, T21, T22	67	86	151	108	142	267
T6, T22, T4	70	86	153	111	147	271
T6. T22, T5	24	44	112	99	123	229
T6, T4, T5	20	44	116	81	129	254
T21, T22, T4	101	109	175	113	143	277
T21, T22, T5	26	39	113	109	136	252
T21, T4, T5	23	48	125	83	134	278
T22, T4, T5	16	35	106	111	148	269

fitness value may provide information a developer can use to debug and localise a fault within the crashing software.

For the EvoCrash fitness function, values between 1 and 0 are a distance measure of how well the result stack trace matches the expected. For example, if a stack trace is ten frames long and eight of these frames are covered, the distance will be significantly closer to 0 than if three are covered. This effectively means that EvoCrash fitness values between 0 and 1 measure how deep the reproduction is and how helpful the generated test may be to a developer. Suppose one frame is covered, but the correct exception has been thrown in the correct location. In that case, the generated test will not provide the developer with helpful information, as the bug that caused the crash may be several frames higher in the stack trace. Conversely, a stack trace may not need to be entirely reproduced for the bug to be captured in the generated tests and, therefore, to provide value to a developer. While we do not look much into this, we identify this as an area for future research and hypothesise that requiring a full reproduction of a stack trace may be necessary to provide beneficial insight for developers looking to debug their programs.

RQ4: Fitness functions which contain T22 reproduce the largest number of crashes. In combination with other terms T2, T4, and T21, the number of crashes reproduced remains high, with combinations of these terms proving best at reproducing crashes.

RQ5: Functions with low single-objective scores tend to perform significantly better in multi-objective configurations as terms which degrade the search can more effectively be ignored. For suitable fitness functions, there is still an increase in reproduced crashes, albeit by a much smaller margin.

A. Comparison to Other Tools

This section provides our analysis of the capability of SynTest-ACR against existing state-of-the-art tools. One of the metrics Soltani *et al.* used to evaluate the performance of EvoCrash on JCrashPack was the number of crashes where EvoCrash could reproduce at least one stack frame. Table VIII presents our findings of the number of stack frames reproduced and the number of crashes with at least one frame reproduced for each fitness function. EvoCrash could reproduce a frame in 87 of the 200 crashes, or 43.5%, in JCrashPack. Our best functions (T2, T22, T4) and (T21, T22, T4) reproduce a single frame in 163 (35.0%) and 183 (40.4%) of the crashes in CrashJS, respectively. While we do not consider counting a single frame reproduction as a suitable measure of a crash reproduction, it is interesting that SynTest-ACR can reproduce a similar percentage of frames as EvoCrash-Java, as this is a metric that has been used in the literature previously.

TABLE VIII
RESULTS FOR FRAMES COVERED. TOTAL FRAMES COVERED AND NUMBER OF CRASHES WITH FRAMES COVERED ARE SHOWN.

	Single Objective		Multi Objective	
Function	Frames	No. Crashes	Frames	No. Crashes
T2, T6, T4	301	131	434	164
T2, T6, T5	184	68	406	151
T2, T6, T21	340	125	437	161
T2, T6, T22	275	110	446	152
T2, T21, T4	362	156	395	178
T2, T21, T5	152	77	346	158
T2, T21, T22	351	132	461	177
T2, T22, T4	330	125	484	163
T2, T22, T5	133	73	453	152
T2, T4, T5	144	70	427	165
T6, T21, T4	317	132	392	174
T6, T21, T5	152	70	361	169
T6, T21, T22	281	110	430	169
T6, T22, T4	294	113	476	160
T6. T22, T5	151	75	433	142
T6, T4, T5	150	79	401	155
T21, T22, T4	331	125	445	183
T21, T22, T5	147	75	406	184
T21, T4, T5	199	79	381	185
T22, T4, T5	130	67	436	163

TABLE IX
RESULTS FOR TOTAL CRASHES REPRODUCED.

	Single Objective		Multi Objective	
Function	No. Repr.	Part. Repr. (≤ 0.5)	No. Repr.	Part. Repr. (≤ 0.5)
T2, T6, T4	48	82	84	121
T2, T6, T5	21	35	77	106
T2, T6, T21	56	85	80	113
T2, T6, T22	67	83	104	118
T2, T21, T4	73	100	80	110
T2, T21, T5	18	38	65	92
T2, T21, T22	100	110	110	129
T2, T22, T4	99	106	114	131
T2, T22, T5	20	37	108	121
T2, T4, T5	21	40	76	112
T6, T21, T4	59	84	81	118
T6, T21, T5	21	36	68	95
T6, T21, T22	67	86	108	125
T6, T22, T4	70	84	110	130
T6. T22, T5	24	42	98	114
T6, T4, T5	20	40	80	111
T21, T22, T4	100	104	111	123
T21, T22, T5	26	37	109	122
T21, T4, T5	23	46	83	114
T22, T4, T5	16	34	110	129

We also ran SynTest-ACR on CrashJS using the EvoCrash fitness function to identify if any of our fitness functions are more suitable for ACR. As the EvoCrash function relies only on concepts (exceptions, line numbers, stack traces) that have exact analogues in JavaScript, we expected that it would be able to guide a search, but weakly hypothesised that it would perform less well than EvoCrash-Java did on Java code, where the fitness function had been developed and optimised. We found that using the EvoCrash fitness function, we could reproduce 96 crashes of the 453 (21.2%) in CrashJS.

Table IX presents the results for total crashes reproduced, rather than targets, by Syntest-ACR on CrashJS, to allow for comparison with these results from the EvoCrash function. Our best fitness functions (T2 - right exception on right line, T21 exception raised and exception line covered, T22 - right exception with partial stack trace match), (T2, T22, T4 - right exception in right function) and (T21, T22, T4) are capable of reproducing 100 (22.1%), 99 (21.9%) and 100 (22.1%) with single-objective search and 110 (24.3%), 114 (25.2%) and 111 (24.5%) with multi-objective search. These results are the first successful application of these ACR techniques to NodeJS programs, and within the ballpark of state-of-the-art ACR tools for other languages such as EvoCrash and Botsing. Furthermore, these fitness functions outperform the EvoCrash fitness function on the same dataset, showing promise for other ACR systems looking for improved fitness functions.

As crashes with the correct exception raised in the correct location and multiple stack frames covered could be useful for developers trying to locate a bug, we consider the results from Table VII. For our best fitness functions (T2, T21, T22), (T2, T22, T4) and (T21, T22, T4), if we consider half the stack trace reproduced as providing information to a developer, we could consider these functions as reproducing 110 (24.3%), 106 (23.4%) and 104 (23.0%) in single-objective search and 129 (28.5%), 131 (28.9%) and 123 (27.2%) in multi-objective search. These results exceed the EvoCrash function, which with the same threshold could reproduce only 101 (22.3%).

RQ6: Compared with the existing EvoCrash fitness function, our functions could fully reproduce 18.9% more crashes and could identify many more potentially-useful partial reproductions.

B. Application in Other Languages

We ported our successful fitness function, which we have tailored for its performance on JavaScript but does not rely on any JavaScript-specific elements, back to EvoCrash-Java, and ran a comparison using the reproduction package from [30]. The main result is perhaps surprising: despite the significant improvement that Syntest-ACR with our selected function has over the EvoCrash function on JavaScript programs, when our best functions are implemented in EvoCrash and run its suite of Java programs, the results are virtually unchanged. Our best function does reproduce a couple of crashes somewhat more frequently, but no additional cases from the test suite are reproduced that base EvoCrash did not already reproduce at least sometimes. There are two primary explanations that come to mind: first, our functions may be tailored to the specific difficulties that JavaScript presents, which a more static language does not benefit from. Second, the reproduction in EvoCrash may be driven primarily by its input generation and search algorithm rather than genetic fitness of the produced input. We have not been able to distinguish these cases, or other possibilities such as a selection bias in EvoCrash's test suite, but report this result to highlight that language-specific evaluation of ACR techniques is required and it is not simply possible to port a generic approach from one language to another. We have shown both significantly worse performance taking the EvoCrash function into JavaScript than it produces for Java, and a lack of benefit from taking our Syntest-ACR function into Java.

VII. Areas for Future Research

An area for future research we identified is to conduct user studies with developers. It would be useful and interesting to implement an ACR tool in a development environment to see how developers interact with it and the generated test cases. This would help us understand which tool's characteristics are more useful to developers, such as speed, capability, or consistency. Furthermore, this would validate the usefulness of generated test cases in aiding developers while debugging.

Another area of future research would be to assess the flakiness of the tests generated by ACR tools. In 2014, Luo *et al.* identified 10 main categories of flaky test causes: Async-wait, concurrency, floating point, I/O, network, order dependency, randomness, resource leak, time, and unordered collection [31]. Eck *et al.* identified another four categories: Too restrictive range, test case timeout, platform dependency, and test suite timeout [32]. Finally, Gruber *et al.* identified another cause: infrastructure [33].

In 2024, Gruber and Roslan *et al.* analysed flakiness in tests generated by test generation tools [34]. In this study, they executed almost 1.2 million tests, of which they discovered 9568 flaky tests, with two-thirds of these flaky tests generated by tools. The remainder were developer-written tests. Furthermore, they identified that flakiness occurs more often in generated tests than in developer-written tests for Java and Python. Of these flaky tests, the majority of these occur due to randomness in the tests or unspecified behaviour. As most ACR tools are built on a test generation tool (EvoCrash [35] and Botsing [3] on EvoSuite [16], [17], SynTest-ACR on SynTest-JavaScript [14]), these findings are highly relevant to the tests generated by ACR tools. Extending the analysis by Gruber and Roslan *et al.* to include SynTest-JavaScript and other tools for other languages, such as C and Python, would undoubtedly provide helpful information about the flakiness of tests generated by ours and other tools. Furthermore, there is the opportunity to research flakiness reduction for these tools.

VIII. Threats to Validity

As with all research, there are some threats to the validity of this study. The first threat is the dataset. While CrashJS is the only suitable dataset for ACR in NodeJS, fully representing

JavaScript programs or crashes is challenging. As CrashJS is a sample of crashes from a collection of sources, future research in this area could be applied to validate these results on crashes from real-world systems and to conduct studies with developers to evaluate the usefulness of tests generated using SynTest-ACR.

The second threat is the implementation. The implementation of SynTest-ACR and the fitness terms and functions we implemented to conduct this research are likely to have bugs and errors. While we would like to assume our implementation works as expected, we know this will not be the case. To mitigate this, any spurious results we found while conducting this study resulted in a check of the implementation of the relevant fitness term, with the implementation often being tweaked and any experiments involving that term rerun.

A third threat is the slightly volatile nature of the grid computing system on which we ran these experiments. As previously mentioned, loss of connection between machines could result in some data loss. We have tried to avoid this as much as possible, but some data likely did not make it into our analysis. Due to this, we have considered our results slightly fuzzy so as not to assume one fitness term may be better than another if it can reproduce as little as 4 or 5 additional crashes.

IX. Conclusions

In this paper, we presented the first tool for Automated Crash Reproduction in JavaScript programs, and an analysis of fitness functions for the same purpose. We investigated several aspects we identified that could affect the success of fitness functions, including coverage targets, the difference between discrete and continuous implementations of specific fitness function terms, and the difference between single and multi-objective functions for ACR in JavaScript. Along with these investigations, we present SynTest-ACR, the first tool for ACR that targets NodeJS programs and can reproduce more crashes (25.2%) within CrashJS than the EvoCrash fitness function (21.2%) for a proportionate increase over the EvoCrash function of 18.9%. Additionally, our best fitness functions had significantly more partial reproductions, which could be useful for developers than with the EvoCrash fitness function. Our functions found possibly useful partial reproductions up to 28.9% of CrashJS, while the EvoCrash function could find 22.3%.

References

[1] N. Chen and S. Kim, "STAR: Stack trace based automatic crash reproduction via symbolic execution," *IEEE Transactions on Software Engineering*, vol. 41, no. 2, pp. 198–220, 2015.

[2] M. Soltani, A. Panichella, and A. van Deursen, "Search-based crash reproduction and its impact on debugging," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1294–1317, 2020.

[3] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for Java," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1278–1282. [Online]. Available: https://doi.org/10.1145/3324884.3415299

[4] A. Bergel and I. S. Muñoz, "Beacon: Automated test generation for stack-trace reproduction using genetic algorithms," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 1–7.

[5] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring LLM-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2312–2323.

[6] S. Li, J. Guo, M. Fan, J.-G. Lou, Q. Zheng, and T. Liu, "Automated bug reproduction from user reviews for android applications," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 51–60. [Online]. Available: https://doi.org/10.1145/3377813.3381355

[7] M. Soltani, A. Panichella, and A. van Deursen, "Evolutionary testing for crash reproduction," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, ser. SBST '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–4. [Online]. Available: https://doi-org.helicon.vuw.ac.nz/10.1145/2897010.2897015

[8] M. Chakraborty, R. Olivares, M. Sridharan, and B. Hassanshahi, "Automatic root cause quantification for missing edges in javascript call graphs," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022, pp. 3–1.

[9] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node. js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 29–41.

[10] ——, "Semantic patches for adaptation of javascript programs to evolving libraries," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 74–85.

[11] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 198–209.

[12] P. Oliver, J. Dietrich, C. Anslow, and M. Homer, "CrashJS: A NodeJS benchmark for automated crash reproduction," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024.

[13] M. Olsthoorn, D. Stallenberg, A. van Deursen, and A. Panichella, "SynTest-solidity: automated test case generation and fuzzing for smart contracts," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 202–206. [Online]. Available: https://doi.org/10.1145/3510454.3516869

[14] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Guess what: Test case generation for JavaScript with unsupervised probabilistic type inference," in *Search-Based Software Engineering: 14th International Symposium, SSBSE 2022, Singapore, November 17–18, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, pp. 67–82. [Online]. Available: https://doi.org/10.1007/978-3-031-21251-2_5

[15] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 209–220.

[16] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the ACM International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New

York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[17] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference On Quality Software (QSIC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40.

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

[19] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 96–138, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09762-1

[20] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.

[21] Y. Huang, J. Wang, Z. Liu, Y. Wang, S. Wang, C. Chen, Y. Hu, and Q. Wang, "CrashTranslator: Automatically reproducing mobile application crashes directly from stack trace," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623298

[22] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639118

[23] S. Feng and C. Chen, "Prompting is all you need: Automated Android bug replay with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608137

[24] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[25] P. Oliver, J. Dietrich, C. Anslow, and M. Homer, "Dataset for Syntest-ACR: Automated crash reproduction for JavaScript," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16345353

[26] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "SecBench.js: An executable security benchmark suite for server-side javascript," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1059–1070.

[27] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "BUGSJS: a benchmark and taxonomy of JavaScript bugs," *Software Testing, Verification and Reliability*, vol. 31, no. 4, p. e1751, 2021, e1751 stvr.1751. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1751

[28] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[29] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163.

[30] P. Oliver, M. Homer, J. Dietrich, and C. Anslow, "A partial reproduction of a guided genetic algorithm for automated crash reproduction," in *2021 IEEE International Conference on Software Maintenance and Evolution*, 2021, pp. 684–688. [Online]. Available: https://arxiv.org/abs/2107.11933

[31] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[32] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.

[33] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in Python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 148–158.

[34] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser, "Do automatic test generation tools generate flaky tests?" in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608138

[35] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen, "Single-objective versus multi-objectivized optimization for evolutionary crash reproduction," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds., Springer. Cham: Springer International Publishing, 2018, pp. 325–340.