Guided, Stochastic Model-Based GUI Testing of Android Apps

Ting Su[1,2], Guozhu Meng[2], Yuting Chen[3], Ke Wu[1]
Weiming Yang[1], Yao Yao[1], Geguang Pu[1], Yang Liu[2], Zhendong Su[4]*
[1]School of Computer Science and Software Engineering, East China Normal University, China
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
[4]Department of Computer Science, University of California, Davis, USA
{suting,gzmeng}@ntu.edu.sg,chenyt@cs.sjtu.edu.cn,sei_wk2009@126.com,ywm0822@qq.com
sei_yaoyao@126.com,ggpu@sei.ecnu.edu.cn,yangliu@ntu.edu.sg,su@cs.ucdavis.edu

ABSTRACT

Mobile apps are ubiquitous, operate in complex environments and are developed under the time-to-market pressure. Ensuring their correctness and reliability thus becomes an important challenge. This paper introduces *Stoat*, a novel guided approach to perform stochastic model-based testing on Android apps. Stoat operates in two phases: (1) Given an app as input, it uses dynamic analysis enhanced by a weighted UI exploration strategy and static analysis to reverse engineer a stochastic model of the app's GUI interactions; and (2) it adapts Gibbs sampling to iteratively mutate/refine the stochastic model and guides test generation from the mutated models toward achieving high code and model coverage and exhibiting diverse sequences. During testing, system-level events are randomly injected to further enhance the testing effectiveness.

Stoat was evaluated on 93 open-source apps. The results show (1) the models produced by Stoat cover 17~31% more code than those by existing modeling tools; (2) Stoat detects 3X more unique crashes than two state-of-the-art testing tools, Monkey and Sapienz. Furthermore, Stoat tested 1661 most popular Google Play apps, and detected 2110 previously unknown and unique crashes. So far, 43 developers have responded that they are investigating our reports. 20 of reported crashes have been confirmed, and 8 already fixed.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Software testing and debugging**;

KEYWORDS

Mobile Apps, GUI Testing, Model-based Testing

ACM Reference Format:
Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106298

*Geguang Pu and Yuting Chen are the corresponding authors.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106298

1 INTRODUCTION

Mobile apps have become ubiquitous and drastically increased in number over the recent years. As recent statistics [29] shows, over 50K new Android apps are submitted to Google Play each month. However, it is challenging to guarantee their quality. First, they are event-centric programs with rich graphical user interfaces (GUIs), and interact with complex environments (*e.g.*, users, devices, and other apps). Second, they are typically developed under the time-to-market pressure, thus may be inadequately tested before releases. When performing testing, developers tend to exercise those functionalities or usage scenarios that they believe to be important, but may miss bugs that their designed tests fail to expose.

To tackle this challenge, many techniques [2, 4, 6, 39–41] have been proposed. Symbolic execution [4, 60] tracks the origins and handles of events at source-code level, and generates tests by exhaustively exploring program paths. Random testing [28, 39] fuzzes apps by generating a stream of random events. Evolutionary algorithm [40, 41] generates tests by randomly mutating and crossovering event sequences to fullfill their optimization goals.

Model-based testing (MBT) [23, 54] is another popular approach to automating GUI testing, which abstracts the app behaviors by a model, and then derives tests from it to validate apps. However, exhaustively generating tests from a model to validate app behavior is overwhelming. For example, *Bites* [20] is a simple cookbook app (shown in Figure 2a) with 1027 lines of code, and its model has 21 states and 70 transitions (generated by our approach). This model can generate 6 one-event sequences, 36 two-event sequences, 567K three-event sequences, which are rather time-consuming to execute. Due to this path-explosion problem, it is practically infeasible to derive all potential tests and execute them. As a result, traditional MBT techniques choose to generate random tests and use model-level coverage criteria [2, 43] (*e.g.*, covering all transitions) as testing goals. However, without a strong guidance, such tests are often redundant and ineffective to detect bugs. In addition, the previous research on model-based GUI testing [1, 2, 7, 15, 18, 31, 32, 42, 48, 57, 67] only considers UI-level events (*e.g.*, `click`, `edit`), and disregards system-level events (*e.g.*, screen rotation, incoming calls) during testing. Without combining both types of events, the effectiveness of MBT may be further limited due to inadequate testing.

Furthermore, most apps are developed without models in practice. Despite much effort [1, 2, 19, 42, 57, 67] in manually or automatically constructing models to represent GUI interactions, MBT's effectiveness is still limited due to incomplete UI exploration. For example, as a recent extensive study [16] shows, the models produced by existing GUI exploration tools achieve fairly low coverage (only half of the coverage achieved by Monkey).

The aforementioned challenges underline the importance of developing effective model-based testing techniques to unleash its potential. To this end, we propose a novel stochastic model-based testing approach, *Stoat*[1] (STOchastic model App Tester), to improve GUI testing of Android apps. It aims to thoroughly test the functionalities of an app from the GUI model, and validate the app's behavior by enforcing various user/system interactions [66]. Given an app as input, Stoat operates in two phases. First, it generates a stochastic model from the app to describe its GUI interactions. In our setting, a stochastic model for an app is a finite state machine (FSM) whose edges are associated with probabilities for test generation. In particular, Stoat takes a dynamic analysis technique, enhanced by a weighted UI exploration strategy and static analysis, to explore the app's behaviors and construct the stochastic model.

Second, Stoat iteratively mutates the stochastic model and generates tests from the model mutants. By perturbing the probabilities, Stoat is able to generate tests with various compositions of events to sufficiently test the GUI interactions, and purposely steers testing toward less travelled paths to detect deep bugs. In particular, Stoat takes a guided search algorithm, inspired by Markov Chain Monte Carlo (MCMC) sampling, to search for "good" models (discussed in Section 4.2) — the derived tests are expected to be diverse, as well as achieve high code and model coverage.

Moreover, Stoat adopts a simple yet effective strategy to enhance MBT: randomly inject various system-level events [44] into UI tests during MCMC sampling. It avoids the complexity of incorporating system-level events into the behavior model, and further imposes the influence from outside environment to detect intricate bugs.

In all, this paper makes the following contributions:

- ***Model construction***. Stoat employs a dynamic analysis technique, enhanced by a weighted UI exploration strategy and static analysis, to effectively explore app behaviors and construct models. The enhancement helps achieve significantly more complete models, which can cover 17~31% more code than the models generated by existing GUI exploration tools.
- ***Fault detection***. We employ Gibbs sampling, an instance of MCMC sampling, to guide stochastic model-based testing. On the 93 open-source apps, Stoat achieves satisfactory coverage and detects about 3X more unique crashes than the state-of-the-art testing tools, Monkey and Sapienz, which clearly demonstrates the benefits of our approach. In particular, Stoat detects 91 more crashes by injecting system-level events during MBT.
- ***Implementation and evaluation***. We have implemented Stoat as an automated tool and further evaluated it on 1661 most popular apps from Google Play. Stoat detects 2110 unique crashes from 691 apps. So far, 20 crashes are confirmed as real faults and 8 are already fixed. The results show that Stoat is effective in testing real-world apps.

[1]The early idea of Stoat, named *FSMdroid*, was presented in [55].

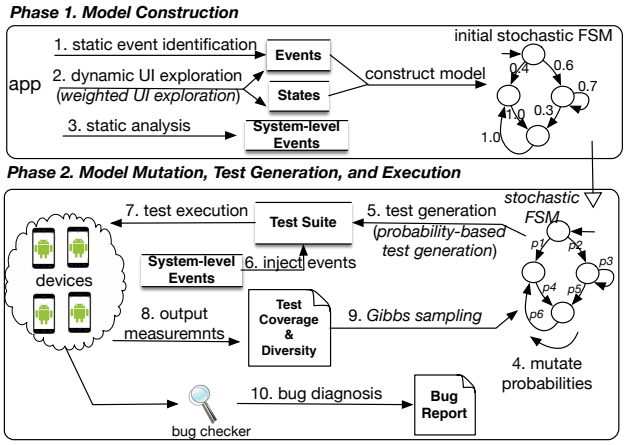


Figure 1: Stoat's workflow.

2 APPROACH OVERVIEW

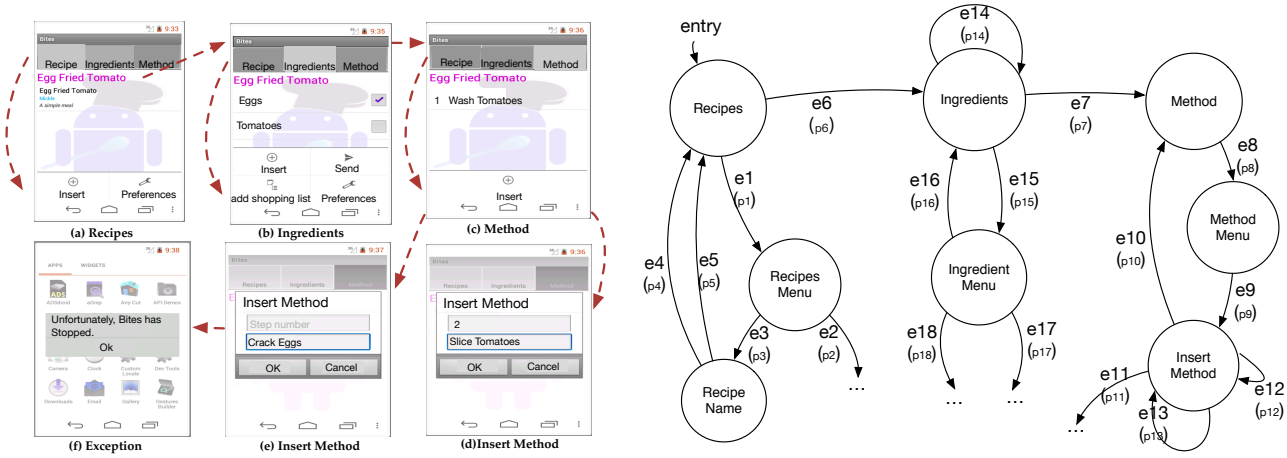
Stoat operates in a unique two-phase process to test an app. Figure 1 shows its high-level workflow.

Phase 1: Model construction. Stoat first constructs a stochastic Finite State Machine (FSM) to describe the app's behaviors. It uses a dynamic analysis technique, enhanced by a weighted UI exploration strategy (step 2 in Figure 1), to efficiently explore app behaviors. It infers input events by analyzing the UI hierarchy of app pages, and dynamically prioritizes their executions to maximize code coverage. In addition, to identify some potentially missing events, a static analysis (step 1) is performed to scan the registered event listeners in the app code. Stoat records the execution frequencies of all UI events during exploration, and later uses them to generate the initial probability values of the transitions in the model. The details will be explained in Section 3.

Phase 2: Model mutation, test generation, and execution. To thoroughly test an app, Stoat leverages the model from *Phase 1* to iteratively guide test generation toward yielding high coverage and exhibiting diverse event sequences. In detail, Stoat works as a loop: randomly mutate the transition probabilities of the current stochastic model (step 4), generate the tests from the model *w.r.t.* the probabilities (step 5), randomly inject system-level events (analyzed by static analysis in step 3) into these UI-level tests to enhance MBT (step 6), replay them on the app (step 7) and collect test results, such as code and model coverage and event sequence diversities (step 8).

Informed by the test results, Stoat exploits Gibbs sampling to decide whether the newly proposed model should be accepted or rejected (step 9), the model with better objective value will be accepted for the next iteration of mutations and samplings; otherwise, it will be rejected with certain probability to avoid local optimal (if rejected, the original model will be reused). Once any bug is detected (*i.e.*, crash or non-responding), further analysis will be performed to diagnose the bug with the corresponding test (step 10). The details will be explained in Section 4.

An Illustrative Example. *Bites* [20] is a simple cookbook app (shown in Figure 2a) that supports recipe creation and sharing. A user can create a recipe by clicking the *insert* menu item in the *Recipes* page (page a). When the user taps the name of a recipe, the app navigates to the *Ingredients* page (page b), where he/she can



(a) Screenshots of a cookbook app *Bites*. (b) App model of *Bites*.

Figure 2: Example app *Bites* and its app model.

view or add ingredients, share them via SMS, or add them into a shopping list. The user can also switch to the *Method* page (page c), where the cooking methods can be viewed. By clicking the *insert* menu item, the user can fill in *Step number* and *Method* (page d).

Figure 2b shows a part of the constructed app model for *Bites*, where each node denotes an app state and each edge a state transition (associated with a probability value). For example, *Recipes* can be navigated to *Ingredients* when e_6 occurs (*i.e.*, *click* a recipe item on the *Recipes* page) with the probability p_6. Stoat generates UI-level tests from this model, and randomly injects system-level events into them during Gibbs sampling. For example, *Bites* can be activated by SMS and Browser to read the recipes shared by others. During testing, Stoat simulates these system-level events by sending specific Broadcast Intents to *Bites*.

3 STOCHASTIC MODEL-BASED TESTING

3.1 Stochastic Model

Stoat uses a stochastic Finite State Machine (FSM) model to represent an app's behaviors. Formally, a stochastic FSM is defined as a 5-tuple $M = (Q, \Sigma, \delta, s_0, F)$, where Q and Σ are the sets of app states and input events, respectively, $s_0 \in Q$ the starting app state, $F \subseteq Q$ the set of final states, and $\delta : Q \times \Sigma \to \mathbb{P}(Q \times [0,1])$ the probabilistic transition function. $\mathbb{P}(\cdot)$ is the powerset operator and each transition is of the form $(s, e, (s', p))$, meaning that the probability of an event e triggering a state transition from s to s' is p. Let an app state s have k event transitions (say $e_1, \ldots, e_i, \ldots, e_k$, $1 \le i \le k$) and p_i is the probability value of e_i. For s, $\sum_{i=1}^{k} p_i = 1$ holds.

In our setting, an app state s is abstracted as an app page (represented as a widget hierarchy tree, where non-leaf nodes denote layout widgets (e.g., `LinearLayout`) and leaf nodes executable widgets (e.g., `Button`)); when a page's structure (and properties) changes, a new state is created (*e.g.*, in Figure 2a, the *Recipes* page and the *Ingredients* page correspond to two app states). If the app exits/crashes, the ending state is treated as a final state (*e.g.*, page f). An edge corresponds to an input event e denoting a UI action (*e.g.*, `click`, `edit`). An app moves from one state s to another state s' by handling an input event e. For example, when the user presses the *menu* key on the *Recipes* page, a menu will pop up, and a new app state (corresponding to *Recipes Menu*) is created. A probability value p is assigned to each transition e, denoting the selection weight of e in test generation. The initial probability values are determined by the execution frequency of each event during model construction — p is initially assigned the ratio of e's observed execution times over the total execution times of all events *w.r.t.* s ($e \in s$).

Test Generation from the Model. Stoat adopts a probabilistic strategy to generate event sequences from a stochastic model. It starts from the entry state s_0, and follows the probability values to select an event from the corresponding app state until the maximum sequence length or the ending state is reached. The higher the event probability value is, the more likely the event will be selected.

3.2 Model Construction

Stoat adopts a dynamic UI exploration strategy, enhanced by static analysis, to construct the stochastic model for the app under test.

Dynamic UI Exploration. An app can navigate among various pages with different UIs to provide its functionalities. In order to efficiently construct more complete behavior models of apps, we investigated 50 most popular Google Play apps from top 10 categories (*e.g.*, Education, Business, Tools) and manually explored as many functionalities as possible. At last, we summarized three observations that are crucial to improve the exploration performance, which constitute the basis of our weighted UI exploration strategy:

- *Frequency of Event Execution.* All UI events are given opportunities to be executed. The less frequently an event is executed, the more likely it will be selected during subsequent exploration.
- *Type of Events.* Different types of events are not equally selected. For instance, compared with normal UI events (*e.g.*, `click`), navigation events (*e.g.*, `back`, `scroll`, and `menu`), are given different priorities to ensure they are triggered at right timing, otherwise they may drastically undermine the exploration efficiency.
- *Number of Unvisited Children Widgets.* If an event solicits more new UI widgets on the next page, it will be prioritized since more efforts should be spent on pages with new functionalities.

To realize these rules, Stoat assigns each event e an execution weight, which is adjusted dynamically at runtime. The weight of an event is defined as:

$$execution\_weight(e) = \frac{\alpha * T_e + \beta * C_e}{\gamma * F_e} \quad (1)$$

where T_e is determined by its event type (1 for normal UI events, 0.5 for *back* and *scroll*, 2 for *menu*), C_e denotes the number of unvisited children widgets, F_e is its history execution frequencies, and α, β and γ are the weight parameters.[2]

Algorithm 1 outlines the stochastic model construction process. It takes an app as input, and outputs its corresponding model M. The algorithm infers the invocable events E from the current app page s according to the UI widgets on it, and then adds them into an event list storing all events during the dynamic analysis (lines 6-7). For example, in the app page *Insert Method* (d) in Figure 2a, since there are two `EditTexts` (*"Step number"* and *"Method"*) and two `Buttons` (*"Ok"* and *"Cancel"*) (the *clickable* properties of them are true), Stoat infers four events, *i.e.*, *edit("Step number")*, *edit("Method")*, *click("Ok")*, and *click("Cancel")*. Before each execution, the weights of all events in the list will be updated *w.r.t.* Formula (1) (lines 8-9).

An event e with the maximum weight (computed by function `getMaxWeightEvent`) on the page s is chosen to execute (lines 10-11). The function `expandFSM` accepts the returned state and the executed event to construct the model (line 15). At last, all transitions of M are assigned with the initial probability values according to their observed execution times (lines 17, and 18-24).

During the UI exploration, if the app enters into an *unknown* state (*e.g.*, the app crashes, exits, becomes non-responding, or navigates to an irrelevant app) after some event is executed, `restoreApp` will be executed to restart the app or navigate the app back to the previous page (lines 12-14). This unknown state is taken as a final state. This event will be added in $Tabu$, and excluded from further executions (line 10) to prevent affecting modeling efficiency.

Static Event Identification. The dynamic analysis technique typically infers events from the UI hierarchy (dumped by Android `UIAutomator` [27]), which only captures static GUI layout information. However, it may miss some dynamic events, *e.g.*, a *menu* action of an Activity that can only be invoked by pressing the menu key, or some events that are programmed in the app code, *e.g.*, a `longClick` action registered on a `TextView`. To further improve modeling capability, Algorithm 1 uses static analysis to identify these potential events that are missed by dynamic analysis (line 4). It detects events by scanning the event listeners in the app code, and then associates these events to the widgets observed at runtime via their unique resource IDs. Stoat detects the events that are registered on UI widgets (*e.g.*, `setOnLongClickListener`) and implemented by overriding class methods (*e.g.*, `onCreateOptionsMenu`).

Model Compaction. The number of an app's states and transitions can be large or even unbounded [7, 15, 42, 50, 67]. To improve the testing efficiency, Stoat compacts the model by identifying only structurally different pages as different states, and merges similar ones. In detail, (1) the hierarchy tree of an state is encoded into a string, and converted into a hash value for efficiently detecting duplicate states; (2) minor UI information, *e.g.*, text changes (*e.g.*, the contents of `TextViews`/`EditTexts`) and UI property changes (*e.g.*, the *checked* property of `RadioButtons`/`CheckBoxs`), is omitted without creating new states; (3) `ListViews` are only differentiated as empty and non-empty. For example, in Figure 2a, the app pages (d) and (e) correspond to the same state, since only the contents in `EditTexts` are different.

[2]The weight parameters are tuned during our investigation on the 50 Google Play apps, but are kept unchanged for all the apps during the final evaluation.

Algorithm 1: App Stochastic Model Construction

Input: the app under test A
Output: the stochastic model M

1 **let** W be a list of events (initialized as empty)
2 **let** s be the starting page of the app
3 **let** $Tabu$ be a tabu event list (initialized as empty)
4 $W \leftarrow W \cup$ {UI events identified by static analysis}
5 **repeat**
6 **let** E be the set of invocable events inferred from s
7 $W \leftarrow W \cup E$
8 **foreach** *event* $e \in W$ **do**
9 updateWeight(e)
10 **let** e = getMaxWeightEvent($E \setminus Tabu$, s)
11 **let** s = execute(e)
  // Tabu special events that trigger unknown states
12 **if** s *is an unknown state* **then**
13 $Tabu \leftarrow Tabu \cup \{e\}$
14 restoreApp()// Restart/recover the app to the previous page
15 $M \leftarrow$ expandFSM(s, e)
16 **until** $timeout$
17 return assignProbability(M)
18 **Procedure** assignProbability(M)
19 $S \leftarrow$ getAppStates(M) // S={s_1,..., s_i,..., s_n}
  // assign the initial probability values for each transitions of s_i
20 **foreach** *state* $s_i \in S$ **do**
21 $E_i \leftarrow$ getEvents(s_i) // E_i={e_1,..., e_j,..., e_k}
22 **foreach** *transition* $e_j \in E_i$ **do**
  // p_j is the probability value of the transition e_j
23 totalTimes = getAllExecutionTimes(e_1,..., e_j,..., e_k)
24 $p_j \leftarrow$ getExecutionTimes(e_j)/totalTimes
25 **return** M

4 GUIDED STOCHASTIC MODEL MUTATION

Stoat exploits Gibbs sampling to guide the mutation of the stochastic model so that a set of representative tests can be generated. In our setting, we intend to find "good" models, from which the test suites can achieve our desired goal. We view this problem as an optimization procedure guided by our fitness function.

4.1 Gibbs Sampling

Gibbs Sampling [5, 64], is a special case of the Metropolis-Hastings algorithm [65]. The Metropolis-Hastings algorithm is one of Markov Chain Monte Carlo (MCMC) methods [14], which are a class of algorithms to draw samples from a desired probability distribution $p(x)$, for which direct sampling is difficult. It iteratively generates samples from a function λ that is *proportional* to the density of $p(x)$. The sampling process generates a Markov chain, where the selection of the current sample only depends on the previous one.

After a number of iterations, these samples can closely approximate the desired distribution $p(x)$, allowing more samples are generated from more important regions (the regions with higher densities).

During sampling, a candidate sample will be accepted or rejected with certain probability, and this probability is determined by comparing the λ values between the current and the candidate sample. Formally, in the t^{th} iteration, the candidate sample x' is generated *w.r.t.* a proposal density $q(x'|x_t)$, and its acceptance ratio is

$$AcceptRatio(x') = min(1, \frac{p(x') * q(x'|x_t)}{p(x_t) * q(x_t|x')}) \quad (2)$$

Usually, q is selected as a symmetric function. Thus Formula (2) can be simplified to

$$AcceptRatio(x') = min(1, \frac{p(x')}{p(x)}) \quad (3)$$

4.2 Objective Function

We designed an objective function favoring test suites that can achieve *high coverage* and contain *diverse event sequences*. Such test suites are expected to trigger more program states and behaviors, and thus increase the chance of detecting bugs.

Our objective function combines three metrics, namely *code coverage*, *model coverage*, and *test diversity*. Code coverage [56, 71] measures how thoroughly the app code is tested; model coverage [43] measures how completely the app model is covered, and test diversity [49] measures how diverse the event sequences are in the test suite, which is a significant complement for the coverage metrics. The objective function is formalized as:

$$f(T) = \alpha * CodeCoverage(T) + \beta * ModelCoverage(T) + \gamma * TestDiversity(T)$$

where T is the test suite generated from a stochastic model M, and α, β, γ are the weights on these metrics[3].

Here, code coverage is computed as either statement coverage for open-source apps [52], or method coverage for closed-source apps [3]. For model coverage, we use edge coverage to compute how many events are covered.

For test diversity, we designed a lightweight yet effective metric to evaluate the diversity of test cases. Let T be an N-size test suite $\{l_1, \ldots, l_i, \ldots l_N\}$, where l_i is an event sequence. A k-length event sequence l can be denoted as $e_1 \rightarrow \ldots e_i \rightarrow \ldots e_k$, where e_i is an event. The key idea is to compute the "centroid" [11] of these N sequences, and take the sum of their Euclidean distances with the "centroid" as the diversity value of T. Intuitively, the larger the distance is, the more diverse T is.

First, we use a binary vector to present each sequence. Let the model M has N_e unique events. The event e can be presented as a N-dimensional vector $\vec{e} = (\varepsilon_1, \ldots, \varepsilon_i, \ldots, \varepsilon_{N_e})$, where ε_i is 0 if the event e is the i-th event in the event list, otherwise 1 (the values are set in this way to avoid the orthogonality of two vectors). Second, let l be a n-length sequence, represented as $\vec{l}_{[n]} = e_1 \rightarrow e_2 \cdots \rightarrow e_n$. We use the function below to recursively transform l into a vector $\vec{l}$ on the basis of its events and their orders:

$$\vec{l}_{[i]} = cos\_sim(\vec{l}_{[i-1]}, \vec{l}_{[i-1]} + \vec{e}_i) \cdot (\vec{l}_{[i-1]} + \vec{e}_i)$$

where $\vec{l}_{[i]}$ is the i-length prefix of $\vec{l}_{[n]}$, *i.e.*, $e_1 \rightarrow e_2 \ldots \rightarrow e_i$, and $\vec{l}_{[1]} = \vec{e}_1$. We use the cosine similarity [63] between $\vec{l}_{[i-1]}$ and $\vec{l}_{[i-1]} + \vec{e}_i$, *i.e.*, $cos\_sim(\vec{l}_{[i-1]}, \vec{l}_{[i-1]} + \vec{e}_i)$, to encode the order relation $l_{[i-1]} \rightarrow e_i$ into the vector $\vec{l}$. By this way, we can take both the contained events and their orders into consideration when computing test diversity. After we obtain the vector set for $T = \{\vec{l}_1, \ldots, \vec{l}_i, \ldots, \vec{l}_N\}$, the centroid $\vec{C}$ of T can be computed as $\vec{C} = \frac{\sum_{i=1}^{N} \vec{l}_i}{N}$. Last, the test diversity is computed by the formula:

$$TestDiversity(T) = \frac{\sum_{i=1}^{N} d(\vec{l}_i, \vec{C})}{N}$$

To fit into the objective function, we scale the test diversity into the range of [0, 1] by dividing N_e.

4.3 Gibbs Sampling Guided Model Mutation

In our problem, we choose Gibbs sampling instead of the standard Metropolis-Hastings algorithm because it is specially designed to draw samples when $p(x)$ is a joint distribution of multiple random variables. In particular, we let all transition probabilities be random variables, and draw samples by iteratively mutating them. It allows samples to be drawn more often from the region with "good" stochastic models. We hypothesize that tests derived from the optimized model can achieve higher objective values. For this reason, we set the target probability density function $p(x)$, by following a common method [25, 53], as

$$p(M) = \frac{1}{Z} exp(-\beta * f(T_M))$$

where M is the stochastic model, and Z a normalizing partition function, β a constant, T_M the test suite generated from the current model M, and f the objective function. According to Formula (3), the acceptance ratio[4] of the newly proposed model M' can be reduced to

$$AcceptRatio(M \rightarrow M') = min(1, exp(-\beta * (f(T_M) - f(T'_{M'}))))$$

Stochastic Model Mutation. Algorithm 2 gives the algorithm of Gibbs sampling guided testing. The search space is the domain of stochastic models, in which each sample is one stochastic model. At each iteration, a new candidate model M' is generated by mutating the transitions' probability values in the current model M.

Let the app have n app states, $s_1, \ldots, s_i, \ldots, s_n$, and each state s_i ($1 \leq i \leq n$) have k event transitions, $e_1, \ldots, e_j, \ldots, e_k$. Stoat randomly decides whether to mutate the transition probabilities or not of each app state s_i (lines 3-9). If the state s_i is selected, Stoat will randomly mutate the original probability value p_j of the transition e_j to a new probability value p'_j, which is the result of p_j+*mSize* or p_j-*mSize*. The intuition is that the newly generated probability value p'_j is around p_j (it can be higher or lower than p_j), so that the new model M' can generate very different event sequences compared with M. To speed up the convergence, *mSize* is set as a fixed value (*e.g.*, 0.1) in implementation. For the remaining transition probabilities, a similar procedure is applied, but the constraint $p'_1 + \ldots + p'_j + \ldots + p'_k = 1$ still holds. For the other unselected states, their transition probabilities are kept unchanged so that the new

[3]The values of α, β, and γ are respectively set to 0.4, 0.2, and 0.4 in the evaluation, which give more weights on code coverage and test diversity without any tuning.

[4]β is empirically selected as -0.33 in our problem, which is tuned to effectively differentiate acceptance ratio.

Algorithm 2: Gibbs sampling Guided GUI Testing

Input: the app under test A, its stochastic model M, and the related system-level events set K_s
Input: the maximum iteration of Gibbs Sampling I_{max}

```
1  repeat
2      S ← getAppStates(M) // S={s_1,..., s_i,..., s_n}
       // M is randomly mutated to M'
3      foreach state s_i ∈ S do
4          if Rand(0,1) > 0.5 then
5              E_i ← getEvents(s_i) // E_i={e_1,..., e_j,..., e_k}
6              foreach Transition e_j ∈ E_i do
7                  p_j ← getProbability(e_j)
8                  p'_j ← randomlyMutate(p_j, mSize)
9                  p_j ← p'_j // p'_j ∈ (0,1) and p'_1 + ... + p'_j + ... + p'_k=1
10     T' ← generateTestSuite(M') // T'={t'_1,..., t'_i,..., t'_n}
       // T' is randomly injected with system-level events
11     foreach event sequence t'_i ∈ T' do
12         if Rand(0,1) > 0.5 then
13             i ← getRandomEventIndex(t'_i)
14             e_s ← selectOneSystemEvent(K_s)
15             injectEvent(t'_i, e_s, i)// insert e_s into the event position i
16     execute(A, T')
17     if A crashes or is non-responding then
18         record the error stack
19     if AcceptRatio(M, M') > Rand(0,1) then
20         M ← M'
21 until I_max is reached OR timeout
```

model M' conditionally depends on the previous model M by those mutated probability values.

To simulate the interactions of environment, Stoat randomly injects system-level events into the generated tests T' from the mutated model M' (lines 11-15). Then T' is replayed on the app to validate its behaviors. The test results of T' are used to determine the acceptance ratio of M'. If T' can improve the objective value, M' will be mutated in the next iteration (lines 19-20). Otherwise, the original M is mutated. The algorithm continues until the testing budget is exhausted. If the app crashes or becomes non-responding, a suspicious bug is recorded (lines 17-18), and the corresponding error stack is dumped for bug diagnosis.

4.4 System-level Events

To incorporate system-level events into mode-based testing, Stoat adopts a simple yet effective strategy by randomly injecting them into UI-level event sequences. This strategy avoids the complexity of including system-level events into the behavior model, and further interleaves both types of events to detect intricate bugs.

Currently, Stoat supports three sources of system-level events: (1) 5 user actions (*i.e.*, screen rotation, volume control, phone calls, SMSs, app switch); (2) 113 system-wide broadcast intents (*e.g.*, battery level change, connection to network) to simulate system messages; (3) the events that the apps are particularly interested in, which are usually declared by the tags `<intent-filter>` and `<service>` in their `AndroidManifest.xml` files.

5 EVALUATION

The evaluation aims to answer the four research questions:

RQ1. ***Model Construction***. Compared with the existing model construction tools for GUI testing, how effective is Stoat?
RQ2. ***Code Coverage***. Compared with the state-of-the-art testing tools, how is the coverage achieved by Stoat?
RQ3. ***Fault Detection***. Compared with the state-of-the-art testing tools, how is the fault detection ability of Stoat?
RQ4. ***Usability and Effectiveness***. How is the usability and effectiveness of Stoat in testing real-world apps?

5.1 Tool Implementation

Stoat is implemented as a fully automated app testing framework, which reuses and extends several tools: Android `UI Automator` [27, 33] and Android Debug Bridge (`ADB`) for automating test execution; Soot [22] and Dexpler [8] for static analysis to identify potential input events; Androguard [58] for analyzing the system-level events that the apps are particularly interested in. Stoat currently supports `click`, `touch`, `edit` (generate random texts of numbers or letters), `navigation` (*e.g.*, `back`, `scroll`, `menu`). During Gibbs sampling, Stoat generates a test suite with the maximum size of 30 tests and each with a maximum length of 20 events at each sampling iteration. Stoat instruments open-source apps by Emma [52] to get line coverage; and instruments closed-source apps by Ella [3] to get method coverage. To improve scalability, Stoat is designed as a server-client mode, where the server can parallelly control multiple Android devices. Stoat is online available at [24].

5.2 Evaluation Setup

Environment. Stoat runs on a 64-bit Ubuntu 14.04 physical machine with 12 cores (3.50GHz Intel Xeon(R) CPU) and 32GB RAM, and uses Android emulators to run tests. Each emulator is configured with 2GB RAM and X86 ABI image (KVM powered), and the KitKat version (SDK 4.4.2, API level 19). Different types of external files (including 5 JPGs/3 MP3s/3 MP4s/10 VCFs/3 PDFs/3 TXTs/3 ZIPs) are stored in the SDCard to facilitate file access from apps.
Subjects. We conducted three case studies. In ***Study 1*** and ***2***, to set up a fair comparison basis, we chose 68 benchmark apps, which have been widely used in previous research work [7, 15, 16, 39–41, 45, 67]. These apps come from F-droid [30], a popular open-source app repository. To further reduce the potential bias, we enriched them by randomly selecting 25 new apps from F-droid. So we totally evaluated on 93 apps. In ***Study 3***, Stoat is applied to test 1661 most popular apps from Google Play of various categories.

In ***Study 1***, we answer ***RQ1*** by comparing Stoat with MobiGUITAR [2] and PUMA [32]. Both tools produce similar FSM models. MobiGUITAR implements a *systematic* and a *random* exploration strategies for constructing models: The former visits widgets in a breadth-first order, and restarts the app when no widgets can be found, and the latter randomly emits UI events. PUMA uses UIAutomator to *sequentially* explore GUIs, and stops exploring when all app states have been visited. Stoat is not compared with other model-based tools because they are either unavailable (*e.g.*, ORBIT [67] and AMOLA [31]) or crash frequently (*e.g.*, Swifthand [15]).

We run each tool on one emulator, and test each app for 1 hour, and measure the *code coverage* to approximate the completeness of

the constructed models, which is the basis of model-based testing. We also record the number of states and edges in the models to measure the complexity. Intuitively, the higher the code coverage, the more compact the model is, the more effective the tool is.

In ***Study 2***, we answer ***RQ2*** and ***RQ3*** by comparing Stoat with these tools: (1) Monkey (random fuzzing), (2) A^{3}E [6] (systematic UI exploration), and (3) Sapienz (genetic algorithm) [41]. Moneky, A^{3}E, and Sapienz are the state-of-the-art GUI testing tools. They have the best performance in their own approach categories [16]. Specifically, Monkey emits a stream of random input events, including both UI and system-level events, to maximize code coverage. A^{3}E systematically explores app pages and emits events by a depth-first strategy, which is also widely adopted in other GUI testing tools [1, 42, 67]. Sapienz uses Monkey to generate the initial test population, and adapts genetic algorithms to optimize the tests to maximize code coverage while minimizing test lengths.

We allocate 3 hours for each tool to thoroughly test each app on one single emulator. Stoat allocates 1 hour for model construction and 2 hours for Gibbs sampling. We record *code coverage* and the number of *unique crashes*. To eliminate randomness, we run each app for five times, and take the average values as the final results. During testing, we identify crashes by monitoring `Logcat` [26] messages. Note each unique crash has a unique error stack; unrelated crashes (*e.g.*, errors from Android system, test harness, and caught exceptions [47]) are excluded. If a tool covers more code and detects more unique crashes, it is more effective.

In ***Study 3***, we answer ***RQ4*** by running Stoat on 1661 most popular apps from Google Play. Stoat run each app for three hours with the same configuration in Study 2. Stoat instruments these apps at the method level to collect code coverage for Gibbs sampling.

5.3 Study 1: Model Construction

Model Completeness Figure 3(a) shows the achieved line coverage *w.r.t.* the models constructed by PUMA (denoted by "PU"), MobiGUITAR-systematic ("M-S"), MobiGUITAR-random ("M-R"), and Stoat ("St") on the 93 subjects (listed in the first column of Table 1). On average, Stoat covers 31% and 17% more code, respectively, than M-S and M-R, and 23% more than PUMA. It indicates that Stoat can cover more app behaviors, and produce more complete models. MobiGUITAR cannot exhaustively explore app behaviors due to its simple exploration strategies. For example, the systematic strategy is surprisingly much less effective than the random strategy, since it visits UIs in a fixed order (breadth-first) and wastes much time on restarting the app when no new UI widgets are found. PUMA continues the exploration until all different app states have been visited, which can save the exploration efforts but may also miss new UI pages due to its abstraction of states is too coarse.

Model Complexity Figures 3(b) and 3(c) show the size of the models in terms of the number of states and transitions (Note the Y-axis uses a logarithmic scale). We can see Stoat achieves much higher code coverage than the other tools (indicated by Figure 3(a)), but its models are more compact without states explosion (Figure 3(b)). In addition, Stoat captures more app behaviors/events (one transition denotes one event in Figure 3(c)) than the other tools, which indicates its models are more complete. In detail, MobiGUITAR determines the equivalence of app states on the basis of the properties (ids and types) of their constitutive UI objects, while PUMA differentiates states according to their UI features (*e.g.*, the number of invocable events). However, these criteria are too coarse to construct representative models. In contrast, Stoat uses the UI layout structures to decide the state similarity and merges states with neglectable differences. Therefore, the models constructed by Stoat would be more effective for Gibbs sampling.

Table 1: Testing results on 93 open-source apps.

Subject Name	ELOC	Coverage (%) A	M	Sa	St	Crashes A	M	Sa	St
a2dp	3576	14	42	39	49	0	0	1	4
aarddict	2200	11	69	13	66	0	0	1	3
aLogCat	846	36	71	72	80	0	0	0	0
Amazed	253	60	77	78	87	0	2	1	0
AnyCut	348	2	67	70	83	0	1	0	1
baterrydog	466	4	72	71	66	0	0	0	0
swiftp	2160	15	13	13	18	0	0	0	1
Book-Catalogue	9847	3	43	25	23	0	1	0	4
bites	1027	3	39	35	57	0	1	1	4
battery	251	51	74	91	93	0	6	4	2
addi	20019	16	17	19	17	0	3	1	3
alarmclock	2453	14	74	71	77	0	5	4	3
manpages	301	44	44	82	75	0	0	1	3
mileage	4699	2	48	48	44	0	6	4	13
autoanswer	387	6	8	9	25	0	0	0	2
hndroid	968	6	2	11	10	1	1	1	1
multismssender	792	13	44	61	76	0	1	0	2
worldclock	1156	83	93	95	98	0	0	1	2
Nectroid	2459	24	36	76	71	0	0	0	3
acal	17453	6	22	29	26	0	3	2	5
jamendo	4398	12	62	55	78	0	2	1	6
aka	1249	15	81	82	82	0	1	5	1
yahtzee	504	3	64	52	71	1	1	0	2
aagtl	11747	9	26	29	35	0	3	2	3
CounterdownTimer	584	40	64	64	86	0	0	0	0
sanity	4935	4	34	19	39	0	1	1	1
dalvik-explorer	1283	23	71	74	75	1	1	2	6
Mirrored	825	2	11	33	50	0	3	1	5
dialer2	897	28	39	42	82	0	0	0	4
DivideAndConquer	768	43	91	88	92	0	1	1	0
fileexplorer	35	62	60	60	61	0	0	0	0
gestures	33	30	46	52	48	0	0	0	0
hotdeath	3890	2	80	57	70	0	1	0	0
adsdroid	153	7	26	38	28	1	1	1	2
myLock	791	5	28	29	46	0	0	0	3
lockpatterngenerator	617	57	88	83	78	0	0	0	0
mnv	3715	5	37	49	56	0	2	1	4
aGrep	862	7	55	-	54	0	3	0	2
k9mail	22823	3	6	7	8	0	0	0	15
LolcatBuilder	578	7	14	18	24	0	0	0	0
MunchLife	163	45	93	87	85	0	0	0	0
MyExpenses	2984	12	53	51	63	0	1	0	3
LNM	399	18	63	62	64	0	0	0	5
netcounter	2370	23	44	68	79	0	1	0	4
bomber	283	76	78	79	78	0	0	0	1
frozenbubble	1643	28	70	-	72	0	0	0	0
fantastichmemo	8886	5	16	42	48	0	1	4	20
blokish	1164	34	46	52	58	0	1	1	2
zooborns	759	16	35	34	36	0	0	0	3
importcontacts	1115	2	81	42	79	0	0	0	1
wikipedia	719	18	36	27	31	0	0	4	0
PasswordMaker	1469	29	61	49	74	1	4	3	13
passwordmanager	10791	4	3	7	7	0	0	0	0
Photostream	1307	6	23	29	28	1	1	1	2
QuickSettings	2883	20	56	51	41	0	0	1	2
RandomMusicPlayer	318	5	61	59	88	0	0	0	2
Ringdroid	2973	-	20	60	-	0	1	5	1
soundboard	18	96	90	54	100	0	0	0	0
SpriteMethodTest	948	72	78	76	87	0	0	0	0
SpriteText	1166	53	61	63	59	0	0	0	0
SyncMyPix	4072	4	21	20	27	0	0	1	2
tippy	995	47	88	86	89	0	0	0	0
tomdroid	1484	1	48	57	58	0	0	2	1
Translate	711	28	49	50	49	0	0	0	0
Triangle	284	56	82	65	75	0	0	0	0
weight-chart	1054	24	74	80	81	0	1	2	3
whohasmystuff	640	43	80	79	84	0	0	0	4
Wordpress	10526	1	5	6	8	0	0	2	13
BabyCareTimer	3048	16	36	45	55	0	1	0	4
Yaab	1921	8	47	48	47	1	1	1	1
campyre	1462	9	6	13	15	0	2	2	5
URLazy	185	25	19	20	26	0	0	0	1
arXiv	2093	9	45	13	62	0	2	2	5
h2droid	917	41	78	84	45	0	0	0	2
Cetoolbox	1118	26	56	53	91	0	0	0	1
CurrencyConverter	774	41	77	66	81	0	0	0	0
charmp	117	60	88	86	97	0	0	0	1
NanoConverter	1199	11	57	43	54	0	1	0	0
anarxiv	1140	22	51	62	63	0	1	0	4
kindmind	1730	28	59	53	57	0	1	1	7
URforms	2560	40	58	75	82	0	2	3	2
Homemanager	1329	21	54	54	57	0	2	2	2
PocketTalk	444	29	33	33	92	0	0	0	0
Rot13	100	64	95	-	96	0	0	0	0
Angulo	627	48	52	59	77	1	0	0	0
RightAlert	238	71	85	91	93	0	1	2	2
AppTrack	1116	38	76	70	84	0	1	6	2
TextEdit	1387	11	63	56	62	0	0	0	1
Diary	195	36	94	92	95	0	1	2	1
Rtltcp	669	6	34	31	42	0	1	2	2
fakedawn	1300	34	58	57	64	0	0	0	7
klaxon	814	11	42	41	69	0	0	0	5
Imcktg	639	42	84	73	90	0	0	1	2

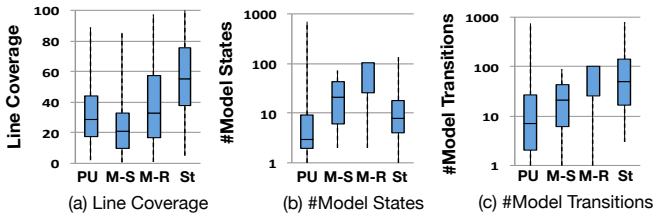


Figure 3: Results of model construction.

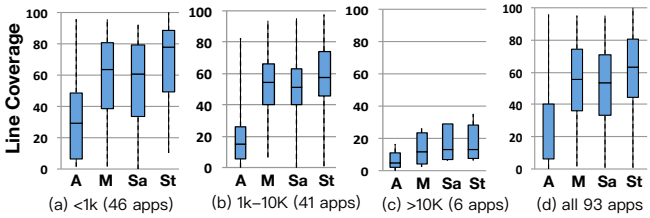


Figure 4: Results of code coverage grouped by app sizes.

5.4 Study 2: Testing Effectiveness

Code Coverage Table 1 lists 93 subjects and their executable lines of code (ELOC), and shows the testing results of A^{3}E ("A"), Monkey ("M"), Sapienz ("Sa") and Stoat ("St") in terms of line coverage and the number of unique crashes (best results are highlighted). On average, they achieve 25%, 52%, 51%, and 60% line coverage, respectively. In particular, Stoat achieves nearly 35% higher coverage than A^{3}E. Figure 4 shows the line coverage of these tools grouped by app sizes. It is clear that Stoat has the best performance.

A^{3}E achieves much lower coverage than the other three tools for two main reasons. First, A^{3}E explores UIs in a depth-first order. Although this greedy strategy can reach deep UI pages at the beginning, it may get stuck because the order of event execution is fixed at runtime. Second, A^{3}E does not explicitly revisit previously explored UIs, and thus may fail in covering new code that should be reached by different sequences. We also note Monkey's coverage is close to Sapienz's when given enough testing time (3 hours).

Unique Crashes Table 2 summarizes the statistics of the four tools in detecting app crashes. Stoat has detected 249 unique crashes from 68 buggy apps, which is much more effective than A^{3}E (8 crashes), Monkey (76 crashes), and Sapienz (87 crashes). We also find Stoat has detected all the crashes that were found by A^{3}E. Fig. 5 gives the pairwise comparison of crashes detected by Monkey, Sapienz, and Stoat. We can see the crashes detected by Stoat have much less overlap with Monkey and Sapienz. In detail, Stoat detected exclusive 227 and 224 crashes than Monkey and Sapienz, respectively. The crashes detected by Monkey and Sapienz are close in number, and they have more overlap (33 bugs are detected by both). The fact that Sapienz uses Monkey to generate the initial population of event sequences may explain this phenomenon.

Method of Calculating Unique Crashes Stoat identifies unique crashes in an accurate way: (1) remove all unrelated exceptions without the keyword of the app's package name; (2) extract the exception lines from the crash stack of the app; (3) use these lines to identify unique crashes. Different crashes should have different sequences of exception lines. However, we find Sapienz simply uses text differences to count unique crashes, which is inaccurate and thus may bring false positives. To set up a fair comparison basis, we modified Sapienz's scripts to follow our method.

Table 2: Testing statistics of A^{3}E, Monkey, Sapienz and Stoat.

Tool	#Buggy Apps	#Unique Crashes
A^{3}E	8	8
Monkey	40	76
Sapienz	43	87
Stoat	68	249

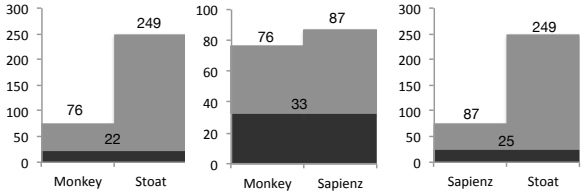


Figure 5: Pairwise comparison of tools in detecting crashes.

5.5 Bug Analysis

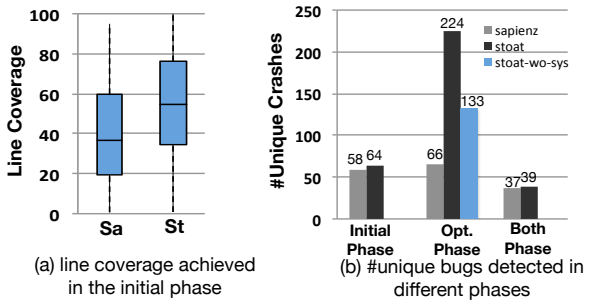
To further investigate the effectiveness of Stoat, we analyzed several typical crashes that were found by Stoat but missed by Monkey and Sapienz. We summarized the following key findings.

Finding 1: Stoat is more effective in UI exploration. Both Stoat and Sapienz are two-phase testing techniques. Sapienz uses Monkey to generate the initial population of event sequences (including *both UI and system-level events*) before genetic optimization, while Stoat constructs app models (*only by UI events*) before Gibbs sampling. In Figure 6(a), we show the coverage achieved by Sapienz (denoted by "Sa") and Stoat ("St") on the 93 subjects in their respective initial phases (*i.e.*, the *population generation* phase and the *model construction* phase). By default setting, Sapienz and Stoat on average take 56 and 60 minutes to finish the initial phase, and require 45 and 23 minutes to reach peak coverage, respectively. We can see that Stoat achieves higher coverage than Sapienz, which enables Stoat to detect more crashes in the optimization phase.

For example, Stoat detects a `CursorIndexOutOfBoundsException` in the app *Bites* [20] (Figure 2a) during model construction. This crash can be revealed by a long event sequence: create a recipe (fill in names, authors, and descriptions), long touch on it, and then select the option of "send by SMS" from the other fours. However, Sapienz has never reached this usage scenario in the initial phase due to its randomness. By utilizing this captured behavior, Stoat further detects a new crash during Gibbs sampling, which can only be revealed when the user fills the ingredients of this recipe but leaves its cooking methods empty, and sends it by SMS. However, Sapienz has never detected this new crash during optimization.

Finding 2: Stoat is more effective in detecting deep crashes. Both Stoat and Sapienz use optimization techniques to guide test generation. However, Sapienz generates new tests by randomly crossovering and mutating sequences. It may produce many "infeasible" ones, and is less likely to reach deep code. In contrast, Stoat guides test generation from an app's behavior model (captures all possible compositions of events), which is more likely to generate meaningful and diverse sequences to reveal deep bugs.

For example, *TextEdit* [59] is a text edit app. Stoat exposes a `NullPointerException` by following a 7-length event sequence.



(a) line coverage achieved in the initial phase

(b) #unique bugs detected in different phases

Figure 6: Comparison between Sapienz and Stoat

The exception is thrown when a non-existing file is accessed after the default file name prefix "/sdcard/" is removed. The code snippet below shows when the user tries to access a non-existing file, the app will remind that the file cannot be found (the case `DIALOG_NOTFOUND_ERROR` at Lines 9-10), and reopen the previous dialog to accept new file names (the case `DIALOG_OPEN_FILE` at Lines 2-7). However, the variable `errorFname` stores the previous non-existing file name, *e.g.*, "test". Thus the app will call `getParent()` at Line 7 and return `null` because the file does not exist. The next call to `toString` crashes the app.

```
1  /* the dialog for opening files */
2  case DIALOG_OPEN_FILE:
3    if (openingError){
4      File f = new File(errorFname.toString());
5      if (f.toString().equals("/")) ...
6      else if (f.isDirectory()) ...
7      else if ( f.getParent().toString() .equals("/")) ...} ...
8  /* the dialog for file not found */
9  case DIALOG_NOTFOUND_ERROR:{ ...
10     showDialog(DIALOG_OPEN_FILE);}...
```

As Figure 6(b) shows, Stoat detects many more crashes than Sapienz in the optimization phase (224 *vs.* 66). The numbers of crashes in their initial phases are close, but Sapienz generates both UI and system-level events while Stoat only generates UI events.

Finding 3: System events can reveal more unexpected crashes. During Gibbs sampling, Stoat randomly injects system-level events into UI-level event sequences, to enhance MBT. As Figure 6(b) shows, by this enhancement, Stoat can detect additional 91 crashes (see the "stoat" and "stoat-wo-sys" columns in the optimization phase). For example, the app *mileage* [21] was crashed by `IllegalArgumentException` when Stoat launches its chart activities and sends them empty intents. The app directly takes the `null` values to make database queries without any sanitization.

From the above analysis, we can see Stoat is more effective than the other tools in bug detection. The models help Stoat generate more meaningful event sequences, and the tests are effectively guided to reach different corner cases. However, Monkey/Sapienz can also detect some crashes that Stoat cannot find. We summarized two main reasons: (1) Monkey supports irregular actions, *e.g.*, `PinchZoom`, `flip`, which have not been included in our app models; (2) Monkey can reveal some stress-testing bugs (it continuously emits events without waiting the previous ones take effect), *e.g.* some concurrency crashes [9] (`IllegalStateExceptions` triggered by the synchronizations between `ListViews` and their data adapters), some `IllegalArgumentExceptions` triggered by the mismatches of service binding/unbinding due to quick switches of activity lifecycle callbacks, and some `OutOfMemoryErrors`.

Table 3: Distribution of the detected crashes by Stoat in Google Play apps.

ID	Exception Type	Number
1	`NullPointerException`	1226
2	Windows Leaked Exception	255
3	`ActivityNotFoundException`	191
4	SQLite Related Exception	71
5	`IllegalStateException`	47
6	`IllegalArgumentException`	37
7	`RuntimeException`	21
8	`ClassCastException`	9
9	`UnsatisfiedLinkError`	8
10	`WindowManager$BadTokenException`	4
11	Other Exceptions	233

5.6 Study 3: Usability on Real-world Apps

To further validate the usability of Stoat, we apply it on the most popular apps from Google Play. Stoat was run on 3 physical machines with 18 emulators and 6 phones (allocate 3 hours per app). In one month, it successfully tested 1661 apps, and detected 2110 unique unknown crashes from 691 apps: 452 crashes from model construction, 1927 crashes from Gibbs sampling, and 269 crashes are detected in both phases. We have sent all the bug reports to the developers. So far, 43 developers have replied that they are investigating our reports (excluding auto-replies). 20 of our reported crashes have been confirmed, and 8 have already been fixed.

Table 4 shows the parts of bugs found by Stoat, where we list the app names, the categories, the installations, the crash types, the brief descriptions of root causes, and their statuses (confirmed or fixed). During the evaluation, we totally found 23 different types of crashes. Table 3 shows the distribution of their numbers. We can see `NullPointerException` is the most common type of exceptions, which aligns to previous case studies [39, 41].

5.7 Limitations and Threats to Validity

Stoat has some limitations. First, during testing, Stoat emits an event, waits until it takes effect, and then emits the next one. This synchronization ensures test integrity, but it may miss those bugs that can only be manifested by swift actions. Second, Stoat may generate "infeasible" event sequences from models. To mitigate this problem, Stoat locates UI widgets by object indexes instead of some volatile properties (*e.g.*, texts), and skips events when the target UI cannot be located. Third, the models produced by Stoat are still not complete since it cannot capture all possible behaviors during UI exploration, which is still an important research goal on GUI testing [16]. For example, Stoat is ineffective on the apps with irregular gestures (*e.g.*, `PinchZoom`, `Drawing`) and specific input data formats. Future work may integrate symbolic execution, string analysis or learning algorithm [38] to tackle such issues.

We mitigate threats to validity in two aspects: (1) eliminate false positives by excluding irrelevant crashes and collecting unique ones, manually inspecting all crashes from open-source apps, and refining crash reporting via developer feedback on the submitted crash reports. (2) apply each testing tool on each app multiple times to mitigate algorithm randomness (Future work may adopt statistical analysis to further strengthen the results).

Table 4: Parts of Bugs found by Stoat and confirmed as real faults.

ID	App Name	Category	Installation	Crash Exception	Description	Status
1	P*	News	10M-50M	NullPointerException	Unable to destroy the `PremiumSettingsActivity` activity	Confirmed
2	M*	Wallpapers	1M-5M	InstantiationException	Fail to instantiate the `CropImageView` activity	Fixed
3	PI*	Pictures	1M-5M	SQLiteCantOpenDatabaseException	Fail to open database files when an activity is launched	Confirmed
4	A*	Photography	50M-100M	StaleDataException	Attempted to access a database cursor after it has been closed	Fixed
5	N*	Email	1M-5M	NullPointerException	Unable to start the `TimeChangeReceiver` receiver	Fixed
6	Ni*	Navigator	5K-10K	NetworkOnMainThreadException	Attempted to perform a networking operation on the main application thread	Confirmed
7	Z*	Utility Tool	10M-50M	ServiceConnectionLeaked	Forget to call unbind to release the service resource	Fixed
8	I*	Browser	1M-5M	NullPointerException	Unable to start the `OrbotActivity` Activity	Confirmed
9	C*	Pictures	10M-50M	ActivityNotFoundException	No Activity found to handle the specified Intent in the activity of `ImageSelectActivity`	Fixed
10	A*	Alarm Clock	5M-10M	WindowManager$BadTokenException	Unable to add window when the `CheckShareService` notifies users from background	Fixed
11	Re*	Life Style	10M-50M	IndexOutOfBoundsException	Access invalid index -1 in the `bottomsheet` list	Fixed
12	PI*	Video	1M-5M	NullPointerException	An error occured while executing `doInBackground()` in `VideosSearchActivityTask`	Confirmed
13	He*	Health	1M-5M	NullPointerException	Unable to start the activity of `SetWeightGoalSuccessActivity`	Fixed
14	PT*	Game	50M-100M	ActivityNotFoundException	Unable to find the explicit activity class `MraidActivity`	Confirmed
15	T*	Utility Tool	10M-50M	ClassCastException	`android.text.SpannableString` cannot be cast to `java.lang.String`	Confirmed

6 RELATED WORK

Model-based GUI testing. Model-based testing (MBT) [23, 54] is a widely used testing approach. One important task is to extract a suitable, abstract (behavior) model for the system under test [17]. However, in GUI testing, manually constructing models is time-consuming and error-prone [36, 57]. Extensive research has created several tools to automate this process. Android-GUITAR [19] uses *event flow graph* [42], which only consists of events. This graph usually generates many infeasible event sequences, and reduces the effectiveness of MBT. AndroidRipper [1], MobiGUITAR [2] (an extension of the former), ORBIT [67] and AMOLA [7] use state machines to represent app models. However, they achieve simple UI exploration (*e.g.*, depth/breadth-first), and thus their performance is limited. SwiftHand [15] uses machine learning techniques to dynamically learn models for apps, but its aim is to improve the exploration strategy and reduce app restarts. MonkeyLab [61] records the execution traces from app users to mine statistical language models, but aims to generate replayable event sequences.

Another important activity in MBT is to generate tests from models. Traditional approaches employ graph traversal algorithms to generate tests, and then fulfill various coverage metrics [43]. Amalfitano *et al.* [2] randomly generate tests from models to satisfy pairwise coverage for apps. Nguyen *et al.* [48] combine model-based testing and combinatorial testing, and enhance the tests with domain input specifications. Brooks *et al.* [10] use probabilistic FSM models populated by software usage profiles [51, 62] to do regression testing of desktop applications. Hierons *et al.* [34] use an extended stochastic model to describe non-deterministic systems, and generate tests from the mutated models to check the conformance between system specifications and their implementations. Compared with these approaches, Stoat uses stochastic FSM models populated by execution profiles to generate tests. The tests are iteratively optimized to detect app bugs with the feedback from test execution. Stoat further enhances MBT by injecting system-level events, which has not been considered by previous work.

Other approaches also exist for app testing. Symbolic execution [4, 36, 46] exhaustively explores program paths to test apps. Dynodroid [39] enforces random testing enhanced with UI exploration heuristics to achieve GUI testing. AppDoctor [35] randomly invokes event handlers in the code, rather than faithfully emitting events on the app screen, to test the robustness of apps. EvoDroid [40] uses evolutionary algorithms to generate high coverage GUI tests. TrimDroid [45] optimizes combinatorial testing for app testing with smaller but effective test suites.

MCMC sampling-driven testing. Markov Chain Monte Carlo (MCMC) sampling techniques have been used for several software testing problems [13, 37, 68–70]. Zhou *et al.* [68] propose a Markov Chain Monte Carlo Random Testing (MCMCRT) approach to enhance traditional random testing. It utilizes the Bayes approach to parametric models for testing, and uses the prior knowledge and previous testing results to estimate parameters. This technique can also improve the performance of random testing [70] and prioritize test case selection [69]. Chen and Su [13] introduce mucert, an approach that adopts MCMC sampling to optimize test certificates for testing certificate validation in SSL/TLS implementations. The test suite is generated and mutated to achieve higher coverage and reveal more discrepancies. MCMC sampling is also used to guide fuzz testing of JVMs' startup process [12], where mutators are selected on the basis of prior knowledge. Le *et al.* [37] adapt MCMC sampling to generate diverse program variants for finding deep compiler bugs. Compared with these MCMC-based testing approaches, Stoat advocates the novel, effective idea of mutating the app model so that tests derived from the model are diverse and lead to high code coverage.

7 CONCLUSION

We have introduced Stoat, a novel, automated model-based testing approach to improving GUI testing. Stoat leverages the behavior models of apps to iteratively refine test generation toward high coverage as well as diverse event sequences. Our evaluation results on large sets of apps show that Stoat is more effective than state-of-the-art techniques. We believe that Stoat's high-level approach is general and can be fruitfully applied in other testing domains.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. Ting Su is partially supported by NSFC Grants 61572197 and 61632005, Geguang Pu by MOST NKTSP Project 2015BAG19B02 and STCSM Project No.16DZ1100600, Yuting Chen by NSFC Grant 61572312, Ke Wu by Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (ZF1213), and Zhendong Su by the United States NSF Grants 1319187, 1528133, and 1618158, and a Google Faculty Research Award. This work is also partially supported by the NTU Research Grant M4061759.020.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. 258–261.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59. DOI: http://dx.doi.org/10.1109/MS.2014.55

[3] Saswat Anand. 2017. ELLA. (2017). Retrieved 2017-2-18 from https://github.com/saswatanand/ella

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. 59.

[5] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. 2003. An Introduction to MCMC for Machine Learning. *Machine Learning* 50, 1 (2003), 5–43.

[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 641–660.

[7] Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 238–249.

[8] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2012. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*.

[9] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2015. Scalable race detection for Android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 332–348.

[10] Penelope A. Brooks and Atif M. Memon. 2007. Automated GUI testing guided by usage profiles. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 333–342.

[11] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *36th International Conference on Software Engineering, ICSE*. 175–186.

[12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[13] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 793–804.

[14] Siddhartha Chib and Edward Greenberg. 1995. Understanding the Metropolis-Hastings Algorithm. (1995).

[15] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 623–640.

[16] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 429–440. DOI: http://dx.doi.org/10.1109/ASE.2015.89

[17] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-based Testing in Practice. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 285–294.

[18] Guilherme de Cleva Farto and Andre Takeshi Endo. 2015. Evaluating the model-based testing approach in the context of mobile applications. *Electronic notes in Theoretical computer science* 314 (2015), 3–21.

[19] Android GUITAR Developers. 2017. Android GUITAR. (2017). Retrieved 2017-2-18 from http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR

[20] Bites Developers. 2017. Bites. (2017). Retrieved 2017-2-18 from https://code.google.com/archive/p/bites-android/

[21] Mileage Developers. 2017. Mileage. (2017). Retrieved 2017-2-18 from https://github.com/evancharlton/android-mileage

[22] Soot Developers. 2017. Soot. (2017). Retrieved 2017-2-18 from https://github.com/Sable/soot

[23] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM, 31–36.

[24] Ting Su et al. 2017. Stoat. (2017). Retrieved 2017-2-18 from https://tingsu.github.io/files/stoat.html

[25] W.R. Gilks, S. Richardson, and D. Spiegelhalter. 1995. *Markov Chain Monte Carlo in Practice*. Taylor & Francis. http://books.google.com/books?id=TRXrMWY_i2IC

[26] Google. 2017. Android Logcat. (2017). Retrieved 2017-2-18 from https://developer.android.com/studio/command-line/logcat.html

[27] Google. 2017. Android UI Automator. (2017). Retrieved 2017-2-18 from http://developer.android.com/tools/help/uiautomator/index.html

[28] Google. 2017. Monkey. (2017). Retrieved 2017-2-18 from http://developer.android.com/tools/help/monkey.html

[29] AppBrain Group. 2017. AppBrain. (2017). Retrieved 2017-2-18 from http://www.appbrain.com/stats/

[30] F-droid Group. 2017. F-Droid. (2017). Retrieved 2017-2-18 from https://f-droid.org/

[31] Vignir Gudmundsson, Mikael Lindvall, Luca Aceto, Johann Bergthorsson, and Dharmalingam Ganesan. 2016. Model-based Testing of Mobile Systems - An Empirical Study on QuizUp Android App. In *Proceedings First Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@IFM 2016, Reykjavík, Iceland, 4th June 2016*. 16–30.

[32] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 204–217. DOI: http://dx.doi.org/10.1145/2594368.2594390

[33] Xiaocong He. 2017. Python wrapper of Android UIAutomator test tool. (2017). Retrieved 2017-2-18 from https://github.com/xiaocong/uiautomator

[34] Robert M. Hierons and Mercedes G. Merayo. 2009. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software* 82, 11 (2009), 1804–1818.

[35] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. 18:1–18:15.

[36] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. 67–77.

[37] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 386–399.

[38] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic Text Input Generation for Mobile Testing. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 643–653.

[39] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 224–234.

[40] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 599–609.

[41] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 94–105.

[42] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*. 260–269.

[43] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2001. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*. 256–267.

[44] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. 2016. Mystique: Evolving Android Malware for Auditing Anti-Malware Tools. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, New York, NY, USA, 365–376.

[45] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 559–570.

[46] Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[47] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. 33–44.

[48] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. 2012. Combining model-based and combinatorial testing for effective test case generation. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 100–110.

[49] Borislav Nikolik. 2006. Test diversity. *Information & Software Technology* 48, 11 (2006), 1083–1094.

[50] Michael Pradel, Parker Schuh, George C. Necula, and Koushik Sen. 2014. EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 33–47.

[51] Stacy J. Prowell. 2005. Using Markov Chain Usage Models to Test Complex Systems. In *38th Hawaii International Conference on System Sciences (HICSS-38 2005), CD-ROM / Abstracts Proceedings, 3-6 January 2005, Big Island, HI, USA*.

[52] Vlad Roubtsov. 2017. EMMA. (2017). Retrieved 2017-2-18 from http://emma.sourceforge.net/

[53] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. 305–316.

[54] Muhammad Shafique and Yvan Labiche. 2010. A systematic review of model based testing tool support. *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04* (2010).

[55] Ting Su. 2016. FSMdroid: Guided GUI Testing of Android Apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 689–691.

[56] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1, Article 5 (March 2017), 35 pages.

[57] Tommi Takala, Mika Katara, and Julian Harty. 2011. Experiences of System-Level Model-Based GUI Testing of an Android Application. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*. 377–386.

[58] Androguard Team. 2017. Androguard. (2017). Retrieved 2017-2-18 from https://github.com/androguard/androguard

[59] TextEdit Developers. 2017. TextEdit. (2017). Retrieved 2017-2-18 from https://github.com/paulmach/Text-Edit-for-Android

[60] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5.

[61] Mario Linares Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. 111–122.

[62] James A. Whittaker and Michael G. Thomason. 1994. A Markov Chain Model for Statistical Software Testing. *IEEE Trans. Software Eng.* 20, 10 (1994), 812–824.

[63] Wikipedia. 2017. Cosine similarity. (2017). Retrieved 2017-2-18 from https://en.wikipedia.org/wiki/Cosine_similarity

[64] Wikipedia. 2017. Gibbs Sampling. (2017). Retrieved 2017-2-18 from https://en.wikipedia.org/wiki/Gibbs_sampling

[65] Wikipedia. 2017. Metropolis-Hastings algorithm. (2017). Retrieved 2017-2-18 from https://en.wikipedia.org/wiki/Metropolis-Hastings_algorithm

[66] Qing Xie and Atif M. Memon. 2006. Studying the Characteristics of a "Good" GUI Test Suite. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*. 159–168. DOI: http://dx.doi.org/10.1109/ISSRE.2006.45

[67] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 250–265.

[68] Bo Zhou, Hiroyuki Okamura, and Tadashi Dohi. 2010. Markov Chain Monte Carlo Random Testing. In *Advances in Computer Science and Information Technology, AST/UCMA/ISA/ACN 2010 Conferences, Miyazaki, Japan, June 23-25, 2010. Joint Proceedings*. 447–456.

[69] Bo Zhou, Hiroyuki Okamura, and Tadashi Dohi. 2012. Application of Markov Chain Monte Carlo Random Testing to Test Case Prioritization in Regression Testing. *IEICE Transactions* 95-D, 9 (2012), 2219–2226.

[70] Bo Zhou, Hiroyuki Okamura, and Tadashi Dohi. 2013. Enhancing Performance of Random Testing through Markov Chain Monte Carlo Methods. *IEEE Trans. Computers* 62, 1 (2013), 186–192.

[71] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427.