AndroFL: Evolutionary-Driven Fault Localization for Android Apps

Vishal Singh[1], Ravi Shankar Das[1], Prajwal H G[2], Subhajit Roy[1]
[1]*Indian Institute of Technology Kanpur*, India
[2]*InMobi*, India
Email: vshlsng@cse.iitk.ac.in, ravishankerdas1998@gmail.com, prajwal.g@inmobi.com, subhajit@iitk.ac.in

***Abstract*—We present our tool, ANDROFL, that provides an infrastructure for an evolutionary algorithm-based test-suite generation backed by a statistical fault localization module for diagnosing faults. ANDROFL's evolutionary test-generator supports configurable fitness functions (e.g., coverage, diagnosability metrics like Ulysis). The statistical fault localization engine supports popular metrics like Ochiai, Tarantula and Barinel, and allows adding custom fault localization metrics. We evaluated ANDROFL on 20 open-sourced apps from F-Droid, and demonstrates significant efficiency gains: it reduces debugging effort by 74% (median EXAM score) compared to random testing—enabling developers to pinpoint faults ≈ 4× faster. Furthermore, ANDROFL localizes 25% and 50% more faults compared to random testing in the top-5 and top-10 ranked list in worst case ranking scenario.**

***Index Terms*—GUI fault localization, Crash faults, Android testing, Automated debugging**

I. INTRODUCTION

The rapid growth of Android has fostered a vast application ecosystem, but this expansion introduces critical quality challenges—particularly GUI-triggered faults causing application crashes, degraded user experience, and compromised reliability [1]. There has been significant progress in automated GUI testing—"many testing techniques and tools" have been developed to detect crash bugs and improve app reliability [2]. Preventing crashes is indeed a top priority for developers, as stability directly affects user satisfaction [2] (crash-prone apps tend to lose users and drop in store ratings). These studies underscore the critical need for efficient diagnosis and repair of GUI-triggered faults in mobile apps.

Current GUI testing tools—including random testing (MONKEY [3]), model-based approaches (APE [4], COMBODROID [5]), deep-learning based (HUMANOID [6]), evolutionary techniques (SAPIENZ [7], STOAT [8]), state-based methods (TIMEMACHINE [9]), and reinforcement learning (Q-TESTING [10])—demonstrate significant effectiveness in detecting crashes through test generation.

However, these tools exclusively address crash *detection*, creating a critical diagnostic gap: They report crashes, coverage and stack traces but provide no systematic fault localization. This forces developers to manually inspect code—a time-intensive process that delays fault resolution. This limitation stems from two key factors: (1) architectural focus on crash induction rather than diagnosis, and (2) inflexibility preventing adoption of localization heuristics.

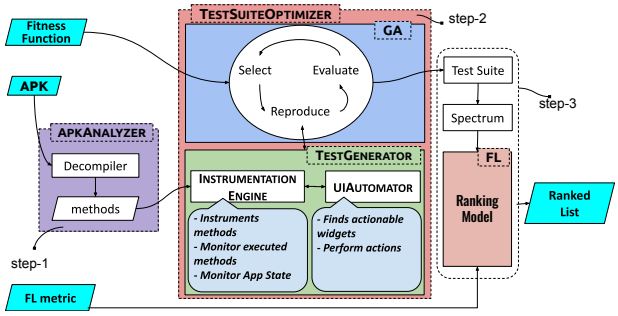


Fig. 1: Workflow of ANDROFL

To address these limitations, we present ANDROFL[1] , a fault localization framework integrating evolutionary test-suite optimization with spectrum-based fault localization (SBFL) [11] to generate ranked suspicious components. Unlike conventional evolutionary approaches maximizing coverage or crash counts, ANDROFL optimizes test suites for fault localization efficacy. Using Ulysis [2] [12] as a fitness function, ANDROFL guides optimization to maximize fault-localization signals rather than merely detecting crashes. Operating at method-level granularity, ANDROFL produces a ranked list of suspicious methods to guide debugging.

Evaluation on 20 open-source Android applications shows ANDROFL reduces debugging effort by 74% (median EXAM score) compared to random testing baseline.

Contributions:

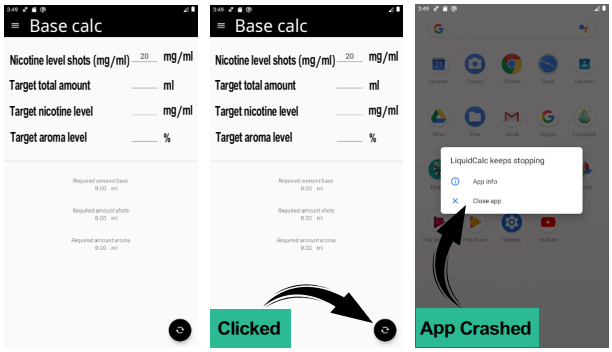
- We propose an integrated framework combining evolutionary test generation with SBFL for diagnosing UI-triggered Android faults, optimizing test suites via diagnostic metrics.
- We provide an extensible interface for user-defined fitness functions and fault localization metrics.

II. PRELIMINARIES

UI Automation. Appium[3] enables cross-platform UI testing via *WebDriver* protocol. It extracts application UI states as XML hierarchies, identifying actionable widgets to generate interaction sequences for systematic interface exploration.

[1]ANDROFL: https://github.com/PRAISE-group/AndroFL
[2]lower scores indicate test suites more amenable to fault localization.
[3]https://appium.io/

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00353



(a) (b) (c)

Fig. 2: Crash sequence: (a) Launch, (b) "Reload" click, (c) Crash.

TABLE I: Actionable UI elements identified in Fig. 2a by UIAUTOMATOR. XP denotes XPath-based selectors; ID refers to `resource-id` attributes.

#	Action	Value(s)
1,2	Click	1-XP:`/*[@content-desc="Open drawer"]`, 2-ID:`fab`
3-6	Text	3-ID:`basis1StaerkeInput`, 4-ID:`basis1MengeInput`, 5-ID:`zielstaerkeInput`, 6-ID:`aromaInput`

Dynamic Instrumentation. Frida[4] enables real-time method tracing by injecting JavaScript hooks into running processes. It instruments methods by logging executions to a binary vector (1: executed; 0: not executed) via entry/exit hooks, providing fine-grained coverage tracing and crash detection.

Evolutionary Test-Suite Generation. Evolutionary algorithms [13] optimize test suites via genetic operations: *selection* prioritizes high-fitness suites, *crossover* combines parent suites, and *mutation* modifies test cases. A *fitness* function defines how to optimize a test suite and can be adapted to enhance diagnostic effectiveness.

Spectrum-Based Fault Localization (SBFL). SBFL [11] correlates coverage with test outcomes to rank program components. For method c, Ochiai [11] metric computes suspiciousness as:

$$\text{sus}(c) = \frac{|T_F \cap \text{covered}(c)|}{\sqrt{|T_F| \times (|T_F \cap \text{covered}(c)| + |T_P \cap \text{covered}(c)|)}}$$

where T_P and T_F denote passing/failing tests. Then, components are sorted by descending suspiciousness scores to produce a ranked list $\mathcal{R}$.

Wasted Effort (WE). WE quantifies debugging efficiency using the EXAM score [14] as $\frac{r}{m}$, where r is the rank of the first faulty component and m is the total components. Lower exam scores indicate more effective localization.

ANDROFL integrates these components: *Appium* automates interactions, *Frida* traces method executions, *evolutionary* algorithm optimizes test suites, SBFL ranks faults, and WE evaluates diagnostic efficacy—completing the test-to-diagnosis cycle.

[4]https://frida.re/

```
m.implementation = function(...args){
        send("enter " + methodName(m));
        var ret = m.call(this, ...args);
        send("exit " + methodName(m));
        return ret;
}
```

Listing 1: JavaScript snippet used by INSTRUMENTATIONENGINE to log method entry and exit for m.

```
public void onClick(View view){
        ...
        throw new RuntimeException("not found");
}
```

Listing 2: Method f_{12} containing a defect that triggers a `RuntimeException`.

III. MOTIVATING EXAMPLE

We illustrate ANDROFL using an open-source app, *LiquidCalc*, exhibits a reproducible crash triggered by specific UI interaction sequences (Fig. 2). Despite being relatively small in scope, the app presents non-trivial GUI logic and internal state transitions—making it an ideal for evaluating fault localization. ANDROFL systematically uncovers and pinpoints root cause of failure through its end-to-end diagnostic workflow.

Step-1. APKANALZER decompiles APK to extract 20 developer methods (Tab. II). These methods enable coverage tracing during test execution through binary instrumentation.

Step-2 To initiate test suite optimization, TESTSUITEOPTIMIZER leverages the provided experimental configurations and begins with configured set of 20 randomly generated test suites $(\mathcal{T}_1 \ldots \mathcal{T}_{20})$. Over the course of configured 50 generations, these suites evolve via selection (guided by Ulysis), crossover, and mutation. This yields an optimized test suite $\mathcal{T}^*$ (Tab. III) of 9 test cases, specifically tailored to enhance fault localization efficacy.

Test Case Generation. During mutation, when a new test case is required, TESTSUITEOPTIMIZER delegates its generation to TESTGENERATOR. The process below demonstrates how test case t_3 was constructed:

- INSTRUMENTATATIONENGINE starts the app, instruments methods using List. 1, initializes a binary vector $\vec{v}$, and signals UIAUTOMATOR when ready (Fig. 2a).
- UIAUTOMATOR parses UI state to XML (Fig. 2a, identifying interactable widgets (Tab. I)). Then selects stochastic action (i.e., clicking "Reload" in Fig. 2b).
- During *click* execution, the INSTRUMENTATIONENGINE updates $\vec{v}$ as methods execute and monitor app state.
- Upon detecting a post-click crash (Fig. 2c), TESTGENERATOR finalizes $\vec{v}$ and records the failed state.

This generates $t_3 = \langle actions, \vec{v}, \text{-} \rangle$ with minimal overhead.

Step-3. Using $\mathcal{T}^*$'s coverage data and outcomes (3 passing, 6 failing tests), ANDROFL constructs a spectrum (Tab. III). Applying the Ochiai metric, FL computes suspicion scores, and generates ranked list $\mathcal{R}$ (last two rows in Table III). The faulty method f_{12} (`MA$1.onClick`) ranks 1 in $\mathcal{R}$ (List. 2) versus rank 9 under random testing—reducing debugging

TABLE II: Instrumented methods from *LiquidCalc*, with abbreviated class names. Refer to the appendix for full method descriptions.

Developer Methods	$\mathcal{F}$
MA.aromarechner()	f_1
⋮	⋮
MA$1.onClick()	f_{12}
⋮	⋮
VerdunFrag.onCreateView()	f_{20}

TABLE III: Test suite spectrum for *LiquidCalc*, including method-level suspiciousness scores and ranks. Ties are broken using the worst-case ranking strategy.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}	E
t_1	1	1	1	1	1	0	0	1	0	1	1	0	1	1	1	0	0	0	0	0	+
t_2	1	1	1	1	1	0	1	1	0	1	1	0	1	0	0	1	1	0	1	0	+
t_3	1	1	1	1	1	0	0	1	0	1	1	1	1	0	1	0	0	1	0	0	-
t_4	1	1	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	-
t_5	1	1	1	1	1	0	0	1	0	1	1	1	1	0	0	0	1	1	1	0	-
t_6	1	1	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	+
t_7	1	1	1	1	1	0	0	1	0	1	1	1	1	0	0	0	0	0	1	1	-
t_8	1	1	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	-
t_9	1	1	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	-
sus	0.82	0.82	0.82	0.82	0.82	0.00	0.55	0.55	0.00	0.82	0.82	0.93	0.82	0.00	0.29	0.00	0.29	0.58	0.47	0.41	
$\mathcal{R}$	9	9	9	9	9	20	12	12	20	9	9	**1**	9	20	16	20	16	10	13	14	

effort by 89% (EXAM 0.05 vs. 0.45) while demonstrating precise diagnostic capability.

IV. Implementation

This section describes AndroFL's architecture and workflow. As shown in Fig. 1, the system comprises three sequentially integrated components that execute in the following order:

APK Analyzer (Step-1). The APKAnalyzer decompiles apps via `dexdump`, identifies developer-written methods ($\mathcal{F}$) through package name heuristics (e.g., `org.*.mainapp`). The output method set $\mathcal{F}$ serves as the foundation for subsequent coverage tracking.

Test Suite Optimizer (Step-2). The TestSuiteOptimizer uses a Genetic Algorithm to evolve *test suites* via configurable fitness functions (e.g., coverage, diagnosability). The evolutionary process selects parent test suites based on fitness scores and generates new offspring test suites through genetic operations. The GA module operates through *crossover* and *mutation* operations at test suite level: *crossover* combines test cases from parent test suites to form new suites; *mutation* modifies test suites by removing, replacing or adding test cases when required. When *mutation* necessitates new test cases, GA module invokes the TestGenerator module.

TestGenerator module coordinates two sub-modules: First, InstrumentationEngine starts app, injects Frida *hooks* (List. 1) into each method $f \in \mathcal{F}$ and concurrently maintaining (1) a coverage bit-vector $\vec{v} \in \{0,1\}^{|\mathcal{F}|}$ where each bit indicates method execution status, and (2) application state monitoring (*running/crashed*). After instrumentation, uiAutomator extracts UI states as XML, finds actionable elements, performs stochastic actions (e.g., button clicks), and observes outcomes. Following each action, InstrumentationEngine updates $\vec{v}$ and verifies app state. Test creation terminates immediately upon crash detection or when reaching developer-defined action limits, yielding a test case as $\langle$actions, $\vec{v}$, state$\rangle$.

GA iterates until convergence/time-limit/max-iteration, outputting optimized test suite $\mathcal{T}^*$.

Fault Localizer (Step-3). The Fault Localizer uses $\mathcal{T}^*$ to identify suspicious code components. It constructs a spectrum matrix $\mathcal{S} = [s_{ij}]$ where rows represent test cases $t_i \in \mathcal{T}^*$ and columns represent methods $f_j \in \mathcal{F}$, with entries $s_{ij} = 1$ if f_j executed during t_i and 0 otherwise. Using configurable fault localization metrics (e.g., Ochiai, Tarantula), the module computes suspicion scores for each method, and outputs a ranked list $\mathcal{R}$ where methods are sorted in descending suspiciousness.

```
@register_fitness_function("CustomFitness")
def custom_fitness(individual):
        # Custom logic
```

Listing 3: Extending AndroFL with user-defined fitness functions via decorator-based registration.

```
@register_metric("myCustom_metric")
def custom_metric(metrics_obj):
        # Custom logic
```

Listing 4: User-defined fault localization metrics can be added to AndroFL via decorator registration in designated files.

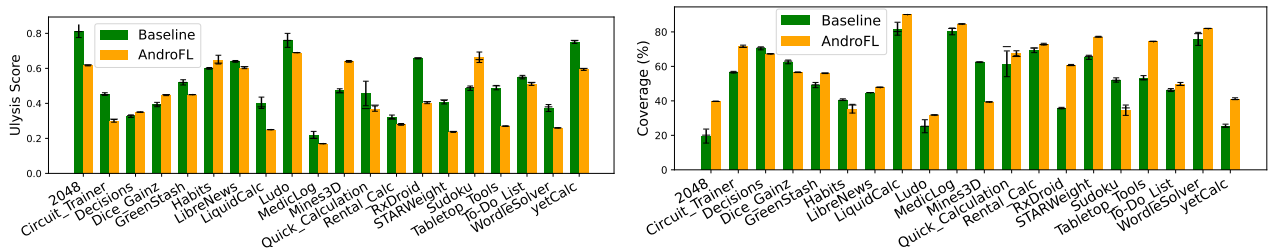
Extensions. AndroFL provides two decorator-based extension points for user-defined functions/metrics. Implement your fitness functions in `user_fitness_functions.py` (List. 3) and your metrics in `user_metrics.py` (List. 4), then register them with `@register_fitness_function` or `@register_metric`. The framework automatically exposes registered functions/metrics through configuration interfaces, requiring only the registered name for activation in test workflows.

V. Empirical Evaluation

This section evaluates AndroFL's core capabilities in test-suite optimization and fault localization.

Experimental Setup. We evaluated AndroFL on 20 open-source apps from F-Droid [5] (500+ LOC each, diverse domains). Due to the lack of fault localization ground truth in existing benchmarks [2], we manually seeded common UI-triggered faults (e.g., RuntimeException, NullPointerException) in UI-accessible methods. We used Android 11 emulators with Frida v16.0.3, Appium v2.19.0. The genetic algorithm employed a maximum of 20 actions per test case, population size 50, elite retention 20, crossover rate 0.1, and mutation rate 0.9 (parameters optimized via grid search on 5 apps). We used Ochiai as fault localization metric and compared against Monkey as baseline, as it represents the state-of-practice in industry crash testing.

[5] https://f-droid.org/en/



(a) Ulysis fitness score (lower is better)

(b) Coverage (higher is better)

Fig. 3: Distributions of Ulysis fitness scores and coverage metrics for 20 apps across 3 independent runs.

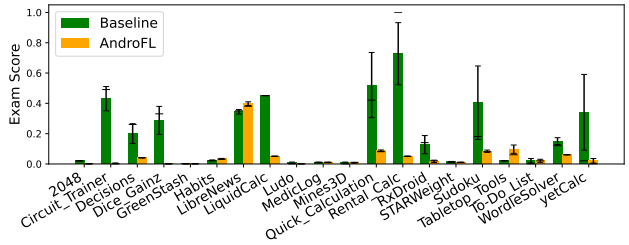


Fig. 4: EXAM scores for 20 applications across 3 independent runs.

TABLE IV: Top-N precision (N=1,5,10) and median EXAM scores comparing ANDROFL against the baseline. The "Improvement" row reports relative gains (↑) and reductions (↓).

Approach	Top-N Precision (%)			Median EXAM
	1	5	10	
ANDROFL	25	50	75	0.019
BASELINE	25	40	50	0.073
Improvement	**0%**	**25%** ↑	**50%** ↑	**74%** ↓

Evaluation Metrics. We measure fault localization effectiveness via Top-N precision (N=1,5,10) [15] and EXAM score, with significance tested by Wilcoxon signed-rank ($\alpha = 0.05$).

Test-Suite Quality. We compared ANDROFL against the BASELINE on diagnosability and coverage on generated test suites. Fig. 3a and 3b show distributions of fitness score and coverage across 3 different runs. ANDROFL significantly outperformed the baseline in fitness (Ulysis) with $p = 0.0164$, as shown in Fig. 3a. While coverage (Fig. 3b) showed no significant difference ($p = 0.0664$), indicating comparable effectiveness in terms of coverage. This confirms ANDROFL's unique focus: optimizing for diagnosability generates tests that amplify fault-revealing patterns in test suites, enhancing SBFL accuracy.

Fault Localization Performance. Figure 4 shows EXAM score distribution of 3 different runs. ANDROFL significantly outperform BASELINE with p = 0.0065. As shown in Tab. IV, ANDROFL outperforms baseline in Top-5 and Top-10 precision (25% and 50% relative improvement respectively) while maintaining equal Top-1 precision. The median EXAM score is reduced by 74%, indicating significantly less code examination is needed to locate faults.

These results confirm that optimization of test suites for diagnosability directly enhances fault localization efficacy, requiring developers to inspect less code to identify faults.

VI. CONCLUSION AND FUTURE WORK

This paper delivers an end-to-end framework for fault localization in Android apps—a capability critically absent in existing tools. By synergizing evolutionary test generation with SBFL, it outperforms random testing in localizing faults.

Our assessment relies on seeded faults which may not fully reflect real-world bug. Future work will extend instrumentation beyond the method level and investigate LLM-guided UI exploration for more efficient test generation.

REFERENCES

[1] H. Cai, "Embracing mobile app evolution via continuous ecosystem mining and characterization," in *MOBILESoft '20*, 2020, p. 5.

[2] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *ESEC/FSE '21*, 2021, p. 12.

[3] Monkey, https://developer.android.com/studio/test/other-testing-tools/monkey.

[4] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *ICSE*. IEEE, 2019, pp. 269–280.

[5] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *ICSE*, 2020, pp. 469–480.

[6] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *ICSE-C*. IEEE, 2017, pp. 23–26.

[7] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *ISSTA*, 2016, pp. 94–105.

[8] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *FSE*, 2017, pp. 245–256.

[9] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 481–492.

[10] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *ISSTA*, 2020, pp. 153–164.

[11] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007, pp. 89–98.

[12] P. Chatterjee, A. Chatterjee, J. Campos, R. Abreu, and S. Roy, "Diagnosing software faults using multiverse analysis." in *IJCAI*, 2020, pp. 1629–1635.

[13] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[14] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *PRDC'06*. IEEE, 2006, pp. 39–46.

[15] J. Zheng, H. Wang, Z. Jin, and X. Tan, "Fault localization based on weighted association rule mining," *CoRR*, vol. abs/1906.02279, 2019.