Search-Based Test Generation Targeting Non-Functional Quality Attributes of Android Apps

Search-Based Test Generation Targeting Non-Functional Quality Attributes of Android Apps

Teklit Berihu Gereziher
Chalmers | University of Gothenburg
Gothenburg, Sweden
teklit@student.chalmers.se

Selam Welu Gebrekrstos
Chalmers | University of Gothenburg
Gothenburg, Sweden
welu@student.chalmers.se

Gregory Gay
Chalmers | University of Gothenburg
Gothenburg, Sweden
greg@greggay.com

ABSTRACT

Mobile apps form a major proportion of the software marketplace and it is crucial to ensure that they meet both functional and non-functional quality thresholds. Automated test input generation can reduce the cost of the testing process. However, existing Android test generation approaches are focused on code coverage and cannot be customized to a tester's diverse goals—in particular, quality attributes such as resource use.

We propose a flexible multi-objective search-based test generation framework for interface testing of Android apps—STGFA-SMOG. This framework allows testers to target a variety of fitness functions, corresponding to different software quality attributes, code coverage, and other test case properties. We find that STGFA-SMOG outperforms random test generation in exposing potential quality issues and triggering crashes. Our study also offers insights on how different combinations of fitness functions can affect test generation for Android apps.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software performance**; **Software verification and validation**.

KEYWORDS

Automated Test Generation, Search-Based Test Generation, Search-Based Software Engineering, Software Quality, Quality Attributes

ACM Reference Format:
Teklit Berihu Gereziher, Selam Welu Gebrekrstos, and Gregory Gay. 2023. Search-Based Test Generation Targeting Non-Functional Quality Attributes of Android Apps. In *GECCO '23, The Genetic and Evolutionary Computation Conference, July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590449

1 INTRODUCTION

Mobile "apps" now form a major proportion of the software being developed, with almost three million apps on the Android Google Play store [1]. As in all other software domains, it is critical to ensure that these apps are reliable and meet their functional and non-functional requirements. This is primarily done through testing, the application of input—in this case, often through a touchscreen-based graphical user interface (GUI)—and the comparison of the resulting output to a set of expectations.

However, software testing is expensive in terms of both effort and time [5]. This is especially true when testing is conducted through a user interface, as a human often performs the test input. Automation has a critical role to play in reducing such costs, through means such as automating the selection and execution of test input [5].

One particularly promising form of automation is *search-based test generation*, where metaheuristic optimization algorithms are applied to identify test cases that best maximize or minimize one or more fitness functions—numeric functions representing properties of interest, e.g., code coverage [25]. Many testing goals can be translated into fitness functions, enabling metaheuristic search to efficiently identify highly effective test input [32].

Automated test generation for Android apps is a growing field of research (e.g., [6, 7, 21, 22, 24, 28]). Several generation approaches have been proposed, including search-based techniques [22, 24]. Search-based test generation for Android has even been deployed in an industrial setting [4]. However, *existing approaches are inflexible and narrowly-focused.* They target a single objective or a small range of objectives, and those objectives are largely focused on source code or interface coverage [33].

In particular, search-based test generation has not been used to assess whether Android apps meet *quality goals.* Quality attributes offer the means to assess *how* the software performs tasks, examining performance, resource usage, availability, and other aspects of software behavior. If an app produces the correct output, but in an unacceptably slow manner, then users may still be dissatisfied. Meeting quality goals is often part of the release criteria for software, and testing is generally used to perform such assessment.

In this study, we propose a multi-objective search-based test generation framework for GUI-based testing of Android apps—STGFA-SMOG[1]. In particular, this framework blends fitness functions based on general test case properties (e.g., crashes discovered), source code (e.g., line coverage), and software qualities (e.g., CPU usage). STGFA-SMOG enables a tester to flexibly choose the subset of fitness functions needed to generate tests for their own testing and quality goals. This framework can also be expanded in the future with additional fitness functions.

We perform an empirical assessment of STGFA-SMOG, comparing its performance over a variety of apps with a common baseline—random test generation—in exposing potential quality issues as

[1]Search-based Test Generation Framework for Android Apps with Support for Multi-objective Generation

GECCO '23, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0119-1/23/07…$15.00
https://doi.org/10.1145/3583131.3590449

well as crashes. With regard to generating tests targeting quality attributes (and code coverage), we observed:

- With a 10 generation search budget, STGFA-SMOG outperforms random generation at maximizing CPU, memory, and network usage—as well as line coverage—in all apps, often with large effect sizes. It outperforms random generation in a subset of apps for method coverage and battery usage. With a 15 generation search budget, STGFA-SMOG outperforms random generation in all configurations.
- STGFA-SMOG shows improved performance when the search budget is increased for CPU, memory, and battery usage, as well as method coverage. Improvements are smaller or more inconsistent for line coverage and network usage.

With regard to triggering crashes:

- STGFA-SMOG is able to trigger more crashes—and trigger the same crashes at a higher frequency—than random generation. More unique crashes were triggered, and the frequency increased, at a higher search budget.
- One crash is only triggered targeting memory or network usage, one when targeting CPU usage, and one when targeting battery usage. However, no fitness function configuration is more effective for triggering crashes across all apps. Instead, the best configuration is app-dependent.
- Tests targeting battery usage benefited more than other fitness configurations from a larger search budget.

Our study offers insights on how different combinations of fitness functions can expose quality issues and crashes in Android apps. We make STGFA-SMOG available (see Section 5) for developers to use on their own apps, and for researchers interested in advancing the capabilities and effectiveness of test generation.

2 BACKGROUND

Android Testing: Android is an open-source operating system for mobile devices, built on the Linux kernel. Apps can be developed in multiple languages, with many written in Java. The Android SDK provides tools and APIs for app development [13].

Android apps can be tested at typical levels of granularity, e.g., unit, integration, and system levels. However, there are particular contextual elements that must be considered [17]. First, the primary means of interaction with apps is through a touchscreen-based GUI. Such interactions can include, e.g., typing, long and short taps, swiping, and pinching. Second, system-level testing requires executing an app on the Android operating system. This execution can be done on either an actual device or through emulation.

In addition, Android applications are often architected using four component types [20]. These include activities (distinct user-facing "screens"), broadcast receivers (event handlers), content providers (interfacing to share structured data between apps), and services (background task execution). As we are focused on GUI-based test generation, we primarily target activities in this research (although activities can trigger execution of other components).

Search-Based Test Generation: Manual creation of test cases is tedious, expensive, and error–prone [19]. Automation of aspects such as test input selection can reduce the cost of this process by reducing and focusing manual effort [5].

Table 1: Examples of Android test generation tools, contrasted by type, whether they generate UI events, whether they generate system events, and the properties targeted.

Tools	Type	UI	System	Targets
Monkey [28]	Random	Yes	Yes	None
Dynodroid [21]	Random	Yes	Yes	Relevancy to App State
GUIRipper [7]	Model-Based	Yes	No	Model Coverage
Q-Testing [27]	Model-Based	Yes	Yes	Model Coverage, Curiosity
SwiftHand [6]	Model-Based	Yes	No	Model Coverage
EvoDroid [22]	Search-Based	Yes	No	Code Coverage
Sapienz [24]	Search-Based	Yes	Yes	Crashes, Length, Code Cov.
STGFA-SMOG	Search-Based	Yes	Yes	Crashes, Length, Code Cov., Qualities

Search-based test generation frames input selection as a search problem, where metaheuristic optimization algorithms attempt to identify test input that best embody properties that testers seek in their test cases [5, 25]. These properties are assessed using one or more fitness functions—numeric scoring functions. The metaheuristic embeds a strategy for sampling solutions from the space of possible inputs, often based on a natural process [14]. In test generation, a "solution" is often either a single test case or a test suite—a collection of test cases. The metaheuristic uses the fitness functions to assess solution quality, offering feedback to guide the selection and improvement of solutions over a series of generations. Search-based test generation has proven to be a flexible [3], scalable [23], and competitive [32] method of automated test generation.

The most common metaheuristic for search-based test generation is a genetic algorithm [11]. Genetic algorithms are modeled after the natural evolution of a population [14]. In short, a "typical" test generation approach often resembles the following process:

- An initial population of test cases is randomly generated. Each test case contains initialization and a random number of interactions with the interface of the app-under-test (AUT).
- Each generation, the fitness scores of each solution are calculated and a new population is created. This population is formed through four sources of solutions:
  - One of the best solutions may be carried over to the new population intact (**reproduction**).
  - At a certain probability, two good solutions will be chosen to create two "children" by combining elements of each solution (**crossover**). For example, the children may blend test cases from the parents.
  - At a certain probability, a good solution can be **mutated**—e.g., a test case may be modified, added, or deleted.
  - At a certain probability, a new randomly generated solution will be added to the population to maintain **diversity**.
- When the **search budget**—typically a limit on the number of generations—expires, the final population is returned (as well as the best solution to date, if it is not in the final population).

3 RELATED WORK

Several approaches to automated test generation for Android apps have been proposed (e.g., [6, 7, 21, 22, 24, 28]). We briefly summarize such approaches here, focusing on the examples in Table 1.

Most approaches can be split into three families. First are random approaches, of which the prototypical is Monkey—a tool included in the Android SDK [28] that is often effective due to its speed [28]. However, test cases are not saved for reuse. Dynodroid observes the current state of the AUT and randomly selects relevant events [21].

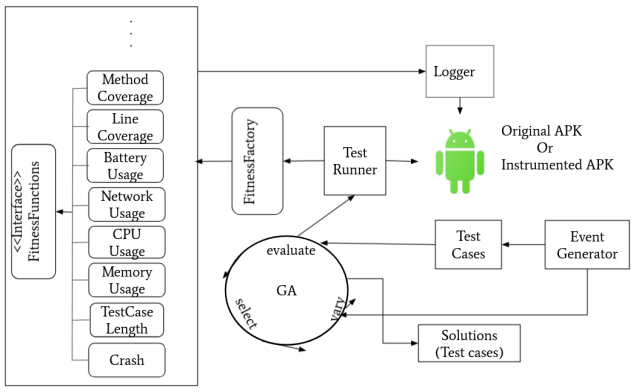

Figure 1: Architecture of STGFA-SMOG.

Model-based approaches (e.g., GUIRipper [7], Q-Testing [27], and Swifthand [6]) extract a model of the AUT GUI and generate input intended to cover transitions through the model. Recent approaches are often based on Reinforcement Learning [27] or other Machine Learning techniques [15]. Other search-based approaches also exist, including EvoDroid [22] and Sapienz [24]. EvoDroid targets code coverage. Sapienz maximizes code coverage and number of crashes while minimizing test case length.

Our approach differs by targeting a greater variety of goals, including crashes, test length, multiple forms of code coverage, and—in particular—multiple aspects of quality (focusing on resource usage). This offers flexibility to assess different testing goals absent from related work. STGFA-SMOG also generates both UI and system events as input, and test cases are saved in a re-executable form.

4 APPROACH

STGFA-SMOG is a multi-objective search-based test generation framework that generates re-executable UI-based tests for Android apps. Its core algorithm follows the process outlined in Section 2. Its architecture is illustrated in Figure 1. In this section, we will discuss important aspects of how it functions.

Solution Representation: In STGFA-SMOG, each solution represents a test case. A test case contains a sequence of atomic input events (considered the "genes" for crossover and mutation). Each input event consists of the *action* performed and any *parameters* for that action. Seven types of events can be applied as test input:

- **System Events:** System-wide operations including pressing the home or back button, increasing/decreasing/muting volume, or starting or ending a phone call.
- **Navigation Events:** Used to navigate the GUI of the AUT, includes scrolling up, down, left, and right.
- **Major Navigation Events:** Pressing a menu button, putting the device to sleep, pressing the button in the center of a d-pad. As in Monkey, these events are rarely used [2].
- **Input Text:** A random string is provided as input.
- **Tap:** A click or long-press is performed at random coordinates (based on the screen size of the device or emulation).
- **Swipe:** A horizontal or vertical swipe is performed (with start and end coordinates determined at random).
- **Drag and Drop:** An item is moved from one location to another (with random start and end coordinates).

The generated test cases are output in a structured text format that STGFA-SMOG can re-execute.

Fitness Functions: STGFA-SMOG supports simultaneous optimization of 1–3 fitness functions[2]. The fitness functions that STGFA-SMOG currently supports include:

- **General Test Properties:** Number of Triggered Crashes (maximized), Test Length (minimized)
- **Code Coverage:** Line, Method Coverage (both maximized)
- **Software Qualities:** CPU Usage, Memory Usage, Network Usage, Battery Usage (all maximized)

In particular, we are interested in the third category in this research. Previous generation frameworks do not allow the generation of tests to check whether software meets quality goals. We have focused, at this time, on fitness functions related to various forms of resource usage. Test cases are evolved to maximize resource use—potentially leading to detection of quality issues.

The factory method design pattern was used to implement fitness functions to support the addition of more functions in the future [12]. Fitness functions are user-selectable, and independent of each other. All fitness functions can be either maximized or minimized. However, we have noted above the *intended* use of each.

Genetic Algorithm: STGFA-SMOG performs multi-objective optimization based on the NSGA-II algorithm [9, 19, 24, 33]. NSGA-II ranks solution quality by finding solutions that balance optimization of the selected fitness functions (i.e., are *Pareto-optimal*) [9]. The solutions are sorted into levels, where each contains individuals that are dominated by the same number of other solutions and that do not dominate each other. Those in the top level represent cases where the fitness of one function cannot be improved without decreasing fitness of another function. Solutions in this level are sorted based on their "crowding distance"—their similarity to other solutions at this level. Solutions from a less-dense space on the Pareto frontier are given higher priority during the formation of a new generation.

To maintain solution quality, NSGA-II combines the parent population and the new offspring population and performs a fast non-dominated sorting on the combined population. This prevents the loss of high-quality solutions from the previous population.

As shown in Figure 1, the *event generator* generates atomic events. The *test case* module represents solutions. The *fitness evaluator* executes tests using the *test runner*, which sends the events to the AUT using Android Debug Bridge (ADB). Before each test execution, the app is initialized to a fixed state. After fitness evaluation, NSGA-II selects the best candidate solutions. The *variation operator* module performs crossover, mutation, reproduction, and introduces randomly generated new solutions to maintain diversity. After the search budget expires, the final population is output as a test suite.

Crossover is performed using uniform crossover. For each atomic event from the first parent solution, a "coin flip" determines which child gets that event and which gets the event at the same index from the second parent. During mutation, the order of events in

[2]The algorithm employed, NSGA-II, has been found to struggle when applied to more than three fitness functions [10, 16, 18]. However, as quality goals often conflict [34], we recommend not targeting many quality-related functions simultaneously.

a solution can be randomized, or the parameters of one or more events can be randomly changed.

User-Adjustable Parameters: The following can be configured:

- Minimum and maximum test case size
- Population size and make-up (proportion created using crossover, mutation, reproduction, and random generation)
- Search budget (number of generations)
- How many test cases to output from the final population

5 METHODOLOGY

We are interested in comparing the capabilities of STGFA-SMOG to the standard baseline for Android test generation—random test generation. We are also interested in performing an exploration of the relationship between fitness functions and fault detection. Therefore, we focus on the following research questions:

RQ1: Are test cases generated by STGFA-SMOG more effective than those generated randomly at meeting tester goals?
RQ2: Are test cases generated by STGFA-SMOG more effective than those generated randomly at triggering crashes?
RQ3: Is a particular combinations of fitness functions more effective than others at causing the assessed apps to crash?
RQ4: Does an increased search budget improve the effectiveness of the resulting test cases?

To implement and evaluate STGFA-SMOG, we followed a design science research methodology [30], an iterative process:

- We developed a random test generation tool (Section 5.1).
- We extended this tool with a genetic algorithm to create STGFA-SMOG (Section 5.1).
- Over a multiple iterations, we added additional fitness functions to STGFA-SMOG (Section 5.1).
- After adding each function, we performed an intermediate evaluation where we gathered fitness values. These fitness values are used to address **RQ1** and **RQ4** (Section 5.2).
- We performed a final evaluation to assess crash detection. This evaluation addresses **RQ2–4** (Section 5.3).

STGFA-SMOG is available at:
https://github.com/TeklitB/STGFA-SMOG
Our random generation tool is available at:
https://github.com/TeklitB/Random-Test-Generation
A replication package containing our experiment data is available at: https://doi.org/10.5281/zenodo.6387568

5.1 Design Science Iterations

Design science is an iterative research methodology intended to produce an artifact [30]. The methodology consisting of cycles of problem identification, implementation, and evaluation. In this study, we implemented a random test generation tool and STGFA-SMOG over three iterations. After each iteration, we performed either an informal or a formal evaluation, and used the results to improve the artifacts.

Iteration 1 (Random Generation): In the first iteration, we designed a random generation tool. Existing random generation tools, like Monkey, do not generate tests in the same format that we selected for STGFA-SMOG, and the resulting fitness of test cases could not be calculated in the same manner. Therefore, we decided to implement our own tool.

The random generation tool generates a population containing a user-set number of test cases. Each test case contains a random number of input events—up to a user-set maximum.

As part of this phase, we implemented two fitness functions—the number of crashes triggered and the test length—as a basis for making comparisons in later stages.

Iteration 2 (Genetic Algorithm): In the second iteration, we created an initial version of STGFA-SMOG by extending the random generation tool with the NSGA-II genetic algorithm. We used the DEAP framework [8] to implement NSGA-II. At this stage, we used the two fitness functions implemented in the first iteration as the optimization targets for the genetic algorithm.

Iteration 3 (Additional Fitness Functions): In the final phase, we iteratively added additional fitness functions to STGFA-SMOG, performing an intermediate evaluation after adding each. Specifically, we added fitness functions based on CPU usage, memory usage, battery usage, network usage, method coverage, and line coverage. The code coverage fitness functions were implemented using ACVTool[3].

Table 2: Apps used for intermediate evaluation.

App Name	Version	Size	Domain	Description
Paris Traffic	2021.04	3.4 MB	Navigation	Simple, responsive map for your trek
Blokish	3.2	2.2MB	Game	Board game
Specie	1.36	0.5MB	Currency	Simple currency conversion
RPNcalc	1.0.3	18MB	Science	A simple, modern calculator
E numbers	1.4.1	0.1MB	Health	Food additives reference

5.2 Intermediate Evaluation

The aim of the intermediate evaluation was to see if STGFA-SMOG was functioning as intended after adding each fitness function. As we focused on examining fitness scores during this evaluation, we use the gathered data to address **RQ1** and to partially address **RQ4**.

Test Subjects: A set of five apps (Table 2) were randomly chosen from the F-Droid store[4]. These apps have been used in past research studies [24, 26, 29]. Apps that require authentication are not supported by STGFA-SMOG in its current form, and were ignored. We also discarded apps that could not be instrumented by the code coverage tool.

Experiment Configurations: Test cases are generated according to the following general fitness function configuration: *(maximize number of crashes) + (minimize test case length) + (maximize fitness function related to testing goal).*

During this evaluation, the third fitness function was one of the following: CPU usage, memory usage, network usage, battery usage, line coverage, and method coverage.

In the experiment, we applied two search budgets—10 and 15 generations. The population size was set to 10 individuals. Both settings were chosen after informal experimentation, largely based on the cost of fitness evaluation. The percentage of the population created through crossover, mutation, reproduction, and addition of random new solutions were set to 30.00%, 30.00%, 15.00%, and

[3]https://github.com/pilgun/acvtool
[4]https://f-droid.org/

Table 3: Apps used for final evaluation

App Name	Version	Size	Domain	Description
Blokish	3.4	2.6MB	Game	Multiplayer board game
SolitaireCG	3.4.1	0.5MB	Game	Solitaire Card Games
PalmCalc	3.0.4	2.9MB	Scientific	Retro scientific calculator
Dib2Calc	0.12.62	3.4MB	Science	The crazy calculator
Mouse Pounce	1.2.1	20MB	Game	Egyptian Rat Screw
xskat	1.6	0.5MB	Game	Play German card game Skat
bmicalculator	4.0.2	2MB	Health	Body Mass Index calculator
Simple Accounting	1.6.1	3.4MB	Money	Simple money tracking
Currencies	1.5.1	3.9MB	Money	A currency converter
Inflation Calc	2.9 (18)	2.2MB	Money	Inflation calculator

25.00% respectively. The minimum and maximum lengths of test cases were set to 20 and 50. The same population and test length settings were applied for random generation as well.

Additional informal experimentation with other search budgets and population sizes was also carried out to identify settings for the final evaluation.

Data Collection: For each app, search budget, and fitness function configuration, five trials were performed. We also performed a paired trial of random generation for each configuration. In this case, we generated 300 test suites with STGFA-SMOG (5 apps, 2 search budgets, 6 fitness configurations, 5 trials) and 300 with random generation. During each trial, we collected the number of crashes triggered as well as the final fitness values for each test case.

The evaluation was conducted on a PC with a five-core 1.60GHz CPU and 8GB RAM running Ubuntu 20.04. A Google Pixel 4 XL emulator with Android API 28 was used to execute the apps.

We perform statistical analysis to assess our observations by comparing each configuration of STGFA-SMOG and random test generation using the following hypotheses:

- H: Test cases generated using configuration A will have a different distribution of fitness values than cases generated using configuration B.
- $H0$: Observations of fitness values for both configurations are drawn from the same distribution.

Where a configuration of STGFA-SMOG represents a particular search budget and fitness function combination. We also use random test generation as a "configuration" in these comparisons.

Our observations are drawn from an unknown distribution. To evaluate the null hypotheses without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [37], a non-parametric test for determining if one set of observations is drawn from a different distribution that another set. We apply the test for each pairing of STGFA-SMOG configuration and random test generation with $\alpha = 0.05$.

In cases of significance, we have also used the Vargha-Delaney A measure to assess effect size [35]. A small effect (A_{12}) is $0.56 \leq A_{12} < 0.64$, medium is $0.64 \leq A_{12} < 0.71$, and large is $A_{12} \geq 0.71$ [36].

5.3 Final Evaluation

After implementing all fitness functions, we performed a final evaluation focused on the ability of generated tests to trigger crashes. The basic evaluation structure was the same, but with an increased number of apps and some changes to STGFA-SMOG settings.

Test Subjects: An expanded set of 10 apps were selected randomly from the F-Droid store (Table 3). We attempted to ensure that apps with varying complexity (e.g., different types and ranges of functionality) and from different product domains were selected.

Experiment Configurations: In this evaluation, we used the same fitness function formulation as in the intermediate evaluation. However, to control experiment costs and to focus on the relationship between quality attributes and crashes, we omitted code coverage-based functions and applied the following fitness functions: CPU usage, memory usage, network usage, and battery usage.

Two search budgets of 10 and 30 generations were used. The population size was set to 20 individuals. The percentage of the population created through crossover, mutation, reproduction, and addition of random new solutions were set to 30.00%, 30.00%, 15.00%, and 25.00% respectively. The minimum and maximum test case lengths were set to 20 and 50. Again, the same population and test length settings were applied for random generation.

Data Collection: For each app, search budget, and fitness function configuration, five trials were performed. We also performed a paired trial of random generation for each configuration. In this case, we generated 400 test suites with STGFA-SMOG (10 apps, 2 search budgets, 4 fitness configurations, 5 trials) and 400 with random generation. During each trial, we collected the number of crashes and fitness values for each test case.

While a single test case could trigger more than one crash—the app is reloaded—we found that the majority of test cases triggered only one crash at most. Therefore, in making comparisons, we instead use the number of crashes across the full test suite (the final population). We apply the Mann-Whitney-Wilcoxon rank-sum test with $\alpha = 0.05$ to the following hypotheses:

- H: Test suites generated using configuration A will have a different distribution of number of triggered crashes than suites generated using configuration B.
- $H0$: Observations of number of triggered crashes for both configurations are drawn from the same distribution.

Again, a configuration can refer to either an execution of STGFA-SMOG with a particular search budget and fitness function combination or to an execution of random test generation. In cases of significance, we again applied the Vargha-Delaney A measure to assess effect size.

6 RESULTS AND DISCUSSION

For brevity, we use the following abbreviated names for particular fitness function combinations: **CLB** (crashes, test length, battery usage), **CLC** (crashes, test length, CPU usage), **CLM** (crashes, test length, memory usage), **CLN** (crashes, test length, network usage), **CLLC** (crashes, test length, line coverage), and **CLMC** (crashes, test length, method coverage).

We also omit Mann-Whitney-Wilcoxon results due to space constraints. If an effect size is not reported, it can be assumed that we could not refute the null hypothesis for Mann-Whitney-Wilcoxon (i.e., results could be drawn from the same distribution).

6.1 Exposing Potential Quality Issues (RQ1,4)

We first assess whether STGFA-SMOG can generate test cases more effective at maximizing fitness functions related to software quality attributes (and code coverage) than random test generation. We also examine the effect of increasing the number of generations allocated to STGFA-SMOG.

Table 4: Median fitness values of final test cases from STGFA-SMOG (10 generation search budget, "Search") and random generation for each app. Cases in bold are where random generation yielded higher median results.

	CLC		CLM		CLB		CLN		CLLC		CLMC	
	Search	Random	Search	Random	Search	Random	Search	Random	Search	Random	Search	Random
Blokish	26.00	11.50	42610.50	25790.00	6.83e-07	0.00	-	-	7.76	6.57	8.53	7.00
Specie	24.00	14.00	46204.50	25398.50	**0.00**	**4.40e-07**	481614.00	37255.00	4.59	4.00	**1.41**	**1.80**
Paris Traffic	27.00	18.00	38872.00	28098.00	6.71e-07	0.00	2350519.00	124255.40	20.40	18.32	26.26	24.63
RPNcalc	68.00	26.50	136372.50	117998.50	4.66e-07	0.00	-	-	22.51	20.30	31.80	29.13
E numbers	29.00	15.00	39955.00	25085.50	4.59e-07	0.00	-	-	2.19	2.02	2.17	1.96

Table 5: Median fitness values of final test cases from STGFA-SMOG (15 generation search budget, "Search") and random generation for each app.

	CLC		CLM		CLB		CLN		CLLC		CLMC	
	Search	Random	Search	Random	Search	Random	Search	Random	Search	Random	Search	Random
Blokish	48.50	10.00	41755.50	24722.50	3.31e-06	1.00e-06	-	-	7.70	6.42	8.16	7.05
Specie	35.50	23.50	45898.00	25511.50	1.70e-05	8.60e-07	1235955.00	37471.66	4.52	4.03	2.05	1.90
Paris Traffic	29.00	13.50	50705.50	29242.50	0.00	0.00	1512818.00	135683.00	20.43	18.46	26.81	23.88
RPNcalc	94.50	52.50	134803.50	120060.50	5.08e-05	0.00	-	-	22.51	20.30	32.39	29.13
E numbers	42.00	21.50	48234.00	25441.00	1.33e-06	3.10e-07	-	-	2.22	2.01	2.17	1.96

Table 6: Effect size results for the fitness values of STGFA-SMOG (10 generation) versus random generation. Large effect sizes are bolded, effect size is only measured when significance is found with Mann-Whitney Wilcoxon test.

	CLC	CLM	CLB	CLN	CLLC	CLMC
Blokish	0.70	**1.00**	0.68	-	**0.75**	**0.89**
Specie	**0.76**	**1.00**	-	**1.00**	**0.83**	-
Paris Traffic	0.62	**0.94**	**0.71**	**1.00**	**0.77**	**0.78**
RPNcalc	**0.86**	**0.89**	-	-	**0.86**	**0.90**
E numbers	**0.82**	**0.97**	0.68	-	**0.91**	**0.89**

Table 7: Effect size results for the fitness values of STGFA-SMOG (15 generation) versus random generation.

	CLC	CLM	CLB	CLN	CLLC	CLMC
Blokish	**0.87**	**0.94**	0.64	-	**0.89**	**0.80**
Specie	0.62	**0.97**	0.70	**1.00**	**0.83**	**0.84**
Paris Traffic	**0.83**	**0.98**	**0.87**	**1.00**	**0.95**	**0.99**
RPNcalc	**0.90**	**0.83**	**0.74**	-	**0.92**	**0.87**
E numbers	0.65	**0.99**	0.69	-	**0.92**	**0.92**

Table 8: Effect size results for the fitness values of STGFA-SMOG, 15 generation versus 10 generation search budgets.

	CLC	CLM	CLB	CLN	CLLC	CLMC
Blokish	**0.79**	-	-	-	-	0.63
Specie	**0.86**	**0.98**	**0.72**	**0.88**	-	**0.90**
Paris Traffic	-	0.63	0.63	0.21	-	0.66
RPNcalc	**0.94**	-	0.68	-	0.62	0.69
E numbers	**0.83**	0.62	-	-	-	-

Tables 4–5 show median fitness values of the final test cases generated by STGFA-SMOG (for 10 and 15 generation search budgets) and random test generation. In each case, we report the fitness values for the quality or code coverage-related function. For example, for the CLC combination, we report the CPU usage fitness—not the test length or crashes. Three apps do not have online functionality, making fitness above 0.00 impossible for network usage.

In almost all cases, STGFA-SMOG yields a higher median fitness than random generation, indicating that the average test is more effective at exposing potential quality issues. There are two exceptions—battery usage and method coverage for the `Specie` app at a 10 generation budget. One reason for this could be that the app is that the majority of valid input events require text input, and purely random input may not be in the expected format or ranges. In the future, STGFA-SMOG's string generation could be improved, e.g., by seeding values extracted from code or documentation [31].

However, at 15 generations, STGFA-SMOG always has a higher median fitness. This indicates clear improvement with a larger search budget. We also use statistical analysis to compare the distributions of fitness values.

Tables 6–7 show the effect size when STGFA-SMOG yielded a different distribution of fitness results than random generation. An effect size larger than 0.5 indicates that STGFA-SMOG outperforms random generation. Large effect sizes (≥ 0.71) are in bold.

With a 10 generation budget, STGFA-SMOG outperforms random generation with a large effect size for almost all fitness functions, in almost all apps. There are only three cases where we cannot refute the null hypothesis—for battery usage for `Specie` and `RPNcalc` and method coverage for `Specie`. However, with a 15 generation search budget, we can refute the null hypothesis in all cases.

> **Exposing Quality Issues (RQ1):** With a 10 generation search budget, STGFA-SMOG outperforms random generation for the CPU, memory, network, and line coverage fitness functions in all apps, often with large effect sizes. It outperforms random generation in a subset of apps for the method coverage and battery fitness functions. With a 15 generation search budget, STGFA-SMOG outperforms random generation for all apps and fitness functions.

This shows the potential of search-based test generation to expose quality issues in Android apps. There is still room for improvement—e.g., cases where the effect size is not large. However, those could be potentially addressed with a higher search budget.

In Table 8, we compare STGFA-SMOG at 15 and 10 generations. In many cases, there is a significant improvement from increasing the budget—particularly for CPU usage, followed by method coverage, memory, and battery usage. Only a small improvement was seen for one app for line coverage. For network coverage, large improvements were observed for `Specie` with a longer search budget. However, performance actually *decreased* for `trafficparis`.

> **Search Budget (RQ4):** STGFA-SMOG shows improved performance when the search budget is increased from 10 to 15 generations for CPU, memory, and battery usage, as well as method coverage. Improvements are smaller or more inconsistent for line coverage and network usage.

Table 9: Median number of times each unique crash is triggered by a test suite generated by STGFA-SMOG targeting a particular fitness function configuration (30 generation budget). "-" means that the crash type was not triggered.

App	Exception	CLB	CLC	CLM	CLN
Blokish	ActivityNotFoundException	6.20	4.00	8.20	2.40
Dib2Calc	IndexOutOfBoundsException	6.80	0.60	1.80	3.60
Dib2Calc	ArrayIndexOutOfBoundsException	2.80	4.20	2.00	3.40
Dib2Calc	NullPointerException	-	-	0.20	0.20
PalmCalc	NullPointerException	-	0.60	-	-
SolitaireCG	ClassCastException	2.40	-	-	-

Table 10: Median number of times each unique crash is triggered by STGFA-SMOG (10 generation budget).

App	Exception	CLB	CLC	CLM	CLN
Blokish	ActivityNotFoundException	0.80	0.40	1.80	1.00
Dib2Calc	IndexOutOfBoundsException	0.60	-	1.60	0.40
Dib2Calc	ArrayIndexOutOfBoundsException	-	3.00	-	0.60
Dib2Calc	NullPointerException	-	-	-	-
PalmCalc	NullPointerException	-	-	-	-
SolitaireCG	ClassCastException	-	-	-	-

Table 11: Median number of times each unique crash is triggered by random generation. "Random-N" refers to trials paired to STGFA-SMOG (N generation search budget).

App	Exception	Random-10	Random-30
Blokish	ActivityNotFoundException	-	-
Dib2Calc	IndexOutOfBoundsException	0.05	0.05
Dib2Calc	ArrayIndexOutOfBoundsException	0.05	-
Dib2Calc	NullPointerException	-	-
PalmCalc	NullPointerException	-	-
SolitaireCG	ClassCastException	-	-

In future work, we will further examine an enlarged pool of apps and search budgets to make clearer recommendations on how to set the search budget.

6.2 Triggering Crashes (RQ2–4)

Beyond assessing whether software meets its quality requirements, the core goal of the testing process is to identify faults in the AUT. This is often accomplished by *making the software crash*. Therefore, in our experiments, we are also interested in (a) assessing whether STGFA-SMOG is better able to trigger crashes than random generation, and (b), whether targeting particular quality-related fitness functions tends to lead to the discovery of more crashes.

Tables 9–11 indicate that six unique crashes (based on the exception and stack trace) were discovered across four of the apps. The tables show the median number of times that each crash was triggered by a test suite generated by STGFA-SMOG (targeting four fitness function combinations) under the two search budgets and by random generation.

All crashes triggered by random generation are also triggered—more often—by STGFA-SMOG under any search budget. In turn, all crashes triggered under a 10 generation search budget are triggered more often under a 30 generation search budget.

Triggering Crashes (RQ2): STGFA-SMOG is able to trigger more crashes—and trigger the same crashes at a higher frequency—than random generation.

Search Budget (RQ4): More unique crashes were triggered, and frequency increased, at a higher search budget.

Table 12: Effect size results for number of triggered crashes between configurations of STGFA-SMOG (10 generation budget) and random generation for `Blokish`.

	CLB	CLC	CLM	CLN	Random
CLB	-	-	-	-	0.52
CLC	-	-	0.46	-	-
CLM	-	0.54	-	-	0.55
CLN	-	-	-	-	0.53
Random	0.48	-	0.45	0.47	-

Table 13: Effect size results for number of triggered crashes between configurations of STGFA-SMOG (10 generation budget) and random generation for `Dib2Calc`.

	CLB	CLC	CLM	CLN	Random
CLB	-	0.43	-	-	-
CLC	0.57	-	-	0.56	0.58
CLM	-	-	-	-	-
CLN	-	0.44	-	-	0.53
Random	-	0.42	-	0.47	-

Table 14: Effect size results for number of triggered crashes between configurations of STGFA-SMOG (30 generation budget) and random generation for `Blokish`.

	CLB	CLC	CLM	CLN	Random
CLB	-	-	-	0.60	0.66
CLC	-	-	0.39	-	0.60
CLM	-	0.61	-	0.64	**0.71**
CLN	0.40	-	0.36	-	0.56
Random	0.34	0.40	0.29	0.44	-

Three of the six crashes can be triggered for any fitness function combination, indicating that the fault is not related specifically to a particular resource being consumed. However, the other three crashes were only triggered under a subset of 1–2 configurations. It is not clear whether the crashes were triggered because a function consumed more of a resource than other functions, but—at the least—focusing on that portion of the code enabled the detection of a functional issue.

Qualities and Crashes (RQ3): One crash is only triggered targeting memory or network usage, one when targeting CPU usage, and one when targeting battery usage.

We performed statistical analysis to compare the configurations of STGFA-SMOG to each other (and to random generation). For **RQ1**, we compared test cases. Because most test cases only triggered a single crash (at most), we instead compare full test suites—the final population.

Tables 12–13 show effect size results comparing multiple STGFA-SMOG configurations generated under a 10 generation budget and random generation for relevant apps. These should be read as the configuration on the row compared to the column. Overall, we primarily see minor differences. Some configurations are better than others at triggering crashes, but all with small effect size. For example, for `Blokish`, CLM slightly outperforms CLC, and for `Dib2Calc`, CLC slightly outperforms CLB and CLN. In many cases, there is no statistical difference in the distribution of number of triggered crashes.

Tables 14–16 show the same for a 30 generation search budget. Again, we see cases where there is no statistical difference in crash triggering. We do see larger differences between some configurations than with a 10 generation budget. For `Blokish`, CLM outperforms CLC and CLN and CLB outperforms CLN. For `Dib2Calc`,

Table 15: Effect size results for number of triggered crashes between configurations of STGFA-SMOG (30 generation budget) and random generation for `Dib2Calc`.

	CLB	CLC	CLM	CLN	Random
CLB	-	0.62	0.64	-	**0.74**
CLC	0.38	-	-	-	0.62
CLM	0.36	-	-	0.42	0.60
CLN	-	-	0.58	-	0.68
Random	0.26	0.38	0.40	0.32	-

Table 16: Effect size results for number of triggered crashes between configurations of STGFA-SMOG (30 generation budget) and random generation for `SolitaireCG`.

	CLB	CLC	CLM	CLN	Random
CLB	-	0.56	0.56	0.56	0.56
CLC	0.44	-	-	-	-
CLM	0.44	-	-	-	-
CLN	0.44	-	-	-	-
Random	0.44	-	-	-	-

Table 17: Effect size results for number of triggered crashes between configurations of STGFA-SMOG (30 versus 10 generation budget).

	CLB	CLC	CLM	CLN
Blokish	0.64	0.59	0.66	-
Dib2Calc	**0.72**	-	0.56	0.66
PalmCalc	-	-	-	-
SolitaireCG	0.56	-	-	-

CLB outperforms CLC and CLM, and CLN outperforms CLM. For `SolitaireCG`, CLB outperforms CLC, CLM, and CLN. However, in all of these cases, the effect size is either small or medium.

From these results, it is difficult to see concrete patterns in whether targeting certain software qualities for test generation leads to the discovery of more crashes than when targeting other qualities. Instead, the results are dependent on the particular AUT. The relationship between crashes and software qualities is complex, and in future work, we will widen the scope of apps and search budgets examined to gather more evidence for the software quality factors that influence crash detection.

Qualities and Crashes (RQ3): No fitness function configuration is more effective for triggering crashes across all apps. Instead, the best configuration is app-dependent. In many cases, there are only minor differences in terms of the number of triggered crashes.

Table 17 assesses whether a larger search budget improves results for each fitness configuration. We see that the number of triggered crashes often improves as the search budget increases for individual configurations. In particular—when targeting battery usage—the number of triggered crashes increased for three of the four relevant apps. For `Dib2Calc`, the number of triggered crashes increased with large effect size. In the other cases, the improvements were more mild. CPU and network usage only discovered more crashes for a single app when the search budget increased.

Search Budget (RQ4): Tests targeting battery usage benefited more than other fitness configurations from a larger search budget, in terms of the number of triggered crashes.

7 THREATS TO VALIDITY

Internal Validity: To control experiment cost, we have only generated five test suites (populations) for each combination of app, budget, and fitness function configuration. Further, we did not vary the parameter settings, e.g., population size, instead using values arrived at following informal experimentation. It is possible that larger sample sizes or different parameter settings may yield different results. However, we believe we have yielded a sufficient sample to draw initial observations. Future work will expand the scope of experiments conducted.

We performed test generation in an emulator. It is possible that quality issues cannot be replicated on real hardware. In future work, we will assess replicability of the discovered issues.

External Validity: Our study has focused on a limited number of apps to control experiment costs. Nevertheless, we believe that such apps are representative of, at minimum, small to medium-sized Android apps. We have randomly selected apps from multiple domains to ensure an unbiased comparison.

We only compare with random test case generation. However, no tool from related work offers comparable fitness functions, making comparisons in terms of quality issues difficult. Moreover, many generation tools are either unavailable or outdated, and random generation has been found to be competitive in past studies [28]. We developed our own random generation tool to ensure that comparisons were made in a fair and equivalent manner (i.e., using the same fitness calculations), and to ensure that generated tests can be re-executed.

Conclusion Validity: When using statistical analyses, we have attempted to ensure all base assumptions are met. We favored non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.

8 CONCLUSIONS

In this study, we have proposed a flexible multi-objective search-based test generation framework for interface testing of Android apps—STGFA-SMOG. This framework allows testers to target a variety of fitness functions, corresponding to different software quality attributes, code coverage, and other test case properties. We find that STGFA-SMOG outperforms random test generation in exposing potential quality issues and triggering crashes. Our study also offers insights on how different combinations of fitness functions can affect test generation for Android apps.

We make STGFA-SMOG available for others to use and improve. In future work, we plan to expand the range of available fitness functions. We observed that fitness evaluation often takes significant time, due to the need to initialize and communicate with an Android device or emulator. Therefore, we will also explore the use of local search (e.g., hill climbing) to generate test cases, as only a single solution would need to be evolved instead of a full population. We will also perform expanded experiments with more apps, fitness combinations, and search budgets to further study the relationship between quality attributes and app crashes.

Acknowledgements: This research was supported by Vetenskapsrådet grant 2019-05275.

REFERENCES

[1] [n. d.]. Number of Android applications on the Google Play store. https://www.appbrain.com/stats/number-of-android-apps

[2] [n. d.]. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey

[3] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder K Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on* 36, 6 (2010), 742–762.

[4] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *Search-Based Software Engineering*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer International Publishing, Cham, 3–45.

[5] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, and Antonia Bertolino. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001. Publisher: Elsevier.

[6] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640. Publisher: ACM New York, NY, USA.

[7] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.

[8] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. Deap: A python framework for evolutionary algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. 85–92.

[9] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197. Publisher: IEEE.

[10] Maha Elarbi, Slim Bechikh, Abhishek Gupta, Lamjed Ben Said, and Yew-Soon Ong. 2017. A new decomposition-based NSGA-II for many-objective optimization. *IEEE transactions on systems, man, and cybernetics: systems* 48, 7 (2017), 1191–1210. Publisher: IEEE.

[11] Robert Feldt and Simon Poulding. 2015. Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary. In *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*. 1–7. https://doi.org/10.1109/SBST.2015.8

[12] Eric Freeman and Elisabeth Robson. 2020. *Head First Design Patterns*. O'Reilly Media.

[13] Suhas Holla and Mahima M. Katti. 2012. Android based mobile application development and its security. *International Journal of Computer Trends and Technology* 3, 3 (2012), 486–490. Publisher: Citeseer.

[14] John Henry Holland. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

[15] Zubair Khaliq, Sheikh Umar Farooq, and Dawood Ashraf Khan. 2022. A deep learning-based automated framework for functional User Interface testing. *Information and Software Technology* 150 (2022), 106969. https://doi.org/10.1016/j.infsof.2022.106969

[16] Vineet Khare, Xin Yao, and Kalyanmoy Deb. 2003. Performance scaling of multi-objective evolutionary algorithms. In *International conference on evolutionary multi-criterion optimization*. Springer, 376–390.

[17] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66. Publisher: IEEE.

[18] Mario Köppen and Kaori Yoshida. 2007. Substitute distance assignments in NSGA-II for handling many-objective optimization problems. In *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 727–741.

[19] Kiran Lakhotia, Mark Harman, and Phil McMinn. 2007. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 1098–1105.

[20] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[21] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[22] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 599–609.

[23] Jan Malburg and Gordon Fraser. 2011. Combining Search-based and Constraint-based Testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 436–439. https://doi.org/10.1109/ASE.2011.6100092

[24] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[25] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156. Publisher: Wiley Online Library.

[26] Iván Arcuschin Moreno. 2020. Search-based test generation for Android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 230–233.

[27] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3395363.3397354

[28] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test*. 34–37.

[29] Samad Paydar. 2020. An Empirical Study on the Effectiveness of Monkey Testing for Android Applications. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering* 44, 2 (2020), 1013–1029. Publisher: Springer.

[30] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. 2007. A design science research methodology for information systems research. *Journal of management information systems* 24, 3 (2007), 45–77. Publisher: Taylor & Francis.

[31] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401. https://doi.org/10.1002/stvr.1601 stvr.1601.

[32] Alireza Salahirad, Hussein Almulla, and Gregory Gay. 2019. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability* 29, 4-5 (2019), e1701. Publisher: Wiley Online Library.

[33] Leon Sell, Michael Auer, Christoph Frädrich, Michael Gruber, Philemon Werli, and Gordon Fraser. 2019. An empirical evaluation of search algorithms for app testing. In *IFIP International Conference on Testing Software and Systems*. Springer, 123–139.

[34] Raed Shatnawi. 2017. Synergies and conflicts among software quality attributes and bug fixes. *International Journal of Information Systems and Change Management* 9, 1 (2017), 3–21. https://doi.org/10.1504/IJISCM.2017.086209 arXiv:https://www.inderscienceonline.com/doi/pdf/10.1504/IJISCM.2017.086209

[35] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101 arXiv:https://doi.org/10.3102/10769986025002101

[36] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. Publisher: Sage Publications Sage CA: Los Angeles, CA.

[37] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), pp. 80–83. http://www.jstor.org/stable/3001968