# TCM: Test Case Mutation to Improve Crash Detection in Android

Yavuz Koroglu(✉) and Alper Sen

Department of Computer Engineering, Bogazici University, Istanbul, Turkey
{yavuz.koroglu,alper.sen}@boun.edu.tr

**Abstract.** GUI testing of mobile applications gradually became a very important topic in the last decade with the growing mobile application market. We propose Test Case Mutation (TCM) which mutates existing test cases to produce richer test cases. These mutated test cases detect crashes that are not previously detected by existing test cases. TCM differs from the well-known Mutation Testing (MT) where mutations are inserted in the source code of an Application Under Test (AUT) to measure the quality of test cases. Whereas in TCM, we modify existing test cases and obtain new ones to increase the number of detected crashes. Android applications take the largest portion of the mobile application market. Hence, we evaluate TCM on Android by replaying mutated test cases of randomly selected 100 AUTs from F-Droid benchmarks. We show that TCM is effective at detecting new crashes in a given time budget.

## 1 Introduction

As of April 2016, there are over 2.6 billion smartphone users worldwide and this number is expected to go up [1]. There is an increasing focus on mobile application testing starting from the last decade in top testing conferences and journals [2]. Android applications have the largest share in the mobile application market, where 82.8% of all mobile applications are designed for Android [1]. Therefore, we focus on Android GUI Testing in this paper.

The main idea of TCM is to mutate existing test cases to produce richer test cases in order to increase the number of detected crashes. We first identify typical crash patterns that exist in Android applications. Then, we develop mutation operators based on these crash patterns. Typically mutation operators are applied to the source code of applications. However, in our work we apply them to test cases.

Typical crash patterns in Android are Unhandled Exceptions, External Errors, Resource Unavailability, Semantic Errors, and Network-Based Crashes [3]. We describe one case study for each crash pattern. We define six novel mutation operators (Loop-Stressing, Pause-Resume, Change Text, Toggle Contextual State, Remove Delays, and Faster Swipe) and relate them to these five crash patterns.


A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 264–280, 2018.
https://doi.org/10.1007/978-3-319-89363-1_15

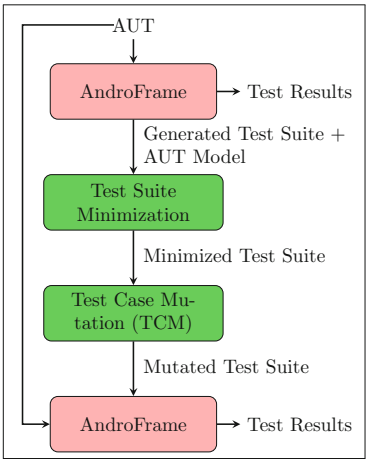



**Fig. 1.** TCM overview

We implement TCM on top of AndroFrame [4], a fully automated Android GUI testing tool. We give an overview of TCM in Fig. 1. First, we generate a test suite for the Application Under Test (AUT) using AndroFrame. AndroFrame obtains an AUT Model which is represented as an Extended Labeled Transition System (ELTS). We then minimize the Generated Test Suite using the AUT Model in order to reduce test execution costs (Test Suite Minimization). We apply Test Case Mutation (TCM) on the Minimized Test Suite and obtain a Mutated Test Suite. We use AndroFrame to execute the Mutated Test Suite and collect Test Results.

We state our contributions as follows:

1. *Test Case Mutation Operators.* We define six mutation operators on Android test cases to uncover new crashes. Our mutation operators are based on typical Android crash patterns described in the literature [3]. All of the mutation operators are novel with the exception of changing text inputs. To the best of our knowledge, ours is the first work to use mutation-based test case generation to detect different crash patterns in Android.
2. *Test Case Mutation (TCM) Algorithm.* We describe a novel algorithm to generate new test cases from existing ones to detect more crashes.
3. *Test Suite Minimization Algorithm.* We propose a coverage-based minimization algorithm to increase the effectiveness of TCM.
4. *Case Studies.* We relate known Android crash patterns to our mutation operators using case studies from F-Droid benchmarks.
5. *Experiments.* We evaluate TCM for crash detection of 100 AUTs downloaded from F-Droid benchmarks. We investigate how coverage and number of detected crashes change with respect to time.

## 2 Background

In this section, we first describe the basics of the Android GUI to facilitate the understanding of our paper.

Android GUI is based on *activities*, *events*, and *crashes*. An *activity* is a container for a set of GUI components. These GUI components can be seen on the Android screen. Each GUI component has properties that describe boundaries of the component in pixels $(x_1, y_1, x_2, y_2)$ and how the user can interact with the component (*enabled*, *clickable*, *longclickable*, *scrollable*, *password*). Each GUI component also has a *type* property from which we can understand whether the component accepts text input. A GUI component accepts text input if its *password* property is *true* or its *type* is *EditText*.

**Table 1.** List of GUI actions

| Non-contextual | Param1 | Param2 | Param3 | Param4 | Param5 |
|---|---|---|---|---|---|
| Click | x | y | - | - | - |
| Longclick | x | y | - | - | - |
| Text | x | y | string | - | - |
| Swipe | x1 | y1 | x2 | y2 | duration |
| Menu | - | - | - | - | - |
| Back | - | - | - | - | - |
| Contextual | Parameter | | | | |
| Connectivity | on/off/toggle | | | | |
| Bluetooth | on/off/toggle | | | | |
| Location | gps/gps&network/off/toggle | | | | |
| Planemode | on/off/toggle | | | | |
| Doze | on/off/toggle | | | | |
| Special | Param1 | Param2 | Param3 | Param4 | Param5 |
| Reinitialize | Package | Activity | - | - | - |

The Android system and the user can interact with GUI components using *events*. We divide events in two categories, *system events* and *GUI events (actions)*. We show the list of GUI actions that we use in Table 1, which covers more actions then are typically used in the literature. Note that GUI actions in Table 1 are possible inputs from the user whereas system events are not. We group actions into three categories; non-contextual, contextual, and special. Non-contextual actions correspond to actions that are triggered by user gestures. *Click* and *longclick* take two parameters, x and y coordinates to click on. *Text* takes three parameters, x and y for coordinates and string to describe what to write. *Swipe* takes five parameters. The first four parameters describe the starting and the ending coordinates. The fifth parameter is used to adjust the speed of *swipe*. *Menu* and *back* actions have no parameters. These actions just click to the menu and back buttons of the mobile device, respectively. Contextual actions correspond to the user changing the contextual state of the AUT. Contextual state is the concatenation of the global attributes of the mobile device (internet connectivity, bluetooth, location, planemode, sleeping). The *connectivity* action adjusts the internet connectivity of the mobile device (adjusts wifi or mobile data according to which is available for the mobile device). *Bluetooth*, *location*, and *planemode* are straightforward. The *doze* action taps the power button of the mobile device and puts the device to sleep or wakes it. We use the *doze* action to pause and resume the AUT. Our only special action is *reinitialize*, which reinstalls and starts an AUT. System events are system generated events, e.g. *battery level*, *receiving SMS*, *clock/timer*.

We report a *crash* whenever a fatal exception is recorded in Android logs similar to previous work [3,5]. Crashes often result with the AUT terminating with or without any warning. Some crashes do not visually affect the execution, but the AUT halts as a result.

We use the *Extended Labeled Transition System* (ELTS) [6] as a model for the AUT. Formally, an ELTS $M = (V, v_0, Z, \omega, \lambda)$ is a 5-tuple, where

- $V$ is a set of *states* (vertices),
- $v_0 \in V$ is the *initial state*,
- $Z$ is the set of all *actions* (input alphabet),
- $\omega : V \times V \times Z$ is the *state transition relation*, and
- $\lambda : V \rightarrow \wp(Z)$ is a *state labeling function*, where $\forall v \in V, \lambda(v) \subseteq Z$ denotes the set of actions enabled at state $v$.

We define a GUI state, or simply a *state* $v$ to be the concatenation of the (1) package name (a name representing the AUT), (2) activity name, (3) contextual state, and (4) GUI components.

Each state $v$ has a set of enabled actions $\lambda(v)$, extracted from its set of GUI components. We say that a GUI action, or simply an *action* $z \in \lambda(v)$ *is enabled* at state $v$ iff we can deduce that $z$ interacts with at least one GUI component in $v$.

A *transition* is a 3-tuple, (start-state, end-state, action), shortly denoted by $(v_s, v_e, z)$. We extend the standard transition and define a *delayed transition* as a 4-tuple, (start-state, end-state, action, delay in seconds), shortly denoted by $(v_s, v_e, z, d)$. We do this to later change the duration of transitions via mutation. We define an execution trace, or simply a *trace* $t$, as a sequence of delayed transitions. An example trace can be given as $t = (v_1, v_2, z_1, d_1), (v_2, v_3, z_2, d_2), \ldots, (v_n, v_{n+1}, z_n, d_n)$ where $n$ is the *length* of the trace.

We say that a trace $t$ is a *test case* if the first state of the trace is the initial state $v_0$ (the GUI state when the AUT is started). A *test suite* $TS$ is a set of test cases. AndroFrame generates these test suites. Then, TCM applies minimization and mutation to generate new test suites.

## 3 Android Crash Patterns and Mutation Operators

In this section, we first describe typical crash patterns for Android applications based on related work in the literature [3]. We give a list of the crash patterns in Table 2 and describe them below.

### 3.1 Android Crash Patterns

**C1. Unhandled Exceptions.** An AUT may crash due to misuse of libraries or GUI components, e.g. overuse of a third party library (stressing) may cause the third party library to crash.

**C2. External Errors.** An AUT may communicate with external applications. This communication requires either permissions or valid Inter Process Communication (IPC) for Android. There are three types of IPC in Android; intents, binders, and shared memory. Intents are used to send messages between applications. These messages are called bundles. Binders are used to invoke methods of other applications. An AUT may crash with an external error due to (1) the AUT attempts to communicate with another application without sufficient permissions, (2) the AUT receives an intent with an invalid bundle from another application, (3) the AUT sends an intent with an invalid bundle and fails to receive an answer due to a crash in the other application, (4) another application uses a binder with illegal arguments, (5) the AUT uses a binder on another application with illegal arguments and fails to receive the return value due to a crash in the other application, or (6) shared memory of the AUT is freed by another application.

**Table 2.** Relating crash patterns and mutation operators

| Crash patterns | Mutation operators |
|---|---|
| C1. Unhandled Exceptions | M1, M3, M6 |
| C2. External Errors | M1, M4, M5, M6 |
| C3. Resource Unavailability | M2, M5 |
| C4. Semantic Errors | M3 |
| C5. Network-Based Crashes | M4, M5, M6 |

**C3. Resource Unavailability.** In Android, an AUT may be paused at any time by executing an *onPause()* method. This method is very brief and does not necessarily afford enough time to perform save operations. The *onPause()* method may terminate prematurely if its operations take too much time, causing a resource unavailability problem that may crash the AUT when it is resumed. Another problem is that an AUT may use one or more system resources such as memory and sensor handlers (e.g. orientation) during execution. When the AUT is paused, it releases system resources. The AUT may crash if it is unable to allocate these resources back when it is resumed.

**C4. Semantic Errors.** An AUT may crash if it fails to handle certain inputs given by the user. For example, AUT may crash instead of generating a warning if some textbox is left empty, or contains an unexpected text.

**C5. Network-Based Crashes.** An AUT may connect with remote servers or peers via *bluetooth* or *wifi*. The AUT may crash and terminate if it does not handle the cases where the server is unreachable, the connectivity is disabled, or the communicated data causes an error in the AUT.

### 3.2 Mutation Operators

We now define the set of Android mutation operators that we developed. We denote these operators by $\Delta$. We describe these mutation operators, then relate them to the crash patterns above, and summarize these relations in Table 2.

**Definition 1.** *A* mutation operator $\delta$ *is a function which takes a test case* $t$ *and returns a new test case* $t'$*. We denote a mutation as* $t' = \delta(t)$*.*

**M1. Loop-Stressing ($\boldsymbol{\delta_{\mathrm{LS}}}$).** $t' = \delta_{\mathrm{LS}}(t)$ reexecutes all looping actions of a test case $t$ multiple times with $d'$ second delay. An action $z_i$ of a delayed transition $t_i = (v_i, v_{i+1}, z_i, d_i)$ in $t$ is *looping* iff $v_{i+1} = v_i$. Let $t_{j\ldots k}$ denote the subsequence of actions between $j^{th}$ and $k^{th}$ indices of test case $t$, inclusively. Then,

$$\delta_{\mathrm{LS}}(t) = t_1^{ls} \cdot t_2^{ls} \cdot \ldots \cdot t_n^{ls} \text{ where } t_i^{ls} = \begin{cases} t_i & v_i \neq v_{i+1} \\ \underbrace{t_i' \cdot t_i' \cdot \ldots \cdot t_i'}_{m \text{ times}} & v_i = v_{i+1} \end{cases} \tag{1}$$

Here $n$ is the length of test case $t$ and $t_i' = (v_i, v_{i+1}, z_i, d')$. We pick $d' = 1$ to avoid double-click, which may be programmed as a separate action than single click. We pick $m = 9$. We have two motivations for choosing $m = 9$. First, in our case studies, we did not encounter a crash when $m < 9$. Second, although we detect the same crash when $m > 9$, we want to keep $m$ as small as possible to keep test cases small. Loop-stressing may lead to an unhandled exception (C1) due to stressing the third party libraries by invoking them repeatedly. Loop-stressing may also lead to an external error (C2) if it stresses another application until it crashes.

**M2. Pause-Resume ($\boldsymbol{\delta_{\mathrm{PR}}}$).** $t' = \delta_{\mathrm{PR}}(t)$ adds two consecutive *doze* actions between all transitions of the test case $t$. Let $t_i^{pr} = (v_i, \textit{doze off}, 2) \cdot (v_i, \textit{doze on}, 2)$. Then,

$$\delta_{\mathrm{PR}}(t) = t_1^{pr} \cdot t_1 \cdot t_2^{pr} \cdot t_2 \cdot \ldots \cdot t_n^{pr} \cdot t_n \tag{2}$$

Pause-resume may trigger a crash due to resource unavailability (C3).

**M3. Change Text ($\boldsymbol{\delta_{\mathrm{CT}}}$).** We assume that existing test cases contain well-behaving text inputs to explore the AUT as much as possible. To increase the number of detected crashes, we modify the contents of the texts.

$t' = \delta_{\mathrm{CT}}(t)$ first picks one random abnormal text manipulation operation and applies it to a random *textentry* action of the existing test case $t$. Abnormal text manipulation operations can be *emptytext*, *dottext*, and *longtext* where *emptytext* deletes the text, *dottext* enters a singe dot character, and *longtext* enters a random string of length >200.

Let $z_i^{ct}$ denote a random abnormal text manipulation action where $z_i$ is a text action and $d_i^{ct}$ denotes the new delay required to completely execute $z_i^{ct}$. We define $t' = \delta_{\mathrm{CT}}(t)$ on test cases as follows:

$$\delta_{\mathrm{CT}}(t) = \begin{cases} t & \nexists z_i = \textit{textentry} \\ t_{1\ldots i-1} \cdot t_i^{ct} \cdot t_{i+1\ldots n} & \text{otherwise} \end{cases} \tag{3}$$

where $n$ is the length of $t$ and $t_i^{ct} = (v_i, v_{i+1}, z_i^{ct}, d_i^{ct})$. An AUT may crash because the corresponding *onTextChange()* method of the AUT throws an unhandled exception (C1). The AUT may also crash if the content of the text is an unexpected kind of input, which causes a semantic error later (C3).

**M4. Toggle Contextual State ($\delta_{\mathbf{TCS}}$).** Existing test suites typically lack contextual actions where the condition of the contextual state is crucial to generate the crash. Therefore, we introduce contextual state toggling with $t' = \delta_{\mathrm{TCS}}(t)$ which is defined as follows.

$$\delta_{\mathrm{TCS}}(t) = t_1 \cdot t_1^{tcs} \cdot t_2 \cdot t_2^{tcs} \cdot \ldots \cdot t_n \cdot t_n^{tcs} \tag{4}$$

where $n$ is the length of test case $t$ and $t_i^{tcs}$ is a contextual action transition $(v_{i+1}, v_{i+1}, z^{tcs}, d')$. $z^{tcs}$ corresponds to a random contextual toggle action. We pick $d' = 10\,\mathrm{s}$ for each contextual action since Android may take a long time before it stabilizes after the change of contextual state. Toggling the contextual states of the AUT may result in an external error (C2), or a network-based crash if the connection failures are not handled correctly (C5).

**M5. Remove Delays ($\delta_{RD}$).** $t' = \delta_{\mathrm{RD}}(t)$ takes a test case $t$ and sets all of its delays to 0. When reproduced, the events of $t'$ will be in the same order with $t$, but sent to the AUT at the earliest possible time.

$$\delta_{\mathrm{RD}}(t) = (v_1, v_2, z_1, 0) \cdot (v_2, v_3, z_2, 0) \cdot \ldots \cdot (v_n, v_{n+1}, z_n, 0) \tag{5}$$

If the AUT is communicating with another application, removing delays may cause the requests to crash the other application. If this case is not handled in the AUT, the AUT crashes due to external errors (C2). If the AUT's background process is affected by the GUI actions, removing delays may cause the background process to crash due to resource unavailability (C3). If the GUI actions trigger network requests, having no delays may cause a network-based crash (C5).

**M6. Faster Swipe ($\delta_{FS}$).** $t' = \delta_{\mathrm{FS}}(t)$ increases the speed of all swipe actions of a test case $t$. Let $z_i^{fs}$ denote a faster version of $z_i$, where $z_i$ is a *swipe* action. Then, we define $\delta_{\mathrm{FS}}$ on test cases with at least one *swipe* action as follows.

$$\delta_{\mathrm{FS}}(t) = t_1^{fs} \cdot t_2^{fs} \cdot \ldots \cdot t_n^{fs} \tag{6}$$

where $n$ is the length of test case $t$ and

$$t_i^{fs} = \begin{cases} (v_i, v_{i+1}, z_i, d_i) & z_i \text{ is NOT a } \textit{swipe} \\ (v_i, v_{i+1}, z_i^{fs}, d_i) & \text{otherwise} \end{cases}$$

If the information presented by the AUT is downloaded from a network or another application, swiping too fast may cause a network-based crash (C3) due to the network being unable to provide the necessary data or an external error (C2). If the AUT is a game, swiping too fast may cause the AUT to throw an unhandled exception (C1).

**Algorithm 1.** Test Suite Minimization Algorithm

**Require:**
$TS$ : A test suite for the AUT
$M$ : AUT Model
**Ensure:**
$TS'$ : Minimized Test Suite

1: $TS' \leftarrow \emptyset$
2: **for** $t \in \{t : t \in TS \wedge t \text{ does not crash}\}$ **do** ▷ Iterate over non-crashing test cases
3: **if** $\text{cov}_M(TS' \cup \{t\}) > \text{cov}_\omega(TS')$ **then** ▷ Take only the test cases that increase coverage
4: $t' \leftarrow \underset{i}{\text{argmin}}\, t_{1\ldots i}$ **s.t.** $\text{cov}_M(TS' \cup \{t_{1\ldots i}\}) = \text{cov}_M(TS' \cup \{t\})$ ▷ Shorten the test case
5: $TS' \leftarrow TS' \cup \{t'\}$ ▷ Add the shortened test case to the Minimized Test Suite
6: **end if**
7: **end for**

**Algorithm 2.** Test Case Mutation (TCM) Algorithm

**Require:**
$TS$ : A Test Suite
$X$ : Timeout of the New Test Suite
$\Delta$: Set of Mutation Operators
**Ensure:**
$TS'$ : New Test Suite

1: $TS' \leftarrow \{\}$
2: $x \leftarrow 0$
3: **repeat**
4: $t \leftarrow$ random $t \in TS$ ▷ Pick a random test case
5: $\delta \leftarrow$ random $\delta \in \Delta$ **s.t.** $t \neq \delta(t)$ ▷ Pick a mutation operator that changes the test case
6: $t' \leftarrow \delta(t)$ ▷ Apply the mutation operator to the test case
7: $TS' \leftarrow TS' \cup \{t'\}$ ▷ Add the mutated test case to the New Test Suite
8: $x \leftarrow x + \sum_{(v_s, v_e, z, d) \in t'} d$ ▷ Calculate the total delay
9: **until** $x > X$ ▷ Repeat until the total delay is above the given timeout

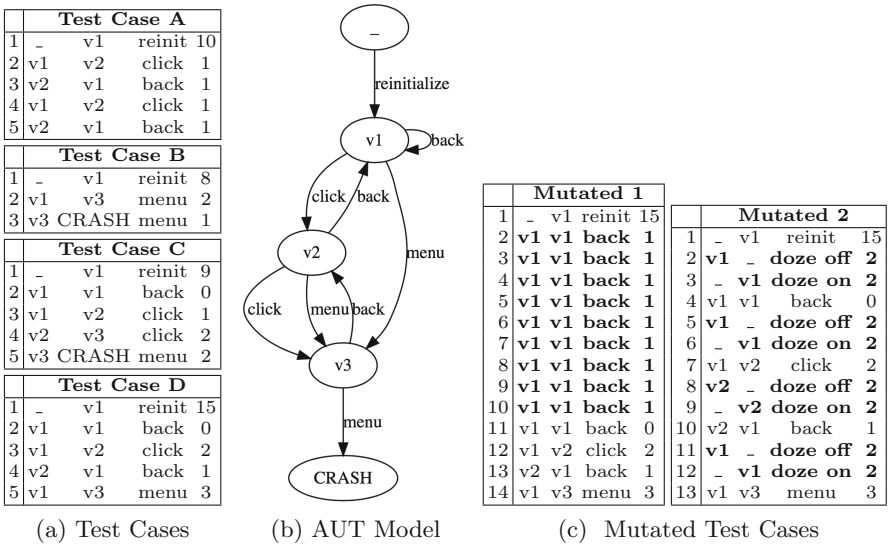
## 4 Test Suite Minimization and Test Case Mutation

Before mutating the existing test cases in a test suite $TS$, we first minimize $TS$. In order to minimize a test suite $TS$, we first define an edge coverage function $\text{cov}_\omega(TS)$ over the AUT model $M$ as follows:

$$\text{cov}_M(TS) = \frac{\#\text{ of unique transitions covered in the AUT Model } M \text{ by } TS}{\#\text{ of all transitions in the AUT Model } M} \tag{7}$$

We present our Test Suite Minimization approach in Algorithm 1. We iterate over all non-crashing test cases of the original test suite $TS$ in line 2. We use non-crashing test cases in Algorithm 1 because our goal is to generate crashes from non-crashing via mutation. We check if the test case $t$ increases the edge coverage in line 3. If $t$ increases the edge coverage, we shorten the test case $t$ from its end by deleting transitions that are not contributing to the edge coverage and add the shortened test case $t'$ to the minimized test suite.

We present our Test Case Mutation approach in Algorithm 2. We pick a random test case $t$ from given $TS$ in line 4. Then, we pick a random mutation operator $\delta$ that changes $t$ in line 5. We mutate $t$ with $\delta$ and add the mutated test case $t'$ to $TS'$ until the total delay of $TS'$ exceeds the given timeout $X$.



| | Test Case A | | | |
|---|---|---|---|---|
| 1 | _ | v1 | reinit | 10 |
| 2 | v1 | v2 | click | 1 |
| 3 | v2 | v1 | back | 1 |
| 4 | v1 | v2 | click | 1 |
| 5 | v2 | v1 | back | 1 |

| | Test Case B | | | |
|---|---|---|---|---|
| 1 | _ | v1 | reinit | 8 |
| 2 | v1 | v3 | menu | 2 |
| 3 | v3 | CRASH | menu | 1 |

| | Test Case C | | | |
|---|---|---|---|---|
| 1 | _ | v1 | reinit | 9 |
| 2 | v1 | v1 | back | 0 |
| 3 | v1 | v2 | click | 1 |
| 4 | v2 | v3 | click | 2 |
| 5 | v3 | CRASH | menu | 2 |

| | Test Case D | | | |
|---|---|---|---|---|
| 1 | _ | v1 | reinit | 15 |
| 2 | v1 | v1 | back | 0 |
| 3 | v1 | v2 | click | 2 |
| 4 | v2 | v1 | back | 1 |
| 5 | v1 | v3 | menu | 3 |

(a) Test Cases



(b) AUT Model

| | Mutated 1 | | | |
|---|---|---|---|---|
| 1 | _ | v1 | reinit | 15 |
| 2 | **v1** | **v1** | **back** | **1** |
| 3 | **v1** | **v1** | **back** | **1** |
| 4 | **v1** | **v1** | **back** | **1** |
| 5 | **v1** | **v1** | **back** | **1** |
| 6 | **v1** | **v1** | **back** | **1** |
| 7 | **v1** | **v1** | **back** | **1** |
| 8 | **v1** | **v1** | **back** | **1** |
| 9 | **v1** | **v1** | **back** | **1** |
| 10 | **v1** | **v1** | **back** | **1** |
| 11 | v1 | v1 | back | 0 |
| 12 | v1 | v2 | click | 2 |
| 13 | v2 | v1 | back | 1 |
| 14 | v1 | v3 | menu | 3 |

| | Mutated 2 | | | |
|---|---|---|---|---|
| 1 | _ | v1 | reinit | 15 |
| 2 | **v1** | **_** | **doze off** | **2** |
| 3 | **_** | **v1** | **doze on** | **2** |
| 4 | v1 | v1 | back | 0 |
| 5 | **v1** | **_** | **doze off** | **2** |
| 6 | **_** | **v1** | **doze on** | **2** |
| 7 | v1 | v2 | click | 2 |
| 8 | **v2** | **_** | **doze off** | **2** |
| 9 | **_** | **v2** | **doze on** | **2** |
| 10 | v2 | v1 | back | 1 |
| 11 | **v1** | **_** | **doze off** | **2** |
| 12 | **_** | **v1** | **doze on** | **2** |
| 13 | v1 | v3 | menu | 3 |

(c) Mutated Test Cases

**Fig. 2.** Motivating example (mutations are denoted as bold)

## 5 Motivating Example

Figures 2a and b show a test suite and an AUT model, respectively. We generate this test suite and the AUT model by executing AndroFrame for one minute on an example AUT. We execute AndroFrame for just one minute, because that is enough to generate test cases for this example. We limit the maximum number of transitions per test case to five to keep the test cases small in this motivating example for ease of presentation. The test suite has four test cases; A, B, C, and D. Each row of test cases describes a delayed transition. The *click* action has coordinates, but we abstract this information for the sake of simplicity.

Among the four test cases reported by AndroFrame, we take only the non-crashing test cases, A and D. In our example, we include D since it increases the edge coverage and we exclude A since all of A's transitions are also D's transitions, i.e. A is subsumed by D. Then, we attempt to minimize test case D without reducing the edge coverage. In our example, we don't remove any transitions from D because all transitions in D contribute to the edge coverage. We then generate mutated test cases by randomly applying mutation operators to D one by one until we reach one minute timeout. Figure 2c shows an example mutated test suite. Test case Mutated 1 takes D and exercises the back button for multiple times to stress the loop at state v1. Test case Mutated 2 clicks the hardware power button twice (doze off, doze on) between each transition. This operation pauses and resumes the AUT in our test devices. We then execute all mutated test cases on the AUT. Our example AUT in fact crashes when the loop on $v_1$ is reexecuted more than eight times and also crashes when the AUT is paused in state $v_2$. When executed, our mutated test cases reveal these crashes both at their ninth transition, doubling the number of detected crashes.

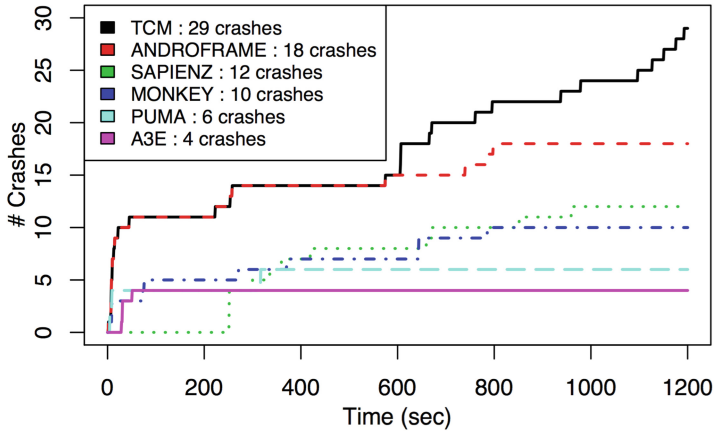



**Fig. 3.** Number of total distinct crashes detected across time

## 6 Evaluation

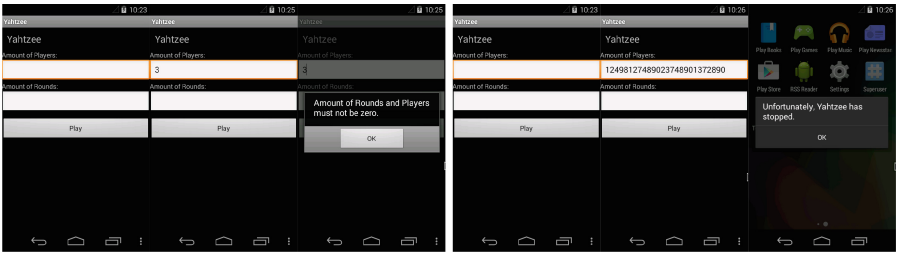
In this section, we evaluate TCM via experiments and case studies. We show that, through experiments, we improve crash detection. We then show, with case studies, how we detect crash patterns.

### 6.1 Experiments

We selected 100 AUTs (excluding the case studies described later) from F-Droid benchmarks [7] for experiments. To evaluate the improvement in crash detection, we first execute AndroFrame, Sapienz, PUMA, Monkey, and A$^3$E for 20 min each on these applications with no mutations enabled on test cases. Then we execute TCM with 10 min for AndroFrame to generate test cases and 10 min to mutate the generated test cases and replay them to detect more crashes. AndroFrame requires the maximum length of a test case as a parameter. We used its default parameter, 80 transitions maximum per test case.

Figure 3 shows the number of total distinct crashes detected by each tool across time. Whenever a crash occurs, the Android system logs the resulting stack trace. We say that two crashes are distinct if stack traces of these crashes are different.

Our results show that AndroFrame detects more crashes than any other tool from very early on. TCM detects the same number of crashes with AndroFrame for the first 10 min (600 s). During that time, AndroFrame detects 15 crashes. In the last 10 min, TCM detects 14 more crashes whereas AndroFrame detects only 3 more crashes. As a result TCM detects 29 crashes in total whereas AndroFrame detects 18 crashes in total. As a last note, all other tools including AndroFrame seem to stabilize after 20 min whereas TCM finds many crashes near timeout. This shows us that TCM may find even more crashes when timeout is longer.




(a) Execution of Test Case $t$ (b) Execution of Test Case $t' = \delta_{\mathrm{CT}}(t)$

**Fig. 4.** An example crash found only by TCM

Overall, TCM finds 14 more crashes than AndroFrame and 17 more crashes than Sapienz, the best among other tools.

We also investigate how much each mutation operator contributes to the number of detected crashes. Our observations reveal that M1 ($\delta_{\mathrm{LS}}$) detects one crash, M2 ($\delta_{\mathrm{PR}}$) detects four crashes, M3 ($\delta_{\mathrm{CT}}$) detects two crashes, M4 ($\delta_{\mathrm{TCS}}$) detects two crashes, M5 ($\delta_{\mathrm{RD}}$) detects four crashes, and M6 ($\delta_{\mathrm{FS}}$) detects one crash. These crashes add up to 14, which is the number of crashes detected by TCM in the last 10 min. This result shows that while all mutation operators contribute to the crash detection, M2 and M5 have the largest contribution.

We present and explain one crash that is found only by TCM in Fig. 4. Figure 4a shows an instance where AndroFrame generates and executes a test case $t$ on the Yahtzee application. Note that $t$ does not lead to a crash, but only a warning message. Figure 4b shows the instance where TCM mutates $t$ and executes the mutated test case $t'$. When $t'$ is executed, the application crashes and terminates. We note that this crash was not found by any other tool. Mao et al. [8] also report that Sapienz and Dynodroid did not find any crashes in this application.

### 6.2 Case Studies

In this section, we verify that the aforementioned crash patterns exist via case studies, one case study for each crash pattern. These studies verify that all of our crash patterns are observable in Android platform. These case studies help us develop and fine-tune our mutation operators.

**Case Study 1.** Figure 5a shows a crashing activity of the *SoundBoard* application included in F-Droid benchmarks. Basically, the *coin* and *tube* buttons activate a third party library, AudioFlinger, to produce sound when tapped. AndroFrame generates test cases which tap these buttons. These test cases produce no crashes. Then, we mutate the test cases with TCM. When we apply *loop-stressing* (M1) on any of these buttons, AudioFlinger crashes due to overuse. AudioFlinger produces a fatal exception (C1) in Android logs. This crash does not cause an abnormal termination, but it causes the AUT to stop functioning (the AUT stops producing sounds until it is restarted).


(a) C1 (b) C2 (c) Resource Unavailability (C3) Example

(d) Semantic Error (C4) Example (e) Network-Based Crash (C5) Example

C1: Unhandled Exception (C1) Example

C2: External Error (C2) Example

**Fig. 5.** Case studies 1–5

**Case Study 2.** Figure 5b shows a crashing activity of the *a2dpVol* application included in F-Droid benchmarks, where AndroFrame fails to generate crashing test cases. We mutate these test cases with TCM. When we activate bluetooth (M4), tapping *find devices* button produces a crash in the external *android.bluetooth.IBluetooth* application due to a missing method (C2) and the AUT terminates.

**Case Study 3.** Figure 5c shows a crashing activity of the *importcontacts* application included in F-Droid benchmarks. The AUT handles the case that it fails to import contacts, as we show in the leftmost screen. Pausing the AUT at this screen causes the background process to abort and free its allocated memory (we show the related screen in the middle). However, the paused activity is not destroyed. If the user tries to resume this activity, the AUT crashes as we show in the rightmost screen, since the memory was freed before. TCM applies a pause-resume mutation (M2) and triggers this resource unavailability crash (C3).

**Case Study 4.** Figure 5d shows a crashing activity of the *aCal* application included in F-Droid benchmarks. AndroFrame generates test cases with well-behaving text inputs. These test cases produce no crashes. Then, we mutate the test cases with TCM. When we apply *change text* (M3) on the last text box and then tap the *configure* button, this produces a semantic error (C4). The AUT crashes and terminates.

**Case Study 5.** Figure 5e shows a crashing activity of the *Mirrored* application included in F-Droid benchmarks. When *wifi* is turned off, the AUT goes into offline mode and does not crash as shown in the leftmost screen. When we toggle *wifi* (M4), the AUT retrieves several articles as shown in the middle, but crashes when it fails to retrieve article contents due to a network-based crash (C5) as shown in the rightmost screen.

## 7 Discussion

Although TCM is conceptually applicable to different GUI platforms, e.g. iOS or a desktop computer, there are three key challenges. First, our crash patterns are not guaranteed to exist or be observable in different platforms. Second, our mutation operators may not be applicable to those platforms, e.g. swipe may not be available as a gesture. Third, either an AUT model may be impossible to obtain or a replayable test case may be impossible to generate in those platforms. When all these challenges are addressed, we believe TCM should be applicable to not just Android, but other platforms as well.

TCM mutates test cases after they are generated. We could apply mutated inputs immediately during test generation. However, this requires us to alter the test generation process which may not be possible if a third party test generation tool is used. Our approach is conceptually applicable to any test generation tool without altering the test generation tool.

We use an edge coverage criterion to minimize a given test suite. Because of this the original test suite covers potentially more paths than the minimized test suite and therefore explores the same edge in different contexts. Without minimization, test cases in the test suite are too many and too large to generate enough mutations to observe crashes in given timeout. Therefore, we argue that by minimizing the test suite we improve the crash detection performance of TCM at the cost of the test suite's completeness in terms of a higher coverage criterion than edge coverage.

Although TCM detects crashes, it does not detect all possible bug patterns. Qin et al. [9] thoroughly classifies all bugs in Android. According to this classification, there are two types of bugs in Android, Bohrbugs and Mandelbugs. A Bohrbug is a bug whose reachability and propagation are simple. A Mandelbug is a bug whose reachability and propagation are complicated. Qin et al. further categorize Mandelbugs as Aging Related Bugs (ARB) and Non-Aging Related Mandelbugs (NAM). Qin et al. also define five subtypes for NAM and six subtypes for ARB. TCM detects only the first two subtypes of NAM, TIM and SEQ. TIM and SEQ are the only kinds of bugs which are triggered by user inputs. If a bug is TIM, the error is caused by the timing of inputs. If a bug is SEQ, the error is caused by the sequencing of inputs.

We note two key points on the crash patterns of TCM. First, testing tools we compare TCM with only detect SEQ bugs. TCM introduces the detection of TIM bugs in addition to SEQ bugs. Second, Azim et al. [3] further divides SEQ and TIM bugs into six crash patterns. We base our crash patterns on these

crash patterns. We present both external errors and permission violations as one crash pattern since permission violations occur as attempts to communicate with external applications with insufficient permissions. As a result, we obtain five crash patterns.

We did not encounter any crash patterns other than the five crash patterns that we describe in Sect. 3. However, it is still possible to observe other crash patterns with our mutation operators due to emerging crash patterns caused by the fragmentation and fast development of the Android platform.

Our mutation operators insert multiple transitions to the test case, creating an issue of locating the fault inducing transition. Given that the mutated test case detects a crash, fault localization can be achieved using a variant of *delta debugging* [10].

We use regular expressions on the Android logs to detect crashes. In the experiment, we only detected *FATAL EXCEPTION* labeled errors as done in previous work [3,5], ignoring Application Not Responding (ANR) and other errors described by Carino and Andrews [11]. Although we believe that TCM would still detect more crashes than pure AndroFrame (fatal exception is the most common crash in Android), we will improve our crash detection procedure as a future work to give more accurate results.

We randomly selected 100 Android applications from the well-known F-Droid benchmarks also used by other testing tools [7]. We show that these applications have similar characteristics with the rest of F-Droid applications in our previous work.

## 8 Related Work

Test Case Mutation (TCM) differs from the well-known Mutation Testing (MT) [12] where mutations are inserted in the source code of an AUT to measure the quality of existing test cases. Whereas in TCM, we update existing test cases to increase the number of detected crashes. Oliveria et al. [13] are the first to suggest using Mutation Testing (MT) for GUIs. Deng et al. [14] define several source code level mutation operators for Android applications to measure the quality of existing test suites.

The concept of Test Case Mutation is not new. In Android GUI Testing, Sapienz [8] and EvoDroid [15] are Android testing tools that use evolutionary algorithms, and therefore mutation operators. Sapienz shuffles the orders of the events, whereas EvoDroid mutates the test case in two ways: (1) EvoDroid transforms text inputs and (2) EvoDroid either injects, swaps, or removes events. TCM mutates not only text inputs, but also introduces 5 more novel mutation operators. Furthermore, Sapienz and EvoDroid use their mutation operators for both exploration and crash detection whereas we specialize TCM's mutation operators for crash detection only. In Standard GUI Testing, MuCRASH [16] uses test case mutation via defining special mutation operators on test cases, where the operators are defined at the source code level. They use TCM for crash reproduction, whereas ours is the first work that uses TCM to discover new crashes. Directed Test Suite Augmentation (DTSA) introduced by

Xu et al. in 2010 [17] also mutates existing test cases but for the goal of achieving a target branch coverage.

We implement TCM on AndroFrame [4]. AndroFrame is one of the state-of-the-art Android GUI Testing tools. AndroFrame finds more crashes than other available alternatives in the literature such as A$^3$E and Sapienz. These tools generate replayable test cases as well. They provide the necessary utilities to replay their generated test cases. We can mutate these test cases but most of our mutations won't be applicable for two reasons. First, A$^3$E and Sapienz do not learn a model from which we can extract looping actions. Second, A$^3$E and Sapienz do not support contextual state toggling. Implementing all of our mutations on top of these tools is possible, but requires a significant amount of engineering effort. Therefore we implement TCM on top of AndroFrame.

Other black-box testing tools in the literature include A$^3$E [18], SwiftHand [6], PUMA [19], DynoDroid [20], Sapienz [8], EvoDroid [15], CrashScope [5] and MobiGUITAR [21]. From these applications, only EvoDroid, CrashScope, and MobiGUITAR are publicly unavailable.

Monkey is a simple random generation-based fuzz tester for Android. Monkey detects the largest number of crashes among other black-box testing tools. Generation-based fuzz testing is a popular approach in Android GUI Testing, which basically generates random or unexpected inputs. Fuzzing could be completely random as in Monkey, or more intelligent by detecting relevant events as in Dynodroid [20]. TCM can be viewed as a mutation-based fuzz testing tool, where we modify existing test cases rather than generating test cases from scratch. TCM can be implemented on top of Monkey or DynoDroid to improve crash detection of these tools.

Baek and Bae [22] define a comparison criterion for Android GUI states. AndroFrame uses the maximum comparison level described in this work, which makes our models as fine-grained as possible for black-box testing.

## 9 Conclusion

In this study, we developed a novel test case mutation technique that allows us to increase detection of crashes in Android applications. We defined six mutation operators for GUI test cases and relate them to commonly occurring crash patterns in Android applications. We obtained test cases through a state-of-the-art Android GUI testing tool, called AndroFrame. We showed with several case studies that our mutation operators are able to uncover new crashes.

As a future work, we plan to study a broader set of GUI actions, such as *rotation* and *doubleclick*. We will improve our mutation algorithm by sampling mutation operators from a probability distribution based on crash rates rather than a uniform distribution. We will find the most optimal timings for executing the test generator and TCM, rather than dividing the available time into two equal halves. We will further investigate Android crash patterns.

## References


1. Piejko, P.: 16 mobile market statistics you should know in 2016 (2016). https://deviceatlas.com/blog/16-mobile-market-statistics-you-should-know-2016
2. Zein, S., Salleh, N., Grundy, J.: A systematic mapping study of mobile application testing techniques. J. Syst. Softw. **117**, 334–356 (2016)
3. Azim, T., Neamtiu, I., Marvel, L.M.: Towards self-healing smartphone software via automated patching. In: 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 623–628 (2014)
4. Koroglu, Y., Sen, A., Muslu, O., Mete, Y., Ulker, C., Tanriverdi, T., Donmez, Y.: QBE: QLearning-based exploration of android applications. In: IEEE International Conference on Software Testing, Verification and Validation (ICST) (2018)
5. Moran, K., Vásquez, M.L., Bernal-Cárdenas, C., Vendome, C., Poshyvanyk, D.: Automatically discovering, reporting and reproducing android application crashes. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 33–44 (2016)
6. Choi, W., Necula, G., Sen, K.: Guided GUI testing of android apps with minimal restart and approximate learning. In: ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 623–640 (2013)
7. Gultnieks, C.: F-Droid Benchmarks (2010). https://f-droid.org/
8. Mao, K., Harman, M., Jia, Y.: Sapienz: multi-objective automated testing for android applications. In: 25th International Symposium on Software Testing and Analysis (ISSTA), pp. 94–105 (2016)
9. Qin, F., Zheng, Z., Li, X., Qiao, Y., Trivedi, K.S.: An empirical investigation of fault triggers in android operating system. In: IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 135–144 (2017)
10. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7), pp. 253–267 (1999)
11. Carino, S., Andrews, J.H.: Dynamically testing GUIs using ant colony optimization. In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 135–148 (2015)
12. Ammann, P., Offutt, J.: Introduction to Software Testing, 1st edn. Cambridge University Press, Cambridge (2008)
13. Oliveira, R.A.P., Algroth, E., Gao, Z., Memon, A.: Definition and evaluation of mutation operators for GUI-level mutation analysis. In: IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10 (2015)
14. Deng, L., Offutt, J., Ammann, P., Mirzaei, N.: Mutation operators for testing android apps. Inf. Softw. Technol. **81**(C), 154–168 (2017)
15. Mahmood, R., Mirzaei, N., Malek, S.: EvoDroid: segmented evolutionary testing of android apps. In: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 599–609 (2014)
16. Xuan, J., Xie, X., Monperrus, M.: Crash reproduction via test case mutation: let existing test cases help. In: 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 910–913 (2015)
17. Xu, Z., Kim, Y., Kim, M., Rothermel, G., Cohen, M.B.: Directed test suite augmentation: techniques and tradeoffs. In: 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 257–266 (2010)

18. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 641–660 (2013)
19. Hao, S., Liu, B., Nath, S., Halfond, W.G., Govindan, R.: PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In: 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys), pp. 204–217 (2014)
20. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: an input generation system for android apps. In: 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 224–234 (2013)
21. Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.M.: MobiGUITAR: automated model-based testing of mobile apps. IEEE Softw. **32**(5), 53–59 (2015)
22. Baek, Y.M., Bae, D.H.: Automated model-based android GUI testing using multi-level GUI comparison criteria. In: 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 238–249 (2016)