Sapienz: Multi-objective Automated Testing for Android Applications

Ke Mao Mark Harman Yue Jia
CREST Centre, University College London, Malet Place, London, WC1E 6BT, UK
k.mao@cs.ucl.ac.uk, mark.harman@ucl.ac.uk, yue.jia@ucl.ac.uk

ABSTRACT

We introduce SAPIENZ, an approach to Android testing that uses multi-objective search-based testing to automatically explore and optimise test sequences, minimising length, while simultaneously maximising coverage and fault revelation. SAPIENZ combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation. SAPIENZ significantly outperforms (with large effect size) both the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-revealing sequence length. When applied to the top 1,000 Google Play apps, SAPIENZ found 558 unique, previously unknown crashes. So far we have managed to make contact with the developers of 27 crashing apps. Of these, 14 have confirmed that the crashes are caused by real faults. Of those 14, six already have developer-confirmed fixes.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** *Search-based software engineering;*

Keywords

Android; Test generation; Search-based software testing

1. INTRODUCTION

There are over 1.8 million apps available from the Google Play marketplace, as of January 2016 [9]. For developed internet markets such as the US, UK and Canada, mobile app usage now dominates traditional desktop software usage [29]. Unfortunately, testing technology has yet to catch up, and software testers are faced with additional problems due to device fragmentation [2], which increases test effort due to the number of devices that must be considered. According to a study on mobile app development [45], mobile app testing still relies heavily on manual testing, while the use of automated techniques remains rare [48].

Where test automation does occur, it typically uses Google's Android Monkey tool [36], which is currently integrated with the Android system. Since this tool is so widely available and distributed, it is regarded as the current state-of-practice for automated software testing [53]. Although Monkey automates testing, it does so in a relatively unintelligent manner: generating sequences of events at random in the hope of exploring the app under test and revealing failures. It uses a standard, simple-but-effective, default test oracle [22] that regards any input that reveals a crash to be a fault-revealing test sequence.

Automated testing clearly needs to find such faults, but it is no good if it does so with exceptionally long test sequences. Developers may reject longer sequences as being impractical for debugging and also unlikely to occur in practice; the longer the generated test sequence, the less likely it is to occur in practice. Therefore, a critical goal for automated testing is to find faults with the *shortest possible* test sequences, thereby making fault revelation more actionable to developers.

Exploratory testing is "simultaneous learning, test design, and test execution" [11], that can be cost-effective and is widely used by industrial practitioners [21, 43, 46] for testing in general. However, it is particularly underdeveloped for mobile app testing [41, 42]. Although there exist several test automation frameworks such as Robotium [10] and Appium [3], they require human-implemented scripts, thereby inhibiting full automation.

We introduce SAPIENZ, the first approach offering multi-objective automated Android app exploratory testing that seeks to maximise code coverage and fault revelation, while minimising the length of fault-revealing test sequences. Our goal is to produce an entirely automated approach that maximises fault revelation with short test sequences. The key insight in our approach is that minimising test sequence length and maximising other objectives can be combined in a Pareto-optimal multi-objective search-based approach to Android testing. By using Pareto optimality, we do not sacrifice longer test sequences, when they are the only ones that find faults, nor where they are necessary to achieve higher code coverage. Nevertheless, through its use of Pareto optimality, SAPIENZ progressively replaces such longer sequences with shorter test sequences when equally good. The paper makes the following primary contributions:

1) The Sapienz approach: the paper introduces the first Pareto multi-objective approach to Android testing, combining techniques used for traditional automated testing, adapting and extending them for Android testing. The approach

Table 1: At a glance: summary of existing tools and techniques for automated Android app testing ('OSS' and 'CSS' refer to Open-Source and Closed-Source Software used as evaluation subjects respectively).

Technique	Venue	Publicly Available	Box	Approach	Crash Report	Replay Scripts	Emulator / Real Device	Eval. Subjects Size		
								Type	OSS	CSS
Monkey [36]	N/A	Yes	Black	Random-based	Text	No	Both	N/A	N/A	N/A
AndroidRipper [15]	ASE'12	Yes	Black	Model-based	Text	No	Emulator	OSS	1	0
ACTEve [16]	FSE'12	Yes	White	Program analysis	N/A	Yes	Emulator	OSS	5	0
A^{3}E [20]	OOPSLA'13	Partially	Grey	Model-based	N/A	Yes	Real device	CSS	0	25
SwiftHand [27]	OOPSLA'13	Yes	Black	Model-based	N/A	No	Both	OSS	10	0
ORBIT [61]	FASE'13	No	Grey	Model-based	N/A	No	Emulator	OSS	8	0
Dynodroid [52]	FSE'13	Yes	Black	Random-based	Text, Image	Yes	Emulator	Both	50	1,000
PUMA [37]	MobiSys'14	Yes	Black	Model-based	Text	Yes	Both	CSS	0	3,600
EvoDroid [53]	FSE'14	No	White	Search-based	N/A	No	Emulator	OSS	10	0
SPAG-C [50]	TSE'15	No	Black	Record-replay	N/A	Yes	Real device	Both	3	2
MonkeyLab [51]	MSR'15	No	Black	Trace mining	N/A	Yes	Both	OSS	5	0
Thor [12]	ISSTA'15	Yes	Black	Adverse conditions	Text, Image	Yes	Emulator	OSS	4	0
TrimDroid [54]	ICSE'16	Yes	White	Program analysis	Text	Yes	Both	OSS	14	0
CrashScope [57]	ICST'16	No	Black	Combination	Text, Image	Yes	Both	OSS	61	0
Sapienz	ISSTA'16	Yes	Grey	Search-based	Text, Video	Yes	Both	Both	78	1,000

combines random fuzzing, systematic and search-based exploration, string seeding and multi-level instrumentation, all of which have been extended to cater for, not only traditional white box coverage (which we term 'skeletal coverage'), but also Android user interface coverage (which we term 'skin coverage').

2) Experimental results: we present the results of two systematic experimental studies on open-source real-world Android apps. The first uses the 68 apps from an Android benchmark suite [28], while the second uses a controlled random sample of 10 apps from the entire F-Droid suite, for which Sapienz always outperforms both Dynodroid and Monkey, statistically significantly and with large effect size in 24 out of 30 cases.

3) The tool, Sapienz: a practical Android testing tool Sapienz, which we make publicly available[1].

4) Demonstration of usefulness: an empirical study of the practical usefulness of the technique on the top 1,000 Google play apps. Sapienz found 558 unique crashes. The crashing behaviour has been verified on real Android devices (as well as Android emulators). At the time of writing, we have started reporting these to the developers, and 14 have been confirmed to be genuine, previously undetected, faults, 6 of which have already been confirmed as fixed by their developers. Since these are the most popular apps in current use, they will likely have been thoroughly tested, not merely by their developers, but also by their many (hundreds of thousands of) users. These results demonstrate that Sapienz is a practical tool for Android developers as well as for researchers. This paper is the first Android app testing work to report a large-scale evaluation on popular Google Play apps with developer-confirmed real-world faults.

2. RELATED WORK AND MOTIVATION

Table 1 presents a brief survey of the characteristics of existing Android testing techniques and tools, which we briefly describe below.

The most closely related work employs search-based methods. Mahmood et al. introduced EvoDroid [53], the first search-based framework for Android testing. EvoDroid extracts the interface model (based on static analysis of manifest and XML configuration files) and a call graph model (based on code analysis by using MoDisco [8]). It uses these models to guide the computational search process. Unfortunately, its implementation is no longer publicly available.

Several previous approaches are based on random testing (fuzz testing), which inject arbitrary or contextual events into the apps. Monkey [36] is Google's official testing tool for Android apps, which is built into the Android platform, and therefore likely to be more widely used than any other automated testing tool for Android apps. Monkey generates (pseudo) random input events, which include both User Interface (UI) events, such as clicks and gestures, and system events such as screen-shot capture and volume-adjustment. Dynodroid [52] is a publicly available and open-source tool that extends pure random testing with two feedback directed biases: BiasedRandom, which uses context adjusted weights for each event, and Frequency, which has a bias towards least recently used events. The implementation supports the generation of both UI and novel system events.

GUI and model-based approaches are popular for testing Android apps [14,15,20,27,37,61]. App event sequences can be generated from models, either manually constructed, or obtained from project artefacts, such as code or XML configuration files and UI execution states. For example, AndroidRipper [15] (subsequently MobiGUITAR [14]) builds a model using a depth-first search over the user interface. Its implementation is publicly available however not open-sourced. A^{3}E [20] consists of two app exploration strategies, the DFS strategy (like AndroidRipper) and a taint-targeted strategy which constructs a static activity transition graph. Although the tool is publicly available, the version does not support taint targeting. SwiftHand [27] dynamically builds a finite state machine model of the GUI, seeking to reduce restart time, while improving test coverage. ORBIT [61] is based on a combination of dynamic GUI crawling and static code analysis, using analysis to avoid generation of irrelevant UI events. PUMA [37] is a flexible framework for implementing various state-based test strategies.

Prior Android testing work also employs several other approaches, such as those that are program-analysis-based or reuse-based. ACTEve [16] is based on symbolic execution and concolic testing and supports the generation of both UI and system events. CrashScope [57] uses a combination of static and dynamic analyses to generate natural language crash descriptions with replicable test scripts. SPAG-C [50] implements a capture-reply based on image comparison of screen-shots to provide reusable and accurate test oracles,

[1]http://github.com/Rhapsod/sapienz

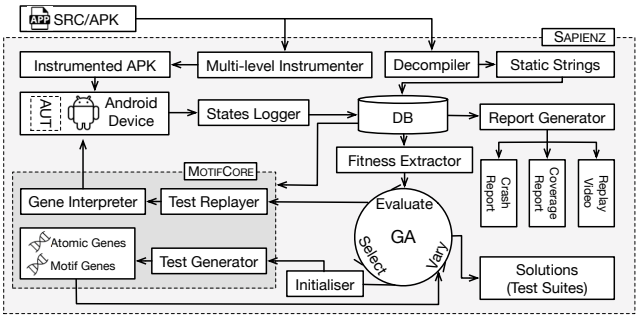

Figure 1: Sapienz workflow.

while Thor [12] makes use of existing test suites, seeking to expose them to adverse conditions. TrimDroid [54] is backed with program analysis by extracting interface activity transition and dependency models.

Collectively, these techniques cover several important test objectives, such as coverage, test sequence length, execution time, readability and replicablity, yet none optimises these competing objectives simultaneously nor provides a set of optimal tradeoff solutions like SAPIENZ. Furthermore, many of these previously proposed techniques require detailed app information, such as source code [16, 53], general UI models [44] and interface and/or activity transition models [20, 53, 54, 55]. While any such additional information can help to guide test data generation, this additional information requirement can be an impediment to easy and widely-applicable automation. Given the pressing need for *fully* automated Android testing, we designed the SAPIENZ approach to require only the binary executable. Of the publicly available tools, Dynodroid and Monkey were found to perform best in the recent comprehensive study by Choudhary, Gorla and Orso [28]. Therefore, we regard these as denoting the state-of-the-art and state-of-current-practice, which we seek to improve by the introduction of SAPIENZ.

3. THE SAPIENZ APPROACH

We first outline the workflow used by our approach. Then we provide component summaries of our evolutionary algorithm. The exploration strategy and app analysers of SAPIENZ are described in Sections 3.2 and 3.3 respectively.

SAPIENZ' overall workflow is depicted in Figure 1. SAPIENZ starts by instrumenting the app under test, which can be achieved in a white box, grey box or black box manner as follows: When the app's source code is available, SAPIENZ uses fine-grained instrumentation at the statement-level (white box). By contrast, should it turn out that only the binary APK file is available (as is often the case in real-world, industrial-strength Android testing scenarios), SAPIENZ uses undexing and repacking to instrument the app at method-level (grey box). However, where the developers disallow repackaging (as is common for commercial apps), SAPIENZ uses a non-invasive activity-level 'skin' coverage, which can always be measured (black box).

SAPIENZ extracts statically-defined string constants by reverse-engineering the APK. These strings are used as inputs for seeding realistic strings into the app, which has been found to improve the performance of search-based software testing techniques for web based testing [13], and traditional application testing [32], and also to improve realism [23],

Algorithm 1: Overall algorithm of SAPIENZ.

Input: AUT A, crossover probability p, mutation probability q, max generation g_{max}, execution time t
Output: UI model M, Pareto front PF, test reports C

```
M ← K_0; PF ← ∅; C ← ∅;                          ▷ initialisation
generation g ← 0;
boot up devices D;                                ▷ prepare app exerciser
inject MotifCore into D;   ▷ for hybrid exploration (see §3.2)
static analysis on A;              ▷ for string seeding (see §3.3)
instrument and install A;
initialise population P;        ▷ hybrid of random and motif genes
evaluate P with MotifCore and update (M,PF,C);
while g < g_max and ¬ timeout(t) do
    g ← g+1;
    Q ← wholeTestSuiteVariation(P, p, q);   ▷ see Algorithm 2
    evaluate Q with MotifCore and update (M,PF,C);
    F ← ∅;                                        ▷ non-dominated fronts
    F ← sortNonDominated(P ∪ Q, |P|);
    P' ← ∅;                                       ▷ non-dominated individuals
    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);      ▷ see equation 3 for operator ≺_c
    P ← P'[0 : |P|];                              ▷ new population
return (M,PF,C);
```

but has not previously been used in Android testing. Test sequences are generated and executed by the MOTIFCORE component, which combines random fuzzing and systematic exploration, which corresponds to two types of genes: the low-level *atomic genes* and the high-level *motif genes*.

SAPIENZ' multi-objective search algorithm initialises the initial population via MOTIFCORE's *Test Generator*. During the genetic evolution process, genetic individuals are assigned to the *Test Replayer* when evaluating individual fitnesses. The individual test scripts are further decoded into executable Android events by the *Gene Interpreter*, which communicates with the the Android device via the Android Debugging Bridge (ADB). The *States Logger* monitors the execution states (e.g., covered activities, crashes) of the App Under Test (AUT) and produces measurement data for the *Fitness Extractor* to calculate the fitnesses. A set of Pareto-optimal solutions and test reports are generated at the end of the search.

3.1 Multi-objective Search Based Testing

Algorithm 1 presents SAPIENZ' top-level algorithm. SAPIENZ optimises for three objectives: code coverage, sequence length and the number of crashes found, using a Pareto-optimal Search Based Software Engineering (SBSE) approach [38, 39].

Each executable test suite $\vec{x}$ for the AUT is termed as a *solution* and a *solution* $\vec{x_a}$ is *dominated* by solution $\vec{x_b}$ ($\vec{x_a} \prec \vec{x_b}$) according to a fitness function if and only if:

$$\begin{aligned} &\forall i = 1, 2, ..., n,\ f_i(\vec{x_a}) \leq f_j(\vec{x_b}) \wedge \\ &\exists j = 1, 2, ..., n,\ f_j(\vec{x_a}) < f_j(\vec{x_b}) \end{aligned} \tag{1}$$

A *Pareto-optimal set* consists of all *Pareto-optimal* solutions (belonging to all solutions X_t), which is defined as:

$$P^* \triangleq \{\vec{x^*} \mid \nexists \vec{x} \in X_t, \vec{x} \prec \vec{x^*}\} \tag{2}$$

SAPIENZ' search-based approach uses NSGA-II to build successively-improved Pareto-optimal sets, seeking new

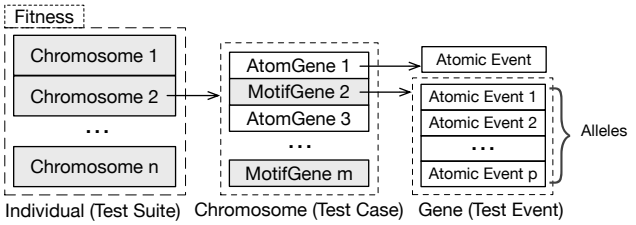

Figure 2: Genetic individual representation.

dominating test vectors. NSGA-II is a widely-used multiobjective evolutionary search algorithm, popular in SBSE research [39], the details of which can be found elsewhere [30].

At the end of search, testers can choose any test suites of interest from the Pareto-optimal set generated by SAPIENZ. In addition to the Pareto-optimal solution, SAPIENZ also produces an all-crash-test-suite with a set of videos for each crashing scenario. This crashing test suite is generated by an archive operator which stores any crashes found during the search process.

SBSE representation: SAPIENZ performs the whole test suite evolution [13, 33] thus each individual corresponds to a test suite. The representation of an individual test suite generated by SAPIENZ is illustrated in Figure 2. SAPIENZ generates a set of these individual test suites, which corresponds to a population of individuals in the evolutionary algorithm. Each individual consists of several chromosomes (test sequences $\langle T_1, T_2, ..., T_m \rangle$) and each chromosome contains multiple genes (test events $\langle E_1, E_2, ..., E_n \rangle$), which consist of a random combination of *atomic* and *motif genes*. An *atomic gene* triggers an atomic event e that cannot be further decomposed, e.g., press down a key, while a *motif gene* is interpreted as a series of *alleles* (atomic events $\langle e_1, e_2, ..., e_p \rangle$).

SBSE variation operator: We define a *whole test suite variation operator* to manipulate individuals. The operator is depicted in Algorithm 2: It applies one of the finer-grained *crossover*, *mutation* and *reproduction* operators on each individual (at test suite level). SAPIENZ' inter-individual variation is achieved by using a uniform set element *crossover* among individuals (test suites). The inner-individual variation is manipulated by a more complex *mutation* operator. Since each individual is a test suite containing several test cases, the operator first randomly shuffles test case orders and then performs a single-point crossover on two neighbouring test cases with probability q, where the prior shuffle operation aims to improve crossover diversity. Subsequently, the more fine-grained test case mutation operator shuffles the test events within each test case with probability q, by randomly swapping event positions. Although atomic events include (mutable) parameters, we choose instead to mutate the execution order of the events, thereby reducing the complexity of the variation operator. Mutants are possible to operate on new GUI widgets not exercised by *any* initial test case, because the timing of the operations are mutated. The *reproduction* operator simply leaves a randomly chosen individual unchanged.

SBSE selection: We use the *select* operator from NSGA-II [30], which defines a crowding-distance-based comparison operator $\prec_c$. For two test sequences $\vec{a}$, and $\vec{b}$. We say $\vec{a} \prec_c \vec{b}$ if and only if:

$$\vec{a}_{rank} < \vec{b}_{rank} \vee (\vec{a}_{rank} = \vec{b}_{rank} \wedge \vec{a}_{dist} > \vec{b}_{dist}) \quad (3)$$

Algorithm 2: The whole test suite variation operator.

Input: Population P, crossover probability p, mutation probability q
Output: Offspring Q

```
Q ← ∅;
for i in range(0, |P|) do
    generate r ~ U(0,1);
    if r < p then                                   ▷ apply crossover
        randomly select parent individuals x1, x2;
        x1', x2' ← uniformCrossover(x1, x2);
        Q ← Q ∪ x1'
    else if r < p + q then                          ▷ apply mutation
        randomly select individual x1;
        ▷ vary test cases within the test suite x1
        x ← shuffleIndexes(x1);
        for i in range(1, |x|, step 2) do
            generate r ~ U(0,1);
            if r < q then
                x[i-1], x[i] ← onePointCrossover(x[i-1], x[i]);
        ▷ vary test events within the test case x[i]
        for i in range(0, |x|) do
            generate r ~ U(0,1) ;
            if r < q then
                x[i] ← shuffleIndexes(x[i]);
        Q ← Q ∪ x
    else Q ← Q ∪ (randomly selected x1); ▷ apply reproduction
return Q;
```

This selection favours test sequences with smaller non-domination rank and, when the rank is equal, it favours the one with greater crowding distance (less dense region).

SBSE fitness evaluation: The fitness value is recorded as a triple for each of the objectives: coverage, length of the test and number of revealed crashes.

SBSE Fitness evaluation can be time-consuming, but it is fortunately also embarrassingly parallel [19, 25, 56, 62]. Therefore, in order to achieve time-efficient search, SAPIENZ supports parallel fitness evaluation, assigning individuals to multiple fitness evaluators, which may run on distributed devices (a single multicore machine was used in our evaluation, when comparing SAPIENZ with other techniques).

3.2 Exploration Strategy

Android apps can have complex interactions between the events triggerable from the UI, and the states reachable and consequent coverage achieved. In manual testing, the testers' knowledge can be deployed to explore such complex interactions [42]. However, for automated testing, some other way to handle complex interactions has to be found. Simple approaches to automated Android testing use only atomic events. Even with combinations of such events, the lack of state and context awareness, makes it difficult to discover complex interactions. This may be one reason why many research tools were found to under-perform by comparison with Monkey in the benchmark study conducted by Choudhary et al. [28].

To address this issue, SAPIENZ uses *motif patterns*, which collect together patterns of lower level events, found to be good at achieving higher coverage. *Motif genes* are based on the UI information available in the current view, which is widget-based for Android apps. *Motif genes* work together to perform behavioural usage patterns on the app, e.g, fill all input fields in the current view and submit.

This is achieved by pre-defining patterns to capture testers' experience regarding complex interactions with the

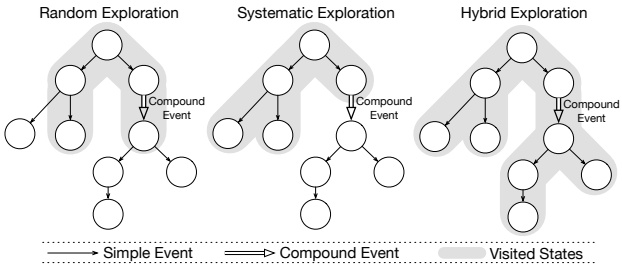

Figure 3: Hybrid exploration strategy.

app. The *motif gene* is inspired by how a DNA motif works: A DNA motif is a short sequence pattern that has a biological function. Motifs are combined with atomic sequences so that, together, they can express the overall DNA function. In our case, our *motif genes* seek to achieve high-level functions (by defining patterns) and to work together with *atomic genes* to achieve higher test coverage. As we explain below, in Section 3.4, our evaluation of SAPIENZ relies solely upon a single obvious, default, generic *motif gene*, to avoid any risk of experimenter bias. However, in future work, we may learn motifs from captured human-led test activities.

Hybrid exploration: *Atomic genes* and *motif genes* are complementary (see Figure 3), so SAPIENZ combines them to form hybrid sequences of test events. Random exploration may (randomly) manage to cover unplanned UI states for compound events (of which consists of a random combination of atomic events), but may generally achieve low overall coverage. Systematic exploration may achieve good coverage within planned UI state regions, but can be blocked by unplanned compounds. The hybrid strategy used by SAPIENZ is shown in Algorithm 3.

3.3 Static and Dynamic Analysis

SAPIENZ performs two types of analysis: static analysis for string seeding and dynamic analysis for multi-level instrumentation. These two features provide necessary information for SAPIENZ to generate realistic test inputs and to guide the search toward optimal test suites with high test coverages.

String seeding: In order to extract statically defined strings, SAPIENZ first reverse-engineers the APK file. SAPIENZ obtains a list of globally applicable strings from the decompiled XML resource files. These natural language strings are randomly seeded into the text fields by the MOTIFCORE component, when performing its hybrid exploration. We found this seeding to be particularly useful when testing apps that require a lot of user-generated content, e.g., Facebook, because it enables SAPIENZ to post and comment in an apparently more human-meaningful way. When the APK file cannot be reverse-engineered successfully, which is a common case for commercial apps, predefined dummy strings[2] will replace the extracted strings from the app.

Multi-level instrumentation for skeleton and skin coverage: In order to be practical and useful, an automated Android testing technique needs to be applicable to both open and closed-source apps. To achieve this, SAPIENZ uses multi-level instrumentation at one or all of the three levels of applicable instrumentation granularity. The coarsest instrumentation granularity is always possible, and is performed through activity/screen interactions to achieve black box testing or 'skin coverage' as we call it, because it only interacts with the 'surface' UI and system actions of the app. Carino and Andrews also use a similar metric based on the change of GUI widgets [26]. We use the term 'skeletal coverage' for the more fine-grained coverages, achieved by grey and white box instrumentation. In some cases, even when source code is unavailable, a finer-grained, grey box coverage is possible at the method level, which we term 'backbone' skeletal coverage. This backbone coverage can be achieved by undexing the APK file, inserting probes and then repackaging the binary file. Of course, where source code is available, we can and do use traditional statement coverage (which we term 'full skeletal coverage'). For such systems we can cover both the 'skeleton and the skin'; white box statement level coverage and black box user interface/activity coverage.

[2]In our particular implementation, a single string of '0' is used to ensure that no fields is empty.

Algorithm 3: The MOTIFCORE exploration strategy.

Input: AUT A, test sequence $T = \langle E_1, E_2, ..., E_n \rangle$, random event list $\mathcal{R}$, motif event list $\mathcal{O}$, static strings $\mathcal{S}$ existing UI Model M and test reports C
Output: Updated (M, C)
for each event E in T **do**
  if $E \in \mathcal{R}$ **then** ▷ `handle atomic gene`
    execute atomic event E and update M;
  if $E \in \mathcal{O}$ **then** ▷ `handle motif gene`
    $currentActivity \leftarrow extractCurrentActivity(A)$
    $uiElementSet \leftarrow extractUiElement(currentActivity)$
    for each *element* w in $uiElementSet$ **do**
      if w is *EditText widget* **then**
        seed string $s \in \mathcal{S}$ into w;
      else
        exercise w according to motif patterns in E;
      update M;
  $(a, m, s) \leftarrow$ get covered activities, methods, statements;
  $C \leftarrow C \cup (a, m, s)$; ▷ `update coverage reports`
if captured crash c **then**
  $C \leftarrow C \cup c$; ▷ `update crash reports`
return (M, C);

3.4 Implementation

We have implemented the SAPIENZ tool on top of the DEAP framework [31] for multi-objective test suite evolution. SAPIENZ achieves full skeletal coverage (statement coverage) using EMMA [6] and backbone coverage (method coverage) using ELLA [5]. It calculates skin coverage (activity coverage) by calling Android's own `ActivityManager` for extracting activity/screen information.

For *atomic genes*, the evaluation version of SAPIENZ supports 10 types of atomic events that originate from Android system source, including `Touch`, `Motion`, `Rotation`, `Trackball`, `PinchZoom`, `Flip`, `Nav` (navigation key), `MajorNav`, `AppSwitch`, `SysOp` (system operations such as 'volume mute' and 'end call'). Regarding *motif genes*, of course, there is a wide range of choices for motif patterns, and we distinguish between those that are generic (applicable to all apps) and those that are bespoke (applicable to only a small homogeneous set of apps). For our evaluation purposes, we resisted the temptation to have any bespoke *motif genes*, since these would require human intuition and intelligence. Furthermore, we imbued our evaluation version of the SAPIENZ tool with only a single (intuitively obvious) generic *motif gene*

that systematically exercises text fields and clickable UI widgets under the corresponding view, which is applicable to all apps. It first seeds strings into all text fields and then attempts to exercise each clickable widget to transfer to the next view. Such a motif pattern might perform appropriate actions in scenarios such as filling in and submitting a form. We used this simple-minded approach for the evaluation version of SAPIENZ, to avoid risking any experimenter bias that might otherwise introduce human ingenuity into the *motif gene* construction process. As a result, the findings reported in the following section can be regarded as lower bounds on the performance of our approach; with a smarter selection of generic motif patterns, results will improve, and would further improve with the construction of bespoke *motif genes* for particular apps.

The SAPIENZ tool generates a set of artefacts for reuse, including reusable *test suites*, detailed *coverage reports* and *crash reports* (with corresponding fault-revealing test cases and automatically captured crash videos as witnesses for the failures induced by test cases).

4. EVALUATION

We evaluate the SAPIENZ approach by conducting three empirical studies on both open-source and popular closed-source Android apps. We investigate whether SAPIENZ can optimise multiple objectives and find previously unknown real faults, within limited (30 minutes per app) execution time on real-world production hardware.

As a sanity check, we first want to establish that we have a reliable experimental infrastructure. This is because there are a number of settings and parameter choices that could affect the results and, as been widely noted in other areas of empirical software engineering [58, 60], the choice of parameter tuning options can have a dramatic effect on results. To ensure reliability, we check that our infrastructure replicates the results previously reported by Choudhary et al. [28].

RQ0 (Reliable replication): Does our experimental infrastructure reliably replicate the results from the recent thorough study by Choudhary et al. [28]?

We call this RQ0 (rather than RQ1) since it merely establishes that our experimental infrastructure replicates recent results, suggesting that it is reliable for answering the subsequent (novel) questions. A natural question to ask for RQ1, once we have established replication of Choudhary et al. in RQ0, is one that is asked by many other studies [20, 27, 51, 52, 53, 54, 61]: 'what coverage is achieved by the newly proposed technique?'

RQ1 (Code coverage): How does the coverage achieved by SAPIENZ compare to the state-of-the-art and the state-of-practice?

Coverage is one useful indicator, simply because failure to achieve coverage leaves aspects of the app untested. Nevertheless, there is evidence that coverage alone, cannot be relied upon to indicate test effectiveness [57]. Therefore, our second question focuses on fault detection; regardless of coverage achieved, the effectiveness of any software testing technique should also be assessed by its ability to reveal faults.

RQ2 (Fault revelation): How do the faults found by SAPIENZ compare to those found by the state-of-the-art and the state-of-practice?

SAPIENZ targets coverage, fault revelation and length of fault-revealing test cases. Longer test sequences might achieve higher coverage, but we need to provide short sequences to testers for debugging purposes [17]. Intuitively, shorter sequences are more likely to be attractive and actionable to developers [34, 49]. This motivates RQ3.

RQ3 (Sequence length): How does SAPIENZ compare to the state-of-the-art and the state-of-practice in terms of the length of the fault-revealing test sequences it returns?

We wish to go further in our empirical analysis, because the Choudhary et al. benchmark suite set [28], although an excellent starting point, consists of only 68 apps, whereas there are, in total (at the time of writing) 1,112 apps in the overall F-Droid community [7]. There could potentially be some sampling or other biases if we restrict ourselves solely to the benchmark apps. Furthermore, since SAPIENZ and the other techniques use randomised algorithms, it is widely regarded as best practice to perform an inferential statistical analysis of the performance of each algorithm, reporting statistical significance and effect size [18, 40]. Therefore, RQ4 investigates the findings that can be reported using statistical significance and effect size on multiple runs of the tools, each applied to a random sample of apps from the 1,112 F-Droid apps publicly available:

RQ4 (Statistical significance and effect size): How does SAPIENZ perform, compare to the state-of-the-art and the state-of-practice, on randomly selected apps, with inferential statistical testing?

Finally, we want to investigate the usefulness of the SAPIENZ technique on real-world commercial apps. Therefore, we follow the practice adopted by some previous authors [37, 52] of applying the technique to a large number of popular apps in Google Play. This avoids the potential bias of applying the technique only to apps chosen from F-Droid, which does not contain any of the most popular apps in current use. Since we do not have access to the source code of these popular commercial apps, it also tests the effectiveness of the technique when used in 'black box mode', where it has least available information to guide the test generation process, and only high level, non-invasive, 'skin coverage' instrumentation is possible.

RQ5 (Usefulness): Can SAPIENZ find any real bugs on popular closed-source real-world apps?

4.1 Experimental Setup

We conduct three studies to answer the above research questions: Study 1 addresses RQ0 to RQ3, Study 2 addresses RQ4 and Study 3 addresses RQ5. Study 1 and Study 2 are based on the execution of the testing approaches under evaluation on a single PC. Study 3 augments this, by using real-world physical (Samsung and Google) devices to demonstrate the practicality of SAPIENZ. For all these studies, we evaluate on Android KitKat version (API 19) because it is the most widely-used version [1] at the time of writing. All techniques under evaluation are fully automated. We choose not to provide manual assistance (e.g., logins) in testing the subjects, because we aim for an unbiased and rigorous assessment of what can be achieved entirely automatically.

Since Dynodroid itself manipulates the emulator and depends on its own customised Android system image, we follow its user guide [4] and use its own image file to execute the tool. For all the approaches under evaluation, we limit only the execution time and the assigned hardware resource, so that our comparison is direct head-to-head test effectiveness achieved in a certain amount of elapsed wall-clock time. This

setting is consistent with the benchmark study conducted by Choudhary et al. [28], which allows us to perform a direct comparison with the results in that previous study.

We set SAPIENZ's crossover and mutation probability to 0.7 and 0.3 respectively. The maximum generation is set to 100 with the population size of 50 and each individual contains 5 test cases. None of the parameters available to SAPIENZ are tuned; all remain set at the same value throughout all our experiments. We adopt this approach in order to ensure that the comparison is strictly fair; results for SAPIENZ might be improved by tuning, but this might also introduce bias and unfairness in the experimentation. We conducted Study 1 and Study 2 on a PC with a single hexa-core 3.50GHz CPU and 16GB RAM on Ubuntu 14.04. For Study 3, we also use a mobile device *Samsung Galaxy Note II* and a cluster of 10 *Google Nexus 7* (2013 version) tablets.

For Study 1, we test each subject for one hour by using each tools under evaluation. We record their achieved coverage every 5 minutes. When comparing fault-revealing test sequence lengths, we need to be careful to normalise the results: each technique might find a different number of faults, so measuring the total length of fault-revealing test sequences would be unfair. Rather, we compare the mean length of the fault-revealing test sequences returned by each approach. We count an atomic event as one event and decompose our high-level *motif genes* into multiple atomic events for a fair comparison.

For Study 2, we use random selection to identify 10 subjects from the 1,112 apps in the overall F-Droid set. We conduct an inferential statistical analysis of the performance of each of the Android testing techniques applied to these randomly selected apps. Details of the 10 randomly selected apps can be found in the left-hand columns of Table 5. Since we cannot rely on Gaussian (aka 'Normal') distribution of test results, we use a non-parametric multiple comparison inferential statistical significance test, the Kruskal-Wallis test [24] (at the 0.05 alpha level) with the Bonferroni correction, and the Vargha-Delaney effect size measure [59], as widely recommended [18, 40]. The differences between approaches are characterised as small, medium and large when the $\hat{A}_{12}$ effect size exceeds 0.56, 0.64, and 0.71, respectively. We repeat each experiment 20 times to provide a sample of runs for statistical analysis. In total, this more rigorous statistical evaluation requires 25 days of execution time.

Since Study 3 concerns the evaluation of SAPIENZ on 1,000 apps, it is inherently time-consuming. Fortunately, since we are interested in the usefulness of the technique, we want to investigate whether it can find faults quickly. Therefore, we restrict the wall-clock execution time for this study to 30 minutes per app per setting. Furthermore, since emulators may not reflect real device behaviour perfectly, we conduct this study under three device settings: on a PC with emulators, on a smart mobile device (*Samsung Note II*) and on a small cluster of 10 tablets (*Google Nexus 7*). The entire computation time of the experiment, on all 1,000 apps under three settings, to answer RQ5 is 1,050 hours (nearly 44 days); 500 hours on emulators, 500 hours on the Samsung Note II and 500/10 hours on the Google Nexus 7 tablets. In this study, we use only the non-invasive 'skin coverage' to guide SAPIENZ, so the results are a lower bound on the performance that would be observed by a developer, who could have access to source code and could therefore exploit the finer granularity levels of coverage.

Table 2: Results on the 68 benchmark apps.

Subject	Coverage			#Crashes			Length		
	M	D	S	M	D	S	M	D	S
a2dp	43	29	46	0	1	3	-	315	148
aarddict	14	46	18	0	0	0	-	-	-
aLogCat	68	49	71	0	0	2	-	-	114
Amazed	66	63	69	1	0	1	1429	-	96
AnyCut	63	65	66	0	0	1	-	-	103
baterrydog	64	66	67	0	1	1	-	81	173
swiftp	13	13	14	0	0	0	-	-	-
Book-Catalogue	46	27	33	1	0	1	1941	-	177
bites	38	25	41	1	0	1	19124	-	116
battery	76	68	79	0	0	4	-	-	198
addi	16	26	20	2	1	2	1367	315	129
alarmclock	72	51	77	4	1	5	1716	170	144
manpages	64	68	75	0	0	3	-	-	120
mileage	40	25	54	2	1	4	878	390	153
autoanswer	13	24	16	0	0	0	-	-	-
hndroid	4	6	10	2	1	2	206	-	117
multismssender	43	49	61	0	0	0	-	-	-
worldclock	93	94	94	0	0	1	-	-	98
Nectroid	69	46	76	1	0	2	416	-	118
acal	15	15	29	1	0	5	62717	-	177
jamendo	62	3	72	0	0	2	-	-	191
aka	79	76	84	1	0	7	42804	-	136
yahtzee	62	51	58	2	0	0	31767	-	-
aagtl	30	29	31	4	0	5	1756	-	188
CountdownTimer	60	62	62	0	0	0	-	-	-
sanity	32	1	19	2	1	2	8377	12	90
dalvik-explorer	69	*	73	2	*	4	3720	*	165
Mirrored	69	68	64	0	0	1	-	-	147
dialer2	38	55	42	0	0	0	-	-	-
DivideAndConquer	85	72	83	0	0	2	-	-	186
fileexplorer	40	56	50	0	0	0	-	-	-
gestures	36	48	52	0	0	0	-	-	-
hotdeath	78	3	79	1	0	3	63975	-	152
adsdroid	23	36	38	2	1	1	356	48	128
myLock	28	33	31	0	0	0	-	-	-
lockpatterngenerator	78	79	81	0	0	0	-	-	-
mnv	49	*	67	2	*	4	30381	*	150
aGrep	*	38	*	*	0	*	*	-	*
k9mail	7	5	7	0	0	1	-	-	238
LolcatBuilder	24	23	31	0	0	0	-	-	-
MunchLife	70	73	76	0	0	0	-	-	-
MyExpenses	51	25	65	0	1	2	-	67	150
LNM	58	66	60	1	0	1	51621	-	48
netcounter	44	63	77	0	0	2	-	-	156
bomber	76	70	73	0	0	0	-	-	-
frozenbubble	*	63	*	*	0	*	*	-	*
fantastischmemo	36	9	60	1	0	6	25375	-	156
blokish	50	50	52	1	1	2	2512	252	194
zooborns	35	38	36	0	0	0	-	-	-
importcontacts	41	43	42	0	0	0	-	-	-
wikipedia	36	32	32	0	0	5	-	-	232
PasswordMaker	63	53	64	3	0	1	3406	-	180
passwordmanager	11	7	16	0	0	0	-	-	-
Photostream	16	23	38	1	1	2	317	29	125
QuickSettings	50	33	50	0	0	1	-	-	134
RandomMusicPlayer	58	82	59	0	0	0	-	-	-
Ringdroid	26	*	29	1	*	2	550	*	161
soundboard	42	60	53	0	0	0	-	-	-
SpriteMethodTest	82	37	83	0	0	0	-	-	-
SpriteText	59	57	62	0	0	0	-	-	-
SyncMyPix	21	20	22	0	0	4	-	-	187
tippy	83	48	83	0	0	0	-	-	-
tomdroid	55	43	58	0	1	1	-	165	91
Translate	48	45	49	0	0	0	-	-	-
Triangle	76	69	79	0	0	0	-	-	-
weight-chart	58	57	77	2	1	4	10588	236	186
whohasmystuff	74	*	80	0	*	0	-	*	-
Wordpress	4	*	7	0	*	1	-	*	137

4.2 State of the Art and Practice

According to the thorough empirical study by Choudhary et al. [28], existing techniques fail to outperform the standard Monkey Android testing tool in 'continuous mode'. In this mode, each testing tool is given one hour execution time and the same hardware configuration. We therefore chose to evaluate in the same way, comparing against Monkey and Dynodroid, which Choudhary et al. found to perform best among the research prototype techniques (beating recently proposed techniques including black box based AndroidRipper [15], A^{3}E [20], PUMA [37] and white-box based ACTEve [16]). Monkey and Dynodroid also performed best in a slightly more recent study [57], and, therefore, if SAPIENZ outperforms both Monkey and Dynodroid, we will also have reasonable evidence to conclude that it is likely to outperform AndroidRipper [15], A^{3}E [20], PUMA [37] and ACTEve [16]. Note that SAPIENZ also yields a Pareto front at the end of its execution, which might be a useful by-product. However, we choose to evaluate SAPIENZ only in the 'continuous mode', for a fair comparison with Monkey and Dynodroid, which do not yield Pareto fronts.

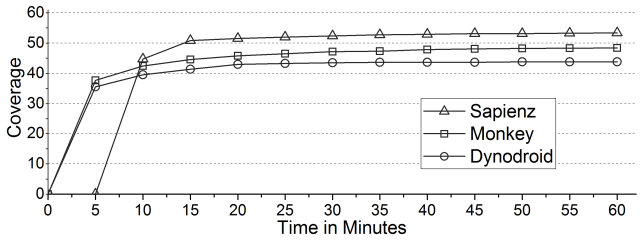

Figure 4: Progressive coverage on benchmark apps.

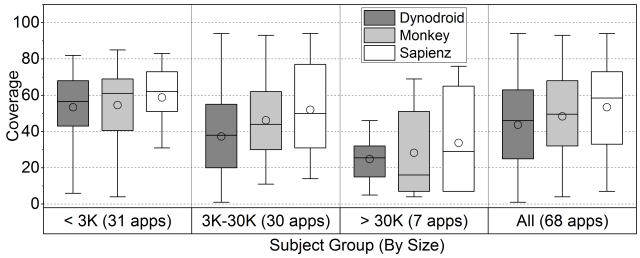

Figure 5: Code coverage on the 68 benchmark apps.

4.3 Results

4.3.1 Study 1: Benchmark Subjects

The detailed experimental results on each subject for Study 1 are given in Table 2, where 'Coverage' reports statement coverage achieved by each of the three tools, '#Crashes' indicates the number of unique crashes detected by each and 'Length' reports the fault-revealing test sequence length for each. The column headings 'M', 'D' and 'S' refer to the three tools we compare; Monkey, Dynodroid and Sapienz. The entry '*' indicates the tool cannot start the corresponding app, while the entry '-' indicates that the fault-revealing length is undefined, because no faults were found.

RQ0 (Experimental replication). We first evaluate Monkey and Dynodroid to check that our experiment infrastructure replicates the results reported by Choudhary et al. [28]. We calculated progressive average coverages across all 68 subjects every 5 minutes for each of the three techniques and report the direct comparison on the final coverages achieved. The progressive coverages of Monkey and Dynodroid are shown in Figure 4. The shape of the growth in coverage over time very closely resembles the results reported by Choudhary et al. [28]. However, the final coverage values achieved by these two tools are slightly higher than those reported by Choudhary et al. This may be caused by the hardware setting: Choudhary et al. ran the experiments on virtual machines while we conducted our experiments on a physical PC which may be faster. Since the overall growth trend closely resembles the results of Choudhary et al., and given that better performance only raises the bar that Sapienz must clear in order to outperform them, we believe these results indicate we have a firm foundation on which to perform our subsequent experiments.

RQ1 (Code coverage). We used an identical evaluation approach for Sapienz as that used in the replication study reported in RQ0 for Monkey and Dynodroid. As can be seen from Figure 4, Sapienz outperformed Monkey and

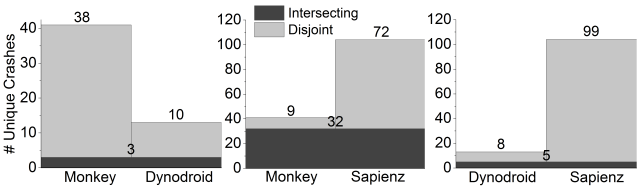

Figure 6: Pairwise comparison on found crashes.

Table 3: Statistics on found crashes.

App Crashes	Monkey	Dynodroid	Sapienz
# App Crashed	24	13	41
# Unique Crashes	41	13	104
# Total Crashes	1,196	125	6,866

Dynodroid from the 10^{th} minute onwards, finally achieving the highest overall statement coverage at the end of the hour's experimental time allowed for each of the 68 subjects. To further investigate these results, Figure 5 presents the boxplots (for which a circle indicates the mean) of the final coverage results for apps grouped by size-of-app. This analysis reveals that Sapienz achieved the highest mean coverage across all four app size groups. We conclude that there is evidence from the 68 benchmark apps that Sapienz can attain and maintain superior coverage after approximately 10 minutes of execution on a standard equipment.

RQ2 (Fault revelation). In answering RQ2, we report not only on the number of crashes found by each technique, but also the overlap between the crashes found by each technique. This allows us to investigate whether the techniques are complementary, or whether one subsumes another, as well as reporting on the overall effectiveness (in terms of number of crashes found). Of course a crash may be triggered by different test sequences, so we report *unique* crashes, considering a crash to be unique when its stack trace differs from all others. We excluded those crashes caused by the Android system or the test harness itself, which were not caused by the faults from the subjects. Such crashes can be identified by checking the corresponding stack traces. A recent study [57] has highlighted this issue and pointed out that these crashes are, essentially false positives, so should not be counted.

As shown in Table 3, Sapienz revealed the largest number of both unique and total crashes in 41 of the 68 apps. Sapienz also found 30 unique crashes in 14 apps for which neither Monkey nor Dynodroid found any crashes. We also provide a pairwise comparison of the unique crashes found in Figure 6 (where the black bars show common crashes; those revealed by both techniques): Across the 68 subjects, Sapienz found 72 and 99 unique crashes, undetected by Monkey and Dynodroid respectively, while it missed only 9 crashes found by Monkey and 8 by Dynodroid. We conclude that there is strong evidence from the 68 benchmark apps that Sapienz outperforms both Monkey and Dynodroid in terms of fault revelation, as measured by the number of crashes found.

RQ3 (Sequence length). Table 4 shows the mean length of fault-revealing test sequences of the three tools, grouped by various subject size ranges (where the group sizes are given in the brackets). On all subject groups except '3K-30K', Sapienz generated the shortest fault-revealing test sequences. On the '3K-30K' subject group, Dynodroid gener-

Table 4: Fault-revealing test sequence length.

		Monkey	Dynodroid	Sapienz
Size	< 3K (31)	13,843	186	132
	3K-30K (30)	14,775	77	153
	> 30K (7)	21,501	276	169
Overall (68)		15,305	161	149

ated the shortest fault-revealing test sequences (although its code coverage and number of found crashes are lower than SAPIENZ). We conclude that there is strong evidence from the 68 benchmark apps that SAPIENZ outperforms the fault-revealing test sequence length of Monkey, and that on larger subjects it also outperforms Dynodroid.

4.3.2 Study 2: Inferential Statistical Analysis

RQ4 (Statistical significance and effect size). For all 10 randomly sampled F-Droid programs, and for all three criteria of interest, SAPIENZ outperformed both Monkey and Dynodroid. However, in this study, we are concerned with the statistical significance in effect size of these results. We first present the boxplots of the performance comparison on 10 F-Droid subjects, as shown in Figure 7.

Table 5 shows Vargha-Delaney $\hat{A}_{12}$ effect size for the three objectives, coverage, the number of crashes found and fault-revealing sequence length. For each objective, the columns contain the effect size comparisons for SAPIENZ-Monkey (S-M), SAPIENZ-Dynodroid (S-D), and, for completeness, Monkey-Dynodroid (M-D), where the result is significant. As shown in the table, SAPIENZ significantly outperforms Monkey with large effect size on 7/10 subjects for coverage, 8/10 for crashes, and 10/10 for length (with large effect size). SAPIENZ significantly outperforms Dynodroid, with large effect size on 9/10 subjects for coverage, 9/10 for crashes and 10/10 for length. We also replicated the finding of Choudhary et al. [28] that Monkey tends to outperform Dynodroid, but further note that it does so less conclusively than SAPIENZ does. The overall results suggest that SAPIENZ outperforms both the state-of-the-art and the state-of-practice approaches on all three objectives.

4.3.3 Study 3: Top 1,000 Popular Apps

RQ5 (Usefulness). In total, SAPIENZ found 558 unique crashes in 329 of the 1,000 Google Play apps to which it was applied. In the previous study of Dynodroid [52], it also tested top 1,000 apps, however the budget used and total number of found unique crashes are not mentioned. The authors found 6 bugs (that led to non-native crashes) in 5 out of 1,000 apps tested. Our found 558 unique crashes were caused by 22 types of errors/exceptions. The distribution of the most common crash types (those with more than 4 crashes each) is shown in Figure 8, revealing that most were caused by 'native' crashes, indicating that the crash occurred outside the Android Java Virtual Machine, while executing the app's native code. Another common class of crashes found were those due to null pointers.

We reported the non-native crashes to the app provider, giving a stack trace for each crash type. In total, we reported 175 crashes[3]. Unfortunately, since these apps are commercial apps, we do not have direct access to the developers, as

[3]For each app, we reported the first found crash that corresponds to each non-native crash type. We did not report native crashes because their stack traces do not explicitly point to the source lines of the potential faults.

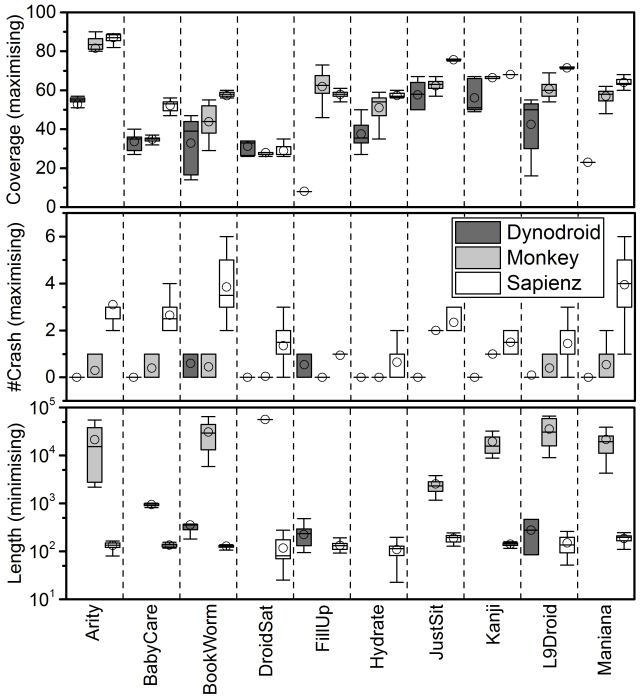

Figure 7: Performance comparison on 10 F-Droid subjects. (Boxplots grouped by subject.)

one might in an open-source environment, but we were able to contact only the associated customer support team. We got 58 replies in total, excluding those that were automatic generated. For such a 'cold call' outreach activity, 58 from 175 emails is relatively high [35, 47].

Of these 58 replies, in 27 cases we got feedback from the app developers (after our email was redirected by their customer support teams). Furthermore, 14 developer teams confirmed that the crashes resulted from real faults in their apps, and 6 of them have already fixed the reported crashes. Among the 13 unconfirmed crashes out of 27 developer replies, 6 indicated that our reports were helpful or that the developers were working on the issue. A further 6 respondents seek additional information. One of the 13 responded that they could not identify the cause of the crash.

We list the anonymised details[4] of these 14 faults confirmed by developers in Table 6: These 14 apps vary greatly in categories and install numbers, with at least 148 million installs in total. The 6 confirmed faults, with further fixes from their developers are labelled as 'Confirmed' in the 'Fixed' column. For the remaining 8 apps, we found that 7 of the confirmed crashes can no longer be observed when testing their most recent versions. However since we have not received confirmation from developers that these faults are definitely fixed, we label them as 'Unconfirmed' in the 'Fixed' column. We observed only one of the confirmed faults was not fixed (still crashes).

4.4 Threats to Validity

Like any empirical study, there are potential threats to validity of our experimental results:

Internal validity: Threats to internal validity concern

[4]App versions are omitted for anonymity.

Table 5: Vargha-Delaney effect size ('-' indicates a statistically insignificant result).

Subject	Description	Ver.	Date	SLOC	Coverage			#Crash			Length		
					S-M	S-D	M-D	S-M	S-D	M-D	S-M	S-D	M-D
Arity	Scientific calculator	1.27	2012-02-11	2,821	-	1.00	1.00	-	1.00	0.98	1.00	1.00	-
BabyCare	Timer for when to feed baby	1.5	2012-08-23	8,561	1.00	1.00	-	0.84	0.92	-	1.00	1.00	-
BookWorm	Book collection manager	1.0.18	2011-05-04	7,589	0.96	1.00	-	0.97	1.00	-	1.00	0.95	-
DroidSat	Satellite viewer	2.52	2015-01-11	15,149	-	-	-	1.00	1.00	-	0.90	0.90	-
FillUp	Calculate fuel mileage	1.7.2	2015-03-10	10,400	-	1.00	1.00	0.73	0.73	-	0.95	0.80	0.23
Hydrate	Set targets for water intake	1.5	2013-12-09	2,728	0.85	1.00	0.92	0.95	-	0.23	0.73	0.73	-
JustSit	Meditation timer	0.3.3	2012-07-26	728	1.00	1.00	-	1.00	1.00	-	1.00	1.00	1.00
Kanji	Character recognition	1.0	2012-10-30	200,154	1.00	1.00	0.84	-	1.00	1.00	1.00	1.00	0.98
L9Droid	Interactive fiction	0.6	2015-01-06	18,040	1.00	1.00	0.99	0.89	0.90	-	0.94	0.91	-
Maniana	User-friendly todo list	1.26	2013-06-28	20,263	0.99	1.00	1.00	1.00	1.00	-	1.00	1.00	-

Table 6: Confirmed app faults identified by Sapienz.

App	Category	Installs	Caused By	Device	Description	Fixed
P*	Photography	10M-50M	`NullPointer`	Nexus 7	Unable to start activity from a customer support SDK.	Unconfirmed
K*	Simulation	10M-50M	`NullPointer`	Nexus 7	Concurrent error while executing `doInBackground()`	Unconfirmed
B*	Business	10K-50K	`NullPointer`	Nexus 7	Null object reference in a third party SDK	No
D*	Education	500K-1M	`NullPointer`	Emulator	Exception from event handler `onOptionsItemSelected()`	Confirmed
T*	Simulation	10K-50K	`NullPointer`	Emulator	Exception from `onAnimationEnd()` in `FlipGameActivity`	Confirmed
T*	Lifestyle	500K-1M	`NullPointer`	Emulator	Error when `CameraUpdateFactory` is not initialized	Confirmed
T*	Transport	1M-5M	`NullPointer`	Emulator	Exception from `onClick()` in `StationInfoFragment`	Confirmed
S*	Education	1M-5M	`NullPointer`	Emulator	Unable to start a third party activity	Unconfirmed
T*	Weather	10M-50M	`NullPointer`	Emulator	Error when `CameraUpdateFactory` is not initialized	Unconfirmed
W*	Weather	10K-50K	`OutOfMemory`	Note II	Error inflating class on binary XML file	Unconfirmed
S*	Puzzle	5M-10M	`ActivityNotFound`	Note II	No Activity found to handle `SHARE_GOOGLE` Intent.	Unconfirmed
F*	Photography	10M-50M	`NullPointer`	Note II	Exception from `onGlobalLayout()` in `ViewUtil`	Confirmed
T*	Music&Audio	100M-500M	`NullPointer`	Note II	Unable to start the activity of `PlayerActivity`	Unconfirmed
P*	Music&Audio	5K-10K	`ActivityNotFound`	Note II	No Activity found to handle a `View` Intent	Confirmed

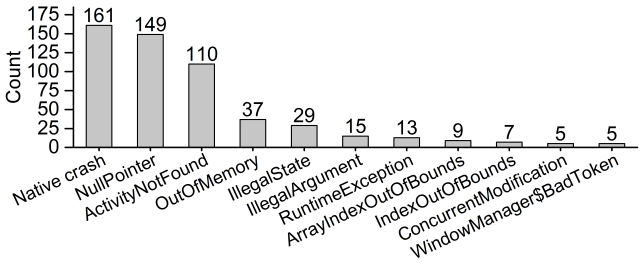

Figure 8: Main crash types on Google Play subjects.

factors in our experimental methodology that may affect our results. For Study 1, 50 of the 68 ASE benchmark subjects originate in a single article [52], which might have resulted in selection bias. To mitigate this issue, we conducted Study 2 on 10 open-source apps, selected using unbiased random sampling. Regarding the particular Sapienz implementation, we implemented only a single motif pattern to exercise all text fields and clickable UI widgets under the corresponding view, which is applicable to all apps. Performance of Sapienz may improve when considering different motif patterns, but could not be worse, since this single option will always be available. Also, the choice of parameter setting for each of the three tools may affect their performance significantly. To reduce this threat, we followed the default configurations for Monkey and Dynodroid, as used in the previous thorough benchmark assessment study Choudhary et al. [28] and we resisted any temptation to tune Sapienz.

External validity: Threats to external validity arise when the experimental results cannot be generalised. Like all empirical studies, we are limited in the number of subject systems to which we can apply our tools and techniques. Our results will not necessarily generalise beyond the 1,078 apps to which we have applied Sapienz. However, we think it promising that the technique applies, out of the box, to so many different apps, none of which have been 'cherry picked' (nor in any other way 'chosen' by the experimenters themselves). It is possible, of course, that the 1,000 most popular apps, and the F-Droid open-source apps, have peculiar characteristics not shared by other classes of apps, for which the performance of the three techniques we studied in this paper may differ. We also only evaluated our approach on a single version of the Android platform. Although the most widely-used version, the rapid evolution of the Android system, means that the performance of three evaluated techniques may vary as subsequent versions become available.

5. CONCLUSIONS

This paper has introduced a novel multi-objective search-based software testing technique and tool Sapienz for automated Android app testing. Sapienz supports multi-level instrumentation and remains applicable, even when only app's APK file (and nothing else) is available. Its evolutionary algorithm continuously optimises for coverage, sequence length and the number of crashes found, seeking to reveal as many crashes as possible, while minimising the length of test sequences.

Our evaluation results on open-source apps have shown that Sapienz outperforms the state-of-the-art technique Dynodroid and the widely-used tool, Android Monkey, on all three objectives for almost all subjects. The only exception is the relatively small (3K-30K lines of code) F-Droid open-source apps in the benchmark suite, for which Dynodroid produced shorter fault-revealing test sequences, although it achieved less coverage and revealed fewer crashes.

We also believe that Sapienz is a practical and useful testing tool, since it was able to find 558 unique crashes in the top 1,000 most popular Android apps, 14 of which have already been confirmed as caused by real faults.

6. REFERENCES

[1] Android dashboards. http://developer.android.com/about/dashboards/index.html.

[2] Android fragmentation visualized. http://opensignal.com/reports/2015/08/android-fragmentation.

[3] Appium: Automation for iOS and Android apps. http://appium.io.

[4] Dynodroid user guide. http://code.google.com/p/dyno-droid.

[5] ELLA: A tool for binary instrumentation of Android apps. http://github.com/saswatanand/ella.

[6] EMMA: A free Java code coverage tool. http://emma.sourceforge.net.

[7] F-Droid. http://f-droid.org.

[8] Modisco. http://www.eclipse.org/modisco.

[9] Number of Android applications. http://www.appbrain.com/stats/number-of-android-apps.

[10] Robotium: User scenario testing for Android. https://github.com/RobotiumTech/robotium.

[11] A. Abran, J. W. Moore, et al. Guide to the software engineering body of knowledge (SWEBOK®). In *2004 Version, IEEE CS Professional Practices Committee*, 2004.

[12] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of Android test suites in adverse conditions. In *Proc. of ISSTA'15*, pages 83–93, 2015.

[13] N. Alshahwan and M. Harman. Automated Web application testing using search based software engineering. In *Proc. of ASE'11*, pages 3–12, 2011.

[14] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[15] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proc. of ASE'12*, pages 258–261, 2012.

[16] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. of ESEC/FSE'12*, pages 59:1–59:11, 2012.

[17] A. Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering*, 38(3):497–519, May 2012.

[18] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. of ICSE'11*, pages 1–10, 2011.

[19] F. Asadi, G. Antoniol, and Y. Guéhéneuc. Concept location with genetic algorithms: A comparison of four distributed architectures. In *Proc. of SSBSE'10*, pages 153–162, 2010.

[20] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proc. of OOPSLA'13*, pages 641–660, 2013.

[21] J. Bach. Exploratory testing. In *The Testing Practitioner*, pages 253–265, 2004.

[22] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[23] M. Bozkurt and M. Harman. Automatically generating realistic test input from web services. In *Proc. of SOSE'11*, pages 13–24, 2011.

[24] N. Breslow. A generalized Kruskal-Wallis test for comparing K samples subject to unequal patterns of censorship. *Biometrika*, 57(3):579–594, 1970.

[25] E. Cantú-Paz and D. E. Goldberg. Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2–4):221–238, 2000.

[26] S. Carino and J. H. Andrews. Dynamically testing GUIs using ant colony optimization. In *Proc. of ASE'15*, pages 138–148, 2015.

[27] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proc. of OOPSLA'13*, pages 623–640, 2013.

[28] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *Proc. of ASE'15*, pages 429–440, 2015.

[29] comScore. The global mobile report. http://comscore.com/Insights/Presentations-and-Whitepapers/2015/The-Global-Mobile-Report, 2015.

[30] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[31] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, July 2012.

[32] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proc. of ICTS'12*, pages 121–130, 2012.

[33] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[34] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proc. of ISSTA'10*, pages 147–158, 2010.

[35] M. T. Frohlich. Techniques for improving response rates in OM survey research. *Journal of Operations Management*, 20(1):53–62, 2002.

[36] Google. Android Monkey. http://developer.android.com/tools/help/monkey.html.

[37] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proc. of MobiSys'14*, pages 204–217, 2014.

[38] M. Harman. The current state and future of search based software engineering. In *Proc. of FOSE'07*, pages 342–357, 2007.

[39] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, November 2012.

[40] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In B. Meyer and M. Nordio, editors, *Empirical software engineering and verification:*

LASER 2009-2010, pages 1–59. 2012. LNCS 7007.

[41] J. Itkonen, M. V. Mantyla, and C. Lassenius. How do testers do it? an exploratory study on manual testing practices. In *Proc. of ESEM'09*, pages 494–497, 2009.

[42] J. Itkonen, M. V. Mantyla, and C. Lassenius. The role of the tester's knowledge in exploratory software testing. *IEEE Transactions on Software Engineering*, 39(5):707–724, 2013.

[43] J. Itkonen and K. Rautiainen. Exploratory testing: A multiple case study. In *Proc. of ESEM'05*, pages 84–93, 2005.

[44] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. of ISSTA'13*, pages 67–77, 2013.

[45] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Proc. of ESEM'13*, pages 15–24, 2013.

[46] C. Kaner, J. Bach, and B. Pettichord. *Lessons learned in software testing*. 2008.

[47] M. D. Kaplowitz, T. D. Hadlock, and R. Levine. A comparison of web and mail survey response rates. *Public Opinion Quarterly*, 68(1):94–101, 2004.

[48] P. S. Kochhar, F. Thung, N. Nagappan, and T. Zimmermann. Understanding the test automation culture of app developers. In *Proc. of ICST'15*, pages 1–10, 2015.

[49] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Proc. of ASE'07*, pages 417–420, 2007.

[50] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai. On the accuracy, efficiency, and reusability of automated test oracles for Android devices. *IEEE Transactions on Software Engineering*, 40(10):957–970, October 2014.

[51] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining Android app usages for generating actionable GUI-based execution scenarios. In *Proc. of MSR'15*, pages 111–122, 2015.

[52] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proc. of ESEC/FSE'13*, pages 224–234, 2013.

[53] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *Proc. of ESEC/FSE'14*, pages 599–609, 2014.

[54] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in gui testing of android applications. In *Proc. of ICSE'16*, 2016. To appear.

[55] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[56] B. S. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *Proc. of WICSA'01*, pages 181–190, 2001.

[57] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *Proc. of ICST'16*, 2016. To appear.

[58] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proc. of ICSE'16*, 2016. To appear.

[59] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[60] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proc. of ESEC/FSE'13*, pages 455–465, August 2013.

[61] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proc. of FASE'13*, pages 250–265, 2013.

[62] S. Yoo, M. Harman, and S. Ur. GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards. *Journal of Empirical Software Engineering*, 18(3):550–593, June 2013.