Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

EMPIRICAL

An Empirical Study on How Sapienz Achieves Coverage and Crash Detection

Iván Arcuschin* | **Juan Pablo Galeotti** | **Diego Garbervetsky**

[1]FCEyN-UBA/ICC-CONICET, Argentina

Correspondence
*Iván Arcuschin, Email: iarcuschin@dc.uba.ar

Present Address
Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina. Pabellón 1, Ciudad Universitaria, Intendente Güiraldes 2160 Buenos Aires (C1428EGA), Argentina

Funding Information
This work is partially funded by UBACYT-2018 20020170200249BA, UBACYT-2018 20020170100419BA, PICT-2015-2741, PICT-2018-3835 and Microsoft Research Azure Grant.

Several tools for automatically testing Android applications have been proposed. In particular, Sapienz is a search-based tool that has been recently deployed in an industrial setting. Although it has been shown that Sapienz outperforms several state-of-the-art tools, it is still to be seen what features of Sapienz impact the most on its effectiveness.

We conducted an extensive empirical study where we compare the impact of the search algorithm and the usage of *motif* genes, a more compact representation of individuals. Our empirical study shows that the usage of *motif* genes improves coverage both for evolutionary algorithms and random approaches. In particular, it also shows that NSGA-II, the multi-objective evolutionary algorithm used by Sapienz, does not have a clear improvement over other algorithms. In terms of number of crashes detected, our study shows that both NSGA-II and Random Search perform similarly. While the usage of *motif* genes improves the crash detection of algorithms, is not enough to make it statistically significant. These facts cast doubts about the use of evolutionary algorithms in the context of Android test generation and suggest that *motif* genes can have a great impact on the overall effectiveness.

KEYWORDS:
Android, Sapienz, Empirical Study, Test Generation, Evolutionary Algorithms, Random Search

1 | INTRODUCTION

As software keeps becoming more important in our daily lives, the use of mobile devices such as smartphones and tablets increases as well. It is estimated that mobile technologies are now used by two-thirds of the global population. Furthermore, mobile users universally consume more digital minutes per person – more than double in the vast majority of countries and regions[1]. In this context, smartphones have become the dominant platform for mobile time consumption, in terms of total minutes across every market. About 80% of all mobile time[1] is spent in application consumption (commonly known as "apps"). As of November 2021, there are over 2.7 million applications available on Google's Play App Store[2].

Despite their growing popularity, apps tend to contain defects which can ultimately manifest as failures (or *crashes*) to end-users. Similarly to other software, testing mobile apps allows developers to ensure a minimum quality threshold for the applications they write. This process typically involves manually writing test cases. Testing intends to assure that new features behave as expected and that changes to the source code do not break previous existing functionality. However, testing is a very time consuming and error-prone task, and hence expensive, activity[3]. To cope with this problem„ different tools for automatically testing Android applications have been proposed[4].

Sapienz[5] is one of such tools, which has been proven to outperform several state-of-the-art tools like Dynodroid[6] and Monkey[7]. In recent years, a re-engineered version of the tool has been deployed in the software company Facebook[8]. Essentially, the Sapienz approach presented in Mao et al.[5] distinguishes from previous Android testing tools due to these two features:

i) A multi-objective evolutionary algorithm (NSGA-II [9]) that generates test sequences, simultaneously maximising statement coverage and fault detection while minimising test length.

ii) The representation of test cases as sequences of *atomic* and *motif* actions.

where an *atomic* action is an event that cannot be further decomposed (e.g., pressing down a key, taping the screen at a certain coordinate, etc.). On the other hand, *motif* actions are composed "events" (i.e., a sequence of *atomic* actions) that represents a usage pattern on the app.

Since these features (i.e., the NSGA-II algorithm combined with *motif* actions) were presented simultaneously, we would like to study the impact of each of them separately. What is more, Mao et al. [5] only performed cross-tool comparisons of their technique. This type of comparisons are undesirable since they might have conflating factors arising from implementation details.

In particular, we are interested in comparing different choices of evolutionary algorithms for Android test generation. Sell et al. [10] presented a study comparing different algorithms for Android test generation, but these algorithms are evaluated on the testing tool MATE [11]. Among other differences, MATE differs from Sapienz in that it uses a widget-based representation of individuals. Widgets are interactive components on an Android UI (such as buttons, text fields, etc.). In contrast, Sapienz individuals are based on sequences of actions that do not depend on widgets (i.e., the *atomic* and *motif* actions mentioned above use only screen coordinates). This difference affects how evolutionary operators (such as crossover and mutation) are performed. Therefore, we would like to study how different evolutionary algorithms might behave by implementing them on the Sapienz tool. Also, it has been shown that (at least for unit test generation), due to flat fitness landscapes and often simple search problems, Random Search [12] can perform as well as evolutionary algorithms, and sometimes even outperform them [13]. Thus, we would also like to study the choice of Random Search for Android test generation.

Therefore, in this paper, we aim to gain more insight into the effects of the main features of Sapienz: the choice of the NSGA-II multi-objective algorithm and the representation of the individuals using *motif* actions. Specifically, the contributions of this paper are the following:

- **Experiment design:** We present an empirical study comparing the effectiveness in terms of statement coverage and crash detection of 9 different algorithms for Android test generation (namely, Random Search, Random Search with *motif* actions, Standard GA, Monotonic GA, Steady-State GA, $(\mu + \lambda)$ EA, (μ, λ) EA, NSGA-II and NSGA-II with *motif* actions) using 38 experimental subjects (8 open-source and 30 popular closed-source real-world Android apps). This study (Section 4) involves both algorithms with and without *motif* actions, as well as a Random Search approach that will serve as a baseline for comparison. The total execution time was 202 days.
- **Experiment results:** We present the results of our empirical study (Sections 4.2, 4.3 and 4.4), leading to a total of 2430 data-points.

Our empirical study yields the following findings:

- Both NSGA-II and Random Search improve their coverage when test cases include *motif* actions.
- Among all the evolutionary algorithms considered in our study, NSGA-II is the one achieving the highest statement coverage. Surprisingly, NSGA-II does not distinguish with statistical significance from Random Search.
- Both NSGA-II and Random Search *marginally* improve their crash detection when test cases include *motif* actions. Nevertheless, this difference is not statistically significant.
- Whether using *motif* actions or not, NSGA-II and Random Search detect a similar number of crashes.

In summary, our experiment provides evidence that the causes for Sapienz performance gains are more attributable to the representation of test cases including *motif* actions rather than to the usage of a specific evolutionary approach.

This article extends our previous paper [14] by:

- Adding two new research questions related to crash detection. In our previous work we analysed the usage of evolutionary algorithms and *motif* actions only in terms of statement coverage. This paper now includes a specific section analysing their impact on Sapienz's crash detection performance. This new analysis follows the same rigour that was used for the previous work: 30 repetitions over 8 different apps and a thorough statistical analysis.
- Adding a research question that specifically targets closed-source real-world subjects. In our previous work we only experimented on subjects taken from the F-Droid[1] repository of open-source Android applications. This paper now includes a specific question analysing whether the results seen before also hold for popular applications taken directly from the Google Play Store[2].

[1] https://f-droid.org
[2] https://play.google.com/store/apps

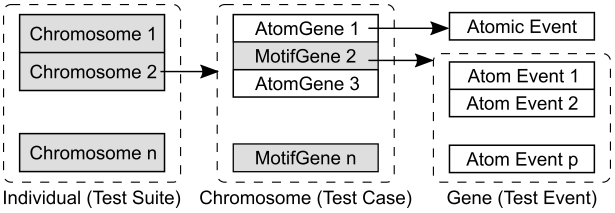

FIGURE 1 Representation of individuals in evolutionary algorithms as presented by Mao et al.[5]

- Providing a detailed description of the 7 algorithms involved in the study: Random Search, Standard GA, Monotonic GA, Steady-State GA, $(\mu + \lambda)$ EA, (μ, λ) EA, and NSGA-II.
- Providing the pseudocode of the MotifCore component which is used in Sapienz for executing both the atomic and *motif* actions.
- Updating the related work section with new empirical studies published since our previous paper.

The remainder of this article is organized as follows: Section 2 and Section 3 present the necessary background. In particular, they briefly introduce the techniques considered for the empirical study. Section 4 presents the empirical study along with the results obtained. Section 5 discusses closely related work. Finally, Section 6 concludes the paper and outlines potential future works.

2 | BACKGROUND

Evolutionary Algorithms (EAs) are a specific type of population-based meta-heuristic. These algorithms are used to solve optimisation problems and work by mimicking the process of natural selection. They typically start with a randomly generated population (i.e., a set of individuals). Each individual in the population represents a solution to the optimisation problem. Then, several iterations evolve the population towards a given goal. To produce a new generation, the fittest individuals are selected according to some selection mechanism (e.g., rank selection, tournament selection, etc.). After this, the new offspring is generated by applying genetic operators like crossover and mutation with certain parametric probabilities.

2.1 | Representation of individuals

The representation of individuals in Sapienz follows the Whole Test Suite generation (WTS)[15] principles. WTS evolves whole test suites for an entire coverage criterion at the same time (i.e., statements in the system under test). In WTS, each individual is a test suite (i.e., a set of test cases). Each test case can be seen as a "chromosome" of the individual. Then, each of these chromosomes will be represented as a sequence of genes (test events).

In our context, and to make comparisons between Sapienz and other algorithms fair, these genes will consist of a combination of *atomic* and *motif* genes. As defined by Mao et al.[5], an **atomic gene** is an event that cannot be further decomposed (e.g., press down a key, tap screen at a certain coordinate, etc.), while a **motif gene** is interpreted as a sequence of *atomic* events $\langle e_1, \ldots, e_p \rangle$. This representation is depicted in Figure 1.

Each *motif* gene defined represents a usage pattern on the app. These patterns follow common user behaviour, such as filling-in all text fields in the current screen and then clicking a button. As such, *motif* actions are based on the User Interface (UI) information available on the current screen.

2.2 | Optimisation goals

To guide the exploration towards a desired goal (i.e., covering all statements in the system), a fitness function must be defined. This function will evaluate each individual in the population. Then, individuals with better fitness values are more likely to survive and propagate their genes to further generations. In the context of test suite generation, the fitness functions are typically based on structural coverage criteria such as statement coverage or branch coverage.

In many cases, it is desirable to optimise the generated test cases towards multiple (possibly conflicting) optimisation goals. A simple mechanism for combining multiple coverage goals is through a weighted linear combination[16]. However, a linear combination requires non-conflicting optimisation goals (e.g., high statement coverage and high mutation score[17]). For instance, a tester would like to obtain a test suite with higher statement coverage and fewer test case length, for debugging and maintenance purposes. It is easy to see that these objectives are conflicting: increasing test length might lead to higher statement coverage while decreasing test length might reduce the coverage. Multi-objective evolutionary algorithms are especially focused on targeting several (possibly conflicting) goals simultaneously.

Algorithm 1: Random Search

Input : Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f
Output: Population of optimised individuals P

$P \longleftarrow \{\}$
while $\neg C$ **do**
  $N_P \longleftarrow$ GenerateRandomPopulation(p_s)
  PerformFitnessEvaluation(δ, N_P)
  $P \longleftarrow$ Selection$(s_f, P \cup N_P)$
return P

Algorithm 2: Standard GA

Input : Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

$P \longleftarrow$ GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, P)
while $\neg C$ **do**
  $N_P \longleftarrow \{\} \cup$ Elitism(P)
  while $|N_P| < p_s$ **do**
    $p_1, p_2 \longleftarrow$ Selection(s_f, P)
    $o_1, o_2 \longleftarrow$ Crossover(c_f, c_p, p_1, p_2)
    Mutation(m_f, m_p, o_1)
    Mutation(m_f, m_p, o_2)
    $N_P \longleftarrow N_P \cup \{o_1, o_2\}$
  $P \longleftarrow N_P$
  PerformFitnessEvaluation(δ, P)
return P

2.3 | Random Search

Random Search [12] (cf. Algorithm 1) is a simple approach for the test suite generation problem. It consists of repeatedly sampling candidates from the search space. Once the budget is exhausted, the fittest sampled individual is returned. Due to its simplicity, it is very useful as a baseline for studying the contributions of any proposed technique.

For unit test generation, it has been shown that Random Search performance is often as effective as other evolutionary algorithms, and it can also outperform them if the system under test is simple enough [13].

2.4 | Genetic Algorithms

In this study we will use the Standard Genetic Algorithm (GA) as described by Campos et al. [18] (cf. Algorithm 2). It starts by generating an initial random population of size p_s. The population is then evaluated using a fitness function δ. Each iteration (i.e., "generation") of the algorithm consists of building a new population and then evaluating each new individual in it. The new population is created by repeatedly choosing a pair of individuals from the current population and then, recombining them into two new individuals. The selection is done with a strategy s_f such as rank-based, elitism or tournament selection. The recombination is done with a crossover function c_f such as single-point or multiple-point, with probability c_p. Before inserting the offspring into the new population, mutation is applied independently on both, with probability m_p. This probability usually is $\frac{1}{n}$, where n is the number of genes in a chromosome. This ensures that, on average, at least one gene is mutated on each offspring, maintaining the diversity in the population.

Several variants of the Standard GA exist that strive to improve effectiveness. In particular, we consider the following alternatives:

Algorithm 3: Monotonic GA

Input : Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p

Output: Population of optimised individuals P

$P \longleftarrow$ GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, P)
while $\neg C$ **do**
  $N_P \longleftarrow \{\} \cup$ Elitism(P)
  while $|N_P| < p_s$ **do**
    $p_1, p_2 \longleftarrow$ Selection(s_f, P)
    $o_1, o_2 \longleftarrow$ Crossover(c_f, c_p, p_1, p_2)
    Mutation(m_f, m_p, o_1)
    Mutation(m_f, m_p, o_2)
    PerformFitnessEvaluation(δ, o_1)
    PerformFitnessEvaluation(δ, o_2)
    if Best(o_1, o_2) is better than Best(p_1, p_2) **then**
      $N_P \longleftarrow N_P \cup \{o_1, o_2\}$
    else
      $N_P \longleftarrow N_P \cup \{p_1, p_2\}$
  $P \longleftarrow N_P$
return P

Algorithm 4: Steady-state GA

Input : Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p

Output: Population of optimised individuals P

$P \longleftarrow$ GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, P)
while $\neg C$ **do**
  $p_1, p_2 \longleftarrow$ Selection(s_f, P)
  $o_1, o_2 \longleftarrow$ Crossover(c_f, c_p, p_1, p_2)
  Mutation(m_f, m_p, o_1)
  Mutation(m_f, m_p, o_2)
  PerformFitnessEvaluation(δ, o_1)
  PerformFitnessEvaluation(δ, o_2)
  if Best(o_1, o_2) is better than Best(p_1, p_2) **then**
    $P \longleftarrow P \setminus \{p_1, p_2\} \cup \{o_1, o_2\}$
  else
    $P \longleftarrow P \setminus \{o_1, o_2\} \cup \{p_1, p_2\}$
return P

- **Monotonic GA:** Similar to the Standard GA, but it only includes either the best offspring or the best parent in the next population (cf. Algorithm 3). This ensures that achieved fitness value does not decrease as the population evolves.
- **Steady State GA:** This algorithm uses the same replacement strategy as the Monotonic GA, but instead of creating a new population in each generation, the offspring replaces parents in the current population immediately after they are mutated and evaluated (cf. Algorithm 4).

Algorithm 5: $(\mu + \lambda)$ Evolutionary algorithm

Input : Stopping condition C, Fitness function δ, Population size μ, Offspring size λ, Selection function s_f, Mutation function m_f, Mutation probability m_p

Output: Population of optimised individuals P

```
P ⟵ GenerateRandomPopulation(μ)
PerformFitnessEvaluation(δ, P)
while ¬C do
    O ⟵ {}
    forall p ∈ P do
        for i ⟵ 1, λ/μ do
            o ⟵ Mutation(m_f, m_p, p)
            O ⟵ O ∪ {o}
    PerformFitnessEvaluation(δ, O)
    P ⟵ select best μ individuals from P ∪ O
return P
```

Algorithm 6: (μ, λ) Evolutionary algorithm

Input : Stopping condition C, Fitness function δ, Population size μ, Offspring size λ, Selection function s_f, Mutation function m_f, Mutation probability m_p

Output: Population of optimised individuals P

```
P ⟵ GenerateRandomPopulation(μ)
PerformFitnessEvaluation(δ, P)
while ¬C do
    O ⟵ {}
    forall p ∈ P do
        for i ⟵ 1, λ/μ do
            o ⟵ Mutation(m_f, m_p, p)
            O ⟵ O ∪ {o}
    PerformFitnessEvaluation(δ, O)
    P ⟵ select best μ individuals from O
return P
```

2.5 | Evolutionary Algorithms

The $(\mu + \lambda)$ Evolutionary Algorithm (EA)[19] is a mutation-based algorithm[20] (cf. Algorithm 5). In this case, μ represents the size of parents and λ the size of the offspring. For each individual in the current population, mutation is applied independently on each gene with probability $\frac{1}{n}$. After mutation, the best μ individuals are selected among a combined pool of parents and offspring to constitute the new population. Therefore, parents will be replaced only if a better offspring is found. A variant of this is a (μ, λ) EA, where the parents are discarded and the new μ individuals are only selected among the offspring (cf. Algorithm 6).

3 | THE Sapienz APPROACH

Sapienz[5] is a multi-objective Android test generation technique aiming at maximising code coverage and fault revelation, while minimising the length of fault-revealing test sequences.

In order to cope with the conflicting goals (i.e., maximising coverage while minimising test length), Sapienz employs the NSGA-II[9] multi-objective evolutionary algorithm (cf. Algorithm 7), which is widely-used in search-based software engineering (SBSE) research[21]. This algorithm uses a fast

Algorithm 7: NSGA-II multi-objective evolutionary algorithm

Input : Stopping condition C, Multi-objective fitness function δ, Selection function s_f, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

```
P ⟵ GenerateRandomPopulation(μ)
PerformFitnessEvaluation(δ, P)
while ¬C do
    O ⟵ {}
    forall p ∈ P do
        o ⟵ Mutation(m_f, m_p, p)
        O ⟵ O ∪ {o}
    F ⟵ sortNonDominated(P ∪ O, |P|)
    P' ⟵ {}
    for each front F in F do
        if |P'| ≥ |P| then
            break
        calculate crowding distance for F
        for each individual f in F do
            P' ⟵ P' ∪ {f}
    P' ⟵ sorted(P', ≺_c)
    P ⟵ P'[0 : |P|]
return P
```

non-dominated sorting with a selection operator which creates a mating pool by i) combining the parent and child populations, and ii) selecting the best N solutions according to fitness and spread. During the selection process, all objectives are combined using a Pareto-optimal [22] search-based approach. Formally, an individual x is said to be *dominated* by another individual y ($x \prec y$) according to a fitness function if and only if x is partially less than y:

$$\forall\, i = 1, \ldots, n,\ f_i(x) \leq f_i(y) \quad \wedge \quad \exists\, i = 1, \ldots, n : f_i(x) < f_i(y)$$

Then, a Pareto-optimal set consists of all the *non-dominated* individuals (belonging to all solutions S):

$$P^* \triangleq \{x \in S \mid \nexists\, y \in S, x \prec y\}$$

Therefore, a solution to the multi-objective optimisation problem is not a single point in the search space (as in WTS generation is), but a family of points. In practice, this means individuals with longer test sequences are not discarded when they are the only ones finding faults, nor where they are necessary to achieve higher code coverage. Thus, through its use of Pareto optimality, Sapienz progressively replaces the longer sequences with shorter test sequences when they are equally good.

In Sapienz, one individual is a test suite. As mentioned previously, each individual consists of several chromosomes (test cases) and each chromosome contains multiple genes (test events), which consist of a random combination of *atomic* and *motif* genes. Event sequences in the test cases are generated and executed by a special component called MotifCore. This component combines random fuzzing and systematic exploration, corresponding to the two types of genes Sapienz supports: *atomic* genes and *motif* genes. The behaviour of each *motif* gene depends on the UI information available at the moment of its execution. These genes are used to perform common user patterns during the exploration, such as filling-in all text fields on the screen and clicking the actionable buttons. The MotifCore's full algorithm is depicted in Algorithm 8.

4 | EMPIRICAL STUDY

We would like to investigate how the evolutionary algorithm and the representation of individuals in Sapienz affect the overall performance of test generation. Thus, we pose the following research questions:

RQ1 (Representation) *What is the contribution of* motif *genes in* Sapienz ***coverage****?*

RQ2 (Algorithm) *What is the contribution of the NSGA-II evolutionary algorithm in* Sapienz ***coverage****?*

Algorithm 8: Simplification of the MotifCore exploration strategy as presented by Mao et al.[5]

```
Input  : App Under Test A, test sequence T = ⟨E_1, E_2, ..., E_n⟩, static strings S, UI Model M
Output: Updated UI Model M
for each event E in T do
    if E is an atomic event then
        execute atomic event E and update M
    if E is a motif event then
        currentActivity ⟵ extractCurrentActivity(A)
        uiElementSet ⟵ extractUiElement(currentActivity)
        for each element w in uiElementSet do
            if w is EditText widget then
                seed string s ∈ S into w
            else
                exercise w according to motif patterns in E
            update M
return M
```

RQ3 (Representation) *What is the contribution of* motif *genes in* Sapienz ***crash detection****?*

RQ4 (Algorithm) *What is the contribution of the NSGA-II evolutionary algorithm in* Sapienz ***crash detection****?*

RQ5 *How does the results on open-source apps compare to real-world* ***closed-source*** *ones?*

In order to answer these questions, we conducted an empirical study. In the following subsection, we describe the experimental setup.

4.1 | Experimental setup

We conduct two studies to answer the above research questions: Study "A" addresses RQ1 to RQ4 and Study "B" addresses RQ5. In both cases, we tried to mimic the experimental setup used in Study #2 presented by Mao et al.[5] as close as possible.

4.1.1 | Selection of Subjects Under Test

For Study "A", we chose to use the 10 subjects already used in Study #2 of Mao et al.[5]. These subjects were randomly picked by the authors of Sapienz from the F-Droid repository of open-source Android applications. Two subjects were preliminary discarded (`BabyCare` and `Hydrate`) due to missing source code (`BabyCare`) and dependencies on libraries that are no longer supported by the Android platform version used in our study (`Hydrate`). The remaining 8 open-source subjects used in our study are shown in Table 1.

For Study "B", we randomly selected 30 closed-source apps from the Google Play Store. These apps were chosen after a meticulous selection process. The selection pool started with 531 apps: 200 taken from the overall Top 200 Free apps ranking, and 331 from the union of all Top 50 Free apps ranking in any specific category (discarding the ones already present in the first 200 apps). From these 531 apps, only 275 support Android KitKat (API 19) and x86 devices. The first is needed because it is the latest Android version supported by the Sapienz prototype publicly available at GitHub. The second is needed to run the apps on emulators. Of those 275 apps, we randomly selected 30 suitable for the ELLA-customized[23] instrumentation. It is worth pointing out that these are real third-party apps that may use obfuscation and anti-debugging techniques, and could be more difficult to instrument. The 30 closed-source subjects are shown in Table 2.

We have not chosen any of the apps presented in studies #1 and #3 of Mao et al.[5]. Although the 68 apps Study #1 of Mao et al.[5] could have been used in our Study "A", we preferred to use the 10 apps presented in Study #2 of Mao et al.[5], since they had already been used in a study with statistical analysis. Regarding the 1,000 Google Play apps used in Study #3 of Mao et al.[5], their package names are not present in the article so we were unable to take a look at them.

Our rationale behind the selection of subjects for our studies "A" and "B" is meant to minimize possible threats to external validity. On one hand, we decided to have one study using 8 open-source subjects. Since open-source software might not always be the best representative of real-world subjects, we opted to favour a larger number of repetitions (30 per combination of subject & algorithm) to gain better statistical significance. On

TABLE 1 Open-source subjects used in Study "A".

Subject	Description	Ver.	Date	LOC
Arity	Scientific calculator	1.27	2012-02-11	2,821
BookWorm	Book collection manager	1.0.18	2011-05-04	7,589
DroidSat	Satellite viewer	2.52	2015-01-11	15,149
FillUp	Calculate fuel mileage	1.7.2	2015-03-10	10,400
JustSit	Meditation timer	0.3.3	2012-07-26	728
Kanji	Character recognition	1.0	2012-10-30	200,154
L9Droid	Interactive fiction	0.6	2015-01-06	18,040
Maniana	User-friendly todo list	1.26	2013-06-28	20,263

TABLE 2 Closed-source subjects used in Study "B".

Subject	Description	Ver.	Date	Downloads
AccuWeather	Weather alerts & live forecast info	5.8.6-free	2021-07-02	100,000,000+
Bitmoji	Create your own personal emoji	10.47.177	2021-05-31	100,000,000+
Body Temperature Records	Fever tracking application	1.0	2021-07-15	10,000+
Calorie Counter by Lose It	A calorie counter & food diary diet app	7.1.1	2021-07-07	10,000,000+
ClipKey	Clipboard manager	1.3.2	2017-05-07	500,000+
Craigslist	Online classifieds	1.14.2	2021-04-13	1,000,000+
CT Prepares	Emergency information for Connecticut residents	1.0.3	2016-11-01	5,000+
Daily Mail Online	News app	4.0.1	2021-08-02	5,000,000+
Dark Horse Comics	Read comics	1.3.17	2020-12-18	1,000,000+
Durham Bus Tracker	Real time school bus location	1.6.0	2019-12-13	50,000+
Ecobee	Smart home devices manager	3.2.2	2021-07-06	1,000,000+
Google Translate	Text translation app	4.4.0.RC01.104701208	2021-06-01	1,000,000,000+
Indeed Job Search	Job search app	4.6	2021-06-02	100,000,000+
Interval Timer	Minimalistic timer	1.2.6	2021-05-24	5,000,000+
Mobills	Budget Planner & Track your Finances	5.3.7	2021-08-23	5,000,000+
Move to iOS	App for migrating to iOS	3.1.2	2021-05-18	50,000,000+
My Baby Firework	Fireworks display with sound	2.116.5	2019-12-16	1,000,000+
My Baby Piano	Piano display with sound	2.146.9	2019-12-16	5,000,000+
MyChart	Health information storage app	5.1	2021-05-17	10,000,000+
Namshi	Fashion & Beauty Online Shopping	3.0.4	2021-08-12	10,000,000+
Pandora	Streaming Music, Radio & Podcasts	8.7.1	2021-05-20	100,000,000+
Remote for Roku by Codematics	Smart TV control app	1.29	2021-06-15	1,000,000+
Roku	Official Roku Remote Control	3.6.0.2281118	2021-05-08	10,000,000+
Snapseed	A professional photo editor	2.19.0.201907232	2020-04-14	100,000,000+
SoftList	Shopping List	2.5.0	2021-05-26	1,000,000+
SwingU	Golf GPS & Scorecard	5.0.62	2021-06-28	1,000,000+
USA Today	News app	3.1.3	2021-06-17	5,000,000+
VINDecoded	VIN Check Report & History	8.3.0.0	2021-05-28	500,000+
Weather by WeatherBug	Live Radar Map & Forecast	5.4.4.38	2021-06-30	10,000,000+
Yelp	Find Food, Delivery & Services Nearby	9.33.0	2021-06-03	50,000,000+

the other hand, we decided to have another study (i.e., Study "B") using closed-source subjects. Since these subjects were taken directly from the Top Free rankings on the Google Play Store, we can assume that they are good representatives of popular real-world subjects. Thus, in this case we settled with a single repetition per combination of subject and algorithm.

4.1.2 | Implementation

As we explained in Section 3, Sapienz[5] implements a multi-objective NSGA-II evolutionary algorithm, including a representation of individuals as sequences of both *atomic* and *motif* genes. We extended the latest publicly available version of Sapienz at GitHub[24], adding the algorithms described in Section 2. These algorithms were implemented using a very simple single-objective fitness function: an individual's fitness is assigned from the accumulated coverage achieved while running. Although it is possible to implement a multi-objective variant, we still decided to use a single-objective fitness function since an important part of this empirical study consists on validating whether the use of a multi-objective evolutionary algorithm (e.g., NSGA-II) actually improves effectiveness or if simpler single-objective evolutionary algorithms might achieve similar results.

As the authors have stated, the current available version of Sapienz at GitHub[24] is regarded as "out-of-date and no longer supported", with the latest activity in the version history recorded in May 2016. Due to this, we enhanced the latest version of Sapienz fixing some issues such as proper time budget management, handling of timeouts when issuing commands to emulators, recovery from an emulator crash. Some of the problems in the original implementation that we had to solve were:

- Initialization of emulators used out-of-date parameters that were no longer supported.
- Lack of time budget management and reliable timeout for commands to the emulator.
- Lack of proper device management to ensure that the failure of one or more devices when running a test case does not stop the whole experimentation.
- Missing runtime information about the efficiency of each algorithm.
- Finally, hardcoded commands all over the source code prevented us from easily extending the runner to use other search-based algorithms.

Besides Sapienz, we have considered 8 algorithms that we have implemented in our extension of Sapienz. To be specific, Random Search (with and without *motif* genes), Standard GA, Monotonic GA, Steady State GA, $\mu + \lambda$ EA, (μ, λ) EA and NSGA-II (i.e., Sapienz without *motif* genes). All the implemented algorithms and the enhanced Sapienz implementation are publicly available on GitHub[3]. For this article, we considered a subset of the algorithms studied in[18]. We discarded many-objective search algorithm (i.e., MOSA[25] and DynaMOSA[26]) since these algorithms were originally designed to work with approach level and branch distance, which are not provided by neither EMMA[27] nor ELLA-customized[23] tools, which we use for collecting coverage information.

It is worth noticing that the MotifCore component of Sapienz (that handles the *motif* genes) was not modified. In other words, the *atomic* and *motif* genes supported in this study are the same that were proposed by Mao et al.[5]. To generate individuals without *motif* genes, we simply filter the content of the test cases generated by the MotifCore component to leave only the *atomic* genes.

4.1.3 | Parameters selection

The parameters were selected following the choices made in Study #2 of Mao et al.[5]. For the crossover function, the uniform crossover operator was used. For the mutation function, a combination of shuffling and one point crossover was used. The crossover and mutation probabilities were set to 0.7 and 0.3, respectively. The selection function used for NSGA-II algorithm was the same as the one depicted by their original authors[9]. The selection function used for the single-objective EAs was roulette selection. The maximum number of generations was set to 100, although none of the evolutionary algorithms accomplished this amount of generations. Population size for each generation was 50 individuals while individuals were composed of 5 test cases. The initial length of each test case was randomly selected between 20 and 500 events. All of these parameters were kept throughout all the executions. We opted to keep the parameters constant to ensure that comparison between the algorithms is fair; since tuning the parameters for each algorithm might change their effectiveness[28,29].

4.1.4 | Experiment Procedure

All test cases were generated on Android KitKat (API 19) because it is the latest Android version supported by the Sapienz prototype publicly available at GitHub. All techniques are fully automated and no manual intervention was provided (e.g., logins) during the execution of the test generators.

For each of these executions, we set a maximum time budget of 2 hours. We conservatively doubled the original 1 hour time budget used in Mao et al.[5] to mitigate any emulator or hardware difference. This is by no means a threat to evolutionary approaches as more time budget allows more fitness evaluations (i.e., more generations).

[3]https://github.com/FlyingPumba/evolutiz

For Study "A", we executed all test generation algorithms in the Microsoft Azure Cloud Computing Platform[4]. The type of virtual machine chosen was D16s_v3, which features 16 cores and 64GB of RAM. The operating system installed on these virtual machines was Ubuntu 14.04. On each 16 core machine, 16 Android emulators were launched. These emulators are all used at the same time when generating test cases for one app.

As was mentioned before, for Study "A" we decided to run 30 repetitions on each open-source subject to gain statistical significance. Therefore, the total experimentation time for Study "A" was 9 *algorithms* $\times 8$ *apps* $\times 30$ *repetitions* $\times 2$ *hours each* $= 4,320hs$ (i.e., 180 days a 16 core machine). If we consider the invested time for emulation, this represents $4,320hs \times 16\ emulators = 69.120hs$ (i.e., $2,880$ days).

On the other hand, Study "B" was executed on a desktop computer with 8 cores and 32GB of RAM running Ubuntu 18.04. On this machine, only 8 Android emulators were launched for each run. In contrast to Study "A", only a single repetition was run for each combination of algorithm and app. Thus, the total experimentation time for the closed-source subjects was 9 *algorithms* $\times 30$ *apps* $\times 1$ *repetition* $\times 2$ *hours each* $= 540hs$ (i.e., 22.5 days of a 8 core machine). If we consider the invested time for emulation, this represents $540hs \times 8\ emulators = 4,320hs$ (i.e., 180 days).

In the Sapienz prototype available at GitHub, the MotifCore component is used for both generating new random test sequences and for executing a given test sequence. The latter is required for obtaining the statement coverage of a test suite. However, we found that this component has a known defect which hinders obtaining the correct fitness value while generating new test sequences. Hence, it was required to re-execute the generated random tests to obtain their correct fitness value. Consequently, to avoid penalizing those approaches that heavily rely on random test generation (such as Random Search) we do not consume any time budget during random test case generation with the MotifCore component.

4.1.5 | Experiment Analysis

For Study "A", statement coverage for the generated test suites was obtained automatically using the EMMA tool[27]. For Study "B", method coverage was used instead, provided automatically by the ELLA-customized tool[23]. This tool is an improved version (e.g., it adds multi-dex support) of the original ELLA tool[30]. It was developed and first used in the paper by Wang et al.[31]. For the single objective genetic algorithms (i.e., Standard GA, Monotonic GA, Steady State GA, $\mu + \lambda$ EA and (μ, λ) EA) we report the statement coverage achieved by the best individual (i.e., the one with the highest coverage) in the last generation. For Random Search, we report the highest coverage achieved by any individual randomly sampled. For the multi-objective NSGA-II algorithm, since the solution is not a single individual but a set of individuals (i.e., the Pareto-optimal front), we report the highest statement coverage achieved by an individual in the Pareto-optimal front.

The number of crashes detected for a given test suite was computed as the number of unique crashes triggered during its execution. This is important since the same crash may arise from different test sequences. As defined by Mao et al.[5], a crash is considered to be unique when its stack trace differs from all others. We also excluded all crashes which were not caused by faults from the subjects (e.g., those caused by the Android system).

For the statistical analysis, we followed the same procedures as Panichella et al.[32] and Campos et al.[33] for comparing different randomized algorithms over a set of subjects. Specifically, we apply the Friedman test[34] with significance level $\alpha = 0.05$.

The Friedman test is a non-parametric test for multiple-problem analysis and it departs from the traditional tests for significance (e.g., the Wilcoxon test) since it computes the ranking between algorithms over multiple independent problems, i.e., Android applications in our case. A significant $p - value$ indicates that the null hypothesis (i.e., no algorithm in the tournament performs significantly different from the others) has to be rejected in favour of the alternative one (i.e., the performance of algorithms is significantly different from each other). If the null hypothesis is rejected, we use the post hoc Conover's test for pairwise multiple comparisons. Such a test is used to detect pairs of algorithms that are significantly different. Finally, $p-values$ obtained with the post hoc test are adjusted with the Holm-Bonferroni procedure to correct the statistical significance level ($\alpha = 0.05$) in case of multiple comparisons.

In the cases where we want to obtain a more detailed comparison between two algorithms for a given subject, we use the Wilcoxon-Mann-Whitney U-test to determine whether there is a statistically significant difference and the Vargha-Delaney A_{12} effect size to measure this difference (if any).

4.2 | Study "A" results: coverage

Table 3 summarises the results of the experiment described in the previous section. We report the overall statement coverage and the rank of each algorithm, procured by the Friedman test based on their average performance. Table 3 also reports the standard deviation and confidence intervals (CI) using bootstrapping at 95% significance level of the statement coverage achieved.

Among all the algorithms evaluated, NSGA-II + *motif* genes (i.e., Sapienz) is the one that achieves the highest overall statement coverage (47%) and CI. The $p - value$ obtained from the Friedman test is 1.97e-09 . This means that we can reject the null hypothesis of the Friedman test (i.e.,

[4]https://azure.microsoft.com

TABLE 3 Summary of coverage results for Study "A": Overall coverage, standard deviation and rank of each algorithm based on their average performance, which is statistically significant according to the Friedman test (p-value is < 0.0001, full data is available on Table 4). For averaged coverage values we also report confidence intervals (CI) using bootstrapping at 95% significance level.

Algorithm	Ranking Mean	Ranking SD	Overall Coverage Mean	Coverage SD	CI
NSGA-II + MG (Sapienz)	1.25	0.71	47.87	17.22	[45.68, 50.06]
Random Search + MG	2.12	0.35	46.95	17.65	[44.71, 49.23]
NSGA-II	3.00	1.41	44.07	18.71	[41.70, 46.47]
Random Search	4.19	0.37	43.78	18.57	[41.46, 46.16]
$(\mu + \lambda)$ EA	5.12	1.36	41.79	17.79	[39.47, 44.06]
Monotonic GA	5.56	0.82	40.70	17.62	[38.46, 42.91]
(μ, λ) EA	7.62	1.06	37.23	17.85	[35.00, 39.51]
Standard GA	8.00	0.76	34.49	19.43	[32.05, 36.96]
Steady State GA	8.12	0.83	33.23	19.80	[30.70, 35.74]

TABLE 4 Full ranking of coverage achieved by algorithms for each subject on Study "A".

	$(\mu + \lambda)$ EA	(μ, λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG	Standard GA	Steady State GA
Arity	5.00	9.00	6.00	1.00	3.00	4.00	2.00	8.00	7.00
BookWorm	3.00	8.00	4.00	6.00	1.00	5.00	2.00	7.00	9.00
DroidSat	8.00	6.00	4.50	2.00	1.00	4.50	3.00	8.00	8.00
FillUp	5.00	7.00	6.00	3.00	1.00	4.00	2.00	8.00	9.00
JustSit	5.00	7.00	6.00	3.00	1.00	4.00	2.00	9.00	8.00
Kanji	5.00	8.00	6.00	3.00	1.00	4.00	2.00	7.00	9.00
L9Droid	5.00	7.00	6.00	3.00	1.00	4.00	2.00	9.00	8.00
Maniana	5.00	9.00	6.00	3.00	1.00	4.00	2.00	8.00	7.00
Mean	5.12	7.62	5.56	3.00	1.25	4.19	2.12	8.00	8.12

there is at least one algorithm that differs from the rest). Table 4 shows the rankings achieved by each algorithm for every application. For example, for subject `Arity`, NSGA-II achieved the best statement coverage, while (μ, λ) EA performed the worst in terms of that metric.

Table 5 shows the $p - values$ obtained by the post hoc Conover's test for pairwise comparison. These $p - values$ indicate whether there is statistical significance or not for each pair of algorithms. For example, although Table 3 shows that $(\mu + \lambda)$ EA achieved higher ranking than Monotonic GA, Table 5 indicates that the $p - value$ obtained for the post hoc Conover's test is far greater than 0.05. Thus, there is not enough evidence to support with statistical confidence that the average for $(\mu + \lambda)$ EA is different from Monotonic GA.

Figure 2 shows visually the overall statement coverage achieved by each algorithm.

4.2.1 | RQ1: What is the contribution of *motif* genes in Sapienz coverage?

To answer this question, we conduct a pairwise tournament between NSGA-II with *motif* genes (i.e., Sapienz) and NSGA-II. Furthermore, to understand whether *motif* genes contribute to major gains (even without using an evolutionary algorithm), we also add to the pairwise tournament Random Search and Random Search with *motif* genes.

Table 6 summarises the results of the pairwise tournament. Given a particular subject, Algorithm X is considered to be better than Algorithm Y for that subject if the result of the Wilcoxon-Mann-Whitney U-test gives a $p - value < 0.05$ (i.e., we can say with statistical confidence that Algorithm X is different from Algorithm Y) and the Vargha-Delaney A_{12} effect size is greater than 0.5. Colloquially, this means that Algorithm X performs significantly better on a higher number of comparisons than Algorithm Y. If the $p - value$ of the U-test is greater or equal than 0.05, we can not conclude that Algorithm X is neither better nor worse than Algorithm Y. The positions in the tournament are decided by ranking the

TABLE 5 Results of the post hoc Conover's test for pairwise analysis of coverage achieved on Study "A". A $p - value$ less than 0.05 for algorithms X and Y means there is enough evidence to claim they are different with statistically significance.

	$(\mu+\lambda)$ EA	(μ,λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG	Standard GA
(μ,λ) EA	< 0.05	-	-	-	-	-	-	-
Monotonic GA	1.000	< 0.05	-	-	-	-	-	-
NSGA-II	< 0.05	< 0.05	< 0.05	-	-	-	-	-
NSGA-II + MG (Sapienz)	< 0.05	< 0.05	< 0.05	< 0.05	-	-	-	-
Random Search	0.410	< 0.05	0.058	0.141	< 0.05	-	-	-
Random Search + MG	< 0.05	< 0.05	< 0.05	0.462	0.462	< 0.05	-	-
Standard GA	< 0.05	1.000	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	-
Steady State GA	< 0.05	1.000	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	1.000

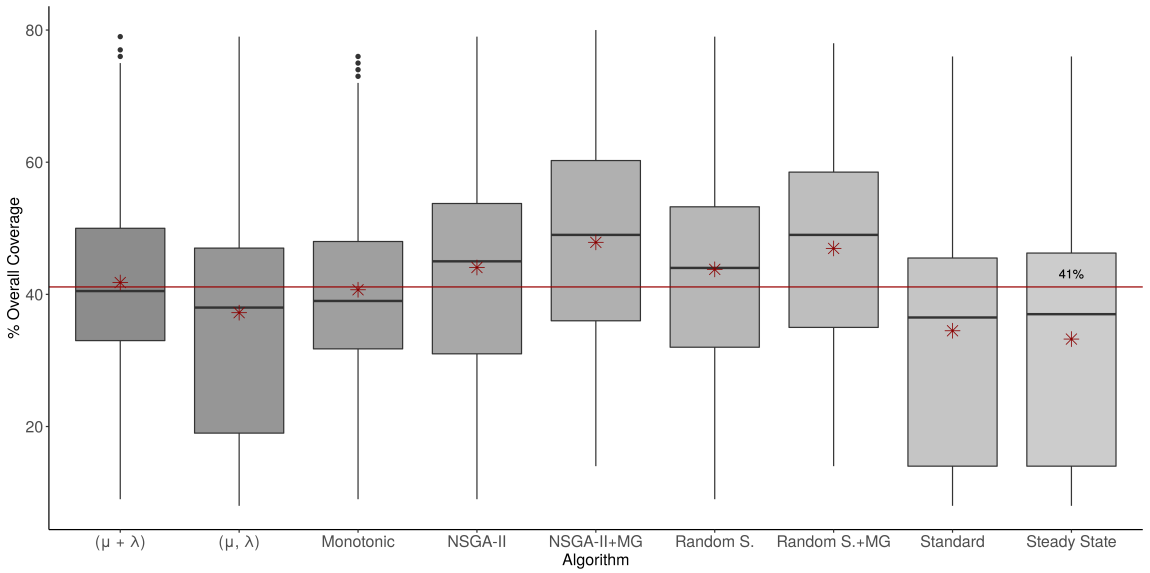

FIGURE 2 Overall coverage achieved by each algorithm on Study "A". Middle line of each boxplot marks the median, black circles represent outliers, $\star$ symbol shows the mean, and the red line represents the mean of all coverages (41%).

differences between the "Better than" and "Worse than" columns. For example, Sapienz has a difference of $16 - 0 = 16$, while Random Search with *motif* genes has a difference of $10 - 3 = 7$, which means the former should be higher in the tournament than the latter.

We can see that the first position is assigned to Sapienz with a significantly better performance in 16 out of 24 comparisons and an average effect size of 0.86. Furthermore, in the remaining 8 of the 24 comparisons, Sapienz is not significantly worse. Surprisingly, the second position goes to Random Search with *motif* genes. This algorithm has a significantly better performance in 10 out of 24 comparisons and an average effect size of 0.90.

Overall, we can see in Table 6 a clear improvement of statement coverage on those algorithms that include *motif* genes over their counterparts without *motif* genes. In terms of statistical significance, Table 5 shows there is enough statistical evidence to hold that there is a difference between both: NSGA-II and NSGA-II with *motif* genes (i.e., Sapienz) and between Random Search and Random Search with *motif* genes. We conjecture that this increment is due to *motif* genes using a more compact representation than *atomic* genes. In other words, a complex user pattern can be represented either with a sequence of N *atomic* events or one *motif* gene. This means that, as long as that particular gene keeps propagating across generations, the pattern will survive in the population. On the other hand, if that same pattern is sprayed out into several dozens of events, it would be easier for crossover and mutation operators to break it and lose its benefits. In summary, a more compact representation of test cases might help trim the search space and achieve higher statement coverage.

TABLE 6 Pairwise comparison of coverage achieved by algorithms with and without *Motif Genes* on Study "A". "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., $p - value$ of Wilcoxon-Mann-Whitney U-test less than 0.05) better and worse, respectively. Columns $\widehat{A}_{12}$ give the average effect size.

Algorithm	Tournament position	Overall Coverage Mean	Better		Worse	
			than	$\widehat{A}_{12}$	than	$\widehat{A}_{12}$
NSGA-II + MG (Sapienz)	1.00	47.87	16/24	0.86	0/24	-
NSGA-II	3.00	44.07	1/24	0.66	10/24	0.07
Random Search + MG	2.00	46.95	10/24	0.90	3/24	0.32
Random Search	4.00	43.78	0/24	-	14/24	0.14

Result 1. RQ1: Motif genes have a *significant* impact on Sapienz coverage. In fact, both NSGA-II and Random Search improve their coverage when test cases include motif genes.

4.2.2 | RQ2: What is the contribution of the NSGA-II evolutionary algorithm in Sapienz coverage?

TABLE 7 Pairwise comparison of coverage achieved by evolutionary algorithms and Random Search on Study "A". "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., $p-value$ of Wilcoxon-Mann-Whitney U-test less than 0.05) better and worse, respectively. Columns $\widehat{A}_{12}$ give the average effect size.

Algorithm	Tournament position	Overall Coverage Mean	Better		Worse	
			than	$\widehat{A}_{12}$	than	$\widehat{A}_{12}$
NSGA-II	1.00	44.07	36/48	0.88	2/48	0.31
Random Search	2.00	43.78	33/48	0.90	1/48	0.34
Standard GA	6.50	34.49	1/48	0.73	30/48	0.15
Monotonic GA	4.00	40.70	17/48	0.78	14/48	0.14
Steady State GA	6.50	33.23	1/48	0.68	30/48	0.09
$(\mu + \lambda)$ EA	3.00	41.79	24/48	0.79	13/48	0.21
(μ, λ) EA	5.00	37.23	5/48	0.83	27/48	0.18

TABLE 8 Comparison of coverage achieved by evolutionary algorithms and Random Search on Study "A". Statistically significant effect sizes are shown in bold.

Algorithm	Overall Coverage Mean	Random Search	
		$\widehat{A}_{12}$	p
NSGA-II	44.07	0.56	0.141
Random Search	43.78	-	-
Standard GA	34.49	**0.12**	$\mathbf{< 0.05}$
Monotonic GA	40.70	0.23	0.058
Steady State GA	33.23	**0.08**	$\mathbf{< 0.05}$
$(\mu + \lambda)$ EA	41.79	0.30	0.410
(μ, λ) EA	37.23	**0.15**	$\mathbf{< 0.05}$

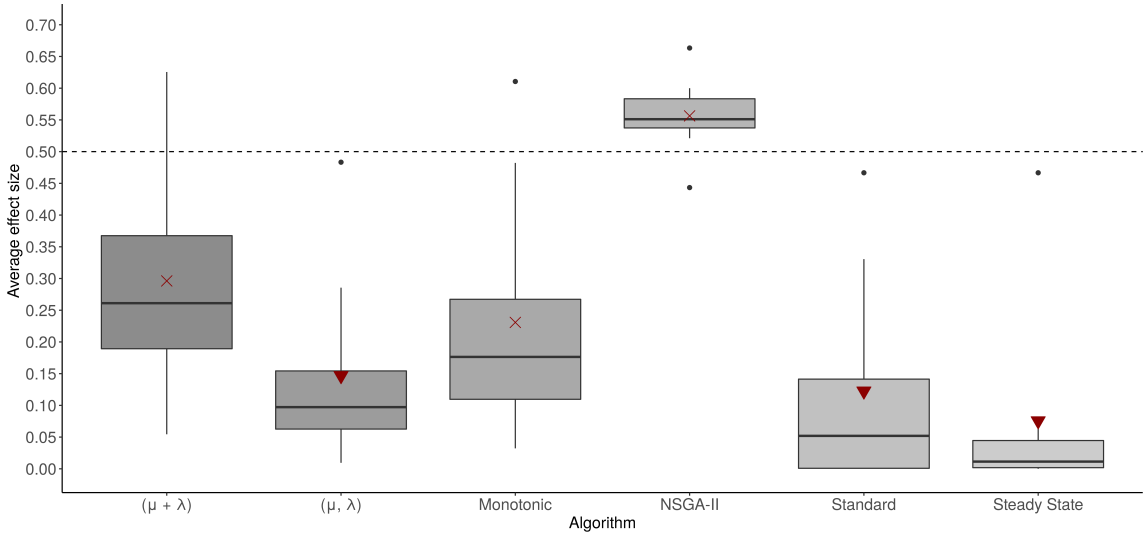

FIGURE 3 Effect size $\widehat{A}_{12}$ of coverage achieved by EA X vs Random Search on Study "A". Middle line of each boxplot marks the median, black circles represent the outliers, ▲ represents the mean of a significant effect size greater than 0.5 (i.e., EA X performs significantly better than Random Search), ▼ the mean of a significant effect size lower than 0.5 (i.e., EA X performs significantly worse than Random Search), $\times$ the mean of a no significant effect size.

To answer this question, we conduct a pairwise tournament among NSGA-II and the evolutionary algorithms presented in Section 2. As a baseline for comparison, we also included Random Search to the tournament.

Table 7 summarises the results of the pairwise tournament. Among all the evolutionary algorithms evaluated, NSGA-II is the one that achieves the highest overall statement coverage (44%). It is also significantly better than the other algorithms in 36 out of 48 comparisons, and only worst in 2 out of 48. An averaged $\widehat{A}_{12}$ of 0.88 means that in the comparisons for which NSGA-II is significantly better than another algorithm, it obtains a highest statement coverage in 88% of the repetitions.

This result is consistent with other studies such as the one performed by Campos et al.[33] for Java unit test case generation in which multi-objective algorithms such as MOSA (a variation of NSGA-II) and DynaMOSA (a latter variation of MOSA) showed higher coverage over single objective search-algorithms.

It is worth noticing that Random Search obtained the second-best place in this pairwise tournament. What is more, the $p-value$ obtained from the Conover's post hoc test when comparing NSGA-II vs Random Search is higher than 0.05. This means that there is not enough evidence to reject the null hypothesis (i.e., that NSGA-II is different from Random Search). To better understand what is the magnitude of this difference between Random Search and evolutionary algorithms, we conducted a more detailed comparison and then calculated the average effect size. Table 8 shows the results of this comparison. Figure 3 shows the results visually. As we can see, NSGA-II is the only algorithm that achieves an average effect size higher than 0.5, but there is no statistical significance.

In other words, Random Search is at least as good as NSGA-II, Monotonic GA and $(\mu+\lambda)$ EA. For all the other evolutionary algorithms, Random Search is better with statistical significance. In summary, this means that EAs are not contributing substantially to gain better statement coverage in Android test generation. This result is also consistent with the study presented by Sell et al.[10] for Android test generation in which single-objective and multi-objective algorithms do not perform better than random algorithms, and sometimes they even perform slightly worse.

In order to optimise a population towards a given objective, evolutionary algorithms require to evolve as many generations as possible. The cost of a fitness evaluation directly affects the number of generations the EA can achieve. In particular, for Android test generation, in order to obtain statement coverage for a given individual, evolutionary approaches need to: push the test case to a device/emulator, start the application, run test case, gather fitness information and pull it from device/emulator. In [10], Sell et al. suggested that high execution costs hamper any meaningful evolution for search algorithms. In our study, we observed that the fitness evaluation might take up to 60 seconds for a test case, depending on its length. Overall, this resulted in approximately 30 generations for each EA on average. Having a population of 50 individuals, the maximum number of fitness evaluations achieved within the time budget of 2 hours was on average $50 \times 30 = 1.500$ fitness evaluations. Similar execution times can also be found in the work of Vogel et al.[35], where authors report execution times of 101 minutes on average, and up to 5 hours, for running 10 generations of Sapienz on one app (using 10 Android emulators).

Finally, Table 5 indicates that there is not enough evidence to hold with statistical significance that NSGA-II with *motif* genes (i.e., Sapienz) is different from Random Search with *motif* genes.

Result 2. RQ2: NSGA-II evolutionary algorithm has a *marginal* impact on Sapienz coverage. Although, NSGA-II is better than the other evolutionary algorithms considered, Random Search is at least as good as NSGA-II.

4.3 | Study "A" results: crash detection

TABLE 9 Summary of crash detection results for Study "A": Overall number of crashes, standard deviation and the rank of each algorithm based on their average performance, which is **not** statistically significant according to the Friedman test (p-value is > 0.05, full data is available on Table 10). For averaged number of crashes we also report confidence intervals (CI) using bootstrapping at 95% significance level.

Algorithm	Ranking Mean	Ranking SD	Overall Crashes Mean	Crashes SD	CI
NSGA-II + MG (Sapienz)	1.75	0.93	1.20	0.78	[1.11, 1.30]
Random Search	2.56	1.18	1.10	0.67	[1.01, 1.18]
Random Search + MG	2.62	1.16	1.16	0.70	[1.07, 1.24]
NSGA-II	3.06	0.73	1.07	0.71	[0.98, 1.16]

TABLE 10 Full ranking of number of crashes achieved by algorithms for each subject on Study "A".

	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG
Arity	3.50	1.00	2.00	3.50
BookWorm	3.00	1.50	4.00	1.50
DroidSat	4.00	3.00	1.00	2.00
FillUp	2.00	3.00	4.00	1.00
JustSit	4.00	1.00	3.00	2.00
Kanji	3.00	1.00	3.00	3.00
L9Droid	2.50	1.00	2.50	4.00
Maniana	2.50	2.50	1.00	4.00
Mean	3.06	1.75	2.56	2.62

To answer the research questions related to crash detection performance, we applied a similar analysis to the one used for statement coverage. Given that the single-objective EAs only aim to improve the coverage of the population, and that NSGA-II was shown to outperform them in that goal (cf. Section 4.2.2), in these RQs we compare only the top 4 algorithms of Table 3: NSGA-II and Random Search, both with and without *motif* genes.

Table 9 summarises the results of the experiment regarding the number of unique crashes detected. Among all the algorithms evaluated, NSGA-II + *motif* genes (i.e., Sapienz) is the one that achieves the highest overall crashes detected (1.2 on average) and CI. The $p - value$ obtained from the Friedman test is 0.188 . This means that we can **not** reject the null hypothesis of the Friedman test (i.e., there is not enough statistical confidence to say that at least one algorithm differs from the rest). Table 10 shows the rankings achieved by each algorithm for every application. Figure 4 shows visually the overall statement coverage achieved by each algorithm.

4.3.1 | RQ3: What is the contribution of *motif* genes in Sapienz crash detection?

To answer this question, we conduct a pairwise tournament among NSGA-II and Random Search, both with and without *motif* genes. Table 11 summarises the results of the pairwise tournament. The first position is assigned to Sapienz with a significantly better performance in 7 out of 24 comparisons and an average effect size of 0.63. Furthermore, in the remaining 17 of the 24 comparisons, Sapienz is not significantly worse. The second position goes to Random Search with *motif* genes. This algorithm has a significantly better performance in 4 out of 24 comparisons and an average effect size of 0.63. It is only significantly worse in 2 out of 24 comparisons.

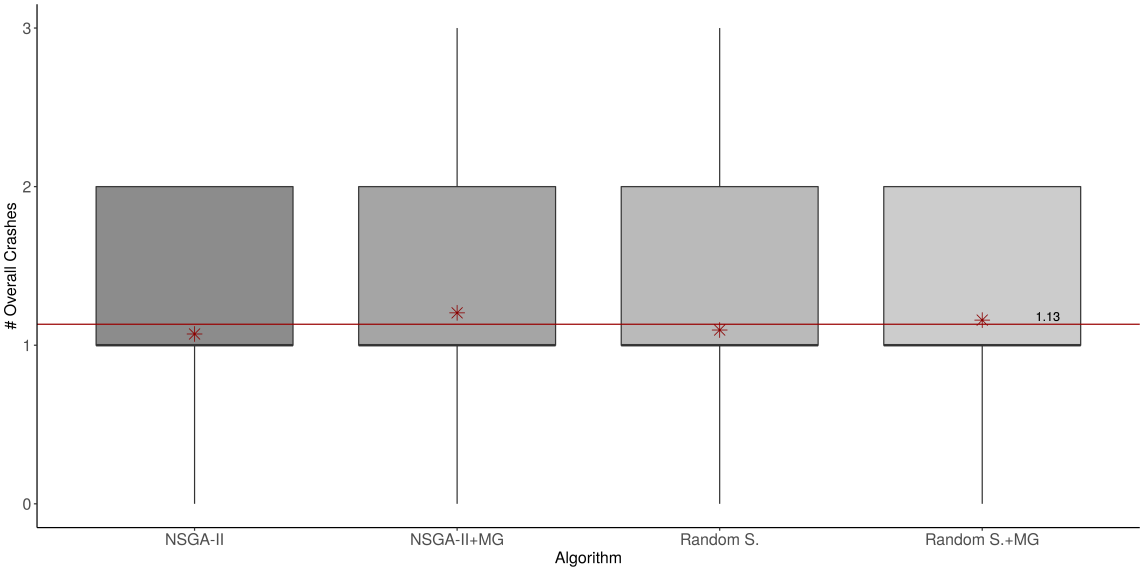

FIGURE 4 Overall number of crashes detected by each algorithm on Study "A". Middle line of each boxplot marks the median, black circles represent outliers, $\star$ symbol shows the mean, and the red line represents the mean of all crashes (1.13).

TABLE 11 Pairwise comparison of crashes detected by NSGA-II and Random Search (with and without *motif* genes) on Study "A". "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., $p - value$ of Wilcoxon-Mann-Whitney U-test less than 0.05) better and worse, respectively. Columns $\widehat{A}_{12}$ give the average effect size.

Algorithm	Tournament position	Overall Crashes Mean	Better		Worse	
			than	$\widehat{A}_{12}$	than	$\widehat{A}_{12}$
NSGA-II + MG (Sapienz)	1.00	1.20	7/24	0.63	0/24	-
NSGA-II	3.00	1.07	0/24	-	4/24	0.34
Random Search + MG	2.00	1.16	4/24	0.63	2/24	0.40
Random Search	4.00	1.10	1/24	0.63	6/24	0.37

Overall, we can see in Table 11 a marginal improvement of crash detection on those algorithms that include *motif* genes over their counterparts without *motif* genes. Nevertheless, the fact that both Sapienz and Random Search with *motif* genes have an average effect size of 0.63 on the few comparisons in which they are significantly different, means that the size of this difference is rather small (i.e., an average effect size of 0.5 means there are no differences). It is also important to note that, in terms of crash detection, we can **not** reject the null hypothesis of the Friedman test ($p - value$ is greater than 0.05), so we can not claim that there are algorithms statistically different from the others.

Result 3. RQ3: Motif genes have a *marginal* impact on Sapienz crash detection. Although, both NSGA-II and Random Search improve their crash detection when test cases include motif genes, this difference is not statistically significant.

4.3.2 | RQ4: What is the contribution of the NSGA-II evolutionary algorithm in Sapienz crash detection?

To answer this question, we use the same pairwise tournament presented for RQ3 (Table 11). Again, because the $p - value$ of the Friedman test is greater than 0.05, we can **not** reject the null hypothesis and claim that there are algorithms statistically different from the others.

Although we can see in Table 11 a very small difference of crash detection when comparing NSGA-II vs. Random Search and Sapienz vs Random Search with *motif* genes, it is not clear whether we can claim that NSGA-II has an *marginal* impact or not. If we look at the algorithms without *motif* genes, NSGA-II does not perform significantly better in any of the 24 comparisons, while Random Search performs significantly better in only 1 of those comparisons. At the same time, Random Search performs significantly worse in 6 out of 24 comparisons, whereas NSGA-II only in 4.

Result 4. RQ4: NSGA-II evolutionary algorithm has a *marginal* or *none* impact on Sapienz crash detection. Whether using motif genes or not, NSGA-II performs similarly to Random Search.

4.4 | RQ5: How does the results on open-source apps compare to real-world closed-source ones?

TABLE 12 Summary of coverage results for Study "B": Overall coverage, standard deviation and the rank of each algorithm based on their average performance, which is statistically significant according to the Friedman test (p-value is < 0.0001). For averaged coverage values we also report confidence intervals (CI) using bootstrapping at 95% significance level.

Algorithm	Ranking Mean	Ranking SD	Overall Coverage Mean	Coverage SD	CI
NSGA-II + MG (Sapienz)	2.63	1.54	19.07	11.43	[15.05, 23.17]
Random Search + MG	3.47	2.15	18.90	11.56	[14.87, 22.94]
$(\mu + \lambda)$ EA	4.65	1.41	18.00	11.48	[14.00, 21.93]
NSGA-II	4.83	1.33	17.90	11.65	[13.72, 22.00]
Standard GA	5.12	1.32	17.63	11.51	[13.59, 21.67]
Steady State GA	5.73	1.68	17.60	11.40	[13.70, 21.57]
Random Search	5.75	1.82	17.37	11.48	[13.37, 21.37]
Monotonic GA	5.92	1.52	17.50	11.53	[13.42, 21.56]
(μ, λ) EA	6.90	1.67	16.67	11.43	[12.63, 20.71]

TABLE 13 Summary of crash detection results for Study "B": Overall number of crashes, standard deviation and the rank of each algorithm based on their average performance, which is statistically significant according to the Friedman test (p-value is < 0.0001). For averaged crashes values we also report confidence intervals (CI) using bootstrapping at 95% significance level.

Algorithm	Ranking Mean	Ranking SD	Overall Crashes Mean	Crashes SD	CI
Random Search + MG	4.13	1.61	0.27	0.52	[0.08, 0.46]
Random Search	4.65	1.09	0.13	0.35	[0.01, 0.25]
NSGA-II	4.95	0.74	0.07	0.25	[-0.02, 0.16]
NSGA-II + MG (Sapienz)	5.10	0.55	0.03	0.18	[-0.03, 0.10]
$(\mu + \lambda)$ EA	5.23	0.50	0.00	0.00	[0.00, 0.00]
(μ, λ) EA	5.23	0.50	0.00	0.00	[0.00, 0.00]
Monotonic GA	5.23	0.50	0.00	0.00	[0.00, 0.00]
Standard GA	5.23	0.50	0.00	0.00	[0.00, 0.00]
Steady State GA	5.23	0.50	0.00	0.00	[0.00, 0.00]

To answer this question, we conducted a similar statistical analysis to the one presented for Study "A". Since we only have one repetition for each combination of closed-source subject and algorithm, we can not perform a pairwise tournament between algorithms. I.e., there is not enough data-points to perform a Wilcoxon-Mann-Whitney U-test between two algorithms for a given subject. Nevertheless, we can still perform a Friedman test, compute its ranking and perform the post-hoc Conover's test. With these statistical tests, we can conclude whether there is any difference in effectiveness among the algorithms considered.

Tables 12 and 13 summarize the results of the Friedman test in terms of coverage achieved and number of unique crashes detected, respectively. Figure 5 shows visually the overall method coverage achieved by each algorithm. As can be seen, the overall mean of coverage for Study "B" is quite lower when compared to the results on Study "A" (18% vs 41%). A possible explanation of such results might be that closed-source real-world apps are more complex than open-source ones, or that their full functionality is usually locked behind login/sign-up screens.

The $p - value$ obtained from the Friedman test for coverage achieved is 5.78e-16 , and for number of unique crashes detected is 2.49e-06 . This means that in both cases we can reject the null hypothesis of the Friedman test (i.e., there is at least one algorithm that differs from the rest). In terms of coverage, this result matches the one in Study "A" (Section 4.2). However, in terms of crashes, this result deviates from the results observed on Study "A". Surprisingly, there is at least one algorithm that is sticking out in Study "B", whereas in Study "A" we did not have enough statistical

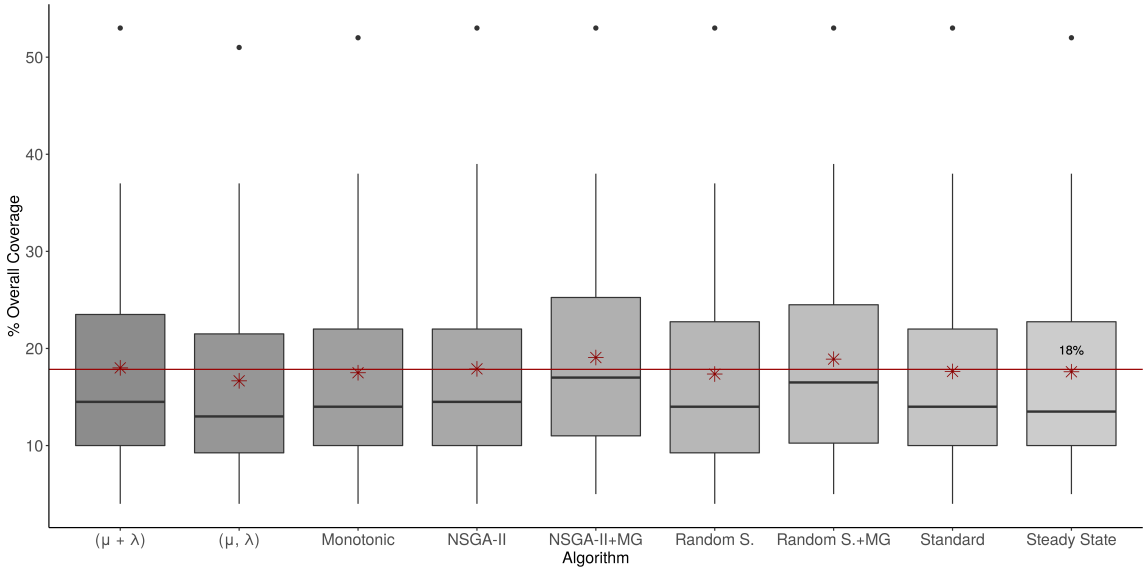

FIGURE 5 Overall coverage achieved by each algorithm on Study "B". Middle line of each boxplot marks the median, black circles represent outliers, $\star$ symbol shows the mean, and the red line represents the mean of all coverages (18%).

TABLE 14 Results of the post hoc Conover's test for pairwise analysis of coverage achieved on Study "B". A $p-value$ less than 0.05 for algorithms X and Y means there is enough evidence to claim they are different with statistically significance.

	$(\mu+\lambda)$ EA	(μ,λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG	Standard GA
(μ,λ) EA	< 0.05	-	-	-	-	-	-	-
Monotonic GA	< 0.05	< 0.05	-	-	-	-	-	-
NSGA-II	1.000	< 0.05	< 0.05	-	-	-	-	-
NSGA-II + MG (Sapienz)	< 0.05	< 0.05	< 0.05	< 0.05	-	-	-	-
Random Search	< 0.05	< 0.05	1.000	< 0.05	< 0.05	-	-	-
Random Search + MG	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	-	-
Standard GA	0.376	< 0.05	< 0.05	1.000	< 0.05	0.094	< 0.05	-
Steady State GA	< 0.05	< 0.05	1.000	< 0.05	< 0.05	1.000	< 0.05	0.099

significance to state any difference. Nevertheless, the overall number of crashes found in Study "B" is smaller than the overall number of crashes in Study "A". This is somehow expected since popular closed-source apps are more robust and mature than open-source apps used for Study "A".

Regarding RQ1, in both studies (i.e., "A" and "B") we observe similar behaviour. NSGA-II with *motif* genes (i.e., Sapienz) is above NSGA-II in the Friedman ranking (Table 12), and the same happens when looking at Random Search with *motif* genes against Random Search. In terms of statistical significance, Table 14 shows there is enough statistical evidence to hold that there is a difference between each algorithm on both pairs. In other words, algorithms that include *motif* genes have greater coverage than their counterparts without *motif* genes, and this difference is statistically significant.

Study "B" take on RQ1: Motif genes have a *significant* impact on Sapienz coverage. In fact, both NSGA-II and Random Search improve their coverage when test cases include motif genes.

For RQ2, our conclusions on Study "B" are similar to Study "A". From the data collected on Study "A", we have previously concluded that NSGA-II has a *marginal* impact on Sapienz coverage. In Study "A", NSGA-II was better than the other EAs but at the same time Random Search was at least as good as NSGA-II. Now, in Study "B", the results are inverted. On one hand, NSGA-II seems to be statistically better than Random Search, with or without *motif* genes. On the other hand, not only did $(\mu+\lambda)$ EA achieved a higher position than NSGA-II in the Friedman ranking (Table 12), but also there is not enough statistical evidence to claim that they are different. Moreover, Standard GA, which ranked below NSGA-II, is not statistically different either.

TABLE 15 Results of the post hoc Conover's test for pairwise analysis of number of crashes achieved on Study "B". A $p - value$ less than 0.05 for algorithms X and Y means there is enough evidence to claim they are different with statistically significance.

	$(\mu+\lambda)$ EA	(μ,λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG	Standard GA
(μ,λ) EA	1.000	-	-	-	-	-	-	-
Monotonic GA	1.000	1.000	-	-	-	-	-	-
NSGA-II	0.450	0.450	0.450	-	-	-	-	-
NSGA-II + MG (Sapienz)	1.000	1.000	1.000	1.000	-	-	-	-
Random Search	< 0.05	< 0.05	< 0.05	0.328	< 0.05	-	-	-
Random Search + MG	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	-	-
Standard GA	1.000	1.000	1.000	0.450	1.000	< 0.05	< 0.05	-
Steady State GA	1.000	1.000	1.000	0.450	1.000	< 0.05	< 0.05	1.000

Study "B" take on RQ2: NSGA-II EA has a *marginal* impact on Sapienz coverage. Even though this algorithm seems to improve the coverage when compared to Random Search, it is not statistically different to some of the other EAs (i.e., $(\mu + \lambda)$ EA and Standard GA).

Regarding RQ3, the answer for Study "A" and Study "B" are different. Although Random Search with *motif* genes is above Random Search in the Friedman ranking (Table 13), the same does not happen for NSGA-II: the version without *motif* genes comes first in the ranking. In terms of statistical significance, Table 15 shows there is enough statistical evidence to hold that there is a difference only between Random Search with and without *motif* genes. The same can not be said of NSGA-II vs. NSGA-II with *motif* genes: they seem to be statistically the same.

Study "B" take on RQ3: Motif genes have a *marginal* or *none* impact on Sapienz crash detection. Only Random Search improves its crash detection when test cases include motif genes. The same does not happen for NSGA-II.

In regards to RQ4, Study "B" has a similar result than Study "A". It is worth noticing that all single-objective EAs failed to detect any crash during execution. In that sense, NSGA-II and Random Search not only are above the rest of the EAs in the Friedman ranking (Table 13), they also managed to detect at least some crashes. In terms of statistical significance, Table 15 shows that there is only a statistical difference between Random Search and NSGA-II when they have *motif* genes. Namely, NSGA-II is not statistically different than Random Search. What is more, NSGA-II is also not statistically different than the the other EAs.

Study "B" take on RQ4: NSGA-II evolutionary algorithm has a *marginal* or *none* impact on Sapienz crash detection. Although, NSGA-II managed to detect some crashes, it is not statistically different than the the other evolutionary algorithms. Also, Random Search is at least as good as NSGA-II.

In summary, for Study "B", as in Study "A", *motif* genes still help to achieve a higher coverage. Additionally, Random Search with *motif* genes keeps being at least as good as NSGA-II with *motif* genes in terms of coverage and crash detection. The main change arises from the fact that NSGA-II is not statistically better than other EAs in Study "B" when we analyze the coverage achieved. It is also less clear that *motif* genes help to detect a larger number of unique crashes. We associate this slight difference in the results related to crash detection to the fact that Study "B" uses real-world closed-source Android apps that might have less (or more difficult to find) crashes.

Result 5. RQ5: The results of RQs 1-2 are mostly maintained. *Motif* genes still have a *significant* impact on Sapienz coverage. Nevertheless, NSGA-II might not be as much better than other EAs, as we believed in RQ2. The results of RQs 3-4 have some slight changes due to the difficulty of finding crashes on popular closed-source apps. From these results, it is not so clear that *motif* genes help detect more crashes.

4.5 | Threats to Validity:

Threats to internal validity might result from how the empirical study was carried out. Since all the studied algorithms are affected by non-determinism, for Study "A" we ran 30 repetitions of each experiment with different random seeds and followed rigorous statistical procedures to evaluate the results. To avoid possible confounding factors when comparing different algorithms, they were all implemented on the same tool. Since parameter tuning can affect the performance of algorithms, we used the same default values for all parameters across experimentation. These values were chosen based on the paper presenting Sapienz[5]. We used roulette selection as a selection function for the single-objective EAs. Although the rank selection function is preferred to avoid premature convergence[29], the average number of generations performed by the single-objective EAs in our study was 30, which mitigates this possible threat.

Another possible threat to internal validity might come from the fact that we used for our experiments a version of Sapienz that might be different from the one that is currently under development at the industrial setting (i.e., Facebook). We chose to use that version (although marked as "out-of-date and no longer supported" by their authors) because it is nevertheless the latest publicly available version used for evaluation by Mao et al.[5].

We measured the success of each algorithm in terms of statement or method coverage. While higher coverage is a desirable goal for test generation, it is only a proxy for the more important goal of fault detection. Therefore, there is a threat to construct validity caused by how we determine which algorithm is better. In this article we extended previous work[14] by reporting how many crashes each algorithm is able to detect. Nonetheless, we believe that this test adequacy criterion is still a reasonable indicator of the effectiveness of different search-based algorithms.

Threats to external validity come from the fact that, due to the very large number of experiments, we only used 8 subjects as case studies for Study "A", which still took a long time even when using a cluster of computers. To avoid selection bias, we explicitly decided to include *only* those apps that have been previously used in a statistical analysis of Sapienz (i.e., Study #2 in Mao et al.[5]). For Study "A", instead of including new evaluation subjects, we opted to favour a larger number of repetitions (30 per combination of subject and algorithm) to gain better statistical significance.

In this article we also extended previous work[14] by adding a new study (specifically, Study "B") using popular real-world closed-source Android apps taken directly from the Google Play Store as subjects. By doing this, we aimed to evaluate how results observed on open-source apps compare to real-world closed-source Android apps that are more representative of real industry applications. It is important to note that while we would like to have hundreds of subjects, as is the case in some empirical studies for Java unit test generation (e.g., Shamshiri et al.[13], Campos et al.[33]), the execution times of Android test evaluation makes it very time consuming, and hence expensive. As a point of comparison, Shamshiri et al.[13] and Campos et al.[33] use 978 and 346 Java subjects each but their allocated time-budget for each run is only of 2 and 1 minute respectively, whereas our time-budget is *2 hours*. Still, another selection of subjects might result in different conclusions.

For the selection of algorithms, we considered the algorithms studied in[18]. Some of the multi-objective algorithms (e.g., MOSA and DynaMOSA) had to be excluded from the study since they were not designed to work exclusively with the statement coverage provided by EMMA. Although we included one multi-objective algorithm (i.e., NSGA-II), including further multi-objective algorithms might also result in different conclusions. In future work, we plan to compare the mentioned algorithms as well as other ones such as SPEA-2[36], NSGA-III[37] and MIO[38].

5 | RELATED WORK

Sell et al.[10], also present a study comparing different algorithms for Android test generation, but these algorithms are evaluated on the testing tool MATE[11]. As we have stated before, MATE uses a widget-based representation of individuals, while Sapienz does not. This means that evolutionary operators such as crossover and mutation are different between both tools, and might influence results obtained. What is more, some classic genetic and evolutionary algorithms in our study are not included in the work by Sell et al.[10], namely: Monotonic GA, Steady-State GA, $(\mu + \lambda)$ EA, (μ, λ) EA. Finally, the work by Sell et al.[10] uses mainly test cases instead of test suites and does not study the effect of *motif* genes.

Vogel et al.[39] conduct a fitness landscape analysis of Sapienz using 5 apps and 5 repetitions. In their analysis, they observe a lack of diversity of the evolved test suites and stagnation of the search after 25 generations. They then propose Sapienz ^{div}, an extension of Sapienz that integrates diversity-promoting mechanisms. This new algorithm is evaluated against the original version in 34 apps with 30 repetitions. The evaluation showed that Sapienz ^{div} is capable of achieving better or similar results than Sapienz in terms of coverage and crash detection. However, Sapienz ^{div} tends to produce longer test sequences and has a significant runtime overhead compared to the original algorithm.

Pilgun et al.[40] introduce a new tool called ACVTool (Android Code coVerage Tool) for instrumenting and measuring code coverage of closed-source apps at class, method and instruction level. They demonstrate the practical value of ACVTool by integrating it with Sapienz and conducting a large-scale experiment where they compare different coverage tools (JaCoCo & ELLA) and granularities (Activity, method, instrumentation and no-coverage at all). In this study, they evaluated the ACVTool prototype on a total of 832 closed-source apps from the Google Play Store sample of the AndroZoo dataset[41], and 446 open-source apps from the F-Droid repository. They report that ACVTool managed to successfully instrument 96.9% of the apps in their experiments. Besides, their findings suggest that even when performing several repetitions, a single coverage metric is not able to find all crashes that were detected by other metrics. Therefore, no coverage granularities clearly outperforms the others in this aspect.

Guo et al.[42] perform a qualitative study of the activity, method and line coverage reported by Monkey[7] and Stoat[43] on 70 open-source Android apps (one repetition each). They report that the unexplored code is mainly due to lack of dependency knowledge on the required events and the widgets state of the app (i.e., which actions cause a screen transition and what is the specific widget state necessary to achieve the transition). In the same work, they propose Gesda, a tool that leverages static dependency analysis to construct a GUI page transition model that captures the widgets with callbacks in a screen and the widgets whose state influence a callback. They evaluate this tool on the same 70 apps and show that it can outperform both Monkey and Stoat.

Choudhary et al.[4] compare several test generation tools for Android on a considerable number of open source applications. The tools considered in their study are: Monkey[7] and Dynodroid[6], EvoDroid[44], GUIRipper[45], PUMA[46],A3E-Depth-first[47], SwiftHand[48], JPF-Android[49], and ACTEve[50]. Although it is an impressive amount of empirical work they do not focus on the specific contributions of the underlying algorithm used by each of the techniques.

Wang et al.[31] also compare several state-of-the-art techniques on industrial applications. The tools evaluated in their study are: Monkey[7], WCTester[51,52], Sapienz, Stoat[43], DroidBot[53] and A3E-Depth-first. The study does not achieve statistical confidence: they only use a few repetitions to compensate for the random nature of algorithms used by the tools.

Campos et al.[33] conducted an empirical study comparing multiple evolutionary algorithms (including some multi-objective) and two random approaches for Java unit test generation. The study was applied to a selection of non-trivial open-source classes. They show that the choice of algorithm can have a substantial influence on the performance of Whole Test Suite optimisation. Panichella et al.[54] also performed an empirical study with different evolutionary algorithms for Java unit test generation and confirmed several of the findings in Campos et al.[33]. Arcuri et al.[29] conduct an extensive study on parameter tuning for search-based algorithms. The results are statistically analysed in the context of test data generation for Java programs using the EvoSuite tool. Their results show that parameter tuning does have an impact on the performance of a search algorithm. Nevertheless, they also show that is it not easy to find good settings that significantly outperform the "default" values suggested in the literature. Our work also analyses search-based evolutionary and random algorithms but in the context of Android apps, paying special attention to the effect of using *motif* genes.

6 | CONCLUSIONS

In this work, we aimed to deepen into how the main features of Sapienz (namely, the NSGA-II algorithm and the representation of individuals using *motif* genes) impact over effectiveness by conducting an extensive empirical study using both experimental subjects previously used in the related literature and popular real-world closed-source subjects taken from the Google Play Store.

Our studies show that, for the case of Android test generation and in terms of coverage, the multi-objective NSGA-II evolutionary algorithm does not have a clear improvement over the other algorithms. In Study "A", NSGA-II is not distinguishable with statistical confidence from Random Search whereas in Study "B" it has a similar performance to other evolutionary algorithms. These results cast doubts about the actual effectiveness of the NSGA-II algorithm for Android test generation. In terms of the impact of *motif* genes, our experimental results provide evidence showing that NSGA-II and Random Search performed statistically better when *motif* genes were included. Both findings suggest that the Sapienz's improvement on coverage is more attributable to adding *motif* genes rather than to the use of a particular choice of multi-objective evolutionary algorithm. Therefore, intra-tool comparisons (as the ones performed in this article and in Sell et al.[10]) should be preferable over cross-tool comparisons (as the one performed by Mao et. al.[5]) whenever possible. In other words, different techniques should be compared using the same tool, aiming to avoid conflating factors behind changes in test suite effectiveness.

In terms of crash detection, surprisingly, we observe no significant difference among the studied algorithms, whether they include or not a *motif* gene representation. However, we observe that the number of detected faults in the subjects is rather small which might hinder the analysis of each tool capability.

References

1. Global Digital Future in Focus 2018 - Comscore, Inc.. https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018; 2018.
2. Number of Android applications on the Google Play store | AppBrain. https://www.appbrain.com/stats/number-of-android-apps; 2021.
3. Joorabchi ME, Mesbah A, Kruchten P. Real Challenges in Mobile App Development. In: IEEE Computer Society; 2013: 15–24.
4. Choudhary SR, Gorla A, Orso A. Automated Test Input Generation for Android: Are We There Yet?. In: IEEE; 2015.
5. Mao K, Harman M, Jia Y. Sapienz: multi-objective automated testing for Android applications. In: ACM; 2016.
6. Machiry A, Tahiliani R, Naik M. Dynodroid: an input generation system for Android apps. In: ACM; 2013: 224–234.
7. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html; 2020.
8. Alshahwan N, Gao X, Harman M, et al. Deploying Search Based Software Engineering with Sapienz at Facebook. In: LNCS. Springer; 2018.

9. Deb K, Agrawal S, Pratap A, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation* 2002; 6(2): 182–197.

10. Sell L, Auer M, Frädrich C, Gruber M, Werli P, Fraser G. An Empirical Evaluation of Search Algorithms for App Testing. In: . 11812 of *LNCS*. Springer; 2019.

11. Eler MM, Rojas JM, Ge Y, Fraser G. Automated Accessibility Testing of Mobile Apps. In: IEEE Computer Society; 2018.

12. Karnopp DC. Random search techniques for optimization problems. *Automatica* 1963; 1(2-3): 111–121.

13. Shamshiri S, Rojas JM, Fraser G, McMinn P. Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?. In: ACM; 2015: 1367–1374.

14. Arcuschin I, Galeotti JP, Garbervetsky D. Algorithm or Representation?: An empirical study on how SAPIENZ achieves coverage. In: ACM; 2020: 61–70.

15. Fraser G, Arcuri A. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 2013; 39(2): 276–291.

16. Rojas JM, Campos J, Vivanti M, Fraser G, Arcuri A. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In: . 9275 of *Lecture Notes in Computer Science*. Springer; 2015: 93–108.

17. Fraser G, Arcuri A. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* 2015; 20(3): 783–812.

18. Campos J, Ge Y, Fraser G, Eler M, Arcuri A. An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation. In: . 10452 of *Lecture Notes in Computer Science*. Springer; 2017: 33–48.

19. Sahin O, Akay B. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. *Appl. Soft Comput.* 2016; 49: 1202–1214.

20. Ter-Sarkisov A, Marsland SR. Convergence Properties of (μ + λ) Evolutionary Algorithms. In: AAAI Press; 2011.

21. Harman M, Mansouri SA, Zhang Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 2012; 45(1): 11:1–11:61.

22. Fonseca CM, Fleming PJ. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In: Morgan Kaufmann; 1993: 416–423.

23. ELLA-customized: An improved version of the ELLA tool. https://github.com/ms1995/ella-customized; 2020.

24. Rhapsod/sapienz: A Prototype of Sapienz (Out-of-date and no longer supported). https://github.com/Rhapsod/sapienz; 2016.

25. Panichella A, Kifetew FM, Tonella P. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In: IEEE Computer Society; 2015: 1–10.

26. Panichella A, Kifetew FM, Tonella P. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Trans. Software Eng.* 2018; 44(2): 122–158.

27. EMMA: a free Java code coverage tool. http://emma.sourceforge.net/; 2020.

28. Arcuri A, Fraser G. On Parameter Tuning in Search Based Software Engineering. In: . 6956 of *Lecture Notes in Computer Science*. Springer; 2011: 33–47.

29. Arcuri A, Fraser G. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 2013; 18(3): 594–623.

30. ELLA: A tool for binary instrumentation of Android apps. https://github.com/saswatanand/ella; 2020.

31. Wang W, Li D, Yang W, et al. An empirical study of Android test generation tools in industrial cases. In: ACM; 2018: 738–748.

32. Panichella A, Molina UR. Java Unit Testing Tool Competition - Fifth Round. In: IEEE; 2017: 32–38.

33. Campos J, Ge Y, Albunian N, Fraser G, Eler M, Arcuri A. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information & Software Technology* 2018; 104: 207–235.

34. García S, Molina D, Lozano M, Herrera F. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization. *J. Heuristics* 2009; 15(6): 617–644.

35. Vogel T, Tran C, Grunske L. Does Diversity Improve the Test Suite Generation for Mobile Applications?. In: . 11664 of *Lecture Notes in Computer Science*. Springer; 2019: 58–74.

36. Kim M, Hiroyasu T, Miki M, Watanabe S. SPEA2+: Improving the Performance of the Strength Pareto Evolutionary Algorithm 2. In: . 3242 of *Lecture Notes in Computer Science*. Springer; 2004: 742–751.

37. Deb K, Jain H. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Trans. Evolutionary Computation* 2014; 18(4): 577–601.

38. Arcuri A. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In: . 10452 of *Lecture Notes in Computer Science*. Springer; 2017: 3–17.

39. Vogel T, Tran C, Grunske L. A comprehensive empirical evaluation of generating test suites for mobile applications with diversity. *Inf. Softw. Technol.* 2021; 130: 106436.

40. Pilgun A, Gadyatskaya O, Zhauniarovich Y, Dashevskyi S, Kushniarou A, Mauw S. Fine-grained Code Coverage Measurement in Automated Black-box Android Testing. *ACM Trans. Softw. Eng. Methodol.* 2020; 29(4): 23:1–23:35.

41. Allix K, Bissyandé TF, Klein J, Traon YL. AndroZoo: collecting millions of Android apps for the research community. In: ACM; 2016: 468–471.

42. Guo W, Shen L, Su T, Peng X, Xie W. Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis. In: IEEE; 2020: 557–568.

43. Su T, Meng G, Chen Y, et al. Guided, stochastic model-based GUI testing of Android apps. In: ACM; 2017.

44. Mahmood R, Mirzaei N, Malek S. EvoDroid: segmented evolutionary testing of Android apps. In: ACM; 2014: 599–609.

45. Amalfitano D, Fasolino AR, Tramontana P, Carmine SD, Memon AM. Using GUI ripping for automated testing of Android applications. In: ACM; 2012: 258–261.

46. Hao S, Liu B, Nath S, Halfond WGJ, Govindan R. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In: ACM; 2014: 204–217.

47. Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of android apps. In: ACM; 2013: 641–660.

48. Choi W, Necula GC, Sen K. Guided GUI testing of android apps with minimal restart and approximate learning. In: ACM; 2013: 623–640.

49. Merwe v. dH, Merwe v. dB, Visser W. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 2012; 37(6): 1–5.

50. Anand S, Naik M, Harrold MJ, Yang H. Automated concolic testing of smartphone apps. In: ACM; 2012: 59.

51. Zeng X, Li D, Zheng W, et al. Automated test input generation for Android: are we really there yet in an industrial case?. In: ACM; 2016: 987–992.

52. Zheng H, Li D, Liang B, et al. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In: IEEE Computer Society; 2017: 253–262.

53. Li Y, Yang Z, Guo Y, Chen X. DroidBot: a lightweight UI-guided test input generator for Android. In: IEEE Computer Society; 2017: 23–26.

54. Panichella A, Kifetew FM, Tonella P. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information & Software Technology* 2018; 104: 236–256.

How to cite this article: I. Arcuschin, J. P. Galeotti, and D. Garbervetsky (2021), An Empirical Study on How Sapienz Achieves Coverage and Crash Detection, *JSEP*, .