Designing Mutation Operators for Android Device Components: A View Through Bluetooth and Location API's

Pedro Henrique Kuroishi
Department of Computing
Federal University of São Carlos
São Carlos, Brazil
phk@ufscar.br

Ana C. R. Paiva
INESC TEC, Faculty of Engineering, University of Porto
Porto, Portugal
apaiva@fe.up.pt

José Carlos Maldonado
Institute of Mathematics and Computer Science
University of São Paulo
São Carlos, Brazil
jcmaldon@icmc.usp.br

Auri Marcelo Rizzo Vincenzi
Department of Computing
Federal University of São Carlos
São Carlos, Brazil
auri@ufscar.br

ABSTRACT

Context: Mutation operators play a crucial role during the mutation testing process due to their capability to model common faults to be injected into an application under test. In the context of the Android OS, the academic literature shows that different studies propose mutation operators, especially focusing on the GUI and configuration. On the other hand, few devise mutation operators for specific Android device resource components such as connectivity, location, and sensors. **Objective:** Therefore, this paper aims to investigate the design and proposition of mutation operators for Bluetooth, Location, and the third-party library AltBeacon. The rationale is that this paper is carried out in an academia-industry partnership, in which the company develops applications that rely on these Android components. **Method**: The design process used a systematic approach named HAZOP to minimize any possible bias. **Results**: This systematic process helped in deriving a set of 16 mutation operators. Next, the paper provides an empirical cost evaluation that assesses the number of generated mutants for two applications and the number of generated mutants per operator, showing the feasibility of the mutation operators and their capabilities in modeling real faults. **Conclusion**: Finally, the paper discusses the future perspectives of extending the operators to other device resource components such as Wi-Fi, NFC, and sensors, as well as automation perspectives by envisioning the implementation and validation of the mutation operators.

KEYWORDS

mutation testing, mutation operator, Android, Android mutation operator, HAZOP, Bluetooth, Location, AltBeacon

1 Introduction

Mutation testing (or mutation) is a test criterion commonly adopted to assess the quality of a given test suite based on its fault detection capabilities [7, 37]. The traditional mutation testing process starts with the generation of faulty versions, named *mutants*, of the original application under test (or AUT). Then, the test suite is executed against each mutant, and the output is evaluated to check whether the test suite can detect the fault injected in the mutant. The quality metric is given by the mutation score, which is defined by the ratio of killed mutants to non-equivalent ones. Observe that a crucial step of mutation testing consists of generating the faulty version of the AUT. To support this process, mutation testing relies on *mutation operators*, i.e., a set of rules that modify the original application by making a simple syntactic change [15]. For example, an arithmetic operator changes the statement $a = b + c$ to $a = b - c$ and $a = b * c$.

Over the years, many studies have proposed mutation operators for different programming languages (e.g., C [1, 5], Java [3, 6, 17, 18, 27, 28], Python [10], JavaScript [29, 31, 33]) and system types (e.g., Android [8, 9, 11, 14, 21, 25, 35], deep learning systems [26], time systems [34], cyber-physical systems [43]). This paper focuses on designing mutation operators for Android.

Android application testing differs from traditional software (e.g., web and desktop applications). An Android app is designed to run on a mobile device. Therefore, when testing an Android app, one has to consider the limitations of the mobile device, such as screen size and density, multiple sensors and connectivity, and constrained resources [30]. Given these heterogeneous characteristics, it is expected that the mutation operators will cover not only the functionality of the app but also other aspects of the Android ecosystem.

The mapping study of Silva et al. [41] showed that few studies focus on mutation testing for Android apps. Consequently, only a subset of the mapped studies proposed Android mutation operators focusing on adapting existing Java operators and designing specific operators for GUI components. Additionally, only four studies [8, 11, 14, 21] present mutation operators for other Android components such as sensors, connectivity, and location.

The present paper aims to propose the design of mutation operators for Bluetooth [22] and Location [23] API, given the lack of studies focusing on these specific Android components. For Bluetooth, this paper considers both the classic and low-energy APIs. Moreover, the present study explores the design of mutation operators for the third-party library AltBeacon [32]. For Location, the operators are designed for the native and the Fused Location Provider API provided by Google Play Services (or GMS) [24].

The rationale for selecting these specific components is that the study is being carried out in an academia-industry partnership. In this case, the company develops different applications that rely on these Android components. Therefore, the mutation operators provided in this study are not only useful in enhancing the quality of the apps developed by the company, but also benefit further academic research and companies dealing with the same difficulties.

In general, the design of mutation operators relies on the experience and knowledge of the proponent [18]. Since there is still no standardized method to facilitate the generation of operators, the paper leverages a systematic technique called Hazard and Operability Studies (or HAZOP) [4, 18]. This approach originated in the chemical industry, and the goal is to identify potential hazards and deviations in the behavior of a system using a set of predefined guide words to support determining the cause and consequences of the deviations [2, 4, 18]. The rationale for using HAZOP is to facilitate the design process and minimize any possible bias that may arise.

In summary, this paper makes the following contributions:

- The design of 16 mutation operators for Bluetooth, AltBeacon, and Location API's.
- A cost evaluation of the proposed mutation operators.
- A discussion of future perspectives on extending the defined set of mutation operators for other Android components and automation aspects.

The remainder of the paper is organized as follows: Section 2 presents an overview of the concepts of this paper. Section 3 describes the study goals and design. Section 4 provides the steps carried out to devise the mutation operators and a description and examples of them. Section 5 presents the empirical cost evaluation of applying the mutation operators. Section 6 describes the related works. Finally, Section 7 presents future perspectives and concludes the paper.

2 Background

This section provides a brief overview of the main concepts related to the present work.

2.1 Mutation Operators for Android

Mutation operators play a crucial role during the mutation process due to their capability of introducing a syntactic modification under the AUT [1]. The academic literature proposes various mutation operators that can be applied to different programming languages, systems, and other purposes.

As mentioned, mutation operators can be described as a transformation rule that injects a possible fault in the source code, mimicking the common mistakes of developers [15]. In general, these operators can modify variables, replace operators from expressions, delete statements, and so on.

In the context of Android, there are still few studies focusing on mutation testing and, consequently, there is still a need to explore this technique and the extension of the existing mutation operators [41]. Linares-Vásquez et al. [20] proposed an insightful taxonomy of Android bugs to support designing the mutation operators. The authors categorized the existing bugs into fourteen categories: Activities and Intent, Back-end Services, Collections and Strings, Data/Objects Parsing and Format, Threading, Android Programming, Non-functional Requirements, GUI, I/O, Device/Emulator, API and Libraries, Connectivity, Database, and General Programming. Moreover, the authors stated that the bugs affect not only Android apps but also Java applications.

According to Silva et al. [41], most Android mutation operators are centered on the GUI and the specific configuration of the Android API. On the other hand, few studies explore mutation operators for sensor and location. Additionally, few studies investigated the design of mutation operators for the Bluetooth API, considering connectivity.

Given the complexity of the ecosystem and the need to explore other Android components, we advocate for a broader investigation of designing mutation operators for Bluetooth and Location apps.

2.2 Bluetooth and Location API

Many Android application relies on the information retrieved by the sensors embedded in the mobile device. For instance, an app may use Bluetooth to connect and exchange data with a peripheral or an IoT device. Or may collect real-time information about the user's location or monitor the motion of a mobile device. Observe that these types of applications resemble the ones developed by our partner company.

According to the documentation [22], Android API supports two types of Bluetooth API: classic and low-energy (BLE). The main difference between these two types relates to energy consumption. The first is commonly used for more battery-intensive operations, whereas the latter is projected to consume less energy and transfer a low quantity of data. The two types of Bluetooth may perform the same operations: finding devices, connecting, and transferring data.

BLE can be used in various contexts such as proximity-based applications, communicating with IoT systems, and interacting with BLE devices such as beacons [12]. The third-party Android Beacon Library [32] supports functionalities to facilitate the interaction of an Android device and beacons. It is worth noting that this library implements the AltBeacon advertisement protocol and is adopted by many applications [13, 19, 32].

Android API also offers support to create location-aware applications [23]. Similar to Bluetooth, the Location API offers a ton of functionalities that facilitate collecting and processing information about the location. A developer may leverage the native functionalities and also the API supported by Google Play services (or GMS) to build location-aware apps. In this paper, the native and the GMS are considered.

2.3 Hazard and Operability Studies – HAZOP

The process for designing and proposing mutation operators is commonly carried out based on previous experience and knowledge [17] since there is still no standardized methodology to facilitate this task. Kim et al. [17] designed mutation operators for the Java programming language using a systematic approach named Hazard and Operability studies (or HAZOP). This method originated in the chemical industry to identify potential hazards and deviations in the behavior of a system using a set of predefined guide words that have to be adapted and interpreted according to the system context [2, 4, 18]. In general, the following guide words are considered:

no/none, *more*, *less*, *as well as*, *part of*, *reverse*, and *other than*. Table 1 describes the interpretations of the guide words.

Table 1: Guide words interpretation adapted from [4, 17].

Guide word	Interpretation
no/none	No part of the intention is achieved
more	Quantitative increase
less	Quantitative decrease
as well as	The design of the intent is achieved but with additional results
part of	Only some part of the intention is achieved
reverse	Reverse the information flow
other than	A result other than the original intent is achieved

Additionally, Chudleigh et al. [4] extends this list with four guide words: *early*, *late*, *before*, and *after*. For the present paper, we considered the seven guide words from Table 1 and also *early* (i.e., earlier execution than intended) and *late* (i.e., delayed execution than intended).

For example, Kim et al. [18] applied the HAZOP approach to derive Java mutation operators by examining the language specification and identifying plausible deviations and their causes and consequences. The operator *Variable replacement operator* can be associated with the guide word *Other than*. It can replace variable names of the same type (*cause*), leading to an anomalous behavior of the system (*consequence*).

3 Study Setup

3.1 Study Goal

The main purpose of this paper is to investigate and design a mutation operator for the Android Bluetooth and Location API. The rationale for selecting these two components is the following:

(1) As presented by Silva et al. [41], few studies focus on mutation operators for specific Android components rather than Java-specific or GUI. Therefore, we intend to provide an investigation toward the viability of proposing mutation operators for Bluetooth and Location apps and envision the possibility for other connectivity components (e.g., Wi-Fi[1] and NFC[2]) and sensors[3] (e.g., gyroscope and accelerometer).

(2) The present study is carried out under an academic-industry collaboration. The company develops mobile applications that interact with IoT devices using Bluetooth (both native and AltBeacon) and Location features. Hence, this paper focuses on these two components and aims to enhance the quality of the testing process of the company. For confidentiality reasons, this paper will not provide any information about the company or specific details of their applications.

3.2 Study Design

Figure 1 illustrates the steps carried out to guide the present study.

The process started by evaluating the Bluetooth, Location, and AltBeacon API and identifying possible mutation points. Then, the collected mutation points were assessed and grouped by their actions to form an initial subset of mutation operators. The subset was then validated using the HAZOP approach. Next, the operators were cross-checked against the existing mutation operators from the academic literature to verify if the operators were previously proposed. Whenever a mutation operator from the literature was not considered in our analysis, it was added to the final set of operators.

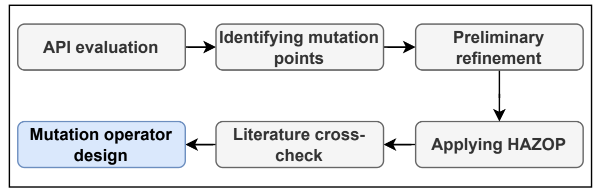

Figure 1: Mutation operator designing process.

3.3 Empirical Cost Evaluation

After defining the mutation operators, this paper evaluates the cost of applying the operators. To this purpose, two apps were considered, and the mutants were manually generated to collect information about (i) the number of mutants generated per application and (ii) the number of mutants generated per operator.

This data provides insights into the viability of the mutation operators in terms of mutant generation and helps us to understand the rationale of the operators that eventually do not generate any mutants.

It is worth noting that the mutants were manually generated because it is still not implemented. That is, whenever a mutation point was found, the source code was manually modified to mimic a possible mutant, and the application was recompiled to check the validity of the syntactic change. Section 7 provides a broad discussion of the automation aspects.

4 Mutation Operator Description

The present section presents the design process carried out to devise the set of mutation operators. Additionally, the section provides a broad description of the operators and examples to enlighten their usage.

4.1 Designing Process

The first step of the designing process started with the API evaluation as displayed in Figure 1. It encompasses an in-depth Bluetooth, Location, and AltBeacon API documentation analysis to understand whether it was reasonable to apply mutation testing in these specific Android components.

During the analysis, it was possible to observe that some mutation points could be categorized based on the component action target. Android offers two types of Bluetooth API support: classic and low-energy. In both cases, the mutation points could be related to configuration, scan, connect/pair, and data/information transfer. Altbeacon also has four action targets: configuration, transmitting, ranging, and monitoring. Location can be divided into two types: native API and GMS (i.e., API that is part of Google Play Services).

[1] https://developer.android.com/develop/connectivity/wifi/wifiscan
[2] https://developer.android.com/develop/connectivity/nfc
[3] https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview

In both cases, the mutation points could be related to configuration and location-specific methods.

The next step consisted of identifying potential mutation points for each component action target. Table 2 presents the number of identified mutation points.

Table 2: Number of identified mutation points.

API	Type	Action	# MP
Bluetooth	Classic	Configuration	5
		Scan	8
		Connect/Pair	4
		Transfer	1
	BLE	Configuration	1
		Scan	3
		Connect/Pair	3
		Transfer	1
	AltBeacon	Configuration	16
		Transmitting	5
		Ranging	7
		Monitoring	5
Location	Native	Configuration	7
		Location-specific	9
	GMS	Configuration	4
		Location-specific	9
Total			88

As can be seen, the previous step generated a larger number of possible mutation points. Moreover, various mutation points had the same code structure, that is, they could represent the same type of mutation operator. Therefore, a refinement step was carried out to aggregate the mutation points and form a more fine-grained group type. After analysis, the 88 mutation points were clustered into 16 groups, which represent the preliminary set of mutation operators.

The name of each mutation operator represents the modeled faults. In a first attempt, we envisioned using the keyword *Sensor* to represent this type of mutation operator. However, Android offers support for Sensor API that does not directly correlate to Bluetooth and Location and hence, using this keyword would be misleading. Since both cases rely on mobile device resources, we decided to use the keyword *Device* to make a more generic name. Note that we intend to extend the mutation operators, considering other mobile device hardware components such as NFC, Wi-Fi, and sensors (e.g., gyroscope and accelerometer). Thus, it would be easier to aggregate new operators using a generic classification. Table 3 presents the preliminary set of mutation operators.

Next, the HAZOP approach was applied to validate the preliminary set of operators. Each operator was assigned to one or more guide words. An operator was excluded from the set whenever the association between the operator and the guide word did not occur. Additionally, we describe the possible cause (i.e., the deviation/mutation operator that should be applied) and the consequence (i.e., system/code behavior when the operator is applied). Table 4 summarizes the results of applying HAZOP.

Nine mutation operators were associated with *OTHER THAN* guide word, which consists of replacing the current instantiation, properties, constants, and method call, causing unexpected behavior or collateral effects. Two mutation operators associated with *AS WELL AS*, that is, it inserts a code snippet without causing any loss of information. Two operators associated with *REVERSE* may cause

Table 3: Set of devised mutation operators.

Operator Name	Acronym
BuggyDeviceCallback	BDC
TrapDeviceCallback	TDC
BuggyDeviceListener	BDL
TrapDeviceListener	TDL
NullDeviceInstanceDeclaration	NDID
DefaultDeviceBuilderInstanceDeclaration	DDBID
SwitchConditionalDeviceMethod	SCDM
ReplaceConditionalParameterDeviceMethod	RCPDM
ShiftDeviceMethodCall	SDMC
DeletionDeviceMethods	DDM
DeviceVariablesOperator	DVO
NullReferenceDeviceMethods	NRDM
RandomActionIntentDeviceDefinition	RAIDD
ReplaceCompatibleTypeDeviceGetMethods	RCTDGM
RandomLocationProviderDeviceReplacement	RLPDR
RandomLocationRequestBuilderPriorityDeviceReplacement	RLRBPDR

an expected result from the application by switching the boolean value of a method or a method parameter. One operator associated with *EARLY* and *LATE* causes an application error. One operator is associated with *MORE* and *LESS*, which increase/decrease the value of a scalar variable, causing an unexpected result. Finally, one operator associated with *NO/NONE* leads to collateral effects. After analysis, no mutation operators were left without an association with the guide word. Thus, the final set of mutation operators is those presented in Table 3.

In the last step, the 16 mutation operators were cross-checked with the existing operators proposed in previous work. According to Silva et al. [41], four studies proposed mutation operators for these Android components. Jabbarvand and Malek [14] designed eight mutation operators for Location and four for Bluetooth. Linares-Vásquez et al. [20] proposed one mutation operator for Location and two for Bluetooth. Deng et al. [9] contributed with one Location operator and Liu et al. [21] designed one operator for Location and one for Bluetooth. Table 5 presents the relationship of the mutation operators from literature (see column *Literature Operators*) and the ones devised in the present paper (see column *Related to*).

The characteristics of the 18 mutation operators proposed in the literature were compared with the 16 operators of this study. An operator subsumes another if both have the same characteristic. For instance, the mutation operators NullGPSLocation and NullBluetoothAdapter from [20] have the same characteristics as NDID. Therefore, NDID subsumes both operators.

After analysis, the mutation operators presented in this paper subsume all the operators proposed in the previous study, showing the feasibility and generalizability of our operators.

4.2 Mutation Operator Description

Table 6 summarizes information on the proposed mutation operators.

As can be observed, six mutation operators are categorized as *Replacement* (DDBID, DVO, RAIDD, RCTDGM, RLPDR, and RLRBPDR). Four mutation operators are categorized as *Null* (BDC, BDL, NDID, NRDM). Two are grouped as *Switch* (SCDM and SCPDM). Two operators are categorized as *Trap* (TDC and TDL). DDM is categorized as *Deletion* and SDMC as *Shift*.

Table 4: Results of HAZOP.

Acronym	Guide Word	Cause	Consequence
BDC	OTHER THAN	Changes the current instantiation of a callback method with a null value.	Possible error, or the application may throw a NullPointerException.
TDC	AS WELL AS	Add a trap method to check the reachability of a callback method.	No loss of information, i.e., the callback method is reachable.
BDL	OTHER THAN	Changes the current instantiation of a listener method with a null value.	Possible error, or the application may throw a NullPointerException.
TDL	AS WELL AS	Adds a trap method to check the reachability of a listener method.	No loss of information, i.e., the listener method is reachable.
NDID	OTHER THAN	Changes the current instantiation of a method with a null value.	Possible error, or the application may throw a NullPointerException.
DDBID	OTHER THAN	Changes the current instantiation of a method with a default instance.	Possible error or a loss of object information.
SCDM	REVERSE	Reverts the current return value of a boolean method.	Unexpected result or behavior of the application.
SCPDM	REVERSE	Reverts the current boolean parameter of a method.	Unexpected result or behavior of the application.
SDMC	EARLY/LATE	Makes an early/late call of a method.	Possible application error.
DDM	NO/NONE	Deletes a declared method.	Unexpected behavior or collateral effects, such as increasing energy consumption.
DVO	MORE/LESS	Increases/Decreases a numeric value.	Possible unexpected result.
NRDM	OTHER THAN	Changes the reference parameter of a method with a null value.	Possible error or an unexpected result/behavior of the application
RAIDD	OTHER THAN	Randomly changes the value of a declared action intent.	Unexpected result or behavior of the application.
RCTDGM	OTHER THAN	Changes the declaration of a get method with a similar with the same return type	Possible application error.
RLPDR	OTHER THAN	Changes the provider of a Location method.	Unexpected behavior or collateral effects, such as increasing energy consumption.
RLRBPDR	OTHER THAN	Changes the priority of a LocationBuilder instantiation.	Unexpected behavior or collateral effects, such as increasing energy consumption.

Table 5: Mutation operator subsumption.

Ref	Literature Operators	Related to
[14]	LUF_T, LUF_D	DVO
	LRP_C, LRP_A	RLPDR
	RLU, RLU_P, RLU_D	DDM
	LKL	RCTDGM
	UAB	SCDM
	FDB_H, FDB_S	DVO
	RBD	DDM
[20]	NullGPSLocation	NDID
	BluetoothAdapterAlwaysEnabled	SCDM
	NullBluetoothAdapter	NDID
[9]	LCM	RCTDGM
[21]	NullLocation	NDID
	NullBluetoothAdapter	NDID

Table 6: Mutation operator categorization.

Category	Mutation Operator
Replacement	DDBID
	DVO
	RAIDD
	RCTDGM
	RLPDR
	RLRBPDR
Null	BDC
	BDL
	NDID
	NRDM
Switch	SCDM
	SCPDM
Trap	TDC
	TDL
Deletion	DDM
Shift	SDMC

The mutation operator marked with *L* refers to those that subsume an operator previously proposed in the academic literature. A brief description of the relationship between the operators is also provided.

4.2.1 Replacement Operators. This category groups six different mutation operators: DDBID, DVO, RAIDD, RCTDGM, RLPDR, and RLRBPDR. The main goal of this type of operator is to replace the current value of variables, method calls, parameters, and constants, or to instantiate an object with a different value instead of null. A brief description of the operators is presented below, and Figure 2 illustrates an example of applying three replacement mutation operators.

- **DefaultDeviceBuilderInstanceDeclaration (DDBID)**: The operator replaces the current instantiation of a Location or Bluetooth class with a default implementation. These classes can be instantiated with different parameters and/or properties. Therefore, this operator declares a builder instance without these parameters/properties.
- **DeviceVariablesOperator (DVO) – L**: The operator mutates scalar variables. In this case, the scalar variable may have its value incremented and decremented. It can be associated with operators LUF and FDB designed by Jabbarvand and Malek [14].
- **RandomActionIntentDeviceDefinition (RAIDD)**: The operator replaces the current declaration of a Bluetooth or Location action intent instance with a random value. It is worth noting that the action intent is mutated with another one of the same class.
- **ReplaceCompatibleTypeDeviceGetMethods (RCTDGM) – L**: The operator replaces the call of a get method of a given return type with another get method of the same return type and attached to the same object. This operator can be associated with the operator LKL from Jabbarvand and Malek [14] and LCM from Deng et al. [9].
- **RandomLocationProviderDeviceReplacement (RLPDR) – L**: The operator replaces the Provider constant parameter of a Location instance with a custom String value or an existing LocationManager provider. Observe that this operator is specific to the Location and can be associated with operator LRP from Jabbarvand and Malek [14].
- **RandomLocationRequestBuilderPriorityDeviceReplacement (RLRBPDR)**: The operator replaces the Priority constant parameter of a LocationRequest builder instance with a custom String value or an existing Priority constant. Similar to the previous operator, it is only applied to Location.

4.2.2 Null Operators. This category encompasses those mutation operators that replace the current instantiation of an object with a *null* value. Following, a description of the operators is presented as well as an example of applying two null operators (see Figure 3).

```java
// Example of DefaultDeviceBuilderInstanceDeclaration
private void example_DDBID() {
    // ORIGINAL CODE
    LocationRequest locationRequest = new LocationRequest
    .Builder(LocationRequest.PRIORITY_HIGH_ACCURACY)
    .setIntervalMillis(5000)  // 5 seconds
    .build();

    // MUTANT
    LocationRequest locationRequest = new LocationRequest();
}

// Example of RandomActionIntentDeviceDefinition
private void example_RAIDD() {
    ...
    // ORIGINAL CODE
    Intent enableIntent = new
    ↪ Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

    // MUTANT
    Intent enableIntent = new
    ↪ Intent(BluetoothAdapter.ACTION_DISCOVERABLE);
}

// Example of ReplaceCompatibleTypeDeviceGetMethods
private void example_RCTDGM() {
    ...
    // ORIGINAL CODE
    double latitude = location.getLatitude();

    // MUTANT
    double latitude = location.getLongitude();
}
```

Figure 2: Example of mutants generated by three different Replacement operators.

- **BuggyDeviceCallback (BDC)**: The operator changes the current instantiation of a Bluetooth or Location callback to a null value. That is, the callback has no response to an event or user interaction.
- **BuggyDeviceListener (BDL)**: The operator changes the current instantiation of a Bluetooth or Location listener to a null value. Similar to the previous operator, the listener has no response to an event or user interaction.
- **NullDeviceInstanceDeclaration (NDID)**: The operator changes a current instance declaration of a Bluetooth or Location class to a null value. This operator can be associated with operators NullGpsLocation and NullBluetoothAdapter from Linares-Vásquez et al. [20] and NullLocation and NullBluetoothAdapter from the study of Liu et al. [21].
- **NullReferenceDeviceMethods (NRDM)**: The operator changes the reference parameters of a method with a null value. This operator tackles method overloading.

4.2.3 Switch Operators. This category encompasses two operators: SCDM and SCPDM. This type of operator switches the boolean value of a method parameter or changes the boolean return value of a method call. A description of each operator is presented below. Figure 4 shows an example of applying SCDM and SCPDM.

- **SwitchConditionalDeviceMethod (SCDM)**: The operator switches the boolean return value of a Bluetooth or Location method to "true" and "false". That is, the operator creates one mutant with a "true" value and another with a "false" value. Additionally, it can be associated with operators UAB from Jabbarvand and Malek [14] and BluetoothAdapterAlwaysEnabled from Linares-Vásquez et al. [20].
- **SwitchConditionalParameterDeviceMethod (SCPDM)**: The operator switches the boolean parameter of a Bluetooth or Location method. Similar to the previous operator, it switches the value "true" to "false" and "false" to "true".

```java
// Example of BuggyDeviceListener
private void example_BDL() {
    ...
    // ORIGINAL CODE
    LocationListener locationListener = new LocationListener()
    ↪ {
        @Override
        public void onLocationChanged(Location location) {
            ...
        }
    };

    // MUTANT
    LocationListener locationListener = null;
}

// Example of NullReferenceDeviceMethods
private void example_NRDM() {
    ...
    // ORIGINAL CODE
    locationManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER,
        5000,
        10,
        locationListener
    );

    // MUTANT
    locationManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER,
        5000,
        10,
        null
    );
}
```

Figure 3: Example of mutants generated by two different Null operators.

```java
// Example of SwitchConditionalDeviceMethod
private void example_SCDM() {
    ...
    // ORIGINAL CODE
    boolean btEnabled = btAdapter.isEnabled();

    // MUTANT
    boolean btEnabled = true;
}

// Example of SwitchConditionalParameterDeviceMethod
private void example_SCPDM() {
    ...
    // ORIGINAL CODE
    beaconManager.setEnableScheduledJobs(false);

    // MUTANT
    beaconManager.setEnableScheduledJobs(true);
}
```

Figure 4: Example of mutants generated by two different Switch operators.

4.2.4 *Trap Operators.*

This operator inserts a trap method to reveal the reachability of a code in the application [1]. It is specifically inserted into the callback or listener interface method. Whenever the trap method is executed, the mutant is killed and hence, the callback or listener method is reachable. A brief description of the operators is presented, and Figure 5 exemplifies their application.

- **TrapDeviceCallback (TDC)**: The operator inserts a trap method inside a callback.
- **TrapDeviceListener (TDL)**: The operator inserts a trap method inside a listener method.

4.2.5 *Deletion Operator.*

This type of mutation operator deletes a statement declaration of the application. Below is a description of DDM and an example of its application (see Table 6).

- **DeletionDeviceMethods (DDM)**: The operator deletes a statement of methods that close, stop, and/or deallocate an existing process or service. It may cause collateral effects such as increasing the energy consumption of an application. This operator is associated with operators RLU and RBD from Jabbarvand and Malek [14].

4.2.6 *Shift Operators.*

This type of mutation operator moves the declaration of a method to another place in the source code. Following is a description of SDMC and an example of its application (see Figure 7).

- **ShiftDeviceMethodCall (SDMC)**: The operator shifts a Bluetooth or Location method call to another part of the source code. It can be shifted into the same method or in a different method from the same class.

4.3 Discussion

In the previous section, all 16 mutation operators were properly described. For space reasons, we did not present an example for all mutation operators. See Section "Data Availability" to access the URL link to the repository containing all supplementary data collected in this work.

The 16 mutation operators were grouped into six categories (Replacement, Null, Switch, Trap, Deletion, and Shift) to mimic the possible faults that may occur during development. Note that the proposed operators may not cover all Bluetooth, AltBeacon, and Location functionalities. To minimize this possible threat, we focused on the main functionalities of each component. Additionally, the mutation operators were designed to be generic and hence, it could be extended for other functionalities and mobile device resources.

Additionally, we cross-checked the proposed operators against those previously defined in the academic literature. As observed, six of these operators (DVO, RLPDR, DDM, RCTDGM, SCDM, and NDID) have subsumed the existing ones, whereas ten of them are new contributions to the present work.

Finally, when observing the characteristics of the proposed mutation operators, we found that parametrization is essential for some operators. For example, the mutation operator RandomActionIntentDeviceDefinition (RAIDD) requires a valid action intent from the same class. That is, if the operator mutates an action intent from "BluetoothAdapter", it is expected that the mutation operator replaces the intent with another one from the "BluetoothAdapter" class.

Note that parametrization makes the approach more flexible because it provides to the developer the possibility of creating a set

```
// Example of TrapDeviceCallback
private void example_TDC() {
    ...
    // ORIGINAL CODE
    private ScanCallback bleScanCallback = new ScanCallback() {
        @Override
        public void onScanResult(int callbackType, ScanResult result)
        ↪ {
            super.onScanResult(callbackType, result);
            ...
        }
    };

    // MUTANT
    private ScanCallback bleScanCallback = new ScanCallback() {
        @Override
        public void onScanResult(int callbackType, ScanResult result)
        ↪ {
            super.onScanResult(callbackType, result);
            TRAP_ON_CALLBACK();
            ...
        }
    };
}

// Example of TrapDeviceListener
public void example_TDL() {
    ...
    // ORIGINAL CODE
    LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            ...
        }
    }

    // MUTANT
    LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            ...
            TRAP_ON_LISTENER();
        }
    }
}
```

Figure 5: Example of mutants generated by two different Trap operators.

```
// Example of DeletionDeviceMethods

// ORIGINAL CODE
private void example_DDM() {
    ...
    fusedLocationProviderClient.flushLocations();
    ...
}

// MUTANT
private void example_DDM() {
    ...

    ...
}
```

Figure 6: Example of a mutant generated by DDM.

```
// Example of ShiftDeviceMethodCall

// ORIGINAL CODE
private void example_SDMC() {
    @Override
    public void onCreate(Bundle savedInstanceState) {
       ...
    }

    @Override
    public void onPause() {
       super.onPause();
       bluetoothServerSocket.close();
    }
}

// MUTANT
private void example_SDMC() {
    @Override
    public void onCreate(Bundle savedInstanceState) {
       ...
       bluetoothServerSocket.close();
    }

    @Override
    public void onPause() {
       super.onPause();
    }
}
```

Figure 7: Example of two mutants generated by SDMC.

of possible mutation operators to be applied. That is, the developer controls the intent, methods, or values to be mutated.

5 Empirical Cost Evaluation

This section provides an empirical cost evaluation of applying the mutation operators by assessing the number of mutants generated per application and the number of mutants generated per operator. For this evaluation, two subject apps were selected. Moreover, these apps were adopted in other studies [14, 36]:

1. **a2dpvolume**[4]: This app automatically adjusts the media volume on connect and resets on disconnect.
2. **runnerup**[5]: This app tracks sports activities using GPS from Android devices.

Each app was carefully evaluated, and the mutants were manually generated whenever a possible mutation point was found. For this purpose, the mutation point was replaced with a possible mutant, and the code was recompiled to check the validity of the mutation. Note that all mutants compiled successfully.

In this analysis, only a single mutant was generated for the mutation point that allows applying the same mutation operators multiple times. For instance, the operator RAIDD may generate a large number of mutants considering a mutation point, but only a single mutant is considered in the analysis. Therefore, this analysis provides the lower boundary mutants generated per app as presented in Table 7. From now on, we refer to the a2dpvolume application as APP_1 and to the runnerup application as APP_2.

As can be observed, the APP_1 is a smaller application but generates 35 more mutants than APP_2. One plausible explanation is that the APP_2 has fewer classes related to Location and Bluetooth that

[4]https://github.com/jroal/a2dpvolume
[5]https://github.com/jonasoreland/runnerup

Table 7: Number of mutants generated per app.

App Id	Name	# Classes	# Mutated Classes	# Mutants
APP_1	a2dpvolume	22	7	86
APP_2	runnerup	150	12	51

can be mutated compared to APP_1. Additionally, APP_1 generates mutants for Bluetooth and Location, whereas APP_2 generates only for Location. Moreover, none of the applications generated mutants for the AltBeacon API, which may reflect the lack of mutant generation for some mutation operators.

Next, Figure 8 presents the number of generated mutants per operator.

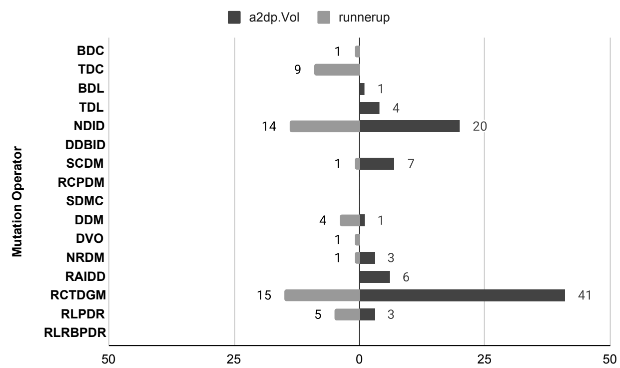

Figure 8: Number of mutants generated per operator.

The results show that the operator RCTDGM generates the largest number of mutants for both apps. In the case of APP_1, 41 mutants were generated, representing a total of 47% of mutants. APP_2 generated 15 mutants, that is, 29% of all mutants. It is worth noting that the mutation operator was applied once per mutation location. Therefore, the number of mutants generated would increase if the operator replaces the call of the get methods from an object with all possible permutations.

The second operator that generated the most mutants was NDID. For APP_1, the operator generated 20 mutants (i.e., 23%) whereas APP_2 generated 14 mutants (i.e., 27%). Additionally, five mutation operators generated mutants for a single application. The operators BDL and TDL only mutated APP_1 while the operators BDC, TDC, and DVO mutated APP_2.

On the other hand, four operators did not generate any mutants for the two applications: DDBID, RCPDM, SDMC, and RLRBPDR. Note that this does not imply that these are useless mutation operators. For instance, the RCPDM operator is useful for applications that rely on the AltBeacon library due to the API structure. Similar to RLRBPDR, which requires the instantiation of a *LocationRequest.Builder* method to generate the mutant. In this case, it was possible to observe that the subject apps did not implement those functionalities that the operators mutate and hence, the statistics may change if a broader set of subject apps is considered.

Observe that this analysis only focuses on the viability of the mutation operators, i.e., if they are capable of generating mutants for Bluetooth and Location applications. Therefore, it is expected that in the future, an in-depth evaluation of the proposed mutation operators regarding their usefulness, i.e., subsumption, equivalence, and triviality [16].

6 Related Work

6.1 Android Mutation Operators

Silva et al. [41] presented a systematic mapping study assessing the studies related to mutation testing for mobile applications. As a result, the study found that a total of 138 mutation operators were proposed across 16 primary studies, as the majority focus on configuration and GUI aspects.

On the other hand, few studies focus on other mobile device components such as connectivity, location, and sensors. That is, only four studies proposed at least one mutation operator for these components. Jabbarvand and Malek [14] investigated mutation operators aiming at energy consumption. That is, the faults are modeled to verify whether they may result in a difference in energy consumption between the mutant and the original AUT. In total, 50 mutation operators were designed, of which eight related to Location and four related to Bluetooth.

Linares-Vásquez et al. [20] proposed an in-depth study that defines a taxonomy of the main Android faults. Then, 38 mutation operators were devised based on the taxonomy. One operator related to the Location injects a null GPS location. Additionally, two mutation operators are associated with Bluetooth: one makes the "BluetoothAdapter" instance always enabled, whereas the other instantiates "BluetoothAdapter" with a null value.

Deng et al. [9] designed 17 mutation operators categorized as: Event-based, Component Lifecycle, XML-related, Common Faults, Context-aware, Energy-related, and Network-related. From this set, only one relates to Location, which mutates attributes such as latitude, longitude, altitude, and speed.

Finally, Liu et al. [21] proposed 32 mutation operators divided in Android-specific operators and Java-specific operators. Only one Bluetooth mutation operator was proposed that instantiates "BluetoothAdapter" with null, similar to the operator proposed by Linares-Vásquez et al. [20].

6.2 HAZOP for Mutation Operators Design

As discussed, there is still no standardized methodology for mutation operator design. However, some initiatives try to use existing systematic approaches aiming at minimizing any possible bias.

The Hazard and Operability approach, or HAZOP, was applied in some studies to facilitate the mutation operator design. Kim et al. [18] uses HAZOP to devise mutation operators for Java programs.

Araujo et al. [2] applied HAZOP to design mutation operators for Dynamic System Models totaling 12 operators. Savarimuthu and Winikoff [40] applied HAZOP for the GOAL language, devising 21 mutation operators. Zhang et al. [45] proposed 191 mutation operators for Restricted Use Case Modelling (or RUCM).

7 Conclusion

This paper provided an initial investigation of the design of Android mutation operators for specific components, i.e., Bluetooth and Location. This study considered the Android native API and the third-party library AltBeacon. The rationale is that the present study is being conducted in an academia-industry partnership in which the company develops applications that heavily rely on these components.

Overall, this work resulted in 16 different Bluetooth and Location mutation operators by using a systematic approach named HAZOP. The operators were categorized based on their types: Replacement, Null, Switch, Trap, Deletion, and Shift. Additionally, we validated the mutation operators by manually generating the mutants, considering two Android applications showing the validity and feasibility of the operators, i.e., their capabilities of generating real faults related to these two components.

By providing a generic set of mutant operators, we advocate that the extension of the mutation operators for other mobile device components, such as Wi-Fi, NFC, and sensors, would be easily carried out. All these components have an API structure similar to Bluetooth and Location, thus, many operators can be reused for these device resources. For instance, they rely on callback or listener functions to handle events or actions, thus, it is possible to extend the operators BDC, BDL, TDC, and TDL to handle these device components. Therefore, we intend to extend the set of mutation operators once they are validated in a controlled experimental environment and an industrial scenario.

Regarding automation aspects, we intend to implement the mutation operators for automatic generation and execution of the mutants. As presented by Silva et al. [41], there are still few tools that support mutation testing for mobile applications. In fact, only three of them support all stages of mutation testing: generation, execution, and analysis. Recently, Vincenzi et al. [44] proposed METFORD, a mutation testing framework that implements a set of mutation operators considering the Mutation Schemata [39, 42] and traditional approach.

In an initial analysis, METFORD appears to address the implementation requirements of the proposed operators, as it is an open-source tool that supports the parametrization of mutation operators. That is, it allows the tester to set mutation operators to be applied and their parameters. For instance, one can set the scalar values to be incremented and decremented from the DVO operator. This provides a more controllable environment for the tester to inject all the desired faults. Additionally, this approach also benefits the reuse of the proposed mutation operators for other device components without enlarging the existing set.

We intend to validate the mutation operators through experimentation using open-source Android applications and conduct case studies in an industrial scenario. The main difficulty is finding Bluetooth, Location, or AltBeacon apps with implemented tests. Pecorelli et al. [38] conducted an empirical study and found that few Android applications (approximately 40% of the evaluated apps) contain at least one test suite. Moreover, Vincenzi et al. [44] showed the poor quality of existing test suites of Android applications, given the defect models proposed in their work. Therefore, we will explore strategies to establish a viable benchmark for validating the mutation operators.

ARTIFACT AVAILABILITY

The following repository contains all the supplementary data collected from this work: https://github.com/phkuroishi/paper-sbes-device-mutation-operator.

ACKNOWLEDGMENTS

The authors would like to thank the funding agencies that helped carry out this work: CAPES (Grant nº 001 and 88887.888653/2023-00), CNPq (Grant nº 141137/2021-5 and 140435/2025-5), and FAPESP (Grant nº 2019/23160-0 and 2023/00001-9).

REFERENCES

[1] Hiralal Agrawal, Richard A DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, and Eugene Spafford. 1989. *Design of mutant operators for the C programming language.* Technical Report. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue.

[2] Rodrigo Fraxino Araujo, José Carlos Maldonado, Márcio Eduardo Delamaro, Auri Marcelo Rizzo Vincenzi, and François Delebecque. 2011. Devising mutant operators for dynamic systems models by applying the HAZOP study. In *International Conference on Software Engineering Advances - ICSEA 2011.* IARIA - The International Academy Research and Industry Association, Barcelona, Spain, 58–64.

[3] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. 2006. Mutation Operators for Concurrent Java (J2SE 5.0). In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006).* IEEE, Raleigh, NC, USA, 11–11. doi:10.1109/MUTATION.2006.10

[4] M. F. Chudleigh, J. R. Catmur, Arthur D. Little, and F. Redmill. 1995. A Guideline for HAZOP Studies on Systems which include a Programmable Electronic System. In *Safe Comp 95*, Gerhard Rabe (Ed.). Springer London, London, 42–58.

[5] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur. 2001. Interface Mutation: an approach for integration testing. *IEEE Transactions on Software Engineering* 27, 3 (2001), 228–247. doi:10.1109/32.910859

[6] Márcio Delamaro, Mauro Pezzè, and Auri Vincenzi. 2001. Mutant Operators for Testing Concurrent Java Programs. In *Anais do XV Simpósio Brasileiro de Engenharia de Software* (Rio de Janeiro/RJ). SBC, Porto Alegre, RS, Brasil, 272–285. doi:10.5753/sbes.2001.23994

[7] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[8] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2017. Mutation operators for testing Android apps. *Information and Software Technology* 81 (Jan. 2017), 154–168. doi:10.1016/j.infsof.2016.04.012

[9] Lin Deng, Jeff Offutt, and David Samudio. 2017. Is Mutation Analysis Effective at Testing Android Apps?. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS).* IEEE, Prague, Czech Republic, 86–93. doi:10.1109/QRS.2017.19

[10] Anna Derezińska and Konrad Hałas. 2014. Analysis of Mutation Operators for the Python Language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk (Eds.). Springer International Publishing, Cham, 155–164. doi:10.1007/978-3-319-07013-1_15

[11] Camilo Escobar-Velásquez, Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2022. Enabling Mutant Generation for Open- and Closed-Source Android Apps. *IEEE Transactions on Software Engineering* 48, 1 (Jan. 2022), 186–208. doi:10.1109/TSE.2020.2982638

[12] Jonathan Fürst, Kaifei Chen, Hyung-Sin Kim, and Philippe Bonnet. 2018. Evaluating Bluetooth Low Energy for IoT. In *2018 IEEE Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench).* IEEE, Porto, Portugal, 1–6. doi:10.1109/CPSBench.2018.00007

[13] S. Gowrishankar, N. Madhu, and T. G. Basavaraju. 2015. Role of BLE in proximity based automation of IoT: A practical approach. In *2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS).* IEEE, Trivandrum, India, 400–405. doi:10.1109/RAICS.2015.7488449

[14] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017).* Association for Computing Machinery, New York, NY, USA, 208–219. doi:10.1145/3106237.3106244

[15] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[16] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring mutant utility from program context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017).* Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/3092703.3092732

[17] Sunwoo Kim, John Clark, and John Mcdermid. 2000. Class mutation: Mutation testing for object-oriented programs. In *Proc. Net. ObjectDays*. 9–12.

[18] Sunwoo Kim, John Clark, and John Mcdermid. 2000. The rigorous generation of Java mutation using HAZOP. In *In Proceedings of the 12 the International Conference on Software and Systems Engineering and Their Applications (ICSSEA'99).* TBA, Paris, France.

[19] M. Lakshmi, Alolika Panja, Naini, and Shakti Mishra. 2019. Customer's Activity Recognition in Smart Retail Environment Using AltBeacon. In *Emerging Research in Computing, Information, Communication and Applications*, N. R. Shetty, L. M. Patnaik, H. C. Nagaraj, Prasad Naik Hamsavath, and N. Nalini (Eds.). Springer Singapore, Singapore, 591–604.

[20] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017).* Association for Computing Machinery, New York, NY, USA, 233–244. doi:10.1145/3106237.3106275

[21] Jian Liu, Xusheng Xiao, Lihua Xu, Liang Dou, and Andy Podgurski. 2020. DroidMutator: an effective mutation analysis tool for Android applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20).* Association for Computing Machinery, New York, NY, USA, 77–80. doi:10.1145/3377812.3382134

[22] Google LLC. 2025. Bluetooth overview. https://developer.android.com/develop/connectivity/bluetooth/. (Accessed on 28/03/2025).

[23] Google LLC. 2025. Location overview. https://developer.android.com/develop/sensors-and-location/location. (Accessed on 28/03/2025).

[24] Google LLC. 2025. Overview of Google Play Services. https://developers.google.com/android/guides/overview. (Accessed on 28/03/2025).

[25] Eduardo Luna and Omar El Ariss. 2018. Edroid: A Mutation Tool for Android Apps. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT).* IEEE, San Luis Potosi, Mexico, 99–108. doi:10.1109/CONISOFT.2018.8645883

[26] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, Memphis, TN, USA, 100–111. doi:10.1109/ISSRE.2018.00021

[27] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt. 2002. Inter-class mutation operators for Java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.* IEEE, Annapolis, MD, USA, 352–363. doi:10.1109/ISSRE.2002.1173287

[28] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for java. In *Proceedings of the 28th international conference on Software engineering (ICSE '06).* Association for Computing Machinery, New York, NY, USA, 827–830. doi:10.1145/1134285.1134425

[29] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. Efficient JavaScript Mutation Testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* IEEE, Luxembourg, Luxembourg, 74–83. doi:10.1109/ICST.2013.23

[30] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *2012 7th International Workshop on Automation of Software Test (AST).* IEEE, Zurich, Switzerland, 29–35.

[31] Muneeb Muzamal and Aamer Nadeem. 2019. Improving test adequacy assessment by novel JavaScript mutation operators. In *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST).* IEEE, Islamabad, Pakistan, 647–652. doi:10.1109/IBCAST.2019.8667222

[32] Radius Network and David G. Young. 2025. AltBeacon overview. https://altbeacon.github.io/android-beacon-library/. (Accessed on 28/03/2025).

[33] Kazuki Nishiura, Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden. 2013. Mutation Analysis for JavaScriptWeb Application Testing.. In *SEKE*, Vol. 2013. KSI Research Inc, Boston, Massachusetts, USA, 159–165.

[34] James Jerson Ortiz Vega, Gilles Perrouin, Moussa Amrani, and Pierre-Yves Schobbens. 2018. Model-Based Mutation Operators for Timed Systems: A Taxonomy and Research Agenda. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS).* IEEE, Lisbon, Portugal, 325–332. doi:10.1109/QRS.2018.00045

[35] Ana C. R. Paiva, João M. E. P. Gouveia, Jean-David Elizabeth, and Márcio E. Delamaro. 2019. Testing When Mobile Apps Go to Background and Come Back to Foreground. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, Xi'an, China, 102–111. doi:10.1109/ICSTW.2019.00038

[36] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications.

In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 153–164. doi:10.1145/3395363.3397354

[37] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, Atif M. Memon (Ed.). Advances in Computers, Vol. 112. Elsevier, 275–378. doi:10.1016/bs.adcom.2018.03.015

[38] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2021. Software testing and Android applications: a large-scale empirical study. *Empirical Software Engineering* 27, 2 (14 Dec 2021), 31. doi:10.1007/s10664-021-10059-5

[39] Macario Polo-Usaola and Isyed Rodríguez-Trujillo. 2021. Analysing the combination of cost reduction techniques in Android mutation testing. *Software Testing, Verification and Reliability* 31, 7 (2021), e1769. doi:10.1002/stvr.1769

[40] Sharmila Savarimuthu and Michael Winikoff. 2013. Mutation Operators for the Goal Agent Language. In *Engineering Multi-Agent Systems*, Massimo Cossentino, Amal El Fallah Seghrouchni, and Michael Winikoff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 255–273.

[41] Henrique Neves Silva, Jackson Prado Lima, Silvia Regina Vergilio, and Andre Takeshi Endo. 2022. A mapping study on mutation testing for mobile applications. *Software Testing, Verification and Reliability* 32, 8 (2022), e1801. doi:10.1002/stvr.1801

[42] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation analysis using mutant schemata. *SIGSOFT Softw. Eng. Notes* 18, 3 (July 1993), 139–148. doi:10.1145/174146.154265

[43] Enrico Viganò, Oscar Cornejo, Fabrizio Pastore, and Lionel C. Briand. 2023. Data-Driven Mutation Analysis for Cyber-Physical Systems. *IEEE Transactions on Software Engineering* 49, 4 (April 2023), 2182–2201. doi:10.1109/TSE.2022.3213041 Conference Name: IEEE Transactions on Software Engineering.

[44] Auri M. R. Vincenzi, Pedro H. Kuroishi, João Bispo, Ana R. C. da Veiga, David R. C. da Mata, Francisco B. Azevedo, and Ana C. R. Paiva. 2025. METFORD – Mutation tEsTing Framework fOR anDroid. *Journal of Systems and Software* 222 (April 2025), 112332. doi:10.1016/j.jss.2024.112332

[45] Huihui Zhang, Tao Yue, Shaukat Ali, and Chao Liu. 2016. Towards mutation analysis for use cases. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems* (Saint-malo, France) *(MODELS '16)*. Association for Computing Machinery, New York, NY, USA, 363–373. doi:10.1145/2976767.2976784